ENSAE

C++ Projet

Rapport

Options Pricing by Monte-Carlo and Black-Scholes

Auteur :
Donovan Landry
Steven Worick

31 janvier 2022

1 Introduction

Dans ce projet nous avons pour but de pricer différentes types d'options via différentes méthodes de calcul. Ainsi le premier fichier se nommant MC-Pricer va permettre de pouvoir calculer le prix d'options via la méthode de Monte Carlo qu'on expliquera ainsi que par la formule de Black Scholes.

Le deuxième fichier se nommant BinPricer, lui va utiliser la méthode d'arbre binomial.

2 Pricer par Monte-Carlo et Black Scholes

2.1 Description du programme

Notre programme se décompose en différentes parties. Nous avons une partie main() où nous allons lancer et instancier chaque options puis le reste du programme est composé de classes pour chaque options. Chaque options se composent de son fichier .h et .cpp

Pour le faire fonctionner il suffit simplement de tout avoir dans un même fichier et simplement lancer le main().

Nous avons donc une première classe abstraite Option : Nous l'avons fait abstraite car nuos n'allons pas instancier d'objet "Option". Ainsi cette classe va nous servir comme la racine de notre arbre.

Option.h : Cette classe Option va être notre classe mère qui va permettre à toutes les autres options d'en hériter.

Tout d'abord on y trouve les <include> des différentes library que nous avons utilisées.
Ainsi nous y avons implanté les paramètres en protected (le taux d'intérêts, la volatilité, le strike etc)
Nous avons fait un enum pour le type d'option (call ou put) nous aurions très bien pu faire un bool type avec 0 pour un put et 1 pour un call.
Enfin on y trouve le constructeur par défaut Option et les différentes méthodes qui vont être utilisées dans les classes filles.

Enfin ce qui prouve que notre **classe Option** est abstraite est l'implantation de minimum une méthode virutelle pure. Nos deux méthodes virtuelles sont

MCPricer() ainsi que payoff() car ce sont des méthodes qui vont changer d'une option à l'autre. Il sera donc possible de les "overrider" dans les classes filles.

On y trouve la méthode **BoxMuller()** qui va nous permettre de simuler la distribution de la loi Normale(0,1) avec le paramètre const qu'on a spécifier après la liste des arguments d'une méthode d'objet pour dire qu'elle est constante, c'est-à-dire qu'elle ne modifie pas l'objet, tous les attributs de l'objets sont considérés constants dans cette méthode.

Nous avons aussi les accès en **Get()** donc en lecture seulement pour les paramètres du prix Sous-jacent et la maturité, ces paramètres pourront être utilisés dans le **main()**.

Et la fonction **BSPricer()** qui va être utilisée simplement pour calcuer le prix d'une option avec la méthode de Black and Scholes. Comme il s'agit d'une formule fermée nous l'avons donc pas mise en virtuelle pure.

Enfin on y trouve la méthode **N()** qui va être utilisé dans la méthode de **BSPricer()** elle va permettre d'avoir la fonction de répartition de la loi normale.

Option.cpp : Le .cpp de notre Option.h Dedans nous allons y préciser toutes les méthodes sauf les virtuelles pures.

-Méthode **Box-Muller()** : Nous nous référons à cette formule Box-Muller

-Méthode **N()** : Formule CDF On s'aide de ces formules pour la fonction de répartition de la loi Normale(0,1)

-Méthode **BSPricer()** : [Formule de Black-Scholes] La valeur d'un call, f_c, et d'un put, f_p, sont données par ces formules :

$$f_c = S_0 N(d_1) - K\mathrm{e}^{-rT} N(d_2),$$
$$f_p = K\mathrm{e}^{-rT} N(-d_2) - S_0 N(-d_1),$$

où

$$d_1 = \frac{\log(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}, \tag{1}$$

$$d_2 = \frac{\log(S_0/K) + (r - \sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma.\sqrt{T}. \tag{2}$$

American.h : La classe pour l'option Americaine celle-ci va hérité de la classe mère abstraite Option.

On y retrouve le constructeur pour l'option Américaine ; **AmericanOpt()** ainsi que son destructeur.

Et en fin on y retrouve les deux méthodes virtuelles pures de la classe mère **MCPricer()**, et **payoff()**

American.cpp :

-**payoff()** : Le payoff normal d'un call et put

-**MCPricer()** : On simule le sous-jacent par rapport à cette formule :

$$S_t = S_0 e^{(\mu - \frac{1}{2}\sigma^2)t + \sqrt{T}N} \tag{3}$$

On enregistre sur un tableau toutes les sommes de prix de chaque échantillon au moment j
Ensuite, nous recherchons le maximum qui sera la meilleure date pour exercer les droits d'option.
On calcule le payoff moyen, S_T , à maturité. Et on actualise par le taux d'intérêt sans risque à l'instant initial t = 0 pour trouver la valeur de l'option.

