Récupération de données de transactions

Alexandre RABHI et Julien ZANIN

16 février 2025

Introduction

Ce projet vise à collecter, traiter et réorganiser une liste désordonnée de transactions sous la forme de carnets d'ordres afin d'obtenir un outil d'analyse et de visualisation sur l'état du marché pour différents actifs. Afin de tester le programme, un module entier est d'abord dédié à la génération aléatoire de transactions et à leur stockage dans un fichier csv. A partir de ce fichier de données, un second module est dédié à la réorganisation des transactions sous forme de carnets d'ordres pour chaque actif échangé dans le pool de transactions. Un troisième module permet à l'utilisateur de lancer une simulation pendant une durée définie au cours de laquelle des ordres aléatoires sont générés puis traités toutes les 20 secondes afin d'illustrer la dynamique de chaque carnet face à la venue de nouveaux ordres.

En complément de ces fonctionnalités, plusieurs modules assurent une gestion complète des transactions et du portefeuille. Un module suit la détention des actifs de l'utilisateur en mettant à jour le prix moyen d'achat lors de nouvelles acquisitions et en calculant les profits et pertes, réalisés comme non réalisés. Un autre module gère le solde disponible pour les achats et les liquidités issues des ventes, garantissant ainsi que chaque opération respecte les fonds disponibles. Un troisième module orchestre l'exécution des ordres en synchronisant les flux financiers et les positions en titres, assurant la cohérence entre le portefeuille et le solde bancaire. Enfin, un dernier module gère l'interface sur laquelle les utilisateurs saisissent leurs ordres d'achat ou de vente, tout en validant ces saisies pour éviter des erreurs évidentes.

Ce projet constitue une base solide pour comprendre les mécanismes de la microstructure de marché à travers l'analyse des dynamiques d'achat et de vente. Il peut être utilisé pour tester l'impact des flux d'ordres sur le prix d'un actif ou pour mieux comprendre le rôle des carnets d'ordres dans la formation de ce prix.

Dans ce rapport nous nous attarderons d'abord sur le fonctionnement et le but des différents modules du programme puis nous évoquerons brièvement les difficultés que nous avons pu rencontrer lors de l'écriture de ces derniers. Enfin nous présenterons les potentielles améliorations qu'il est encore possible d'apporter à ce programme.

1 Présentation des modules

1.1 Génération des ordres

Afin de tester l'intégralité du programme, le module `OderGenerator.cpp` génère des transactions aléatoires et les sauvegarde dans un fichier csv pour qu'elles puissent être utilisées par la suite. Au lancement de la génération, l'utilisateur doit préciser plusieurs paramètres :

— Le nombre d'actifs pour lesquels il souhaite générer des ordres (variable de type `int`) ;

— Le nombre de transactions qu'il veut générer pour chaque actif (il peut s'agir d'une variable de type `int` s'il veut le même nombre de transactions pour tous les ordes ou d'une variable de type `vector<int>` sinon) ;
— Les prix autour desquels il souhaite que les transactions générées fluctuent (il peut s'agir d'une variable de type `double` s'il veut le même prix pour tous les actifs ou d'une variable de type `vector<double>` sinon) ;
— La part d'ordres de vente à découvert qu'il désire pour chaque actif (il peut s'agir d'une variable de type `double` s'il veut la même part pour tous les actifs ou d'une variable de type `vector<double>` sinon).

Chaque ordre généré possède 8 caractéristiques différentes comme indiqué dans la table 1.

ID	Asset	Timestamp	Type	Short Sell	Price	Quantity	Amount
1	AAPL	2025-02-15 14 :30 :56	Buy	No	150.00	10	1500.00
2	TSLA	2025-02-15 14 :30 :21	Sell	Yes	800.50	5	4002.50
3	AMZN	2025-02-15 14 :30 :17	Buy	No	3300.75	2	6601.50

TABLE 1 – Exemple de transactions générées

Le champ ID correspond à l'identifiant de l'opérateur à l'origine de l'ordre. Pour le générer on utilise une loi uniforme discrète telle que :

$$ID \sim U(\{1, 2, \ldots, 300\})$$

Ce choix de modélisation permet de maximiser le réalisme des ordres générés. Un même opérateur pourra en effet être à l'origine de plusieurs transactions.

