Projet de C++ : Pricer d'options par Simulations Monte-Carlo dans le modèle de Black Scholes

Alexis Aubert
Axel Chedri
Alan Loret
Alexandre Taranoff

Janvier 2021

1 Problématique

L'objectif de ce projet est de développer un pricer d'options. Il existe de multiples manières d'évaluer des options : l'approche par arbres, par équations aux dérivées partielles, ... Ici, nous nous concentrons sur l'approche Monte-Carlo. Elle consiste à simuler un grand nombre de réalisations et d'en déduire par la loi des grands nombres une approximation du prix. Nous nous plaçons dans le modèle de Black-Scholes, qui en fonction d'un paramètre de taux d'intérêt et d'un paramètre de volatilité propose une génération de la trajectoire de prix. Nous nous appuyons donc sur ce modèle pour simuler l'évolution du prix d'un sousjacent fictif et déterminer le prix d'une option dérivée de celui-ci.

2 Programme

2.1 Structure

Le programme se compose de 19 fichiers. On distingue les fichiers codant les méthodes de simulations, des fichiers codant les options. Ainsi, 7 fichiers sont destinés à la mécanique de simulation et du calcul du prix :

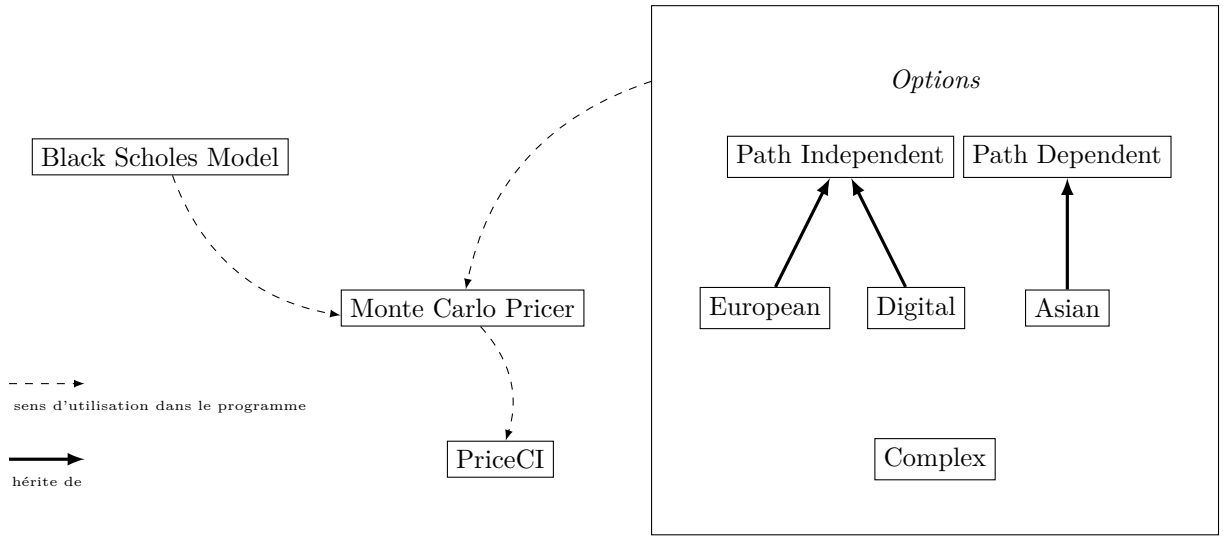
- `main.cpp` constitue le fichier principal contenant la fonction *main* avec quelques exemples d'utilisation du programme.
- `BlackScholesModel.h, BlackScholesModel.cpp`, classe contenant l'objet *BlackScholesModel* permettant de générer des prix, ou des trajectoires de prix selon le model de Black Scholes.
- `MonteCarloPricer.h, MonteCarloPricer.cpp`, classe contenant l'objet *MonteCarloPricer* permettant d'appliquer la méthode Monte-Carlo afin d'estimer le prix d'option.
- `PriceCI.h, PriceCI.cpp`, classe contenant l'objet *PriceCI* rassemblant les méthodes et attribus nécessaires au calcul de la variance asymptotique et du prix.

