

PROJET C++

STOCK EXCHANGE SIMULATOR

Simulateur de marché financier

13 février 2015

Amin Redwane

Beji Yanis

Vassaux Florent

Ecole Nationale de la Statistique et de l'Administration Economique

Table des matières

1	Présentation de la structure	4
1.1	Les produits financiers	4
1.1.1	Les actions : l'objet Stock	4
1.1.2	Les options : l'objet Option	4
1.1.3	Les obligations : l'objet Bond	5
1.2	Les Agents	6
1.2.1	Le portefeuille : l'objet Portfolio	6
1.2.2	Les ordres de bourse : l'objet Order	6
1.2.3	Les agents : l'objet Agent	6
1.3	Les infrastructures :	8
1.3.1	Les entreprises : l'objet Company	8
1.3.2	Le Trésor : l'objet Treasure	8
1.3.3	La bourse : l'objet Stock Exchange	9
1.4	Les autres supports :	10
1.4.1	L'objet Database :	10
1.4.2	Les statistiques : l'objet Stat	11
1.5	Schéma récapitulatif de la structure	12
2	Les comportements des agents :	13
2.1	Le marché actions :	13
2.1.1	L'agent Random	13
2.1.2	L'agent Company :	13
2.1.3	L'agent Sharpe	14
2.2	Le marché options :	16
2.2.1	L'agent Tree	16
2.2.2	L'agent Black-Scholes	17
2.3	Le marché obligataire :	18
3	Limites du projet et améliorations futures possibles	19

Introduction

Le but est ici de créer un simulateur de marché financier où des agents pourront s'échanger des actions, des obligations et des options. Pour cela, nous allons créer des modèles économiques de chacun des acteurs jouant un rôle sur les marchés (ex : le Stock Exchange, les agents, l'Etat, les entreprises, etc.) et les interconnecter entre eux. Alors, nous pourrons observer si ce marché simulé recrée d'une manière assez fidèle un véritable marché financier.

Le premier chapitre présente la structure du programme informatique tandis que la deuxième s'attarde sur le comportement des agents. Enfin, une troisième partie, fera une critique générale de l'ensemble du projet après que nous ayons pu tester les premiers résultats de ce simulateur.

En lançant le programme, l'utilisateur choisit le type d'agent qu'il veut voir agir. Puis un panneau affiche toutes les informations de l'agent 0 du programme pour une durée de 100 tours. (Attention certains personnages peuvent être passif sur plusieurs tours).

Chapitre 1

Présentation de la structure

1.1 Les produits financiers

1.1.1 Les actions : l'objet Stock

L'objet Stock comporte toutes les informations sur le cours de l'action que l'on peut observer sur le marché. Cet objet ne sera donc que détenu par la bourse¹ ("StockExchange").

Voici les attributs que comporte une action :

1. *int _compagnie_id* : le numéro de la compagnie qu'elle représente,
2. *vector<double> _stock_values* : l'historique de toutes les valeurs de l'action,
3. *vector<double> _median_buy* : l'historique des prix médians d'achat,
4. *vector<double> _median_sell* : l'historique des prix médians de vente,
5. *int _number_of_stocks* : le nombre d'actions de la même compagnie mises en circulation.

Il y a une seule méthode importante ici :

`NewPrice(double new_price, double median_buy, double median_sell)`

Elle consiste à apporter les trois nouveaux prix mis en argument à la fin de chaque historique.

1.1.2 Les options : l'objet Option

L'objet Option est un contrat détenu par le vendeur du susdit contrat et auquel les autres agents peuvent avoir accès par l'intermédiaire de la bourse ("Stock Exchange"). Ces produits ne sont pas revendables par l'acheteur. Les agents programmés considéreront que ce sont des options européennes alors qu'en réalité elles seront américaines. Les évaluations faites par les agents seront donc d'un "style" européen. Si le programme permet à un joueur d'être un agent, celui-ci n'aura aucun problème pour exécuter ses options étant donné qu'il n'aura pas à l'exercer dans un intervalle de temps très réduit.

Voici les attributs que comporte une option :

1. *int _id* : le numéro de référence de l'option par rapport au vendeur,
2. *int _id_seller* : le numéro de référence du vendeur de l'option,
3. *int _id_buyer* : le numéro de référence de l'acheteur de l'option,
4. *int _maturity* : la période de temps entre le début du contrat optionnel et la maturité de ce produit financier,
5. *int _time* : date de l'achat de l'option et donc du début du contrat.

