C++ Final Project : Option Pricer

Report

Aim of the project :

Create an option pricer which allows us to determine the price of an option (European, Digital, Asian or American) from given option's characteristics such as the interest rate r, the volatility sigma, the strike price K, the maturity T, the underlying price S and the cost of carry b.

In order to determine the option's price, we can use different method according to the type of the option (European, Digital, Asian or American) and its nature (call or put).

In this project, we have implemented the different method of pricing so that we might be able to determine the option's price by using different way and then compare the prices obtained from the diverse methods of pricing.

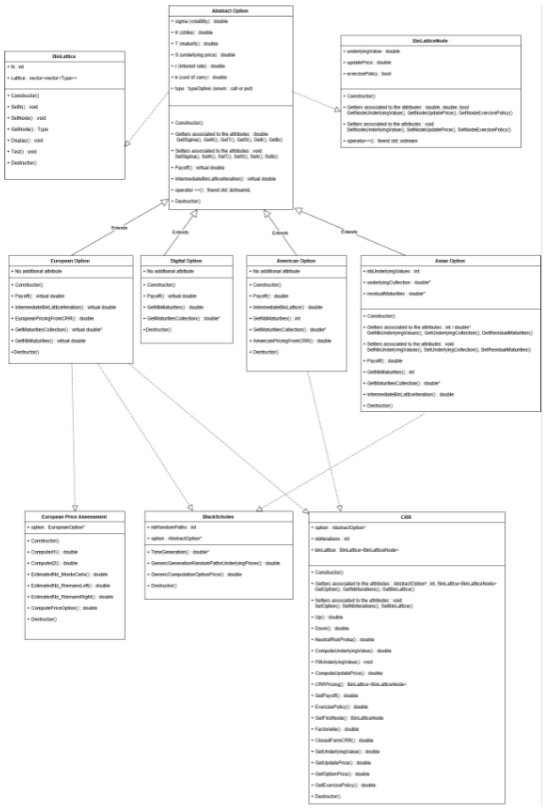
The final aim is that the option's prices obtained thanks to our pricer be as close as possible to the market price.

Architecture of the project :

We have chosen to structure our project by distinguishing the characteristics and properties specific to each type of options. To do that, we have created several classes with interactions between themselves.

We have implemented 10 classes which are the following :

- Abstract Option Class
- European Option Class
- Digital Option Class
- Asian Option Class
- American Option Class
- BinLattice Node Class
- BinLattice Class
- CRR (Cox Ross Rubinstein) Class
- BlackScholes Class
- European Price Assessment Class



You can see the JPG document to have a better view of the diagram.

Implementation :

Abstract Option Class

In order to determine the price of the options, we have to use different methods according to their types (European, Digital, Asian, American). Indeed, the type of an option determines the characteristics which are specific to this option as well as the way to use to compute its price. Some characteristics are shared by all the options wathever their types whereas others are specific to one particular type of option. In order to represent as best as possible this architecture, we have chosen to create a parent class called Abstract Option from which all the types of options inherit. This class contains the attributes and properties shared by all the types of options. That is to say, each option, wathever its type, belongs to the Abstract Option's class.

The attributes of this class are the following : the interest rate r , the volatility sigma, the strike price K, the maturity T, the underlying price S, the cost of carry b which are shared by all the types of options.

The methods implemented in this class are the following : a virtual method called « Payoff » which takes as inputs a number of underlying values and an array of underlying values and which returns a double, a virtual method called « Intermediate BinLattice Iteration » which takes as inputs an update price and a temporary payoff and which returns a double, a display method and the Getters and Setters of the attributes.

The constructor enables us to create an Abstract option object from the attributes of the class.

The getters associated to each attribute of the class allow us to have access to these attributes.

The setters associated to each attribute of the class allow us to modify the value these attributes.

The Method « Payoff » computes the gain that we can obtain with our option when a maturity T and a collection of underlying values are given.

The Method « Intermediate BinLattice Iteration » is especially used in the European and American class and returns a double corresponding either to the update price or to the current payoff according to which is upper to the other.

The virtual methods are first defined in the Abstract Option Class without being implemented in this class, and then they are implemented in each child class in different way according the specificities of the class. The virtual methods enable us to define the same function for several class but with different implementation according the characteristics of each class.

