Rapport de projet .NET

ZUBER Zachary
RABUSSON Gregoire
BENSOUSSAN Julien
Groupe Teide n°3

Septembre 2022

Table des matières

1 Introduction

Le but de ce projet est de créer puis de **backtester** une stratégie systématique de couverture d'un portefeuille d'un portefeuille d'option "Call" sur n sous-jacents :

$$(\sum_{i=1}^{n} \omega_i S_T^i - K)_+$$

Avec T la date de maturité de l'option, K le prix d'exercice et pour tout $i \in 1...n$, $(S_t^i)_{t\in[0,T]}$ représente les prix et $\omega_i > 0$ la quantité de sous-jacent.

2 Stratégie de couverture systématique

Un stratégie systématique est une stratégie qui peut s'effectuer sans intervention humaine. Comme dit précédemment, le but est de couvrir le portefeuille d'option précédent contre le risque. Le pseudo-code d'une stratégie systématique est donné ci-dessous.

```
Input  : Initial value V_{t_0} of the portfolio (how much to invest)
Input  : Market data for the underlying assets
Output: Portfolio values
1  begin
2      UpdateCompo(t_0) // nb: there can be additional parameters
3      foreach date t > t_0 do
4          V_t := UpdatePortfolioValue(t)
5          if RebalancingTime(t) then
6              UpdateCompo(t)
7          end
8      end
9      return (V_{t_0}, ..., V_{t_N})
10 end
```

Figure 1: Pseudo-code d'une stratégie systématique

Pour chaque date $t_i \in t_0, t_1, ..., t_N$ notre portefeuille est composé de deux types d'actifs :

- des actifs risqués $S_{t_i}^1, ..., S_{t_i}^n$ dans les proportions $(\delta_{t_i}^1, ..., \delta_{t_i}^n)$;
- Une quantité d'actif sans risque B_{t_i} empruntée ou placée à un certain taux r_{t_i}.

Dans la suite, pour l'actif sans risque on utilisera R_{t_i} qui représente la valeur cumulée au taux r_{t_i} d'une unité de monnaie.

Tout l'enjeu du projet est de savoir comment et quand rebalancer le portefeuille pour pouvoir avoir une couverture de risque optimale.

Nous allons détailler dans un premier temps la fonction *UpdateCompo()* du pseudo-code qui implémente une stratégie de portefeuille autofinancé pour rebalancer le portefeuille, c'est-à-dire comment mettre à jour sa composition. Puis nous verrons la fonction *RebalancingTime()* qui nous permet de décider quand mettre à jour notre portefeuille pour qu'il couvre notre position.

2.1 Formalisation mathématique d'un portefeuille

Formellement, un portefeuille peut être décrit à chaque instant t_i par le vecteur de $\mathbb{R}^{n+1}$:

$$C_{t_i} = \begin{pmatrix} \delta^1_{t_i} \\ \delta^2_{t_i} \\ \vdots \\ \delta^n_{t_i} \\ B_{t_i} \end{pmatrix}$$

Et on peut obtenir la valeur du portefeuille en effectuant de produit scalaire de ce vecteur C_{t_i} avec le vecteur S_{t_i} qui concatène les prix et le taux au même instant.

$$S_{t_i} = \begin{pmatrix} S^1_{t_i} \\ S^2_{t_i} \\ \vdots \\ S^n_{t_i} \\ R_{t_i} \end{pmatrix}$$

L'autofinancement signifie que la valeur du portefeuille juste avant chaque rebalancement est la même qu'après celui-ci.

2.2 Mise à jour de la composition du portefeuille autofinancé

Nous choisissons la composition du portefeuille en actifs risqués avec la méthode du delta-hedging. En effet nous avons une composition à chaque instant N de $(\delta^1_{t_N}, ..., \delta^n_{t_N})$ en proportion d'actifs risqués. Ces deltas sont calculés à l'aide d'un priceur spécifique et nous n'avons pas cherché à savoir comment les calculer dans ce projet.