1. La bourse est ici une entreprise monopolistique qui sert d'intermédiaire aux agents

6. *int _id_stock* : le numéro de référence de l'action sur laquelle l'option porte,
7. *double _strike* : le strike de l'option,
8. *bool _is_call* : *True* si l'option est un call (resp. *False* si c'est un put),
9. *int _number_of_stock* : le nombre d'actions qui seront transférées si l'exercice a eu lieu.
10. *double _price* : prix fixé par le vendeur de l'option,
11. *int _offer* : le prix donné par l'acheteur et qui consiste avant que l'option ne soit vendue à l'enchère la plus haute si deux agents veulent l'acheter à la même date (on a forcément $_offer \geq _price$),
12. *bool _exercice* : indique (par la valeur *True* que l'acheteur veut exercer son option (la valeur par défaut étant *False*)),

Les méthodes à noter ici sont :

— *NewBuyer(int id, double offer)* :

Ces arguments remplacent les informations de l'acheteur de l'option si une (meilleure) offre a été faite.

— *Exercice()* :

L'acheteur indique son intention d'exercer l'option, l'attribut *_exercice* prend la valeur *True*.

1.1.3 Les obligations : l'objet Bond

L'objet Bond est une obligation. Ici, seul l'Etat peut en émettre par l'intermédiaire de son Trésor ("Treasure"). Les Bond sont des obligations zéro-coupons, i.e. elles ne versent aucun intérêt avant la maturité de l'obligation. Le nominal de l'obligation sera le prix de celle-ci, i.e. le montant versé par l'agent à la création de ce titre. Le taux du titre représente les intérêts versés par le Trésor à une maturité de référence (constante pendant l'exécution du programme).

Voici les attributs d'un Bond :

1. *int _time* : date de création de l'obligation,
2. *int _maturity* : période qui va de la création de l'obligation à la maturité de celle-ci,
3. *double _rate* : taux,
4. *double _price* : argent prêté à l'Etat par l'agent,
5. *bool _sold* : indique si l'agent a vendu son titre à la bourse.

La méthode *Sold()* permet au propriétaire d'indiquer à la bourse qu'il souhaite vendre ce contrat, en faisant passer la variable *_sold* en vrai.

Calculs :

Pour un prix P , un taux R , une maturité T_m et une maturité de référence T_r , on a :

$$\begin{cases} I = P \times (1 + R)^T \\ T = \frac{T_m}{T_r} \end{cases}$$

Remarque : le taux R s'exprime sur une période T_r . Si le propriétaire vend ce titre à la bourse au prix P_0 , celle-ci devra recevoir à maturité un flux monétaire correspondant au résultat d'un placement au taux r d'aujourd'hui sur une durée $(T_m - t)$ où t est le temps entre le rachat et la maturité du titre.

$$\begin{aligned} P \times (1 + R)^T &= P_0 * (1 + r)^{\frac{T_m - t}{T_m} T} \\ \Leftrightarrow P_0 &= P \frac{(1 + R)^T}{(1 + r)^{\frac{T_m - t}{T_m} T}} \end{aligned}$$

1.2 Les Agents

1.2.1 Le portefeuille : l'objet Portfolio

Contrairement à ce que laisse supposer son nom, l'objet Portefeuille n'est pas un portefeuille, celui-ci est en fait un vecteur d'objets Portfolio. L'objet Portfolio comporte principalement le nombre d'actions possédées pour une société donnée ainsi que des données que nous qualifierons de comptables, essentielles dans les stratégies de trading.

Voici les attributs de Portfolio :

1. *int _id_stock* : indice de référence de l'action (de la compagnie),
2. *int _nb_stock* : nombre d'actions¹ détenues par l'agent,
3. *double _medium_price* : prix moyen auquel le vendeur a acheté ces titres (resp. vendu s'il effectue des ventes à découvert),
4. *double _actual_price* : valeur de l'action¹ donnée par le marché,
5. *double _dividends* : dénombre la quantité totale de dividendes perçue par l'agent pour ces actions¹,
6. *double _earnings* : totale des plus-values et des moins-values obtenues par l'achat et la vente d'actions¹.

La méthode importante ici est :

SetPortfolio(int delta, double price) qui modifie les données de l'objet Portfolio en sachant que je viens d'acheter Δ actions (si Δ est négatif, je vends ces actions) au prix *price*.

1.2.2 Les ordres de bourse : l'objet Order

L'objet Order va servir de support aux ordres de bourse que les agents vont "envoyer" à la bourse ("StockExchange").

Voici les attributs de l'objet Order :

1. *int _id* : numéro de référence de l'ordre pour l'agent qui le crée,
2. *int _time* : date à laquelle l'ordre a été émis,
3. *int _stock_id* : numéro de référence de l'action sur lequel l'ordre porte,
4. *int _agent* : numéro de référence de l'agent émettant l'ordre,
5. *bool _is_buy* : cette variable est vraie si c'est un ordre d'achat (resp. fausse pour la vente),
6. *int _number_of_shares* : le nombre d'actions que l'émetteur veut échanger,
7. *double _price* : prix unitaire de l'action échangée.

Dans le cas d'exécution partielle de l'ordre, on le mettra à jour avec la méthode :

void SetNumber(int n) : qui remplace le *_number_of_shares* actuel par *n*.