European Option Class

The European Option Class inherits from the Abstract Option Class which contains the attributes and properties shared by all the types of options. The European Option Class does not contains other attributes in addition to those contained in the Abstract Option Class. In this class are implemented the following methods :

A constructor which aims at creating a European Option Object from its attributes : the interest rate r, the volatility sigma, the strike price K, the maturity T, the underlying price S, the cost of carry b. It's the same constructor than the one used to create an Abstract Option.

The method « Payoff » aims at computing the gain that we are able to obtain with our option at a maturity T, when a number of underlying prices and a collection of underlying prices are given.

The method « Intermediate BinLattice Iteration » is a method implemented so that we might be able to applicate the pricing of American options to the European Options with the Binomial Tree and the Cox Ross Rubinstein Method. This method takes as inputs a temporary payoff as well as an update price and returns the update price value (for the American Option : either update price if it is upper than the current payoff or the payoff if it is upper than the update price).

Two methods « Get Nb Maturities » and « Get Maturities Collection » which respectively allows to return the number of residual maturities that are required for a European Option as well as the collection of residual maturities that corresponds.

A method « European Pricing From CRR » which allows to return the first node of the Binomial tree corresponding to the European Option price.

Digital Option Class

The Digital Option Class inherits from the Abstract Option Class what means that the Digital Option Class owns the same attributes than the Abstract Option Class and it has no additional attributes.

In this class, we have implemented a constructor so that we might be able to create a Digital Option from the attributes contained in the Abstract Option Class.

The method called « Payoff » allows to compute the payoff of a digital option when a number of underlying values and a collection of underlying values are given.

The method called « Get Nb Maturities » returns the number of residual maturities that we need to take into account to compute the payoff of the option. In the case of a Digital Option, in the same way than in the case of the European or the American options, the number of residual maturities that we consider in the computation of the payoff is equal to 1.

The method called « Get Maturities Collection » returns the collection of residual maturities at which we consider the underlying values to compute the payoff of the option.

Asian Option Class

The Asian Option Class takes as attributes both the attributes of the Abstract Option Class and additional attributes which are specific to the Asian Options : the number of underlying values, a collection of underlying values and the associated collection of residual maturities.

In this class, we have implemented the properties Getter and Setter associated to the attributes of the class.

We have implemented a method called « Payoff » which computes the gain that we can hope obtaining at a maturity T when the number of underlying values as well as the collection of underlying values are given.

The two methods « Get Number Maturities » and « Get Maturities Collection » are two methods which respectively return the number of residual maturities and the collection of residual maturities associated to the underlying values that we take into account in the payoff computation.

The method called « Intermediate BinLattice Iteration » is not used for the Asian Option but we have to implement the method in each child class so this method returns a double corresponding to the update price.

American Option Class

The American Options differ from the European options accordingly that we can exercise the option at each date between the purchase of the option and its maturity date. The American Option Class inherits from the Abstract Option Class. It owns the same attributes than the Abstract Option class.

We have implemented a constructor to create an American Option from the attributes inherited from the Abstract Option Class.

The method called « Payoff » aims at computing the gain that we can hope to obtain at each residual maturity date when the number of underlying values and the collection of underlying values required to compute the gain are given.

The method called « Intermediate BinLattice Iteration » is a method which allows to determine the price of both American and European options thanks to the binomial tree. If we consider the binomial tree in which each node contains an underlying price, an update price and a boolean representing the exercise policy, for a European option we compute its price by calculating at each iteration of the tree, the updated value of the mean of the payoffs at the following iteration and we continue this process until the root of the binomial tree, whereas for an American Option we compute its price by calculating at each iteration of the tree, the updated value of the mean of the

prices at the following iteration which are either equal to the payoff if it is upper to the update price or to the update price if it is upper to the payoff.

The method called « GetNbMaturities » returns the number of residual maturities required to compute the payoff of an American Option.

The method called « GetMaturitiesCollection » returns the collection of residual maturities at which we take into account the underlying values to compute the payoff.

The method called « American Pricing From CRR » allows to return the price of an American Option when a binomial tree of prices is given and when index of height and depth of the tree which correspond to the date at which the user wants to exercise its option are given.

BinLattice Node Class

We have implemented the BinLattice Node Class so as to create the nodes of our binomial tree. This class takes as parameters : an underlying price, an update price and a boolean for the exercise policy at a given date.