Soient deux instants successifs t_i et t_{i+1}. Nous avions à l'instant t_i une composition C_{t_i} et une valeur $V_{t_i} =< C_{t_i}, S_{t_i} >$.

Juste avant l'instant t_{i+1}, la valeur de ce portefeuille se calcule avec la combinaison linéaire des prix à l'instant t_{i+1}^- avec la composition à l'instant t_i aditionnée à la quantité d'argent placée B_{t_i} capitalisée au taux R_{t_i}. On a donc la formule :

$$V_{t_{i+1}}^- = \sum_{k=1}^{n} \delta_{t_i}^k \times S_{t_{i+1}}^k + B_{t_i} R_{t_i}$$

Or à l'instant t_{i+1}, nous avions dit que l'on voulait avoir une composition $(\delta_{t_{i+1}}^1, ..., \delta_{t_{i+1}}^n)$ d'actifs risqués, connue car donnée par notre pricer et une certaine quantité $B_{t_{i+1}}$ d'actif sans risque. Donc pour respecter notre stratégie d'autofinancement, il faut que les deux valeurs (juste avant et après rebalancement) du portefeuilles soient égales. C'est-à-dire que :

$$\sum_{k=1}^{n} \delta_{t_i}^k \times S_{t_{i+1}}^k + B_{t_i} R_{t_i} = < C_{t_{i+1}}, S_{t_{i+1}} > = \sum_{k=1}^{n} \delta_{t_{i+1}}^k \times S_{t_{i+1}}^k + B_{t_{i+1}}$$

On a une équation à une inconnue $B_{t_{i+1}}$ et on connait alors la nouvelle composition de notre portefeuille $C_{t_{i+1}}$ à l'instant t_{i+1}.

2.3 Oracle de rebalancement

Il s'agit maintenant de savoir quand rebalancer notre portefeuille autofinancé. En effet, on peut choisir d'effectuer la mise à jour de sa composition de façon quotidienne, hebdomadaire ou autre. Dans le cadre de ce projet, nous avons implémenté deux stratégies de période de rebalancement pour *Rebalancing-Time()*.

La première constitue en un rebalancement hebdomadaire. En effet, on choisit un jour de la semaine (Lundi, Mardi, Mercredi etc...) et cette fonction renverra true ce même jour de toutes les semaines suivantes et false sinon. La deuxième est un rebalancement régulier d'une période P, c'est-à-dire qu'elle renverra true tous les **P-jours ouvrés** et false sinon.

Il sera par la suite très facile d'implémenter des nouvelles stratégies pour cet oracle de rebalancement car nous avons utilisé une interface. Il n'est en effet pas nécessaire de savoir si l'oracle est hebdomadaire ou journalier pour notre portefeuille de couverture. Cette architecture nous permet d'ajouter et de "brancher" facilement une nouvelle startégie de rebalancement.

3 Librairie

3.1 Architecture et implémentation

On a dans ce projet une application console BacktestConsole dont l'utilisation pratique est expliquée dans le mode d'emploi qui va suivre. On a également

une bibliothèque de classe AutofinancingPortfolioLibrary qui contient les classes décrivant les portefeuilles décrite sur la Figure 2 et une classe de test TestUnit pour les tests unitaires.