1.2.3 Les agents : l'objet Agent

L'objet Agent est ici une classe mère de tous les types d'agents différents qui vont agir sur les marchés. Cette classe virtuelle a donc été créée pour pouvoir lister l'ensemble des actions (ou méthodes que pourront utiliser les agents).

1. le mot "actions" fait référence aux actions émises par la compagnie *_id_stock*.

Voici les attributs de l'objet Agent :

1. *int* _id : indice de référence de l'agent,
2. *double* _bank : indice de référence de la banque auquel il appartient,
3. *double* _cash : disponibilités de l'agent,
4. *vector*<*Portfolio**> _portfolio : portefeuille de l'agent qui possède un objet Portfolio par type d'action et donc ce tableau a une taille égale au nombre d'actions de types différents qu'il existe sur le marché, ce nombre ne sera pas modifié pendant l'exécution du programme,
5. *vector*<*Order*> _orders : ordres passés par l'agent,
6. *vector*<*Bond*> _bonds : obligations détenues par l'agent,
7. *vector*<*Option*> _option_store : regroupe les options que l'agent met en vente et les options déjà vendues,
8. *vector*<*Option**> _options : regroupe les options que l'agent a achetées.

Plusieurs méthodes sont ici à considérer, elles peuvent être réparties en plusieurs familles suivant le produit financier qu'elle traite.

Les actions : (par les objets Order)

- *void* MakeAnOrder(*int* time, *int* id_stock, *bool* is_buy, *int* number_of_stocks, *double* price) :
Enregistre un ordre de bourse dans la variable _orders de l'agent. L'ordre résumera : à la date time, l'agent ici présent veut acheter (si is_buy est vrai, sinon il veut vendre) number_of_stocks actions de référence id_stock au prix price,
- *void* EraseOrder(*int* order_id) :
Efface du vecteur _orders l'ordre d'identifiant order_id,
- *void* EraseAllOrder() :
Efface tous les ordres du vecteurs _orders

Les actions : (par les objets Portfolio)

- *void* SetPortfolio(*int* stock_id, *int* number, *double* price, *bool* isbuy) :
Active la fonction SetPortfolio($(2\mathbb{I}_{is_buy} - 1) \times number, price$) à l'objet _portfolio[stock_id]

Les obligations :

- *void* MakeBond(*int* time, *int* maturity, *Treasure**¹ treasure, *double* price) : créer une obligation de maturité maturity, à l'instant time, la variable price est égale au montant du placement.
- *void* SellBond(*int* bond_id) : Active la méthode Sold du i-ème élément du vecteur des obligations (où i = bond_id).
- *void* EraseBond(*int* i) : efface le i-ème élément du vecteur _bonds, cette méthode ne sera utilisée qu'après qu'une obligation ne soit vendue à la bourse ou que celle-ci ne soit arrivée à maturité et rémunérée.

Les options :

- *void* SellOption(*int* id_stock, *int* maturity, *double* strike, *double* price, *bool* is_call) : met en vente dans l'option store de l'agent l'option décrite par les paramètres de la méthode,
- *void* BuyOption(*int* id, *int* seller, *double* offer, *vector*<*Option**> options) : demande d'achat d'une option mise en vente par un autre agent, l'agent n'obtient l'action que s'il fait l'offre la plus onéreuse,
- *void* AddOption(*Option** poption) : si l'agent réussit à acheter une option, grâce à cette méthode, celle-ci va se retrouver dans son vecteur _options,

1. Treasure est l'objet représentant le Trésor de l'Etat et c'est lui qui émettra les obligations et déterminera leur taux.

- `void EraseOption(int id_option)` : efface une option de son `_option_store`,
- `void EraseBoughtOption(int id_option, int id_seller)` : efface une des options que l'agent a acheté (soit parce qu'elle a expirée ou qu'elle a été exercée),
- `void ChangePriceOption(int _id, double price)` : modifie le prix d'une option déjà mise en vente précédemment mais n'ayant reçu aucune demande d'achat,
- `void ExerciceOption(int i)` : active la méthode de la i-ème option de son vecteur `_options`.

La fonction Do : Cette fonction est définie de la manière suivante :

virtual void Do(int time, vector<Stock> stocks, vector<Option> options, Treasure* tres, vector<Company*> companies, vector<Stat*> stats).* Cette fonction comporte en arguments tous les objets représentant le marché, les premiers représentent la date, les actions sur le marché, les options mises en vente, l'état ou le trésor public (émetteur d'obligations), les entreprises cotées et un objet Stat comportant des statistiques. Tous les objets non encore décrits le seront ultérieurement. Cette fonction sera appelée à chaque tour pour chaque agent. Tous les agents seront définis dans des sous-classes de l'objet Agent et auront leur propre procédé au sein de cette fonction *Do*. On utilise alors la virtualité de cette fonction commune à tous les agents pour pouvoir accéder simplement à leur spécificité.

1.3 Les infrastructures :

1.3.1 Les entreprises : l'objet Company

La modélisation des entreprises est une modélisation originale créée dans le seul but de modéliser des flux monétaires de dividendes réalistes, le modèle utilisé sera expliqué dans le document Firm Modelling. Je vais tout de même le résumer rapidement.