Asian.h : La classe pour l'option Asiatique celle-ci va hérité de la classe mère abstraite Option.

On y retrouve le constructeur pour l'option Asiatique ; **AsianOpt()** ainsi que son destructeur.

Et en fin on y retrouve les deux méthodes virtuelles pures de la classe mère **MCPricer()**, et **payoff()**

Asian.cpp :

-**payoff()** : Le payoff normal d'un call et put

-**MCPricer()** : On simule le sous-jacent par rapport à cette formule :

$$S_t = S_0 e^{(\mu - \frac{1}{2}\sigma^2)t + \sqrt{T}N} \tag{4}$$

Le payoff ici sera différent car le payoff d'une option asiatique est la moyenne des payoff. Ici la variable $sample_size$ correspond au nombre de prix simulés alors que n_sample correspond au nombre de dates simulées.
Puis nous calculons le payoff moyen, S_T , à maturité. Et on actualise par le taux d'intérêt sans risque à l'instant initial t = 0 pour trouver la valeur de l'option.

European.h : La classe pour l'option Européenne celle-ci va hérité de la classe mère abstraite Option.

On y retrouve le constructeur pour l'option Européenne ; **European()** ainsi que son destructeur.

Et en fin on y retrouve les deux méthodes virtuelles pures de la classe mère **MCPricer()**, et **payoff()**

Cependant nous ne sommes pas obligés de rajouter la méthode **MCPricer()** car nous calculons le prix de l'option Européenne à l'aide de la méthode **BSPricer()** qui se trouve dans la classe mère **Option**.

European.cpp : On y retrouve tout simplement le payoff de l'option Européenne et la méthode **MCPricer()** qui ne sera pas utilisée.

main.cpp :

Notre main où nous allons tout simplement tester notre programme et y afficher les différents à l'aide de différentes méthodes de calculs de prix.
Tout d'abord on y instancie à chaque fois nos objets donc nos options avec différents paramètres puis nous stockons dans des variables "double" le prix à l'aide de la méthode choisie ou disponible.
Et enfin nous affichons le résultat pour différents type d'option (call/put).

2.2 Critique des problèmes rencontrés et des solutions adoptées

Pas trop de problèmes rencontrés, peut être le passage d'un IDE à l'autre il fallait gérer les différences pour chaque IDE.

Et la difficulté également présente réussir à lier tous les fichiers entre eux. Ainsi que l'utilisation des paramètres dans différentes classes, il a fallu penser à rajouter les accès en lecture et écriture ; les get et set

2.3 Description de l'architecture générale du programme

Toutes les classes sont faites d'un .h et de son .cpp certaines comme j'ai pu évoquer toutes les classes vont hérédite d'une classe mère et ainsi pouvoir overrider les fonctions de cette classe mère. Le polymorphisme est présent dans ce projet, on le retrouve pour toutes les fonctions propres à chaques classes comme la fonction **payoff()** ainsi que la fonction **MCpricer**. Toutes les classes héritent de la classe mère **Option**

3 Pricing Binomial et affichage de l'arbre

3.1 Description du programme

Notre programme se décompose en différentes parties. Nous avons une partie main() où nous allons lancer et instancier chaque options puis le reste

du programme est composé de classes pour chaque options. Chaque options se composent de son fichier .h et .cpp

Et nous avons en plus une classe BinLattice qui va servir à instancer l'arbre binomial avec différentes méthodes.

Pour le faire fonctionner il suffit simplement de tout avoir dans un même fichier et simplement lancer le main().

Nous avons donc une première classe abstraite Option : Nous l'avons fait abstraite car nuos n'allons pas instancier d'objet "Option". Ainsi cette classe va nous servir comme la racine de notre arbre.

Option.h : Cette classe Option va être notre classe mère qui va permettre à toutes les autres options d'en hériter.

Tout d'abord on y trouve les <include> des différentes library que nous avons utilisées.
Ainsi nous y avons implanté les paramètres en protected (le taux d'intérêts, la volatilité, le strike etc)
Nous avons fait un enum pour le type d'option (call ou put) nous aurions très bien pu faire un bool type avec 0 pour un put et 1 pour un call.
Enfin on y trouve le constructeur par défaut Option et les différentes méthodes qui vont être utilisées dans les classes filles.