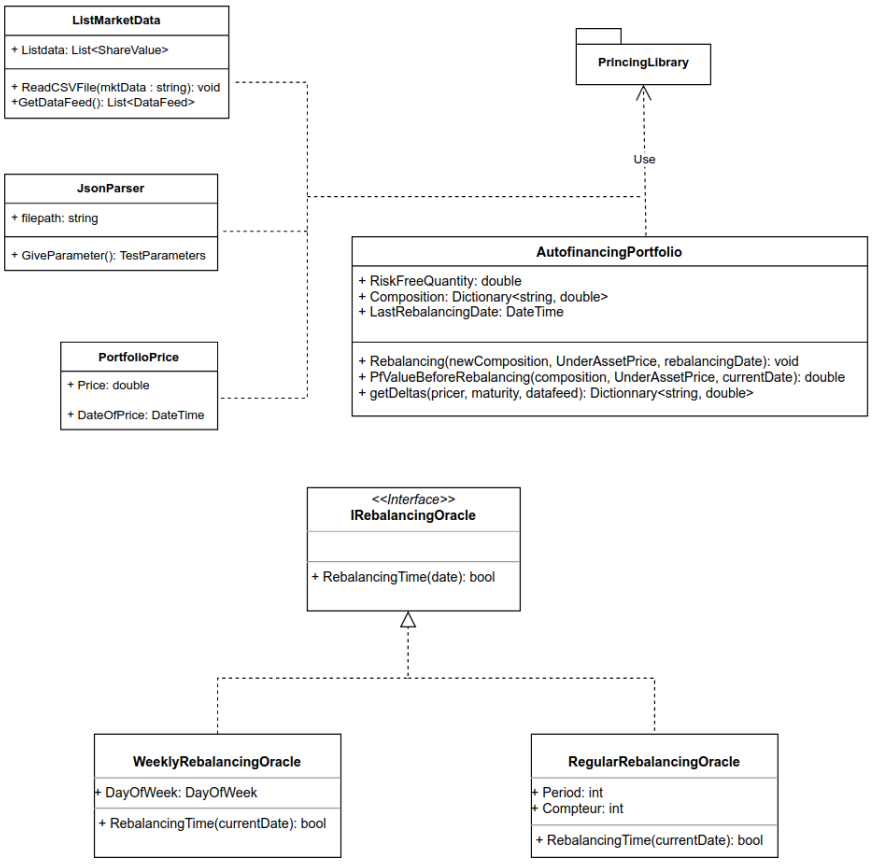

Figure 2: Diagramme de classe de la bibliothèque de classse

3.2 Mode d'emploi

Notre librairie s'utilise comme une application console .NET nommée Backtest-Console. Elle peut être exécutée depuis le terminal en utilisant la commande suivante *BacktestConsole.exe test-params mkt-data pf-vals th-vals*

- *test-params*: chemin vers le fichier JSON contenant les paramètres du test.
- *mkt-data*: chemin vers le fichier CSV contenant les données du marché sur lequel le test sera exécuté.

- *pf-vals*: chemin vers le fichier CSV de sortie contenant les valeurs du portefeuille répliqué.
- *th-vals*: chemin vers le fichier CSV de sortie contenant les valeurs théoriques du portefeuille répliqué.

Les fichiers CSV que nous produisons par la console contiennent deux colonnes: une avec les dates et une avec les valeursavec des en-têtes.

4 Tests et performance

4.1 Tests unitaires

Pour voir si notre application de Backtest fonctionne correctement il est essentiel de savoir si nos fonctions qui donnent la valeur du portefeuille et de recomposition fonctionnent. C'est pourquoi nous avons effectué des tests NUnit dans la classe de test AutofinancingPortfolioTest du répertoire UniTest.

Les deux premiers tests, *TestPfInitialValue()* et *TestPfValue()* nous permettent de tester si la méthode de calcul de valeur du portefeuille PfValueBeforeRebalancing() est correcte et renvoie bien la bonne valeur du portefeuille. Ensuite, le test *TestPfRebalancing()* nous permet de vérifier si la méthode de rebalancement du portefeuile *Rebalancing()* est correcte et respecte bien les règles du portefeuille autofinancé expliquées dans la partie théorique. La validation de ces deux tests nous permet d'implémenter la stratégie systématique en sachant que nos consignes de rebalancement sont respectées.

Le deuxième point important à tester pour voir si notre stratégie systématique permet de bien couvrir le portefeuil est relatif aux oracles de rebalancement de l'interface *IRebalancingOracle*. Nous n'avons cependant pas effectué de test Nunit sur les méthodes des classes qui implémentent cette interface car leur code est relativement simple et des tests simples avec l'exécutable Backtest.exe nous ont permis de voir la librairie donne le résultat attendu de backtest.