L'entreprise verse un dividende tous les `_div_period`. Le prochain versement aura lieu à la date `_paytime`. A chaque tour on simule une loi normale de paramètre `_mu` et `_sigma`. Entre deux versements, on somme ces lois aléatoires dans le vecteur `_results`. Arrivé à une date `paytime` on va répartir ce "résultat" entre l'investissement dans l'entreprise par une accumulation de son cash, l'épargne de l'entreprise (`_savings`) et les dividendes versés aux actionnaires.

Pour éviter (en espérance) que la valeur de l'entreprise ne baisse (i.e. que son `_cash` baisse), elle va d'abord dépenser une somme calculée dans le vecteur `_objectives` de son résultat dans son capital. Si l'entreprise n'a pas encore dépensé tout son résultat, elle va regarder son épargne. Une épargne négative signifie qu'elle a octroyé un prêt, elle va donc payer les intérêts de celui-ci (au taux sans risque) pour ne pas que sa dette augmente. Si, après ces deux observations, l'entreprise a encore de l'argent, elle va commencer à payer un dividende. En effet, nous considérons que le dividende mis en définition de l'entreprise est équivalent à un dividende minimum à versé dans une bonne conjoncture.

Si après tous ces premiers paiements, l'entreprise possède encore de l'argent, elle va le distribuer de manière à maximiser sa fonction d'utilité dépendant de paramètres dits comportementaux :

1.3.2 Le Trésor : l'objet Treasure

Le Trésor est ici le représentant de l'Etat qui doit émettre des obligations suivant les recommandations de l'Etat et les demandes du marché. Le Trésor ne fait qu'une chose : déterminer le nouveau rendement sur une période donnée par le paramètre `_maturity`, maturité de référence dans lequel tous les taux sont exprimés. Les agents choisissent ensuite d'investir un certain montant à ce nouveau taux r_0 sur une période de leur choix T_m . Le taux d'intérêt de leur obligation sera donc :

$$r = (1 + r_0)^{\frac{T_m}{\text{period}}} - 1$$

Remarque : le paramètre `_rate` de l'obligation est toujours considéré sur la période de référence `_periode`. On considère que l'Etat impose certains paramètres :

- `_maturity = Tr_aim_demand : objectif de demande totale obtenir par tour`
- `_aim_rate` : taux fixé si l'on a une demande constante dans le temps à `_aim_demand`.
- `_high` : le taux le plus élevé que le trésor peut offrir
- `_low` : le taux le plus bas que le trésor peut offrir.

L'analyse du trésor se fera sur les vecteurs de double `_rate` et `_demand` contenant respectivement l'historique des taux du trésor et l'historique des demandes totales par tour. Plus particulièrement, on récupère d la moyenne mobile des demandes totales sur `_period` tours et r le dernier taux du trésor. En notant d^* et r^* `_aim_demand` et `_aim_rate` on obtient le nouveau taux par les formules :

$$\Delta D = \frac{\text{aim_demand}}{d} - 1$$

$$\Delta r = \frac{\text{aim_rate}}{r} - 1$$

$$x = \frac{|\Delta D| \times \Delta D + |\Delta r| \times \Delta r}{|\Delta D| + |\Delta r|}$$

$$r_{new} = r \times (1 + x)$$

Les coefficients delta s'interprètent de la manière suivante : si d dépasse de 1% la valeur `_aim_demand`, alors $\Delta \approx -1\%$.

Les deux informations Δ peuvent être interprétées comme des recommandations : ne pas trop s'éloigner du taux objectif et augmenter de ΔD le taux r en supposant que la demande va alors diminuer de ΔD .// On obtient enfin le nouveau taux r_{new} en considérant la moyenne pondérée de ces deltas, eux-mêmes pondérés par l'amplitude de leur différence avec les objectifs ($|\Delta|$).// Ceci reste, bien sûr, une façon très simplifiée de fixer les taux sans risque mais elle reste dans les faits assez efficace.

1.3.3 La bourse : l'objet Stock Exchange

Le Stock Exchange est ici une entreprise chargée de faire communiquer les informations de chaque agent au marché et inversement. Cette entreprise se fait des marges sur les actions et les options en empochant la différence entre le prix auquel le vendeur veut acheter et le prix auquel l'acheteur veut vendre.

Son premier paramètre est très important : `_time`, cette variable correspond au numéro du tour en train de s'écouler. La majorité des transactions en particulier les options et les obligations seront dépendantes de cette information.

Le Stock Exchange est l'objet qui détiendra les objets Stocks auxquels les agents auront accès dans un vecteur `_stocks`.

