Projet de Modélisation et Programmation

Conception et Implémentation d'un pricer Monte-Carlo générique en C++

Lucas COUTÉ Lucas LEBIHAN Léonard MOMMEJA

Raphaël PRÊTRE–HECKENROTH

24 Septembre 2019

1 Présentation du Projet

Ce projet a pour finalité de créer une bibliothèque fonctionnelle permettant de calculer des prix d'options par une méthode de Monte-Carlo ainsi que leurs dérivées par rapport au spot.

Théoriquement, le prix d'une option en $t \in [t_0, t]$ portant sur des sous-jacents de cours $(S_s)_{s\in[t_0,T]}$ s'exprime sous la forme d'une fonction $v(t, (S_s)_{s\in[t_0,t]})$ que l'on peut donner de manière explicite en utilisant des espérances conditionnelles. Pour pouvoir implémenter des algorithmes pour calculer ces prix, il faut tout d'abord discrétiser le temps pour que les prix puissent s'exprimer sous la forme $v(t, (S_{t_i})_{i\in[\![0;N]\!]/t_i<t}, S_t)$.

On utilise ensuite une méthode de Monte-Carlo pour approximer l'espérance conditionnelle par une moyenne empirique de M simulations des variables aléatoires de la formule.

Dans ce projet C++, on peut générer deux exécutables permettant respectivement de pricer une option (`price0`) et de la couvrir par l'achat de sous-jacents (`hedge`).

Une documentation précise de la bibliothèque peut être générée avec un `make doc`. Il est ensuite possible de la visualiser à partir du dossier `build` à l'aide de la commande : `firefox ../doc/html/index.html` .

Le diagramme de Gantt se situe dans le dossier `AUTHORS`. Il s'y trouve au format `Planner` ou au format html.

2 Implémentation

L'architecture de notre projet est proche de celle du squelette fournie avec les implémentations demandées en plus. En effet, on a toujours trois classes principales (`MonteCarlo`, `BlackScholeModel` et la classe absraite `Option`), les trois classes qui héritent d'`Option` (`Basket`, `Asian` et `Performance`), les deux classes pour stocker les résultats (`PricingResult` et `HedgingResult`) et les deux classes pour implémenter les exécutables (`price0` et `hedge`).

Les codes sources des deux exécutables sont séparés du reste du code et ont été placés dans le dossier `src/apps`.

Intéressons-nous maintenant aux méthodes de la classe `MonteCarlo`. Tout d'abord, on constate qu'il y a deux implémentations de `price`, l'une permettant de calculer le prix en 0 et l'autre en t. Pour bien comprendre la méthode de Monte-Carlo, on a commencé par implémenter cette méthode en 0 mais celle-ci n'est jamais appelée dans les deux exécutables car nous avons surchargé la méthode `price` pour tout t de manière à ce qu'elle puisse également donner le prix en 0. Nous avons réussi à faire cela en implémentant correctement la méthode `asset` (en t) de `BlackScholesModel`. On décrit précisément ce que fait `asset` par la fonction :

$$(t, (S_{t_i})_{i\in[\![0;N]\!]/t_i<t}, S_t) \mapsto ((S_{t_i})_{i\in[\![0;N]\!]/t_i<t}, (S_{t_i})_{i\in[\![0;N]\!]/t\leq t_i})$$

En français, `asset` a en sortie la matrice des cours des sous-jacents aux temps strictement inférieurs à t (donnés en argument par past) auxquels on ajoute ceux aux temps supérieurs ou égaux à t (calculés à partir des cours en t donnés dans `path`). (Le sujet laissait penser que `asset` devait être implémenté de manière différente en considérant que $t_i \leq t < t_{i+1}$ alors que nous considérons ici que $t_i < t \leq t_{i+1}$)

Les deltas pour chaque sous-jacent sont calculés en utilisant la méthode `shift_asset` sur les mêmes simulations de trajectoire du marché (cela permet d'éviter de faire M simulations pour

chaque sous-jacent, ce qui engendre un gain de temps non négligeable car la génération de nombres aléatoires prend beaucoup de temps).