Le champ Asset correspond à l'actif concerné par la transaction. Au lancement de la simulation, le programme va chercher au hasard dans une liste d'actifs le nombre d'actifs souhaités par l'utilisateur à l'appel de la fonction à l'aide d'une variable de type `set<int>` pour éviter le problème des doublons.

Le champ Timestamp correspond à l'horodatage de l'instant où l'ordre est placé. Cet horodatage est généré aléatoirement à l'aide de lois uniformes avec toutefois l'année (2025) et le mois (février) fixés arbitrairement.

Le champ Type correspond à la direction de l'ordre (achat ou vente). Lorsque l'utilisateur spécifie le nombre de transactions qu'il souhaite, la fonction génére automatiquement 50% d'odres d'achat et 50% d'ordres de vente.

Le champ Short Sell correspond à une variable de type `bool` qui indique s'il s'agit d'un ordre de vente à découver ou non. Dans la suite du projet, cette variable n'a pas de réelle utilité car le carnet d'ordre traite tous les autres de vente de manière identique indépendamment de s'il s'agit d'une vente à découvert ou non. Nous l'avons toutefois laissée pour rester fidèle à la consigne.

Les champs Price et Quantity correspondent au prix et à la quantité d'actif de l'ordre. Ils sont tous les deux générés à partir de lois de probabilité. Le prix dépend de l'actif sur lequel porte l'ordre et est tiré d'une loi normale centrée sur le prix moyen indiqué en entrée par l'utilisateur pour ledit actif avec une variance de 1. Le quantité est quant a elle tirée d'une loi uniforme continue avec pour support $[0, 1; 1000]$. Le champ Amount correspond au montant de la transaction et n'est autre que le produit du prix de l'ordre par sa quantité.

1.2 Construction des carnets d'ordres

Le module `OrderBookManager.cpp` a deux objectifs : construire des carnets d'ordres à partir d'un fichier csv contenant des transactions non triées et gérer, une fois le carnet d'ordre établi, l'arrivée de nouveaux ordres. Le module se sert de structures pour agréger et manipuler les données comme `Order` pour gérer les ordres du fichiers de transactions ou encore `OrderBookStatistics` et `OrderBookEntry`.

Bid Volume	**Price**	**Ask Volume**
	150.3	354.6
	150.2	2206.68
	150	3023.59
1200.4	149.9	
1800.1	149.8	
1450.8	149.7	

TABLE 2 – Exemple de carnet d'ordres

Un carnet d'ordres est une représentation structurée de l'offre et de la demande sur le marché d'un actif. Comme présenté dans la table 2, il se compose de deux colonnes principales :

- Le côté "Bid" (offre d'achat) contient les prix et volumes proposés par les acheteurs pour acquérir l'actif ;
- Le côté "Ask" (offre de vente) contient les prix et volumes proposés par les vendeurs pour céder l'actif.

Les bids sont toujours classés par ordre décroissant, le meilleur bid (le plus élevé) étant en haut. Inversement, les asks sont classés par ordre croissant, le meilleur ask (le plus bas) étant en haut. Par conséquent dans le code on se sert de `map` pour modéliser ces deux côtés du carnet en utilisant le prix en clé et le volume correspondant en valeur. Ces objets sont particulièrement adaptés à ce contexte car ils permettent le tri automatique des paires clé-valeur en fonction de la clé, et donc du prix.

A partir du carnet d'ordre on peut construire deux métriques clés pour l'actif échangé : le spread Bid-Ask et le prix mid. Le Bid-Ask Spread représente la différence entre le meilleur prix d'achat (highest bid) et le meilleur prix de vente (lowest ask). Il est donné par la formule :

$$\text{Spread}_{\text{Bid-Ask}} = P_{\text{ask}}^{\text{best}} - P_{\text{bid}}^{\text{best}}$$

Avec $P_{\text{ask}}^{\text{best}}$ le prix de vente le plus bas (meilleur ask), et $P_{\text{bid}}^{\text{best}}$ le prix d'achat le plus élevé (meilleur bid).