Ensuite, afin de faire toutes les transactions possibles, il va parcourir les produits mis en vente par chaque agent et les inscrire dans ses vecteurs `_orders` (pour les ordres de bourse), `_options` (pour les options mises en vente) et `_options_sold` (pour les options qui ont été vendues). Enfin, parce que les agents peuvent revendre leurs obligations à cette entreprise, elle va conserver toutes ces obligations revendues dans son vecteur `_bonds`.

Parmi les méthodes, les méthodes `AddNewOrders` et `AddNewOptions` permettent de "scanner" les informations d'un ou des agents. Pour le marché actions, la fonction `UpdatePrices` permet de calculer les nouveaux prix de chaque action. Elle va connecter le prix d'achat le plus important avec le prix de vente le plus petit puis s'intéresser aux ordres restants. La fonction continue de s'exécuter jusqu'à ce que le prix d'achat soit inférieur au prix de vente. On regarde alors la moyenne des deux derniers prix relevés et on obtient le nouveau prix d'équilibre. Cette méthode permet en effet de croiser les courbes d'offre et de demande du marché pour ainsi déterminer le prix d'équilibre. On calculera aussi les prix médians d'achat et de vente de chaque action.

Puis dans un second temps on utilise la fonction `UpdateOrders` qui réunit par paire les acheteurs et les vendeurs puis la fonction `ExecuteOrders` permettant de réaliser la transaction.

De manière analogue, la fonction `UpdateOptions` parcourt toutes les options dans le vecteur `_options` pour voir si elles ont été vendues et faire ainsi la transaction. Ensuite, elle regarde dans les options déjà vendues dans `_sold_options` pour voir si elles ont été exercées par leur acheteurs et effectuer, dans ce

cas, la transaction associée grâce à la fonction `ExerciseOptions` ou pour effacer les options expirées avec `EraseExpiredOptions`.

Toutes ces fonctions sont ainsi regroupées dans la méthode `NextDay` (celle du `Stock Exchange`). Cette fonction combine le fait de tout effacer dans les vecteurs `_orders`, `_options` et `_sold_options` du `Stock Exchange` tout en récupérant les nouvelles informations des agents, faire les calculs adéquats décrits ci-dessus. Pour finir, cette fonction regarde les obligations souhaitant être vendues par les agents et réalise donc ces échanges.

1.4 Les autres supports :

1.4.1 L'objet Database :

L'objet `Database` est ici l'objet central de cette simulation. L'objectif de cet objet est de regrouper tous les acteurs et toutes les informations de cette simulation. Puis dans un second temps cet objet permet d'initialiser la simulation et de la faire vivre en appelant les mises-à-jour de tous les agents ou infrastructures de ce modèle. Ces deux fonctions ont été mises à l'écart de l'objet dans les documents "Général".

Dans la fonction `Initialisation`, tous les objets sont créés puis regroupés dans un objet `Database` dont le pointeur est donné en sortie. C'est ici que l'on pourra modifier certains paramètres pour changer les comportements des entreprises, des agents, etc. Ensuite les actions de chaque entreprises sont distribuées (et vendues) aléatoirement entre les agents qui partent chacun avec un capital de même montant.

La fonction `NextDay` fait, dans l'ordre :

1. l'update du Trésor en lui faisant calculer son nouveau taux,
2. l'update du `Stock Exchange` qui va faire toutes les transactions possibles entre les agents,
3. la méthode `BondPayment()` de l'objet `Database` qui exécute les obligations arrivées à maturité,
4. l'update de toutes les entreprises cotées,
5. la fonction `Do` de tous les agents.

1.4.2 Les statistiques : l'objet Stat

L'objet Stat permet de regrouper dans une même classe toutes les informations que les agents (certains, pas tous) utilisent dans la gestion de leur portefeuille. L'intérêt de l'objet Stat est donc de calculer puis de stocker une fois pour toutes des informations sur les marchés communes aux agents (telles que les cotations des actions ajustées des dividendes, les rendements individuels des actions sur une période donnée, la matrice de variance-covariance entre les rendements des actions sur une période donnée, etc.), et ainsi de ne pas avoir à répéter tous ces calculs pour chacun des types des agents.