4.2 Performance de couverture

A partir des fichiers CSV en sortie de l'exécution de l'application console, nous pouvons voir si notre stratégie de portefeuille autofinancé pour se couvrir est efficace en regardant d'une part l'écart entre les valeurs du portefeuille autofinancé et les valeurs données par le pricer à chaque rebalancement de portefeuille ce qui est illustré par la Figure 3. Nous avons choisi ici un rebalancement régulier tous les jours ouvrés, et les données de marché contenues dans **data_share_5_3.csv** et nous pouvons voir que la couverture semble bonne puisque les deux courbes se chevauchent.

Dans un second temps nous pouvons regarder l'indicateur de qualité q de la couverture pour avoir une idée un peu plus précise. Cet indicateur, aussi

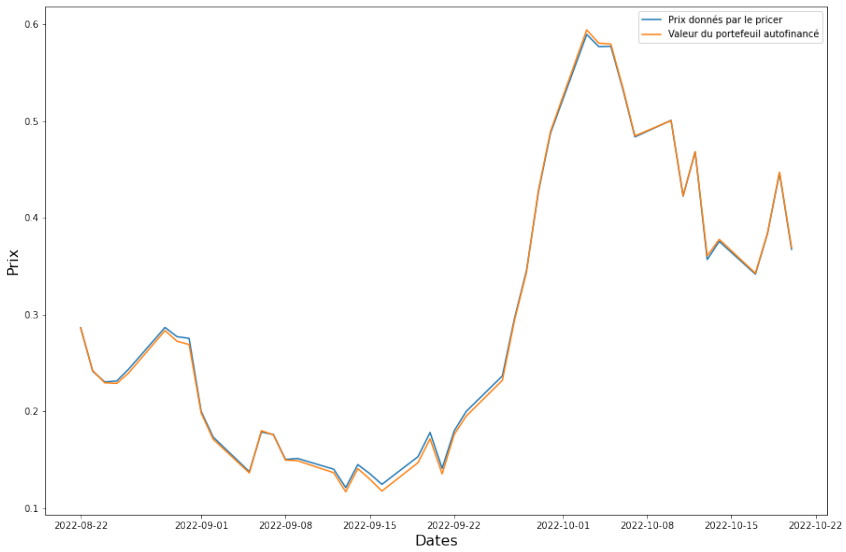

Figure 3: Comportement du portefeuille de réplication autofinancé et prix de l'option à couvrir

appelé **TrackingError** est calculé en fonction de l'écart de la manière suivante à chaque instant :

$$TrackingError = \frac{|Prix_{pricer} - Prix_{Pfautofinance}|}{P_0}$$

Ainsi une couverture parfaite aura $TrackingError = 0$ et plus on s'éloigne de 0 plus la couverture sera mauvaise. Nous avons donc tracé sur la Figure 4 l'évolution