We have also implemented the Getters and the Setters associated to each attribute of the BinLattice Node Class so that we might have access to the attributes of the nodes by reading and writting.

BinLattice Class

We have implemented the BinLattice Class in order to create a binomial tree which allows to determine the price of the options at each given date between the current moment and the maturity date.

This class takes as inputs a number of iterations N as well as a vector of vectors which represents the binomial tree that we want to build.

We have implemented a constructor which takes as inputs the attributes of the class that is to say the number of iterations N corresponding to the depth of the tree and a vector of vectors. This constructor allows to build a binomial tree which is a vector of N vectors, each of them having a variable size. The depth of the tree is equal to N and the height is variable.

The Setter of the number of iterations N called « SetN » allows to modify the value of N and to size or resize the binomial tree according to this value. We create a vector of N vectors and we resize each of the N vectors in such a way that each of the N vectors has a size going from 1 to N+1.

The Setter called « SetNode » allows to modify the value of the attribute stored in each node.

The Getter called « GetNode » allows to return the value stored in each node of the binomial tree.

We have also implemented a method called « Display » which allows to display the index representing the height and the depth of each node in the binomial tree as well as the values stored in each node.

In this class, we have created a template which allows us to create a BinLattice object taking as parameters different types of variables. The BinLattice objects is an object created from the following attributes : an integer corresponding to the number of iterations in the binomial tree and a vector of vector containing variables whose the type is variable according to which elements we

would like to store in our vector of vector that's to say in our binomial tree. The template allows to store different objects in the binomial tree.

CRR Class

The CRR Class takes as attributes an Abstract Option, a number of iterations which corresponds to the depth of the binomial tree, a vector of vector from the BinLattice and BinLatticeNode Class which represents the binomial tree.

The CRR Constructor enables to create a binomial tree from two parameters : the vector of vectors which represents the architecture of the binomial tree and the number of iterations that we want to take into account to build our binomial tree.

We have implemented the Getters and Setters associated to the attributes of the CRR Class so that we might be able to have access to the attributes by reading and writting.

We have implemented two methods called « Up » and « Down » which allow to compute the underlying values from the root of the binomial tree to the leaves in a forward way. At each time that we go up, the previous underlying value is multiplied by the value of U and at each time that we go down, the previous underlying value is multiplied by the value of D. Thus, to compute a given underlying value we just have to know at which number of iterations it corresponds in the binomial tree and either the number of U or the number of D (the other is equal to the number of iterations minus the number of U or D) which has enabled us to reach the underlying value that we want to determine. Then we compute this underlying value by multiplying the underlying value at time 0 by the value of U at the power of the number of U and by the value of D at the power of the number of D.

We have implemented the method called « Neutral Risk Proba » in order to compute the probability p corresponding to the probability that enables us to go up between to adjacent steps. The probability that we go down between to adjacent steps is then equal to (1-p). In order to compute p we use the Up and Down methods with the following parameters : the number of iterations in the binomial tree and 1 for the number of Up and the number of Down as we compute the probability allowing to pass from a step to another adjacent step.

The method called « Fill Underlying Values » enables us to fill the binomial tree with the correct underlying values at each iterations. To reach this aim, we start by filling the highest branch of the binomial tree as in this branch each underlying value at each iteration is computed by going up in the tree from the previous iteration and then by multiplying the previous underlying value by the U value. We store these underlying values in the nodes of the binomial tree. Then we compute the other underlying values which are obtained by going down once time from the previous underlying value and then by multiplying the previous underlying value by the D value. Thus we store these underlying values in the nodes of the binomial tree.

The method called « Compute Underlying Value » aims at computing the underlying value when a number of U values and a number of D values are given that's to say at a given position in the tree when we know the number of U and D that we have crossed to reach this position. This method computes the underlying value at a given position in the binomial tree by multiplying the initial underlying value S corresponding to the time 0 by the U value at a power equal to the number of U crossed to reach the position of the underlying value that we want to compute and by the D value at a power equal to the number of D crossed to reach the position of the underlying value that we want to compute.