Le prix mid, ou prix médian du marché, est défini comme la moyenne arithmétique du meilleur bid et du meilleur ask :

$$P_{\text{mid}} = \frac{P_{\text{ask}}^{\text{best}} + P_{\text{bid}}^{\text{best}}}{2}$$

Lorsque que la méthode `processOrders` de la classe `OrderBookManager` récupère les transactions du fichier de données, elle suit la logique suivante : tant qu'un ordre d'achat (bid) est supérieur ou égal au meilleur prix de vente (ask), elle considère qu'une transaction peut avoir

lieu. Dans ce cas, l'ordre d'achat est apparié avec le ou les ordres de vente correspondant(s), et les transactions sont exécutées (elles sont supprimées) jusqu'à ce qu'il ne reste plus d'ordres compatibles. Si après ces transactions, aucun bid ne dépasse le meilleur ask restant, le carnet d'ordres trouve son équilibre et l'exécution s'arrête.

Lorsque les quantités des ordres ne correspondent pas exactement, la méthode applique un principe de First In, First Out (FIFO). Autrement dit, les ordres les plus anciens sont exécutés en premier. Ainsi :

- Si un ordre d'achat est supérieur au meilleur ask mais avec une quantité insuffisante pour absorber tout le volume disponible, il est entièrement exécuté et l'ordre de vente restant est partiellement exécuté ;
- Si un ordre de vente est inférieur ou égal au meilleur bid mais ne couvre qu'une partie du volume disponible, il est entièrement exécuté, et l'ordre d'achat restant attend qu'un autre vendeur se positionne.

Mathématiquement, en notant Q_{bid} et Q_{ask} les quantités disponibles respectivement au meilleur bid et meilleur ask :

$$\begin{cases} Q_{\text{bid}} \geq Q_{\text{ask}} \Rightarrow Q_{\text{bid}} - Q_{\text{ask}} \text{ reste dans le carnet} \\ Q_{\text{ask}} > Q_{\text{bid}} \Rightarrow Q_{\text{ask}} - Q_{\text{bid}} \text{ reste dans le carnet} \end{cases}$$

Une fois les carnets d'ordres construits, la méthode affiche pour chacun d'eux différentes métriques :

- Le prix d'exécution moyen ;
- Le volume total échangé des ordres considérés comme passés ;
- Le Montant total échangé des ordres considérés comme passés ;
- Le meilleur prix d'achat du carnet ;
- Le meilleur prix de vente du carnet ;
- Le prix mid du carnet ;
- Le spread Bid-Ask ;
- La profondeur du côté Bid (le nombre de niveaux de prix des ordres d'achat) ;
- La profondeur du côté Ask (le nombre de niveaux de prix des ordres de vente) ;
- Le montant total ($prix \cdot volume$) du côté Bid ;
- Le montant total ($prix \cdot volume$) du côté Ask.

1.3 Simulation d'order flows

Le module `OrderBookSimulator.cpp` permet de lancer une simulation pour une durée (en secondes) choisie par l'utilisateur à l'appel de la fonction. Lors de cette simulation, toutes les 20 secondes, un nouvel ordre aléatoire est généré pour chaque actif et les carnets d'ordres sont mis à jour en conséquence.

Les ordres générés peuvent être soit des ordres de marché, soit des ordres limite. Un ordre de marché est une instruction d'achat ou de vente d'un actif au meilleur prix disponible immédiatement dans le carnet d'ordres. Il garantit l'exécution mais pas le prix exact car si le volume au meilleur prix n'est pas suffisant pour satisfaire à l'ordre, celui-ci remonte le carnet d'ordres jusqu'à ce que tout son volume soit exécuté. Un ordre limite est quant à lui une instruction d'achat ou de vente à un prix spécifique ou meilleur. L'ordre n'est exécuté que si le marché atteint ce prix.

La génération des autres suit la logique suivante :

- Le volume du nouvel ordre est tiré d'une loi uniforme continue sur le support $[0,1; 1000]$;

— L'ordre a 50% de chance d'être un ordre de marché et 50% de chance d'être un ordre limite ;
— L'odre a 50% de chance d'être un ordre de vente et 50% de chance d'être un ordre d'achat ;
— S'il s'agit d'un ordre limite, le prix de l'ordre est tiré d'une loi normale centrée au prix mid avec une variance de 3.