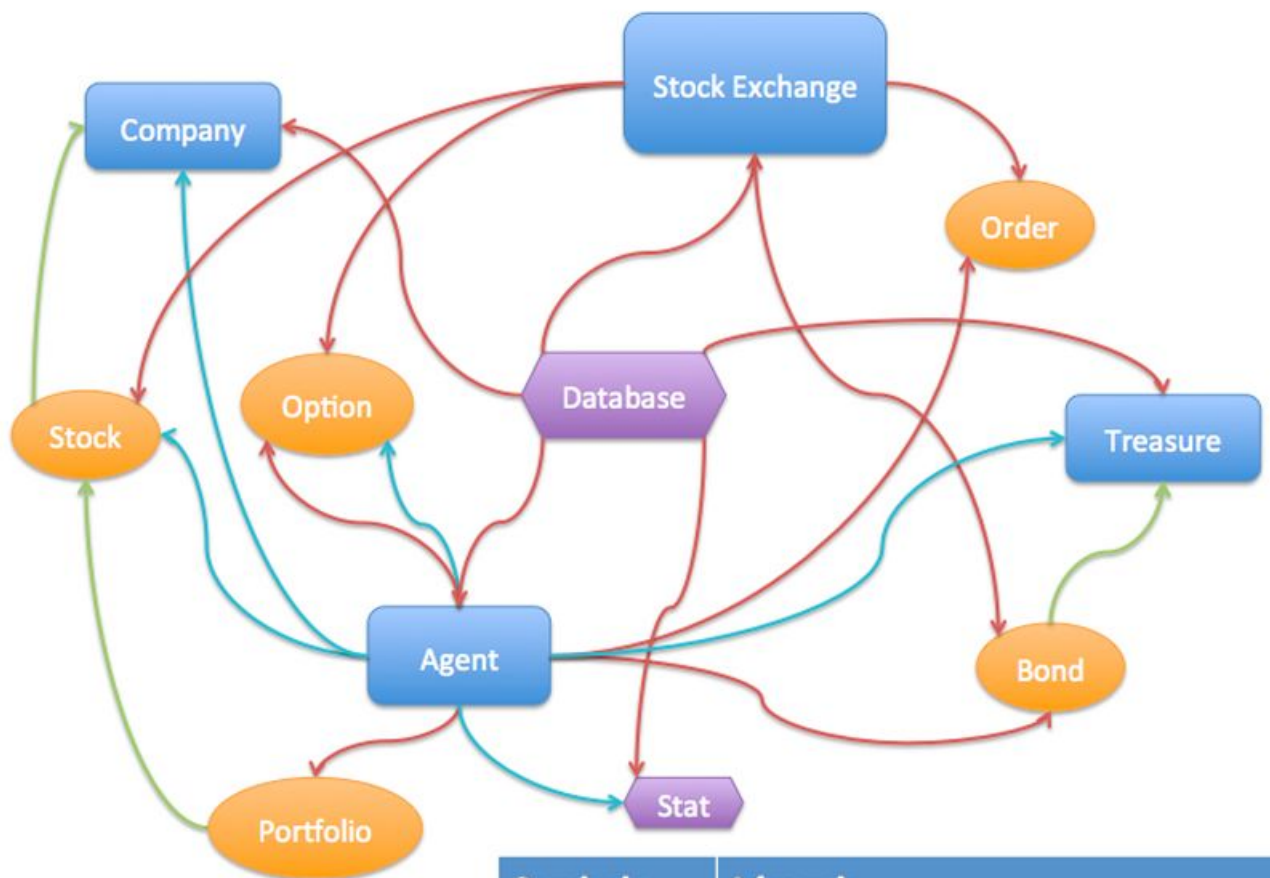
A ce stade d'avancement du projet, l'objet Stat possède en attribut les éléments suivants :







- L'entier naturel *_period*, qui correspond au nombre de tours de cotations passés pris en compte,
- Un vecteur de réels *_yields*, qui correspond au vecteur des rendements de chacune des actions sur *_period* tours passés. Ce vecteur est donc de taille *nb_companies*, et est calculé en moyennant les rendements quotidiens de chaque action sur la période considérée,
- Une matrice de réels *_covariance*, qui correspond à la matrice de variance-covariance des rendements de chacune des actions sur *_period* tours passés. Cette matrice est donc de taille *nb_companies*nb_companies*, et est établie en calculant la covariance empirique entre les rendements quotidiens de chaque couple d'actions sur la période considérée.
- Une matrice de réels *_ajusted_values* de taille *nb_companies*_period* qui regroupe les cotations passées (toujours sur la même période considérée) des actions du marchés, mais ajustées de l'émission éventuelle de dividendes par l'entreprise. En effet, ce calcul de cotation ajustée est important car l'émission de dividendes par une entreprise provoque une baisse mécanique du cours de son action. Cette baisse ne doit surtout pas être prise en compte dans les calculs des rendements quotidiens des actions, qui s'en trouveraient alors faussés.

En ce qui concerne ses méthodes, l'objet Stat possède surtout les trois fonctions *Adjust_stocks*, *Update_yields* et *Update_covar* (à exécuter dans cet ordre), qui permettent respectivement de mettre à jour le calcul des cotations ajustées, le vecteur des rendements, et enfin la matrice de variance-covariance.

Notons enfin que cet objet Stat n'est pas figé. Au contraire, il peut s'ajuster facilement et incorporer de nouveaux éléments utiles aux agents. Par exemple, il est tout à fait possible de rajouter à l'objet stat un attribut *highest_loss* qui correspondrait au numéro de l'action qui aurait le plus fortement chuté sur la période. La force de cet objet est donc de pouvoir incorporer l'ensemble des données et indicateurs relatifs au marché (il en existe des centaines), que les agents n'auront plus qu'à "piocher" lorsqu'ils optimiseront leur portefeuille.

