Grenoble-INP ENSIMAG, UGA

3A IF 2021-2022

Rapport du projet .NET

Réalisation d'un outil d'aide à la décision en finance

Nicolas Peyrichou
Matthieu Rambaud
Nathan Salleyrette
Loïc Simeon

08/09/2021

Table des matières

Vous trouverez ici le rapport du projet de Stratégies systématiques en .NET de l'équipe : Nicolas Peyrichou - Matthieu Rambaud - Nathan Salleyrette - Loic Simeon. Nous avons recueilli et détaillé ici l'architecture de notre solution, les fonctionnalités implémentées pour répondre aux besoins du client ainsi que les grands principes mathématiques et financiers que nous avons utilisés et implémentés.

1 Introduction

L'application proposée est un outil d'aide à la décision en finance dans lequel l'utilisateur peut visualiser des données sur des produits financiers. L'application dispose d'une interface graphique permettant d'afficher le prix théorique d'une option choisie à partir de cours d'actions simulés ou réels et la valeur d'un portefeuille de couverture à partir de paramètres définis au préalable.

L'objectif de ce projet est de nous faire découvrir plusieurs aspects du développement logiciel avec la plateforme .NET de Microsoft et l'utilisation de l'IDE Visual Studio.

Au cours de ce projet, nous avons appris à travailler avec Visual Studio, mais aussi à programmer en C# qui est un des languages utilisés par la plateforme .NET. De plus, nous avons appris à appeler des librairies développées avec la plateforme .NET, nous avons lu des données stockées dans des fichiers mais aussi dans des bases de données. Nous avons aussi passé beaucoup de temps sur l'interface graphique, à apprendre comment elle fonctionnait, à rajouter des fonctionnalités, régler l'affichage et corriger des erreurs. Durant ce projet, nous avons découvert le language C#, ainsi que des concepts liés aux bases de données et le patron de conception MVVM.

2 Organisation du projet

Lors de ce projet, nous avons formé des binômes pour effectuer chacune des tâches. Cette façon de faire nous a permis d'avoir une architecture que chacun d'entre nous maîtrise pour ensuite expliquer à tout le monde chaque fonctionnalité. De plus, nous avons pu éviter de nombreuses erreurs et les phases de débugging étaient, elles aussi, plus rapides. La communication a été au cœur de notre projet amplifié par une bonne entente entre les membres et un retour en présentiel.

3 Architecture du projet

Nous avons structuré notre projet en suivant le patron MVVM. Tout d'abord, Model-view-viewmodel (MVVM) est un modèle d'architecture logiciel qui facilite la séparation du développement de l'interface utilisateur graphique (la vue) du développement de la logique métier ou back-logique de fin (le modèle) de sorte que la vue ne dépende d'aucune plateforme de modèle spécifique. De plus, nous avons essayé de rendre notre code le plus modulaire possible. Ainsi à la place de faire une distinction à chaque instant dans notre code via l'utilisation d'intructions de type instanceOf, nous avons décidé de créer des interfaces puis une classe abstraite par entités possibles. Enfin, nous pensons avoir assez peu de code dupliqué, même s'il reste certaines fonctions qui possèdent un code similaire.

3.1 Points particuliers :

3.1.1 AbstractController

Cette classe abstraite a pour but de gérer le comportement global de notre simulateur. Cette classe permet de calculer les paramètres de pricing à chaque instant (soit estimé avec la classe ControllerHistoric, soit en dur avec la classe Controller). Il permet aussi d'obtenir dans le cas de données estimées, la fenêtre d'estimation. Nous avons choisi comme structure une classe abstraite car il y a beaucoup de fonctions en commun avec les deux sous-classes.

3.1.2 AbstractPortfolio

Cette classe abstraite correspond à toutes les caractéristiques de notre portefeuille. Il contient le nombre d'actifs risqués en notre possession, ainsi que la quantité d'argent placée au taux sans-risque. Il existe une classe portfolio propre aux VanillaCall et une autre propre aux Basket. Il était nécessaire de faire cette distinction car nous calculons le prix et les deltas avec les fonctions du Pricer.

3.1.3 Estimator

Cette classe permet d'obtenir nos paramètres estimés, donc il calcule la volatilité ainsi que la matrice de corrélation. Le tout en faisant appel à la librairie WRE. La classe Estimator est appelé par les sous classes d'abstractController pour calculer les tableaux de volatilités et les matrices de corrélation.

3.1.4 HistoricDataManager

Cette classe permet de se connecter à la base de données et de récupérer les données sous la forme d'un DataFeed.

3.1.5 Services

Ces classes sont présentes pour assurer le lien entre la fenêtre déroulante de l'interface graphique et les choix possibles. Il y a actuellement trois fenêtres déroulantes : - Une pour la provenance des données, - Une pour choisir l'estimation ou non des paramètres, - La dernière pour choisir quel test lancer (ces tests d'options sont écrit dans un fichier json).