Les fichiers restant codent les options :

- `PathIndependent.h, PathIndependent.cpp`, classe virtuelle contenant la structure basique des options dont le payoff ne dépend pas de la trajectoire du prix.
- `PathDependent.h, PathDependent.cpp`, classe virtuelle contenant la structure basique des options dont le payoff dépend de la trajectoire du prix.
- `European.h, European.cpp`, classe contenant les options *EuropeanCall*, *EuropeanPut*, *BullSpread*, *BearSpread*, *Strangle* et *Butterfly*.

— `Digital.h, Digital.cpp`, classe contenant les options *DigitalCall*, *DigitalPut* et *DoubleDigital*.
— `Asian.h, Asian.cpp`, classe contenant les options *AsianArithmeticCall*, *AsianArithmeticPut*, *AsianGeometricCall* et *AsianGeometricPut*.
— `Complex.h, Complex.cpp`, classe permettant de générer des options personnalisées en combinant **uniquement** les options suivantes *EuropeanCall*, *EuropeanPut*, *DigitalCall*, *DigitalPut*, *AsianArithmeticCall*, *AsianArithmeticPut*, *AsianGeometricCall* et *AsianGeometricPut*. On réplique, grâce à cette nouvelle classe, quelques options classiques comme le *BullSpread* ou le *BearSpread*.

Le schéma suivant détaille l'articulation de ces différents fichiers :



2.2 Utilisation

L'utilisateur doit spécifier plusieurs valeurs pour faire fonctionner le programme :

1. Création du modèle de Black Scholes
   — S, le prix actuel du sous-jacent.
   — r, le taux d'intérêt sans risque.
   — v, la volatilité.
2. Création de l'optimisateur
   — N, nombre de simulations dans l'algorithme de Monte-Carlo.
   — *steps* (facultatif), nombre d'étapes pour la génération de la trajectoire de prix.
3. Création d'une option
   — K le ou les strikes de l'option.
   — T la maturité.

Exemple : Calcul du prix d'un Call Européen.

```
1 BlackScholesModel model(S, r, v); // 1
2 MonteCarloPricer optimizer(N, steps); // 2
3 EuropeanCall eu_call(K, T); // 3
4 optimizer.priceAndPrint(model, eu_call); // Compute and print
```

Exemple : Création d'une option personnalisée et calcul du prix.

```
1 BlackScholesModel model(S, r, v);
2 MonteCarloPricer optimizer(N, steps);
```

```
EuropeanCall eu_call(K, T); // Eu call
AsianArithmeticCall asian_call(K2, T); // Asian call

Complex custom_option("Customized option"); // Custom Option
custom_option += eu_call; // Long Eu call
custom_option -= asian_call; // Short Asian call
optimizer.priceAndPrint(model, custom_option); // Compute and print
```

Attention : Deux méthodes sont disponibles pour calculer le prix d'une option : *priceAndPrint* et *priceAndPrintClassic*. La première accepte toutes les options, et génère une trajectoire complète du prix à chaque itération de l'algorithme de Monte-Carlo. Cela permet d'évaluer les options selon la trajectoire du prix. Pour les options qui ne dépendent pas de la trajectoire du prix, on peut gagner du temps en ne calculant que le prix à maturité sans générer la trajectoire du prix grâce à la méthode *priceAndPrintClassic*. On ne peut donc pas utiliser cette méthode pour estimer le prix d'une option Asiatique qui dépend de la trajectoire du prix.

