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).

We successfully replaced the GeneralizedBlackScholesProcess with our ConstantBlackScholesProcess. Please refer to the code in the same directory for the implementation.

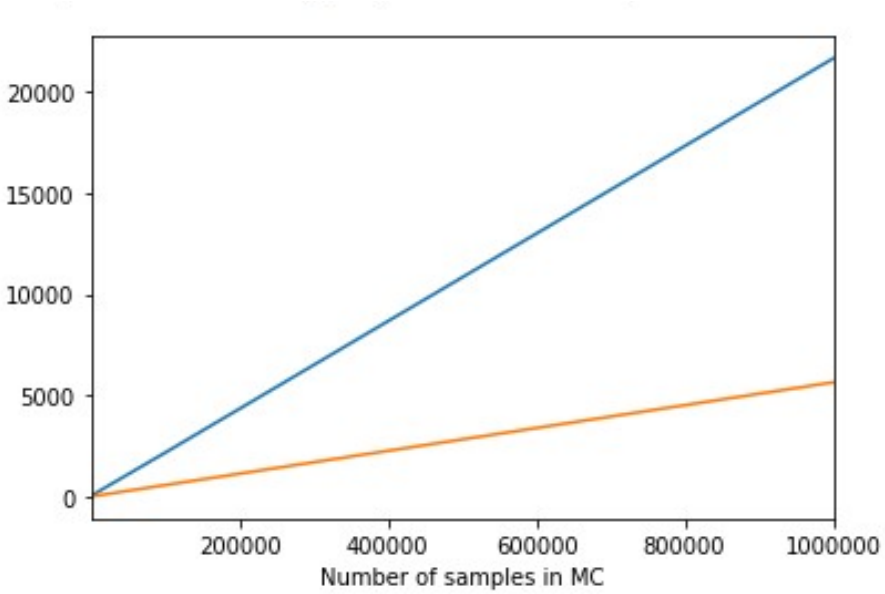
Let's do a benchmark using these two processes. In order to have enough data for comparison, we will study the evolution of both processes when we increase the number of samples in the MC simulation.

Generalized Black Scholes Process :

Number of samples in MC	**NPV Value**	**Error estimate**	**Total elapsed time**
100	3.93214	0.41839	5 ms
1000	3.75903	0.13018	23 ms
10000	3.82003	0.04288	244 ms
100000	3.84793	0.01363	2168 ms
1000000	3.83694	0.00431	21659 ms

Constant Black Scholes Process :

Number of samples in MC	**NPV Value**	**Error estimate**	**Total elapsed time**
100	3.93214	0.41839	~0 ms
1000	3.75903	0.13018	6 ms
10000	3.82003	0.04288	57 ms
100000	3.84793	0.01363	565 ms
1000000	3.83694	0.00431	5645 ms



Evolution of execution time (in ms) with number of samples for both processes (blue : generalized, orange:constant)

We observe that the execution time is linearly dependant with the number of samples in the MC simulation. We have a linear complexity for both processes.

The constant black scholes process performs better since the execution time is divided by appromitavely 3.83.

Fortunately, we didn’t notice any flinch in the other parameters (NPV value and error estimate). They remained exactly equal whichever the process we use (which is a bit odd since we were expecting a loss in accuracy).

All in all, the constant black scholes process outperforms the generalized process.