3.1.6 Models

Dans ce dossier, on retrouve les classes relatives au dessin du graphique liveCharts.

3.1.7 Strategies

Les classes strategy définissent la composition du portefeuille. Il en existe une propre aux VanillaCall et une autre propre aux Basket. Comme ces classes ont certaines fonctions en commun, nous avons décidé de créer une classe abstraite (AbstractDeltaNeutralStrategy) et dans l'hypothèse où nous créerions une nouvelle stratégie, nous avons créé une interface de stratégie (IStrategy.cs)

3.1.8 Views

Ici, nous avons tous les éléments liés à l'affichage de notre interface. C'est à dire la SelectionView qui regroupe les menus déroulants et la ChartView qui permet d'afficher le graphique dans l'interface.

3.1.9 ViewModels

Dans ce dossier, il y a les classes et interfaces nécessaires à la liaison entre la MainWindowViewModel et l'interface graphique.
Tout nos fichiers XAML ont été traités à part.
Pour finir, voici un diagramme de classe résumant les différentes intéractions de nos différentes classes :

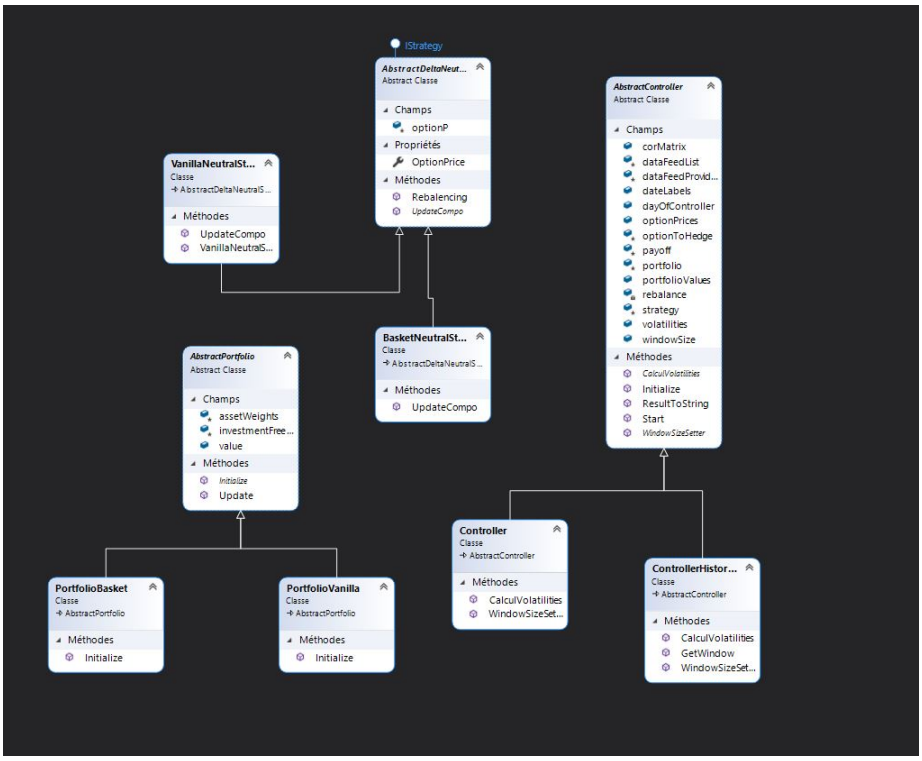

Figure 1: Diagramme de classes

4 Fonctionnalités implémentées

Nous avons pu lors de cette semaine de projet implémenter les fonctionnalités suivantes :

- Création d'une stratégie de couverture pour les options de type Vanilla Call
- Création d'une stratégie de couverture pour les options de type Basket
- Possibilité de chosir entre des données simulées, semi-historiques ou historiques, choix de la date de début et de fin du test.
- Calcul de la Tracking Error pour les stratégies ci-dessus, indication du payoff de l'option et valeur du portefeuille de couverture.
- Création de graphes permettant de suivre la valeur de notre portefeuille et le prix de l'option à couvrir au cours du temps.

- Ajout de DatePicker, ainsi que d'un menu déroulant permettant de choisir le type de données à utiliser.
- Utilisation de la bibliothèque Wall Risk Engine pour estimer les différents paramètres.
- Réalisation d'un estimateur des paramètres de pricing.
- Possibilité de choisir si les données sont estimées ou codées en dur.
- Réalisation d'un fichier JSON permettant de lancer des tests facilement.
- Possibilité de saisir la taille de la fenêtre d'estimation
- Possibilité de saisir la fréquence de rebalancement

