Rapport d'avancement

Groupe numéro 2 :
- WANG Ziyuan
- Liu Ziyu
- Amine ROUATBI

Projet numéro 1 :
« Use constant parameters in Monte Carlo engines »

Dernière mise à jour de ce rapport : Jeudi 12/03/2020

Notre approche a été de commencer par bien lire et comprendre le sujet, puis d'explorer le code et de savoir ce qu'il faut coder.

La toute première difficulté qu'on a recontrée était de savoir comment utiliser la librairie et les classes dans le main.cpp. En effet, dans l'énoncé du sujet, il est question de (1) estimer les performances du moteur Monte Carlo donné (value, accuracy, elapsed time) puis de trouver une solution pour changer l'architecture de la classe du moteur afin d'utiliser un processus de black-scholes à paramètres constant, plutôt qu'un processus généralisé.

Le premier travail fourni est donc d'estimer la performance du modèle original, c'est-à-dire avec un processus généralisé.

Nous avons trouvé dans la documentation de QuantLib un exemple où on utilise plusieurs modèles (américain, européen, etc.) pour effectuer plusieurs simulations. Voici le lien précis de l'exemple : « https://github.com/lballabio/QuantLib/blob/master/Examples/EquityOption/EquityOption.cpp »

Cet exemple nous a aidés à comprendre le fonctionnement de la librarie et nous l'avons adapté dans le main.cpp en enlevant beaucoup de parties (méthodes autres que Monte Carlo) et en changeant quelques paramètres. Le code adapté se trouve dans le main.cpp du répertoire du projet.

Voici les changements les plus importants que nous avons effectué :

1) Remplacer le processus BlackScholesMertonProcess par ce qui est demandé, i.e. un GeneralizedBlackScholesProcess :

```
-        ext::shared_ptr<BlackScholesMertonProcess> bsmProcess(
-                 new BlackScholesMertonProcess(underlyingH, flatDividendTS,
-                                               flatTermStructure, flatVolTS));

+        boost::shared_ptr<GeneralizedBlackScholesProcess> gbsProcess(
+                 new GeneralizedBlackScholesProcess(underlyingH, flatDividendTS,
+                                               flatTermStructure, flatVolTS));
```

2) Utiliser le moteur fourni dans le projet, c'est-à-dire le McEuropeanEngine**_2** (le 2 est important sinon on va utiliser le moteur local, par défaut, de QuantLib).

```
-        ext::shared_ptr<PricingEngine> mcengine1;
-        mcengine1 = MakeMCEuropeanEngine<PseudoRandom>(bsmProcess)
-            .withSteps(timeSteps)
-            .withAbsoluteTolerance(0.02)
-            .withSeed(mcSeed);

+        boost::shared_ptr<PricingEngine> mcengine;
+        mcengine = MakeMCEuropeanEngine_2<PseudoRandom>(gbsProcess)
```

```
+            .withSteps(timeSteps)
+            .withAbsoluteTolerance(0.02)
+            .withSeed(mcSeed);
```

3) Ajouter les instructions permettant d'effectuer le benchmarking :
- Un chronométrage du temps d'exécution (value dans l'énoncé)
- La valeur de l'erreur avec la méthode errorEstimate() (accuracy dans l'énoncé)
- La valeur retournée par la simulation avec la méthode NPV() (elapsed time dans l'énoncé).

Voici quelques mesures de benchmarking que nous avons effectué :

timeSteps	Valeur (NPV)	Erreur (errorEstimate)	Temps d'exécution total
1	3.83452	0.0198782	64 ms
10	3.82361	0.0198285	336 ms
20	3.84767	0.0198165	605 ms
50	3.81936	0.0199537	1455 ms

Interprétation :

Nous n'avons pas analysé, pour l'instant, ce que représente la valeur NPV et ses évolutions.

Il y a un comportement aberrant dans l'évolution de l'erreur. Nous avons intuité que plus on met de timeSteps, plus on pousse le calcul loin et donc l'erreur devrait diminuer (on augmente le nombre de tirs dans la simulation de Monte Carlo?). L'erreur diminuait bien au début, mais en passant de 20 steps à 50, elle a au contraire augmenté. Nous diagnostiquerons le problème plus en détail s'il reste du temps.

Le plus important : Le temps d'exécution total est très significatif et il augmente assez vite. Même si la complexité n'est que linéaire (en O(n)), il y a quand même une certaine attente. Nous supposons que les entreprises spécialisées en finance, lorsqu'elles réalisent des simulations de Monte Carlo, font des calculs beaucoup plus gourmands (timeStep de l'ordre des millions?) et donc la problématique posée par le sujet est tout à fait logique.

L'idée, si nous avons bien compris, est de faire un compromis sur l'erreur pour améliorer le temps d'exécution. Elle consiste, pour rentrer plus dans les détails, à remplacer le processus de Black-Scholes généralisé qu'on a utilisé en un processus à paramètres constants.

Nous sommes actuellement en train de travailler sur la nouvelle classe et l'implémentation est en cours. Ce rapport sera mis à jour avec les nouvelles observations très prochainement.