Enfin ce qui prouve que notre **classe Option** est abstraite est l'implantation de minimum une méthode virtuelle pure. Notre méthode virtuelle est **payoff()** car c'est une méthodes qui va changer d'une option à l'autre. Il sera donc possible de l' "overrider" dans les classes filles.

Et la fonction **BinPricer2()** qui va être utilisée simplement pour calcuer le prix d'une option avec la méthode de Cox-Rox Rubinstein. Comme il s'agit d'une formule fermée nous l'avons donc pas mise en virtuelle pure.

Il y a également l'implémentation de la méthode CRR en mode itératif qu'on retrouve dans la méthode **BinPricer()**

Enfin on y trouve la méthode **combi()** qui va être utilisé dans la méthode de **BinPricer2()** elle va permettre d'avoir les coefficients binomiaux dans la méthode fermée **BinPricer2()**.

Option.cpp : Le .cpp de notre Option.h Dedans nous allons y préciser

toutes les méthodes sauf les virtuelles pures.

-Méthode **BinPricer2()** : Nous nous référons à cette formule de Cox Rox Rubinstein (formule fermée différente de celle en itératif) :

On simule le sous-jacent par rapport à cette formule :

$$S_t = S_0 e^{\sigma\sqrt{\Delta t}} \tag{5}$$

Puis à chaque iteration on calcul le payoff de l'option afin de remonter l'arbre jusqu'à la dernière valeur par cette méthode.

$$f_{u^j,d^i} = \mathrm{e}^{-r\Delta t}(p f_{u^{j+1},d^i} + (1-p) f_{u^j,d^{i+1}}).$$

CRR Binomial Model 1979

-Méthode **BinPricer()** : Nous nous référons à cette formule itératif de Cox Rox Rubinstein Formule CRR

$$f = \mathrm{e}^{-rn\Delta t}\left[\sum_{j=0}^{n}\binom{n}{j} p^j (1-p)^{n-j} f_{u^j,d^{n-j}}\right]. \tag{6}$$

-Méthode **combi()** : Méthode afin de calculer les coefficient binomiaux Combinaison

DigitalOpt.h : La classe pour l'option Digitale celle-ci va hérité de la classe mère abstraite Option.

On y retrouve le constructeur pour l'option Digitale ; **DigitalOpt()** ainsi que son destructeur.

Et en fin on y retrouve la méthodes virtuelle pure de la classe mère **payoff()**

DigitalOpt.cpp : Nous allons utiliser par l'héritage les méthodes de la classe mère Option nous avons juste à y rajouter la méthode overrider du payoff.

-**payoff()** : Le payoff d'une option Digitale

EuropeanOpt.h : La classe pour l'option Européenne celle-ci va hérité de la classe mère abstraite Option.

On y retrouve le constructeur pour l'option Européenne ; **EuropeanOpt()** ainsi que son destructeur.

Et en fin on y retrouve la méthodes virtuelle pure de la classe mère **payoff()**

EuropeanOpt.cpp : Nous allons utiliser par l'héritage les méthodes de la classe mère Option nous avons juste à y rajouter la méthode overrider du payoff.

-**payoff()** : Le payoff d'une option Européenne.

BinLattice.h : Cette classe va représenter la structure de données (arbre des chemins) utilisée pour la méthode binomiale.

Cette classe est implémentée en tant que classe template afin que le type de données tenu par le vecteur de vecteur Lattice peut être personnalisé comme bon lui semble. Cela nous permet de faciliter le changement du type de retour

La classe contient deux variables propres à elle :

- N : afin de stocker le nombre de temps de pas dans l'arbre binomial -Lattice : un vector de vector afin de contenir la data de type double. Puis le reste de paramètre que nous retrouvons dans la classe Option par exemple (le taux d'intérêt, strike etc)

Elle contient différentes méthodes :

-**SetN()** : Une fonction void qui simplement assginer la taille du Lattice de vecteur N+1.

SetNode(int n , int i , double val) : Elle permet de stocker la valeur mise en paramètre à l'étape n et noeud i.

GetNode(int n , int i , double val) : Elle permet de retourner la valeur mise en paramètre à l'étape n et noeud i.

Display() : Elle permet d'afficher les valeurs stocker dans l'arbre binomial.

CalNode(int n , int i) : Elle permet de nous permet de calculer la valeur à l'étape n et au nœud i grâce à la formule fermée du prix de l'option.

