Projet C++ - Réplications de portefeuille

Mathieu DUARTE, Jun JING

2014-2015

Sommaire

Introduction

Ce projet proposait la réalisation d'un outil de réplication de portefeuille par les payoffs. Généralement, les premiers types de réplication de portefeuille que l'on rencontre sont des réplications *statiques* sur des modèles très simples (souvent des modèles discrets comme celui de l'arbre binomial par exemple). Pour ce projet, nous avons choisi de nous placer dans le cadre continu **sous les hypothèses du modèle de Black-Scholes** et de développer une méthode de réplication *dynamique* des **options européennes** basée sur l'utilisation des grecques. En particulier, nous choisissons de nous reposer sur $\Delta = \frac{\partial P}{\partial S}$ (P prime de l'option, S prix de l'action) pour effectuer du **delta-hedging**. Il s'agit d'une méthode de couverture classique que nous décrirons par la suite. Enfin, nous avons choisi d'implémenter en plus une simulation de couverture, c'est-à-dire une fonction qui simule l'évolution future de l'actif sous-jacent et qui indique les gains/pertes qui découleraient d'un delta-hedging effectué avec une régularité plus ou moins importante.

Chapitre 1

Description de ce que fait le programme

La compilation sous Qt du projet lance une fenêtre générale intitulée *Portfolio Replicating Program* qui donne deux possibilités à l'utilisateur :

1. **Portfolio Replicating** : il s'agit d'une réplication instantanée à l'instant $t = 0$ du portefeuille. Cette réplication repose sur l'utilisation des grecques. L'utilisateur spécifie des paramètres et reçoit notre suggestion pour répliquer son portefeuille.
2. **Simulation** : via les paramètres spécifiés à l'instant $t = 0$ par l'utilisateur, cette fenêtre lance une simulation de l'évolution de l'actif sous-jacent jusqu'à maturité. Cette simulation n'est pas - dans la version actuelle du programme - affichée à l'utilisateur. Le programme effectue du delta-hedging sur cette simulation et retourne à l'utilisateur le résultat (gains/pertes) avec notre couverture.

1.1 Portfolio Replicating

Nous allons maintenant préciser - sans trop rentrer dans les détails - comment s'effectue cette réplication. L'utilisateur indique un certain nombre de valeurs :

- *Initial value* : il s'agit de la valeur à l'instant $t = 0$ de l'actif sous-jacent
- *Strike* : il s'agit du prix d'exercice de l'option
- *Interest rate* : il s'agit du taux d'intérêt de l'actif sans risque (du numéraire)
- *Volatility* : il s'agit de la volatilité sur le marché de l'actif sous-jacent
- *Maturity* : il s'agit de la maturité de l'option
- *Position* : long indique une position d'achat, short indique une position de vente
- *Option type* : call indique une option d'achat, put indique une option de vente
- *Number of options* : il s'agit du nombre d'options dans le portefeuille. Nous avons restreint l'utilisateur à un seul type d'option, c'est-à-dire que l'utilisateur ne peut - via notre programme du moins - répliquer un portefeuille ayant à la fois des options d'achat et des options de vente. Pour ce cas particulier, il

devra répliquer le portefeuille des call séparément, puis le portefeuille des put séparément.

Via ces valeurs, il est possible de calculer le Δ d'un call et d'un put. En effet, en notant $\mathcal{N}$ la fonction de répartition de la loi normale centrée réduite et d_1 une fonction des paramètres indiqués par l'utilisateur (notations du modèle de Black-Scholes), on a :

$$\Delta(call) = \mathcal{N}(d_1)$$

et

$$\Delta(put) = \mathcal{N}(d_1) - 1$$

Sachant que $\Delta = \frac{\partial P}{\partial S}$, si l'on suppose par exemple que $\Delta = 0.52$, cela impose que lorsque S varie d'un petit montant, P varie de 0.52 fois ce montant. Couvrir une option européenne via son Δ impose donc d'acheter ou de vendre une certaine quantité d'actif afin d'être sûr de pouvoir tenir ses engagements à maturité. Etant donné que Δ varie avec le temps, le trader doit périodiquement revoir son *delta-hedging*.