The method called « Compute Update Price » enables to compute the update price at a given date from two known prices at the following iteration (an up price and a down price). To compute the update price at the given date we compute the weighted average of the two prices (up and down) respectively weighted by the neutral risk probability and 1 minus the neutral risk probability, multiplied by an update factor. Then if the aim is to compute the price of a European option, we keep the update price and we continue this process until the root of the binomial tree. But if the aim is to compute the price of an American option, we have to compare the computed update price to the current payoff, we keep the one that is the highest and we throw the other. That is for this reason that we have implemented the method called « Intermediate BinLattice Iteration » which is mainly used in the European and American class. This method allows to return at each iteration the update price if the option that we are pricing is european and either the update price or the payoff (the highest of themselves) if we are pricing an American option.

The method called « CRRPricing » allows to compute for each node the price of the option if the owner of the option wants to exercise its option. To compute the price of the American option in each node of the binomial tree, we start by computing the payoff corresponding to the underlying values stored in the last iteration of the binomial tree. These payoffs at maturity are equal to the update prices. We compute the exercise policy which is a binomial variable equal to true if the current payoff is upper to the update price and false otherwise. Then we cross the tree in a backward way from the iteration number « nbIterations – 1 » and for each iteration we cross the nodes from the top to the bottom of the tree. At each time, we consider two adjacent iterations of the tree, we compute each pair of update prices (update price up, and update price down) stored in the iteration j+1 and from this pair of update prices we are able to compute the update price of the iteration j by computing the weighted average of these two update prices multiplied by the update factor thanks to the method called « Compute Update Price ». Then we update the exercise policy attribute of the node by comparing the current payoff to the update price and if the current payoff is upper to the update price we assign the value true to the exercise policy, otherwise we let it at false. The function « CRRPricing » returns then the binomial tree filled with following attributes : underlying value, update price and exercise policy.

We have implemented Getters so that we might able to get the attributes stored in each node back :

The method called « Get Payoff » allows to compute the payoff of the underlying value stored in a node of the binomial tree whose the index of height and depth are given and to get this value back.

The method called « Get Update Price » allows to get the update price stored in a node whose the index of height and depth are given, back.

The method called « Get Exercise Policy » allows to get the exercise policy stored in a node whose the index of height and depth are given, back.

The method called « Get First Node » allows to return the root node of the binomial tree. This method is used when we want to compute the price of a European option as we can not exercise the option at each date between the purchase and the maturity date of the option but we have the obligation to exercise the option at the maturity date. Thus, in this case we want to compute the update price at the maturity updated to the current date (root node).

We have implemented the method called « Factorielle » so as to compute the binomial coefficient in the closed form of Cox Ross Rubinstein method.

The method called « Closed Form CRR » allows to compute the price of a European option without using a binomial tree. This function implements the closed form CRR formula which is given by the following formula :

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

Where h(S(N,i)) is the payoff of the option at the iteration N and the node i of the binomial tree.

BlackScholes Class

In this class, we have implemented all the methods which are specific to the pricing of Asian Options and which are also suitable to the European Options. This class takes as attributes an Abstract Option and the number of generated random paths.

In the method called « Generic Generation Random Paths Underlying Prices » , we compute the m underlying values required to compute the payoff of the option. So as to compute the m underlying values, we generate m Gaussian random variables and then we use the Wiener Process. We store the m underlying values in an array. Finally, we compute the payoff of the option from the collection of the m underlying values. This method returns the payoff of the option.

In the method called « Generic Computation Option Price », we generate random paths in which we call the method called « Generic Generation Random Paths Underlying Prices » so that for each random path of m underlying values, we compute the corresponding payoff. Once the generation of random paths ended, we obtain a number of payoffs equal to the number of generated random paths, thus we compute the empirical mean of the payoffs. The updated mean of the payoffs corresponds to the price of the option.

The Wiener Process formula is given by :

$$S(t_1) = S(0)e^{\widehat{\left(r-\frac{\sigma^2}{2}\right)t_1+\sigma\sqrt{t_1}\widehat{Z_1}}}$$

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

We have implemented the method called « Time Generation » which takes as inputs a minimal boundary, a maximal boundary and a step and which returns an array of residual maturities. This method allows to easily obtain an array of residual maturities to pass in parameters of the other computation methods.

European Price Assessment Class

The Class « European Price Assessment » is a class created to implement all the methods specific to the European Options.

This class takes as attributs an Abstract Option and as methods all the methods require to compute the estimation of the European Option price with Black-Scholes Model.