Petite subtilité : si l'ordre généré est un ordre limite d'achat (de vente) dont le prix est supérieur (inférieur) au meilleur prix de vente (d'achat), l'ordre est automatiquement convertit en ordre de marché comme le font les vrais exchanges. Ainsi dans les faits, la simulation génèrera plus souvent des ordres de marché que des ordres limite.

Le traitement des nouveaux ordres et la mise à jour des carnets d'ordre se fait essentiellement par le biais de la classe `OrderBookManager` du module `OrderBookManager.cpp` qui possède une méthode dédiée . Une fois le carnet à jour, la méthode affiche les carnets accompagnés de leurs nouvelles statistiques.

1.4 Gestion du portefeuille (Portfolio)

Lorsqu'un ordre d'achat est exécuté, la classe `Portfolio` calcule un prix moyen de revient pour l'actif concerné selon la formule suivante :

$$P_{\text{moyen}} = \frac{\left(Q_{\text{ancienne}} \times P_{\text{ancienne}}\right) + \left(Q_{\text{nouvelle}} \times P_{\text{nouvelle}}\right)}{Q_{\text{ancienne}} + Q_{\text{nouvelle}}}$$

où Q_{ancienne} et P_{ancienne} représentent respectivement la quantité et le prix moyen avant la nouvelle transaction, tandis que Q_{nouvelle} et P_{nouvelle} sont la quantité et le prix du nouvel achat. Cette approche par prix moyen s'avère idéale dans le contexte d'un carnet d'ordres, chaque exécution pouvant intervenir à un prix différent tout en devant être agrégée dans une vision globale de la position.
Lors d'une vente, la méthode `updateSell` soustrait la position vendue et calcule le *profit ou la perte réalisé(e)* en comparant le prix d'exécution P_{exec} au prix moyen P_{moyen} :

$$\text{PnL}_{\text{ralis}} = \left(P_{\text{exec}} - P_{\text{moyen}}\right) \times Q_{\text{vendue}}$$

Cette différence est consignée dans `pnlHistory` et contribue directement au *PnL global*. Parallèlement, un *PnL non réalisé* peut être estimé si la quantité résiduelle reste en portefeuille, en comparant le prix moyen au dernier prix constaté sur le carnet (P_{courant}) :

$$\text{PnL}_{\text{non réalisé}} = \left(P_{\text{courant}} - P_{\text{moyen}}\right) \times Q_{\text{restante}}$$

Cette portion non réalisée demeure sujette aux fluctuations du marché, d'où l'intérêt de maintenir un prix moyen à jour. De plus, la méthode `updateBuy` met en cohérence la quantité globale de l'actif et son prix moyen, tandis qu'un historique interne (`tradeHistory`) enregistre chaque transaction. Les enregistrements complets (achats et ventes) peuvent être exportés via `logTradesToCSV`, permettant ainsi de retracer toute l'activité de l'utilisateur.

Enfin, `Portfolio` met à disposition diverses fonctions d'affichage, notamment `printHoldings`, qui donne un aperçu de la quantité et du prix moyen pour chaque actif détenu, `printGlobalPnL` pour visualiser la somme totale de profits ou pertes réalisés, et `printAssetPerformance` afin de déterminer la rentabilité actuelle de chaque titre (en incluant, si besoin, la partie non réalisée). Ces mises à jour reposent également sur la classe `TransactionResolver`, qui déclenche la séquence *banque-portefeuille* : on y vérifie d'abord que le solde est suffisant pour un achat (ou

qu'une vente est réalisable), puis on actualise à la fois le solde bancaire et la position dans `Portfolio`.

1.5 Gestion du compte bancaire (BankAccount)

La classe `BankAccount` est dédiée à la gestion du solde disponible pour l'achat et à la réception des produits de vente lors de chaque transaction. Quand une opération a lieu, la fonction `deposit` enregistre un apport de liquidités (par exemple, à la suite d'une vente), alors que la fonction `withdraw` contrôle et effectue le débit nécessaire pour couvrir un achat. Chacune de ces opérations est tracée dans un historique (`transactionHistory`) propre au compte, qui peut ensuite être exporté via `logTransactionsToCSV` si l'utilisateur souhaite garder une preuve des versements et retraits successifs. `BankAccount` agit donc comme un garde-fou sur l'ensemble du processus, interdisant toute tentative d'achat qui excéderait le solde courant. Lorsqu'une vente se produit, elle crédite en retour le montant équivalent au prix multiplié par la quantité vendue. Cette relation bidirectionnelle avec `Portfolio` est pilotée par `TransactionResolver`, qui s'assure que le solde bancaire et la position en titres soient tous deux parfaitement synchronisés après chaque transaction.