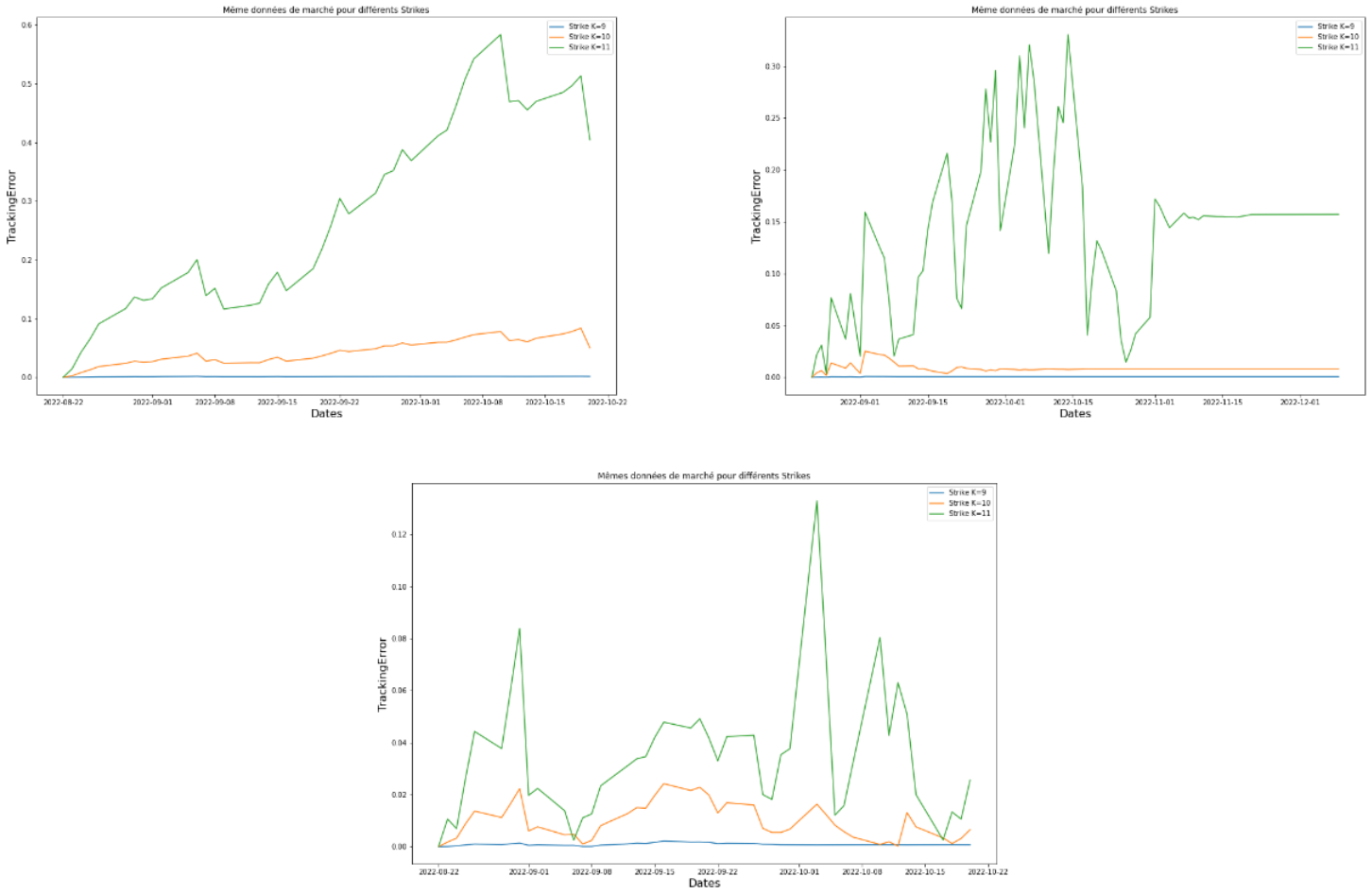

Figure 4: Ecart relatif illustrant la qualité de couverture du portefeuille autofinancé

Une première étude possible est de comparer la valeur de la tracking error pour un même marché pour des strikes différents. On observe sur les graphes ci-dessus que la tracking error est négligeable pour un strike de 9 ou de 10 comparé à un strike de 11 quelque soit le marché étudié. On peut alors affirmer que la valeur du strike impacte la tracking error, et emmettre l'hypothèse que plus le strike est elevé, plus la tracking error augmente, même si il nous ai difficile de trouver une justification à cela. Nous pouvons également emmettre l'hypothèse que la valeur de la tracking error se stabilise pour un marché et un strike auourt