Black-Scholes Model is given by :

$$Call : C(S,t) = S_t N(d_1) - Ke^{-r(T-t)} N(d_2)$$

$$Put : P(S,t) = Ke^{-r(T-t)} N(-d_2) - S_t N(-d_1)$$

Before implementing the Black-Scholes Model, we have to estimate two unknown parameters of the formula which are :

$$N(d_1)\ and\ N(d_2)$$

These two unknown parameters correspond to the area under the Gaussian density curve respectively marked out by the two parameters $d_1\ and\ d_2$.

First, we compute the parameters $d_1\ and\ d_2$ thanks to the following formulas :

$$d_1 = \frac{\ln\left(\frac{S_t}{K}\right) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

In this aim we have implemented the two methods : « Computed1 » and « Computed2 » which return the two parameters $d_1\ and\ d_2$.

Then, we can either estimate $N(d_1)\ and\ N(d_2)$ by using the Monte-Carlo Method or by the Riemann Approximation. In order to do that we have implemented the Monte-Carlo method which consists in generating a large number of Gaussian variable, in counting the number of Gaussian variables that are lower than the parameters $d_1\ or\ d_2$ which is passed as parameter of the function, and in computing the ratio of the number of Gaussian variables lower than the parameter over the total number of Gaussian variables generated. The Monte-Carlo method returns this ratio as an approximation of $N(d_1)\ or\ N(d_2)$.

We have also implemented two methods which compute the Riemann Sum respectively Left and Right in order to compare the approximations by Riemann methods to the ones by Monte-Carlo methods. We can not directly compute the Riemann Sum (left or right) marked out by the parameters $d_1\ or\ d_2$ as one of the boundary is infinity so we start by computing the Riemann Sum between one of the parameters and 0 and as the Gaussian density is symetric we then compute the Riemann Sum between infinity and the parameter by removing to 0.5 the value of the Riemann Sum between the parameter and 0. To compute the Riemann Sum we compute the Sum of the area of each rectangle under the Gaussian density whose the width is equal to (0 - d_1) or (0 - d_2) divided by the number of the generated variables and the height which is equal to the Gaussian density applicated to the abscissa equal to a multiple of the width (step).

Once this approximation is made, we can implement the Black-Scholes model. We implement this model in the method called « Compute Option Price » by distinguishing the computation of the price according to if it is a put or a call and according to which approximation method we have chosen to approximate the parameters $N(d_1)\ and\ N(d_2)$.

Main

In the Main, we have created for each different option class, an option object (European, Digital, American, Asian, Abstract). We have defined the parameters required to create the different option

objects at the top of the main so that the user might be able to easily change their values in order to make several simulations.

Then we have created several sections, one specific to each type of options. In each of these sections, we have created an option object according to the type considered in the given section. Then for each created option object, we apply the corresponding methods stored in the appropriated class. To apply the methods associated to the created option object, we have to create an object of the class containing the methods that we want to apply to our option which also contains the option as attribute. Once this object is created, we can call the methods that we want to apply to our option.

Parameters of the created objects :

AbstractOption.cpp | AbstractOption.h | CRR.cpp | CRR.h | main.cpp

```cpp
#include <stdio.h>
#include "AbstractOption.h"
#include "EuropeanOption.h"
#include "AsianOption.h"
#include "AmericanOption.h"
#include "BlackScholes.h"
#include "CRR.h"
#include "EuropeanPriceAssessment.h"
#include <string>
using namespace std;
int main(int argc, char **argv)
{
    cout<< "------ PARAMETERS ------" << endl;
    double interestRate = 0.03;//0.05;
    double volatility = 0.05;//0.3;
    double strike = 1800; //36;
    double maturity = 1;
    double underlying = 1800; //35;
    double constb = 3; //0.5;
    typeOption type = typeOption::call;

```

AbstractOption.cpp AbstractOption.h CRR.cpp CRR.h main.cpp

```
22      cout << endl;
23      cout << "Option Properties : " << endl;
24      cout << "Interest Rate : " << interestRate << endl;
25      cout << "Volatility :" << volatility << endl;
26      cout << "Strike : " << strike << endl;
27      cout << "Maturity :" << maturity << endl;
28      cout << "Underlying Value : " << underlying << endl;
29      cout << "Constant b : " << constb << endl;
30
31      cout << endl;
32      double nbIterations = 10000;
33      string assessmentMethod1 = "Monte-Carlo";
34      string assessmentMethod2 = "Left Sum Riemann";
35      string assessmentMethod3 = "Right Sum Riemann";
36
37      cout << endl;
38      cout << "Iterations Number : " << nbIterations << endl;
39      cout << "Assessment Method 1 : " << assessmentMethod1 << endl;
40      cout << "Assessment Method 2 : " << assessmentMethod2 << endl;
41      cout << "Assessment Method 3 : " << assessmentMethod3 << endl;
42
```