1.2 Simulation

La partie simulation :

1. simule l'évolution future du sous-jacent jusqu'à la maturité de l'option, ceci étant fait en simulant un mouvement brownien géométrique (ce qui est licite puisque l'on se place sous les hypothèses du modèle de Black-Scholes)
2. effectue un delta-hedging à divers instants choisis par le programmeur (les arguments N et $NbHedge$ de la fonction SimulationHedge qui définissent la précision de la simulation du mouvement brownien ainsi que du hedging sont fixés par nous-mêmes dans l'exécution des fenêtres)

Chapitre 2

Problèmes rencontrés et solutions choisies

2.1 Utilisation de librairies externes

Quand nous avons commencé le développement du projet, certaines parties comme deltaneutral.cpp et simulation.cpp étaient réalisées sous Visual Studio à l'aide de la librairie GSL, notamment :

- l'utilisation de la fonction de répartition de la loi normale (deltaneutral.cpp)
- la génération de nombres aléatoires (simulation.cpp)

Or malgré des recherches sur plusieurs forums, l'utilisation de GSL sous Qt s'est révélée moins facile à mettre en place. Ne voulant pas trop perdre plus de temps sur l'installation de GSL sous Qt, nous avons décidé de procéder différemment en :

- implémentant directement une estimation de la fonction de répartition de la loi normale, via la méthode de Simpson (http ://math60082.blogspot.fr/2013/02/question-write-function-to-calculate-nx.html)
- implémentant un générateur congruentiel (celui implémenté en cours)

Une remarque concernant la génération de nombres aléatoires : au départ, nous avons souhaité utilisé la fonction de GSL car celle-ci est nettement plus performante que la fonction rand initiale de C++. N'ayant pu résoudre le problème de GSL sous Qt, nous avons ajouté l'implémentation de UniformGenLC vue en TD mais nous aurions pu simplement utiliser la fonction rand.

2.2 Utilisation de Qt

Qt est un logiciel libre qui permet de créer des interfaces graphiques d'utilisateurs (GUI, Graphical User Interface). Un programme avec GUI apporte beaucoup plus d'avantages qu'un programme sans GUI. On peut penser à la lisibilité, à la simplicité d'utilisation, etc. C'est en grande partie pour cela que l'on a opté pour la programmation en Qt. Ceci a été largement possible grâce à la documentation présente sur le site OpenClassRooms.

A la création de notre programme, nous avons choisi un projet vide de type *Application Qt avec Widgets*, ceci avec l'idée que dans la suite, grâce à des classes

comme QObject et QWidget (qui sont propres à Qt), on pourrait par exemple créer des classes héritières de QWidget pour faciliter le travail.

Au lieu d'utiliser des userforms (.ui) nous avons choisi de coder directement dans des classes C++ pour une raison simple : faciliter le déboggage. Par exemple, une fois que l'on commence la création des boutons en mode design, il devient très compliqué de switcher en mode éditeur. Prenons l'exemple de MainWindow : après avoir créé deux boutons en mode design, nous avons voulu ajouter une image .png et à défaut d'une idée pour le faire sans code, nous sommes ensuite passé en mode éditeur pour utiliser SetPixmap et QGridLayout, mais Qt a refusé en indiquant qu'il existait déjà un Layout dans la fenêtre. Finalement, après avoir passé une demi-journée en déboggage, nous avons laissé tomber la fenêtre MainWindow et nous avons recréé une autre fenêtre en définissant une classe proprement.