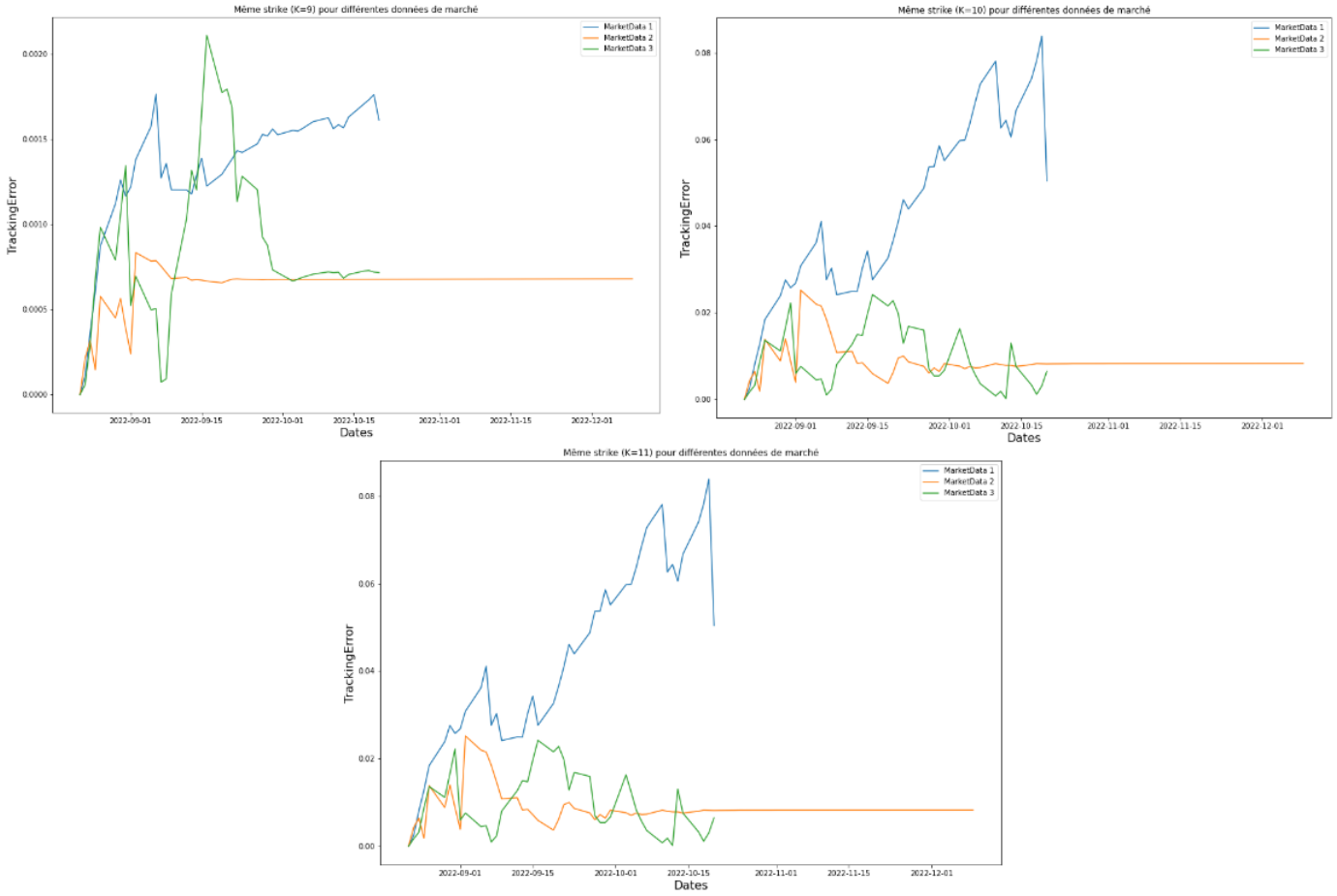

Figure 5: Ecart relatif illustrant la qualité de couverture du portefeuille autofinancé

d'une valeur au bout d'une certaine date, même si plus d'écheance nous aurait permis de de vérifier cette hypothèse.

Une autre étude possible est de comparer la valeur de la tracking error pour un même strike pour des marchés différents comme on peut l'observer ci-dessus. Ici, on se rend compte que le marché à moint d'impact que le strike sur la valeur de la tracking error. Même si il semblerait que le marché 1 ne soit pas le plus favorable, la tracking error reste dans un même ordre de grandeur.

De plus nous avons essayé de jouer sur la période de rebalancement P. Nous pouvons voir sur la Figure 6 que la TrackingError est plus importante pour une période P = 3 jours ouvrés que pour un rebalancement quotidien de P = 1 jour ouvré.