AbstractOption.cpp AbstractOption.h CRR.cpp CRR.h main.cpp

```
43      cout << endl;
44      int crrDepth = 2;
45      cout << "CRR Tree Depth : " << crrDepth << endl;
46
47      cout << endl;
48      int nbVal = 5;
49      cout << "Number of Maturity Values : " << nbVal << endl;
50      double* maturValues = (double*) malloc (sizeof(double)*nbVal);
51      for (int i=0; i<nbVal; i++){
52          maturValues[i] = i;
53      }
54
55      cout << endl;
```

European Option :

AbstractOption.cpp | AbstractOption.h | CRR.cpp | CRR.h | main.cpp | EuropeanOption.cpp | EuropeanOption.h | AmericanOption

```
cout << "------ EUROPEAN OPTION ------" << endl;
EuropeanOption* optionE = new EuropeanOption(interestRate, volatility, strike, maturity, underlying, constb, type);
// CRR appliqué à européenne et méthodes de monte-carlo et riemann

cout << endl;
cout << "PRICING WITH BLACK SCHOLES METHOD" << endl;
EuropeanPriceAssessment euroAssess;
euroAssess.option = optionE;
double euroPriceByMonteCarlo = euroAssess.ComputePriceOption(nbIterations, assessmentMethod1);
cout << "European Price By Monte-Carlo :" << euroPriceByMonteCarlo << endl;
double euroPriceByRiemannLeft = euroAssess.ComputePriceOption(nbIterations, assessmentMethod2);
cout << "European Price By Riemann Left :" << euroPriceByRiemannLeft << endl;
double euroPriceByRiemannRight = euroAssess.ComputePriceOption(nbIterations, assessmentMethod3);
cout << "European Price By Riemann Right :" << euroPriceByRiemannRight << endl;

cout << endl;
cout << "PRICING WITH CRR METHOD" << endl;
CRR crrE((AbstractOption*)optionE, 2);
crrE.FillUnderlyingValue();
crrE.CRRPricing();
cout << "European Option Price : " << optionE->EuropeanPricingFromCRR(crrE) << endl;
```

American Option :

```
cout << endl;
cout << "------ AMERICAN OPTION ------" << endl;
cout << endl;
cout << "Pricing With Cox Ross Rubinstein (CRR) Method / Binomial Tree Method : " << endl;
AmericanOption* optionA = new AmericanOption(interestRate, volatility, strike, maturity, underlying, constb, type);
CRR crrA((AbstractOption*)optionA, crrDepth);
crrA.FillUnderlyingValue();
crrA.CRRPricing();
crrA.binlattice.Display();
```

Asian Option :

AbstractOption.cpp | AbstractOption.h | CRR.cpp | CRR.h | main.cpp | EuropeanOption.cpp | EuropeanOption.h | AmericanOption.h

```
cout << endl;
cout << "------ ASIAN OPTION ------" << endl;
cout << "Pricing With Wiener Process : " << endl;
cout << endl;
BlackScholes bs;
//srand(time(NULL));
srand(0);
//maturValues = bs.TimeGeneration(10,1,2);
AsianOption* optionas = new AsianOption(interestRate, volatility, strike, maturity, underlying, constb, nbVal, maturValues, t
bs.nbRandomPaths = 500;
bs.option = (AbstractOption*)optionas;
//bs.GenericGenerationRandomPathsUnderlyingPrices();
double test = bs.GenericComputationOptionPrice();
cout << endl;
cout << "Asian Option Price : " << test << endl;

system("pause");
return 0;
```

Conclusion :

This project allowed us to learn how to code in C++ but also to use different notions of programming such as the templates, the abstract class, the virtual methods. Thanks to this project we have also learnt financial concepts such as the properties of different types of options (European, Digital, American, Asian) as well as financial mathematics methods such as Monte-Carlo, Box-Muller, Balck-Scholes, Wiener Process, Cox Ross Rubinstein and the way to apply them to financial concepts and to interpret the results obtained.