4.1 Explication détaillée

4.1.1 La stratégie

Pour les stratégies, nous avons décider d'utiliser celle du delta-neutre. C'est à dire que l'on va essayer de couvrir notre option en achetant uniquement des sous-jacents de cette option. Voici comment ce déroule la première itération pour un vanilla call :
A $t = 0$:
$portfolio_{value} = premium$
On achète δ_0 d'actif
On place $premium - \delta_0 S_0$ dans l'actif sans risque.
A $t = 1$ (premier rebalancement) :
$portfolio_{value} = (premium - \delta_0 S_0)(1 + r) + \delta_0 S_1$
On achète δ_1 d'actif
On place $portfolio_{value} = (premium - \delta_0 S_0)(1 + r) + \delta_0 S_1 - \delta_1 S_1$

4.2 Tracking error

Pour vérifier la cohérence de notre modèle, nous avons calculé la tracking error. Cette erreur correspond simplement à l'écart entre la valeur de notre protefeuille et le payoff de l'option. Nous avons réussi à obtenir un modèle qui parvient à avoir une erreur d'estimation souvent inférieure à 30% dans les pires cas.

4.3 Respect du cahier des charges

Notre application permet d'effectuer du forwardtest et du backtest sur différents produits financiers avec un portefeuille de couverture. En effet, un forwardtest exécute un algorithme sur de la data simulée tandis qu'un backtest exécute un algorithme sur de la data existante. Dans un premier temps, il était plus intéressant d'effectuer du forward testing car on saisissait les paramètres comme la volatilité directement dans notre programme et on savait quelle valeur on

devait obtenir. Une fois cette vérification effectuée sur plusieurs exemples, nous sommes passés au back testing.

4.4 Json

Pour pouvoir tester des scénarii différents, précis et définis à l'avance, nous avons mis en place un fichier de configuration Json dans lequel nous retrouvons de nombreux scénarii et pouvons en ajouter de nouveaux (toujours dans le même fichier). Ces scénarii sont sélectionnables depuis l'interface.

5 Modèles

Les options à couvrir étaient le Vanilla call $(S_t - K)_+$ et l'Average basket $(\sum \omega_i S_T^i - K)_+$. Pour cela, nous disposions d'informations des marchés sur les 5 dernières années, elles étaient disponibles sur une base de données. De plus, nous disposions d'une librairie de pricing avec plusieurs fonctions sur le Vanilla call et l'Option basket que nous avons utilisées dans notre algorithme. En plus de cette librairie de pricing, nous avons utilisé la librairie d'analyse de risque WRE. En effet, cette librairie développée en C nous a permis d'estimer la volatilité des produits mais aussi d'accéder aux matrices de covariance.

Une fois que notre application est lancée, l'interface graphique s'affiche et l'utilisateur peut choisir les caratéristiques de l'option, la date à partir de laquelle il veut récupérer des données de marché, la date de fin de son test. Il peut aussi choisir si les données sont simulées, semi-historiques ou historiques ainsi que la fenêtre d'estimation des paramètres.
L'application retourne en sortie le payoff de l'option, la valeur du portefeuille de couverture correspondant qu'on trace avec livecharts pour suivre leurs évolutions. De plus, notre application calcule et renvoie la tracking error du test : $\frac{VL_{pf} - payoff}{prix\_option}$

6 Présentation des tests réalisés

6.1 Stratégie globale

Pour cette partie, nous avons principalement utilisé les graphiques LiveChart pour vérifier la couverture de notre portefeuille. Grâce à notre interface graphique, nous pouvons facilement tester nos modèles en sélectionnant simplement les dates de début et de fin puis le type de données à considérer. De plus lorsque nous souhaitions tester une fonctionnalité, nous avions pour habitude de la tester dans un premier temps dans un programme console avant de le relier au reste de notre projet. Cependant, étant donné le peu de temps dont nous disposions, nous n'avons pas penser à réaliser des tests de non-régréssion. De plus la structure complexe de notre modèle a rendu complexe le test de certaines fonctionnalités.

6.2 Estimation des paramètres

Pour vérifier la qualité de nos estimateurs, nous avons créé un nouveau projet dans lequel nous avons simplement essayé de calculer la volatilité et la matrice de covariance sur des données simulées. Le projet s'appelle projet console. Pour vérifier la fiabilité, nous avons utilisé plus de 10000 points.

7 Bilan collectif du projet et conclusion

Lors de ce projet, nous avons pu découvrir l'utilisation de la plateforme .NET, ainsi qu'une application concrète de la finance à l'informatique. Ce projet fut certe intense pour nous, mais nous avons réussi à produire une interface graphique qui satisfait les principaux objectifs attendus. Chacun a su trouver sa place au sein du groupe pour nous permettre d'avancer au mieux. A cela, nous pouvons ajouter une très bonne communication qui nous a permis d'avancer au plus vite. Les principaux challenges que nous avons dû surmonter sont les suivants :

- Familiarisation avec le langage C# et avec la PricingLibrary
- Utilisation de WRE pour estimer les paramètres
- Rendre notre interface robuste (surtout avec les DatePickers)
- La factorisation du code qui nous a posé beaucoup de problème et de réflexion.

Nous avons réussi à les relever pour répondre au mieux aux différentes attentes du cahier des charges.