En effet, par récurrence, nous prouvons que :

$$H(n,i) = e^{R\Delta(N-n)} * \sum_{k=0}^{N-n} (\binom{N-n}{k})(1-p)^k.p^{N-n-k}.h(S(N,k+i))$$

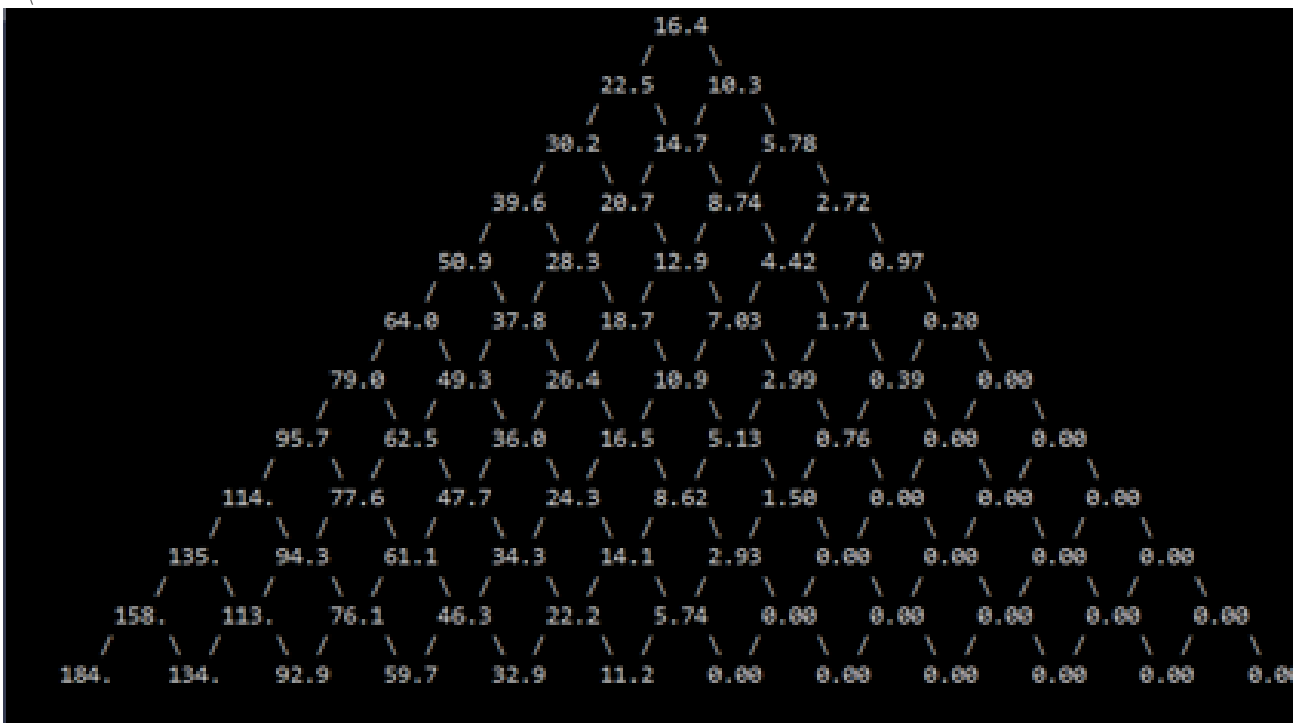
Pour la fonction **Display()** :

Nous avons décidé d'affecter 4 unités d'espace pour chaque prix. Par exemple, 12.32897 sera affiché comme 12.3 dans le Binlattice.

Au début, nous avons une chaîne d'espaces qui sont les espaces avant les premiers caractères afin d'avoir une bonne pyramide.

Lorsque le nombre de lignes est impair, nous affichons les prix.

Lorsque le nombre de lignes est pair, nous affichons la barre oblique "/" et " \".



3.2 Critique des problèmes rencontrés et des solutions adoptées

Même problèmes que le premier Pricer, changement d'IDE ainsi l'idée de vouloir faire un seul .h pour le BinLattice. Et la difficulté également présente réussir à lier tous les fichiers entre eux. Ainsi que l'utilisation des paramètres

dans différentes classes, il a fallu penser à rajouter les accès en lecture et écriture ; les get et set

3.3 Description de l'architecture générale du programme

De même, implémentation des classes en .h et .cpp sauf pour la class BinLattice. Le polymorphisme est présent dans ce projet, on le retrouve pour toutes les fonctions propres à chaques classes comme la fonction **payoff()**. Toutes les classes héritent de la classe mère **Option** sauf la classe BinLattice