Exemple : Utilisations correctes et incorrectes de *priceAndPrintClassic*

```
BlackScholesModel model(S, r, v);
MonteCarloPricer optimizer(N, steps);

EuropeanCall eu_call(K, T); // Eu call
optimiser.priceAndPrint(model, eu_call) // OK
optimiser.priceAndPrintClassic(model, eu_call) // OK

AsianArithmeticCall asian_call(K2, T); // Asian call
optimiser.priceAndPrint(model, asian_call) // OK
optimiser.priceAndPrintClassic(model, asian_call) // WRONG
```

3 Résultats

3.1 Comparaison des types d'options

Les résultats concernant les prix des différentes options d'achat sont logiques : les options offrant le plus d'avantages ou tout simplement étant les moins risquées coûtent plus cher. De plus, nous obtenons des valeurs de prix comparables au résultats d'articles de revues scientifiques qui traitent des applications de la méthode de Monte-Carlo en finance.

3.2 Problème rencontré

Une des problématiques importantes des simulations de Monte-Carlo réside dans la génération des nombres aléatoires. En particulier, puisque nous nous basons sur le modèle de Black-Scholes, nous devons simuler une loi normale pour calculer les payoffs et en déduire une estimation du prix des options. Le choix de notre générateur pseudo-aléatoire est donc crucial. Nous avons utilisé le générateur pseudo-aléatoires de la librairie C++11 qui est plus robuste que la fonction *rand()*. L'implémentation dans notre projet est faite dans le fichier `BlackScholesModel.cpp` :

```
#include <random>
#include <iostream>

// Define our Uniform random generator on [-1,1]
std::random_device rd; // Define our seed rd
std::mt19937 mt(rd()); // Seed the random number generator mt
std::uniform_real_distribution<double> dist(-1.0, 1.0); // Create our random variable :
    Uniform on [-1,1]

dist(mt) // Return a random number
```

Ensuite, à partir de ce générateur qui simule une variable aléatoire de loi $\mathcal{U}([-1,1])$, nous utilisons la méthode d'inversion de la fonction de répartition. La méthode polaire (Box Müller) est implémentée dans la méthode *gaussian_box_muller* de la classe *BlackScholesModel*.

3.3 Améliorations

Réduction de la variance

La simulation par la méthode de Monte-Carlo nécessite un grand nombre de simulations pour être pertinente. Ainsi, pour avoir des estimations avec une précision appropriée, nous pouvons augmenter le nombre de simulations, ce qui se fait au détriment d'une augmentation du temps de calcul. Il faut donc trouver un juste-milieu.

Dans certains cas, notre estimateur donne des résultats très satisfaisants pour des temps d'exécution de l'ordre de la seconde, notamment pour les options indépendantes du chemin suivi par le sous-jacent (utilisez la méthode *priceAndPrintClasssic* avec $N = 10^7$). Cependant, lorsqu'on traite les options dépendantes du chemin suivi, il n'est parfois pas possible d'obtenir un intervalle de confiance suffisamment étroit en un temps raisonnable.

Pour améliorer la rapidité de la convergence de notre estimateur nous aurions pu utiliser une autre méthode : la réduction de variance. En effet, réduire la variance de notre estimateur améliore la vitesse de convergence. Cependant, il faut toutefois veiller à ce que les calculs supplémentaires engendrés par cette méthode n'augmentent pas trop le temps de calcul. Nous avons pensé à implémenter la méthode de contrôle antithétique.

Calculer des Grecques

Nous avons envisagé le calcul des Grecques : $\Delta = \frac{\partial V_t}{\partial S_t}$, $\Gamma = \frac{\partial^2 V_t}{\partial S_t^2}$, $\Theta = \frac{\partial V_t}{\partial T - t}$, $\vartheta = \frac{\partial V_t}{\partial \sigma}$, mais nous n'avons pas mené cette partie à son terme. D'une part le manque de temps nous a contraint à établir certaines priorités, et d'autre part cet ajout tardif s'intégrait mal à la structure du programme que nous avions jusqu'alors codé. Néamoins, l'ajout de cette fonctionnalité serait souhaitable dans un futur programme.

Interface graphique

Nous aurions souhaité coder une interface graphique permettant de faciliter l'utilisation de notre pricer d'options par les utilisateurs. Malheureusement, nous avons manqué de temps, et nous nous sommes aperçus que la bibliothèque que nous comptions utiliser est devenue payante.