La plus grande difficulté réside dans l'allocation dynamique de la mémoire. Pour que les variables survivent à la fonction dans laquelle elles ont été créées et qu'elles soient utilisables partout dans la classe, on a besoin de créer des variables globales (déclarées dans la classe) et d'utiliser des pointeurs avec allocation dynamique. On se rappelle que ces variables sont instanciées par new et doivent être détruites à la fin de l'utilisation, c'est-à-dire à la sortie de la classe. Ce qui nous arrivait parfois, c'était un message warning à la fermeture des fenêtres, qui indiquait que le programme s'était terminé subitement. Il s'agissait bien d'un problème de destructeur. Il est peut être bête de re-préciser la différence entre variable globale et variable locale, mais il est très important d'y penser, sinon on peut facilement se tromper et être obligé d'utiliser le déboggage pendant des heures...

Une particularité des programmes Qt ? Un objet type est défini par les attributs et les méthodes. En revanche, un objet défini en Qt est constitué des attributs, des méthodes, des signaux et des slots. Ces deux derniers permettent de gérer des évènements. La fonction Connect permet de lier un signal à un slot, une fois le signal émis, le slot va être déclenché de suite. On peut également lier un signal à un autre signal. Un exemple type dans le code est que quand on clique sur sur un bouton, il se passe quelque chose : la fermeture de la fenêtre, une fonction spécifique (open result(), disable(), etc). Les types de signal pourraient être clicked(), pessed(), released(), toggled(), etc. Les slots peuvent être quit(), QMessageBox() ou des fonctions personnalisées. Il faut faire attention au fait que dans la définition de la classe, il est indispensable d'ajouter des Q OBJECT et déclarer les slots sous *public slots :*.

Chapitre 3

Description de l'architecture du programme

3.1 deltaneutral.cpp

La fonction deltaneutral définit dès le début la valeur d_1 du modèle de Black-Scholes et distingue clairement deux sous-parties dans chaque partie du programme. Soit il s'agit d'un *call* et dans ce cas là on a la sous-partie *long* ou la sous-partie *short*. Soit il s'agit d'un *put* et on dispose de la sous-partie *long* ou de la sous-partie *short*. Ces parties/sous-parties sont choisies via des booléens (calll pour un call et loong pour une position long : nous n'avons pas utiliser le mot long étant donner le conflit avec le type long présent d'origine dans C++ pour les int par exemple).

3.2 simulation.cpp

La fonction BrownGeom simule un mouvement brownien géométrique. En ce qui concerne la fonction SimulationHedge, on utilise la fonction BrownGeom puis une version de deltaneutral pour couvrir le trader à des instants définis par le programmeur. On calcule ensuite les gains via le double *earnings* en choisissant une fois de plus la bonne partie (call ou put) ainsi que la bonne sous-partie (long ou short).

3.3 La construction des fenêtres

La fenêtre d'accueil est nommée GuideWindow, qui comprend deux boutons et deux labels (l'un affichant une image et l'autre une phrase). Les deux boutons dirigent vers deux fenêtres filles : MyWindow et SimWindow. Elles ont les mêmes interfaces et se différencient par la sortie, MyWindow donnant des conseils d'ordres à placer sur le marché, tandis que SimWindow simule la perte potentielle. Enfin, la fenêtre ResultWindow affichant le résultat ou la simulation prend en argument deux types de classe, on peut les définir séparément dans le constructeur en spécifiant les arguments.

Attention ! Pour que l'image se charge correctement, il faut que celle-ci soit stockée dans le dossier de l'exécutable et non pas dans celui des ressources.

Chapitre 4

Guide d'utilisation du programme

Nous donnons ici quelques indications concernant les valeurs à indiquer dans les fenêtres :

- le taux d'intérêt *Interest Rate* est une valeur évidemment comprise entre 0 et 1.
- la volatilité *Volatility* est une valeur évidemment comprise entre 0 et 1.
- la maturité *Maturity* est en années. Ainsi, pour 20 semaines par exemple on prendra une valeur de 0.3846

Une fois de plus, on rappelle que les résultats sont basés sur les hypothèses du modèle de Black-Scholes. Des valeurs absurdes données en paramètres amènent souvent à des résultats absurdes...