1.6 Résolution des transactions (TransactionResolver)

La classe `TransactionResolver` se charge d'orchestrer chaque exécution d'ordre en communiquant avec `BankAccount` et `Portfolio`. Lors d'un ordre d'achat, elle appelle la méthode `withdraw` de `BankAccount` pour vérifier la disponibilité des fonds, puis sollicite `Portfolio` (`updateBuy`) afin que la position de l'actif concerné soit mise à jour. À l'inverse, lors d'une vente, elle commence par avertir `Portfolio` (`updateSell`) pour diminuer la quantité de titres détenus et calculer la plus-value ou moins-value éventuelle, avant d'appeler la méthode `deposit` de `BankAccount` pour créditer le montant de la vente. De cette manière, `TransactionResolver` assure en permanence l'alignement des données entre le solde bancaire et la quantité d'actifs détenus, tout en préservant un historique de chaque transaction et de son horodatage. Ce pont logique est essentiel pour la robustesse du projet, puisqu'il garantit qu'aucune inconsistance n'apparaisse entre l'argent réellement disponible et la détention effective d'actifs au sein du portefeuille.

1.7 Saisie des ordres (OrderInputHandler)

La classe `OrderInputHandler` permet à l'utilisateur de formuler ses instructions d'achat ou de vente de manière interactive. Elle regroupe des méthodes facilitant la récupération des informations nécessaires, telles que le type d'ordre (`BUY` ou `SELL`), l'actif visé et le prix auquel l'utilisateur souhaite négocier. Elle valide également les saisies pour éviter les erreurs évidentes, comme la mention d'un titre qui n'existe pas ou un format de prix incorrect. Dans le cadre du projet, ce module est fondamental car il rend la simulation plus réaliste en impliquant l'utilisateur directement dans la prise de décisions. Il constitue, en somme, l'interface de saisie par laquelle tout ordre manuel transite avant d'être transmis au reste du système, qu'il s'agisse du carnet d'ordres ou du portefeuille.

2 Défis rencontrés

2.1 Choix des structures de données pour le carnet d'ordres

L'une des difficultés majeures a été de sélectionner la structure de données la plus adaptée pour stocker et gérer les ordres du carnet. Nous avons initialement envisagé plusieurs options comme `std::vector` ou `std::list`, mais celles-ci présentaient des limitations :

- Utiliser des Vecteurs (`std::vector`) nécessitait un tri explicite après chaque insertion, ce qui augmentait la complexité ;
- Utiliser les istes (`std::list`) permettait des insertions rapides mais rendait le tri et la recherche de valeurs inefficaces ;

Nous avons finalement opté pour des `std::map` imbriquées :

- `map<string, map<double, OrderBookEntry, greater<>> bidBooks;` ;
- `map<string, map<double, OrderBookEntry» askBooks;`.

Cette approche présente plusieurs avantages :

- La première `map<string, ...>` permet de stocker un carnet distinct par actif.
- Cet structure permet également de trier plus facilement le carnet :
  - `bidBooks` trie automatiquement les prix en **ordre décroissant** (`greater<>`), facilitant l'accès au meilleur bid.
  - `askBooks` trie les prix en **ordre croissant**, garantissant un accès immédiat au meilleur ask.

Ce choix de structure garantit une gestion optimale des carnets d'ordres de plusieurs actifs.

2.2 Génération de données aléatoires

Une autre difficulté a été de comprendre la génération de nombres aléatoires en C++, bien plus complexe qu'en Python où une simple fonction suffit. En C++, il faut plusieurs éléments pour obtenir un tirage selon une distribution normale :

- il faut utiliser `random_device rd;` pour générer une source d'entropie matérielle pour initialiser le générateur pseudo-aléatoire et ainsi assurer une bonne qualité d'aléa ;
- il faut se servir de `mt19937 gen(rd());` pour initialiser un générateur pseudo-aléatoire qui assure une meilleure qualité et une période bien plus longue que `rand()`, réduisant ainsi les corrélations entre les nombres générés.