1.5 Schéma récapitulatif de la structure



Symbole	Légende
	Objet de la structure économique
	Objet financier
	Objet de la structure informatique
	L'objet contient l'objet pointé
	L'Agent observe les objets suivants
	L'objet fait référence à un autre objet

Chapitre 2

Les comportements des agents :

2.1 Le marché actions :

Sur le marché des actions, il existe des centaines de manières, de techniques et d'analyses permettant de valoriser un portefeuille. Certains gestionnaires à long-terme se concentrent sur l'analyse économique des entreprises (le cours des actions comme représentant de la performance de l'entreprise associée), d'autres sur l'analyse financière (en prenant en compte l'évolution de différents indicateurs financiers/boursiers pour prédire l'impact sur le cours d'une action), et enfin certains ne se concentrent que sur l'analyse graphique (et supposent que les schémas historiques du cours d'une action se répéteront dans le futur). Evidemment, il ne nous a pas été possible de modéliser toutes les techniques de gestion de portefeuille possibles (et là n'est pas le but de ce projet). Nous avons donc choisi de modéliser trois types d'intelligence : un agent aléatoire, un agent qui pratique l'analyse financière en utilisant le ratio de Sharpe, et enfin un agent qui se concentre sur l'analyse économique en s'intéressant aux bénéfices des entreprises.

2.1.1 L'agent Random

L'Agent Aléatoire (*RandomAgent*) a pour but de créer du "bruit" dans le mécanisme de formation des prix des actions disponibles sur le marché. En effet, ce dernier, en achetant et vendant des actions de façon aléatoire, permet de créer de l'offre et de la demande et ainsi de faire vivre le marché. L'important ici est de souligner le fait que le comportement aléatoire de ces agents fait varier les prix sans tendance : ils donnent de la consistance au marché sans le perturber à long-terme (ce que l'on peut vérifier simplement en faisant tourner le programme avec uniquement des agents aléatoires et en constatant que les prix des actions fluctuent autour de leur valeur initiale).

Cependant, si les agents aléatoires ne donnent pas de tendance aux cours des actions, ils sont responsables d'une bonne partie de leur volatilité. Une première amélioration de ces agents a donc consisté à contrôler cette volatilité par le paramètre *_margin* propre à chaque agent aléatoire : ce paramètre modifie le prix demandé par l'agent aléatoire (à l'achat ou à la vente d'une action) en imposant une marge plus ou moins importante. Il permet donc d'implémenter des agents aléatoires qui passent des ordres à des prix plus ou moins éloignés des cours constatés des actions, d'où une influence sur la volatilité du marché des actions.

2.1.2 L'agent Company :

Cet agent va regarder la différence entre le prix de marché et le prix de l'action réelle, c'est-à-dire en regardant la situation des entreprises par rapport à leurs flux de dividendes. Ces agents vont faire l'hypothèse que le dernier dividende versé sera d'un montant égal à tous les dividendes perçus dans le futur et que le taux d'intérêt sans risque restera constant.

Si les dividendes viennent d'être distribués (versement de la somme D par agent), leur évaluation du prix de l'action sera :

$$P_0 = \sum_{i=1}^{\infty} \frac{D}{(1+r)^i} = \frac{D}{r}$$

r est ici le taux d'intérêt sans risque pour une durée égale à l'intervalle de temps entre deux paiements de dividendes (T).

Ainsi, en actualisant ce prix à la date d'aujourd'hui (espacé de d du prochain versement de dividendes) et en ayant rajouté le prochain dividende on obtient la formule :

$$P = \frac{P_0 + D}{(1+r)^{\frac{d}{T}}}$$

$$P = \frac{D}{r} \times \frac{1}{(1+r)^{\frac{d}{T}-1}}$$

A chaque tour, l'agent Company va donc sélectionner aléatoirement une action, l'évaluer regarder si le prix de marché diffère d'un taux `_margin` avec son évaluation. Si oui, il va passer un ordre d'achat de 50 actions au prix de marché augmenté de son taux `_margin`. Cet agent a pour but dans notre modélisation de faire premièrement le lien entre la situation économique de l'entreprise et les prix sur le marché, et dans un second temps avec le marché obligataire ce qui devrait relier la dynamique du taux sans risque à la dynamique du marché action.

2.1.3 L'agent Sharpe

L'agent Sharpe (*SharpeAgent*) met en place une analyse financière du marché des actions bien connue en gestion de portefeuille : il essaye de constituer un portefeuille optimal, c'est-à-dire un portefeuille (une allocation d'actifs) qui se trouve sur (ou en tout du moins au plus près possible) de la frontière efficiente du marché. La frontière efficiente du marché consiste en l'ensemble des portefeuilles dont le rendement est maximal pour un niveau de risque donné (de manière équivalente, c'est aussi l'ensemble des portefeuilles dont le niveau de risque est minimal pour un rendement donné). En d'autres termes, un portefeuille qui se trouve à l'instant t sur la frontière efficiente atteint un compromis gain/risque optimal.