Enfin, la méthode permettant de calculer à la fois le prix de l'option et l'erreur de couverture a également été implémentée avec la méthode `MonteCarlo::profitAndLoss` en respectant les formules :

$$\begin{cases} V_0 = p_0 - \delta_0.S_0 \\ V_i = V_{i-1}.e^{\frac{r.T}{H}} + (\delta_{i-1} - \delta_i).S_{\tau_i} \quad \forall i \in [\![1; H]\!] \\ error = V_H + \delta_H.S_{\tau_H} - p_{t_H} \end{cases}$$

Cette méthode telle qu'elle est implémentée est particulièrement longue à l'exécution car on appelle une fois la méthode `price`, puis $H + 1$ fois la méthode `delta` et une fois la méthode `payoff`. La fonction `asset` est donc appelée $H + 2$ fois, ce qui entraîne la génération d'une très grande quantité de nombres aléatoires (génération qui demande beaucoup de temps quand on utilise `gprof`).

Afin de tester nos couvertures sur d'autres valeurs que que les valeurs historiques fournies, nous avons implémenté et testé la méthode `simul_market`.

3 Tests et gestion de la mémoire

Le dossier `src/test/` regroupe un ensemble de programmes C++ nous permettant de tester les différentes fonctions implémentées lors de ce projet en fixant nous-même les paramètres que nous souhaitions pour pouvoir déboguer facilement.

Nous avons également naturellement prêté attention à réparer toutes les fuites mémoires à l'aide de `valgrind` afin d'avoir une meilleure qualité de code. En revanche, pour la clarté du code, nous n'utilisions pas le mot clé `auto` pour la déduction automatique du type par le compilateur bien que nous comprenions l'intérêt d'une telle coding practice.

Nous avions initialement eu l'idée de rajouter des tests unitaires à l'aide de Google Tests mais après réflexion il était compliqué d'implémenter ces tests dû à la nature aléatoire des résultats. Et même en fixant la graine du RNG, comment s'assurer que nos résultats soient les bons ? Nous ne souhaitions pas y consacrer trop de temps et risquer de ne pas avancer sur le projet, donc nous sommes arrivés avec ce compromis.

4 Problèmes rencontrés

La compréhension du projet a mis plus de temps que prévu. En effet le formalisme se heurtait à nos connaissances et nos intuitions, comme par exemple lors du calcul du prix de l'option en t avec les $\widetilde{S_k}$.

Malgré nos efforts, un warning de données non initialisées est affiché dans le rapport de valgrind dans l'exécutable `test-couverture` mais ne crée pas pour autant de fuites mémoires. Même avec l'option `-track-origin=yes`, nous n'avons pas réussi à réparer ce problème.

Les temps d'exécution du pricer et de la couverture sont aussi bien trop grands. Une étude du profiling à l'aide de gprof nous a montré que nous appelions énormément de fois (de l'ordre de 48 millions) les méthodes payoffs des options. Nous n'avons toujours pas trouvé comment réduire ce nombre d'appels, ni une optimisation réduisant significativement le nombre de calculs par appel des différentes méthodes en jeu.

5 Améliorations envisageables

Nous n'avons pas encore réussi à optimiser suffisamment notre couverture d'option pour que toutes les options puissent être couvertes dans le temps imparti par le testeur automatique.

Nous pouvons étendre la bibliothèque à d'autres options exotiques.

On peut également introduire de nouveaux modèles de calcul pour les trajectoires de marché (par exemple le modèle de Black-Scholes-Merton dans lequel les actifs peuvent verser des dividendes) puisque le modèle de Black Scholes n'est pas représentatif de la réalité.

Nous n'avons pas fait de gestion d'exceptions ce qui causera des problèmes lors des fichiers input invalides. Le temps d'exécution étant déjà trop long nous nous étions concentrés sur l'optimisation sans gérer ces petits détails à l'aide de try/catch.