2.3 Optimisation de la mémoire

Nous avons également réalisé à quel point en C++ la gestion explicite de la mémoire est cruciale. Une difficulté rencontrée a été par exemple de comprendre qu'il vallait mieux passer des paramètres par référence constante (`const &`) plutôt que directement une valeur. Cela permet en effet d'éviter une copie inutile et ainsi améliorer la performance. Cette optimisation est particulièrement utile lorsqu'on manipule des objets volumineux comme des carnets d'ordres très détaillés et profonds sur une simulation de longue durée.

2.4 Simultanéité de la simulation et des ordres clients

Une difficulté majeure que nous avons pu recontrer est d'une part la volonté de garder une simulation de carnet d'ordre en continu ainsi que conserver la possibilité d'avoir un utilisateur qui puisse interagir directement avec le marché. En effet, la simulation fonctionnait telle qu'il était impossible d'avoir un message fixe et un ordre du client qui impacterait le marché ponctuellement. Nous avons donc opté pour une solution qui n'est pas optimale mais fonctionnelle :

le threading.

Avec le threading, on peut avoir d'une part la simulation en continu et l'ordre ponctuel du client. Cependant, cela a provoqué un nouveau problème : l'enchevêtrement des messages lorsque des ordres aléatoires générés par la simulation coexistent avec les ordres manuels saisis par l'utilisateur. Cela perturbe grandement la lisibilité et provoquait une soupe de lettres que nous avons également pu régler d'une part en modifiant la manière dont étaient générés les ordres pour ne pas qu'ils se chevauchent et un affichage grâce à mutex qui permet de bloquer un affichage avant d'en avoir un suivant.

3 Améliorations à apporter

3.1 Analyse individuelle des ordres

Bien que les transactions d'origine contiennent des informations détaillées, telles que l'horodatage, ces données sont perdues lors de l'agrégation des ordres par niveaux de prix. Le carnet fournit de ce fait uniquement une vue globale du marché mais ne permet pas d'accéder à une vue locale détaillée des ordres individuels.

Une amélioration intéressante consisterait donc à conserver en mémoire la liste des ordres à chaque niveau de prix. Cela permettrait :

- De déterminer l'ordre de priorité des ordres en attente d'exécution (principe FIFO : First In, First Out).
- De suivre le temps moyen d'attente d'un ordre avant exécution.
- De mieux analyser la dynamique du carnet d'ordres, en observant l'évolution des volumes et le comportement des participants.

Cette amélioration offrirait ainsi une granularité plus fine et permettrait d'effectuer des analyses avancées sur la dynamique individuelle des ordres.

3.2 Etendre la diversité des typologies d'ordres simulés

Dans l'état actuel, la simulation génère uniquement des ordres limites et des ordres de marché. Pour enrichir la simulation et la rendre plus réaliste, il serait pertinent d'introduire d'autres types d'ordres, notamment :

- Des ordres d'annulation qui permettent de supprimer ou de réduire un ordre limite en attente d'exécution. Cela influerait par conséquent sur la liquidité et la profondeur du carnet d'ordres ;
- Des Ordres "Fill or Kill" (FOK) qui doivent être exécutés immédiatement en totalité, sinon ils sont annulés. Leur introduction permettrait d'étudier l'impact des ordres agressifs sur la structure du carnet ;
- Des Ordres "Immediate or Cancel" (IOC) qui exécutent la partie disponible immédiatement et annulent le reste. Cela simulerait des comportements de trading plus flexibles.
- Des Ordres iceberg qui affichent seulement une partie du volume total, le reste étant caché. Cette amélioration introduirait une notion de profondeur cachée du marché.

L'ajout de ces nouveaux ordres apporterait une dynamique plus réaliste au carnet et permettrait d'analyser plus finement les comportements des participants.

3.3 Exploiter les modifiers pour optimiser la mémoire

L'optimisation mémoire est un enjeu important dans tout programme de simulation intensif. Une amélioration possible serait d'utiliser des modificateurs de type adaptés pour réduire la taille des variables :

- Utiliser `short` au lieu de `int` pour les valeurs qui ne dépasseront jamais 32 767 comme les ID des ordres ;
- Employer `unsigned` lorsque l'on sait qu'une variable ne peut prendre que des valeurs positives (comme les volumes des ordres ou volumes à chaque niveau de prix des carnets) ;
- Réduire l'usage des `double` lorsque des `float` suffisent, afin de diminuer l'empreinte mémoire sans perte significative de précision.