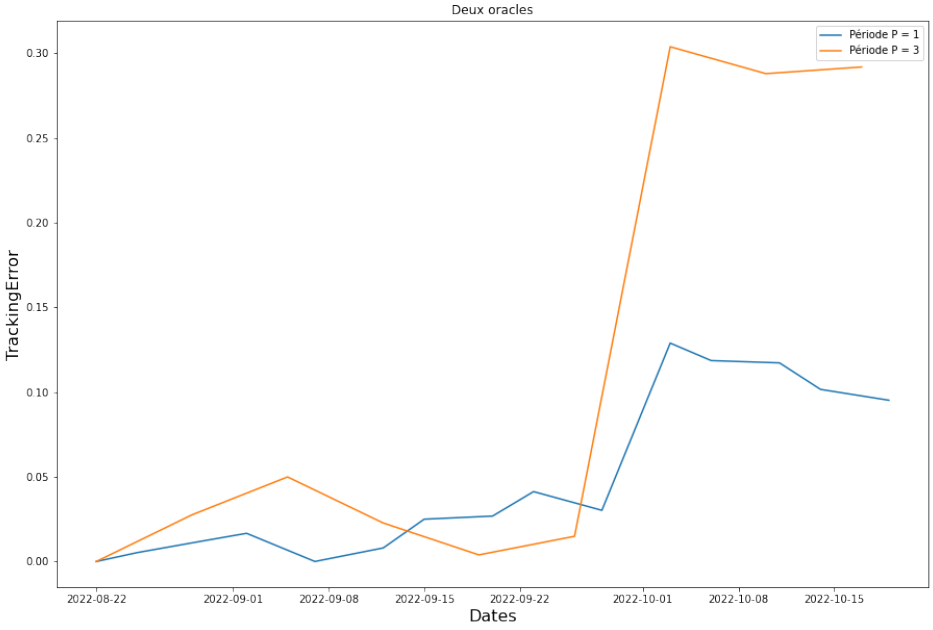

Figure 6: Ecart relatif illustrant la qualité de couverture pour d'autres périodes de rebalancement

4.3 gRPC server

L'idée ici a donc été d'utiliser grpc pour créer un server. Nous avons essayé d'implémenter le coté serveur qui semble fonctionnel. En effet, nous avons récupéré les "testParams" et "MarketData" via le BacktestRequest. Il a donc fallu convertir ces éléments récupérés en classes de la console, pour pouvoir réutiliser le même code que nous avons utilisé pour calculer les valeurs du portefeuille de réplication. Un des gros problèmes que nous avons rencontrer, et qui nous a fait perdre un temps précieux a été de comprendre qu'il fallait convertir une première fois du coté client les données et les convertir une deuxième fois dans l'autre sens. Nous n'avons pas pu tester le code du serveur a cause d'une erreur dans la partie client. En effet, lorsque nous affichons les valeurs, nous obtenons 255 fois la derniere valeur du prix. Ainsi, nous esperons que le server fonctionne correctement, il a été quand meme testé pour des valeurs par défaut mais donc pas pour de vrais paramètres.

5 Conslusion et améliorations possibles

La prochaine étape serait de finir l'étape 4 sur le gRPC server et l'interface MAUI qui permet de séléctionner les paramètres de test, les données du marché et utilise le client gRPC pour effectuer les tests.

De plus, nous n'avons utilisé notre librairie que pour effectuer du Backtest avec des données de marché fournies mais il serait également possible d'effectuer du Forward-test grâce à la libraire fournie qui permet de simuler des données de marché. Ces forwardtests auraient pu nous donner plus de matières pour analyser les performances et les résultats de notre apllication.

Puis dans un second temps nous pourrons nous intéresser à l'oracle de rebalancement. Pour le moment cet oracle ne prenait en compte que la date et renvoyait un booléen si il fallait mettre à jour le portefeuille à l'intant T. Cependant, on sait qu'il y a des commissions à chaque achat et vente d'un actif donc il n'est pas judicieux de faire un rebalancement si la qualité de couverture de notre portefeuille est suffisamment bonne.

Enfin, notre server est largement perfectible du au manque de test qu'il a subi.