Ces ajustements permettraient de diminuer l'empreinte mémoire et d'optimiser la gestion des ressources, rendant ainsi la simulation plus efficace.

3.4 Réprésentation graphique dynamique du carnet d'ordres

À l'heure actuelle, les carnets d'ordres sont uniquement affichés sous forme de tableaux statiques. Une amélioration majeure consisterait à intégrer une visualisation graphique dynamique, similaire au graphique ci-dessous :

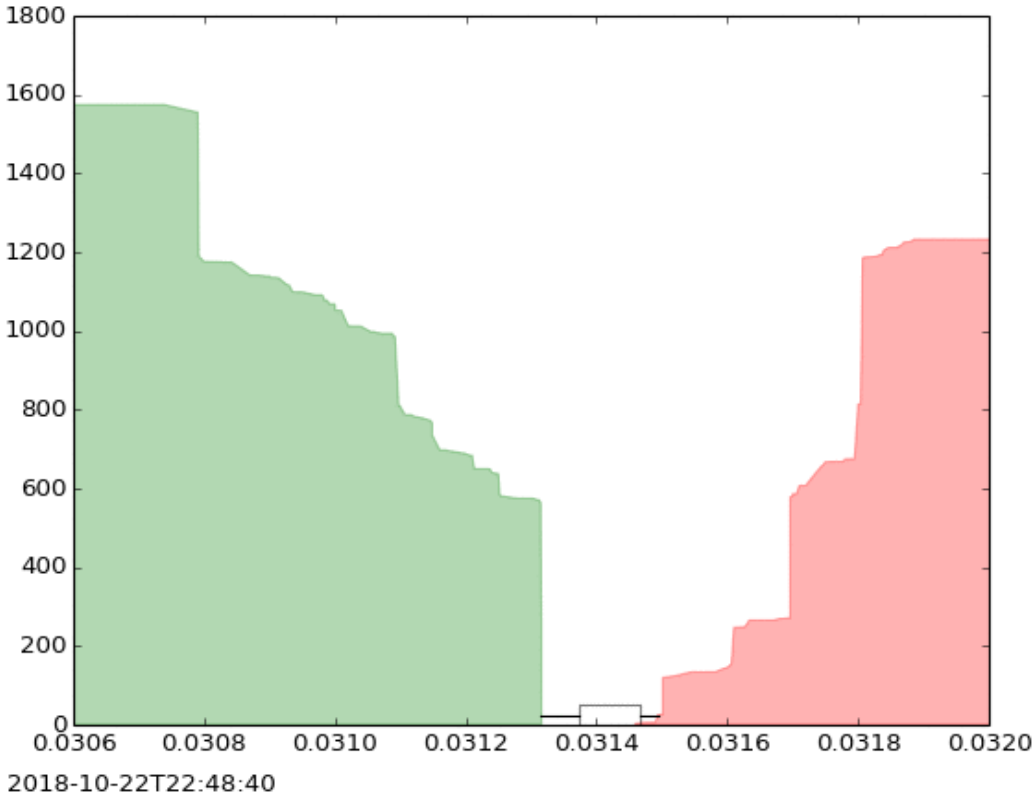

FIGURE 1 – Exemple de représentation graphique d'un carnet d'ordres

Cette visualisation permettrait :

- De voir l'évolution en temps réel des carnets d'ordres ;
- D'identifier immédiatement les déséquilibres entre l'offre et la demande ;
- De mieux analyser l'impact des nouveaux ordres sur la liquidité du marché.

Une implémentation possible serait d'utiliser des bibliothèques graphiques comme `Qt` ou `SFML`.

3.5 Séparation des interfaces

Une dernière amélioration possible serait de scinder l'affichage en deux interfaces distinctes :
— Une pour le « carnet d'ordres » (dynamique des bids et asks).
— Une pour le solde bancaire et le portefeuille, avec passage d'ordres.

Une architecture client/serveur avec sockets ou interface graphique améliorerait la lisibilité et la flexibilité et permettrait ainsi un confort supplémentaire pour l'utilisateur.