Il existe une multitude de manières de rendre un portefeuille optimal (de le "placer" sur la frontière efficiente). Une de ces méthodes consiste à maximiser le ratio de Sharpe du portefeuille.

En effet, le ratio de Sharpe mesure l'écart de rentabilité d'un portefeuille d'actifs financiers (actions par exemple) par rapport au taux de rendement d'un placement sans risque (autrement dit la prime de risque, positive ou négative), divisé par un indicateur de risque, l'écart-type de la rentabilité de ce portefeuille (autrement dit sa volatilité). Formellement, $S = \frac{(R-r_0)}{\sigma_\alpha}$, où R est l'espérance des rentabilités du portefeuille, r_0 étant le référentiel de comparaison choisi (en général le taux de placement sans risque), et σ l'écart-type du taux de rendement du portefeuille considéré. Pour simplifier, c'est un indicateur de la rentabilité (marginale) obtenue par unité de risque pris dans cette gestion.

Concrètement, nous nous plaçons sur un marché avec `nb_companies` actions. L'agent Sharpe doit trouver à chaque tour l'allocation ω_i optimale de l'action i dans son portefeuille (ce qui revient à résoudre un programme de maximisation dans \mathbb{R}^n). Nous ne procédons pas exactement ainsi pour deux raisons :

- D'abord, pour la complexité des calculs engendrés. Il y aura des dizaines d'agents Sharpe avec des paramètres différents, ce qui représente une charge de calcul considérable à chaque tour.
- Ensuite et surtout, parce que le but du projet est aussi de pouvoir comparer les intelligences mises en place en matière de gestion de portefeuille. Pour se faire, il est fondamental de faire en sorte que tous les agents aient la même capacité de calcul à chaque tour, et ne passent qu'un seul ordre par tour. Si nous laissons l'agent Sharpe résoudre à chaque tour un programme de maximisation dans \mathbb{R}^n , sa puissance de calcul s'en trouverait clairement disproportionnée par rapport aux autres.

Pour résoudre ce problème, nous implémentons un agent Sharpe qui constitue un portefeuille **asymptotiquement** optimal en résolvant à chaque tour un programme de maximisation dans \mathbb{R} et en améliorant son portefeuille en passant un ordre (à l'achat ou à la vente) sur une seule action à la fois (et non plus sur $nb_companies$ actions à chaque tour). L'agent procède donc comme suit :

- De la même façon que pour l'objet Stat, chaque agent Sharpe possède un attribut *_period* qui renseigne du nombre de tours de cotations passés que l'agent prend en compte. Tant que ce nombre de tours n'est pas atteint, l'agent se comporte de manière passive, en constituant un portefeuille équilibré.
- Une fois que ce nombre de tours est atteint, l'agent répète les mêmes opérations à chaque tour. D'abord, il choisit une action au hasard (que nous appelons l'action référence). Son portefeuille se divise alors en deux : *portfolioA* constitué uniquement de ses actions référence, et *portfolioB* constitué de toutes ses actions sauf de ses actions référence. Il choisit enfin l'allocation $(\omega, 1 - \omega)$ entre ses deux sous-portefeuilles qui maximise son ratio de Sharpe.

Pour ce faire, il n'a qu'à calculer les rendements et volatilités de *portfolioA*, *portfolioB*, et de *portfolioToT* (où $portfolioToT = \omega portfolioA + (1 - \omega) portfolioB$) en utilisant le vecteur des rendements et la matrice de variance-covariance déjà présents dans l'objet Stat prévu à cet effet.

Au niveau de la programmation, l'agent Sharpe possède les méthodes importantes suivantes :

- Les méthodes *distributionA* et *distributionB* qui renvoient les vecteurs de répartitions en actions des portefeuilles A et B définis comme ci-dessus.
- La méthode *MaxiSharpe*, qui avec tous les rendements, volatilités et covariances donnés en argument, renvoie le ω optimal qui maximise le ratio de Sharpe (en itérant sur ω avec un pas donné). Cependant, afin d'éviter des passages d'ordres trop importants, nous avons pris l'initiative de réduire l'intervalle dans lequel évolue ω dans le programme de maximisation de l'agent. En effet, ce n'est plus l'intervalle $[0, 1]$ mais un intervalle "proche" de ω_0 (avec une marge ajustable), où ω_0 désigne le ω au début du tour (ie. la répartition initiale entre les deux portefeuilles A et B dans le portefeuille total de l'agent). Ainsi, l'agent ne fera pas varier cette répartition initiale de manière drastique, et par conséquent ne passera pas des ordres extrêmes (qui dépassent son capital et qui ne seront jamais exécutés entièrement).
- Enfin la méthode *Do* qui fait que le programme s'exécute, finit par faire passer un ordre à l'agent.

De manière plus calculatoire on a :

La fonction *MaxiSharpe* détermine : $\omega^* = \operatorname{argmax}(\frac{R_P - r_0}{\sigma_P^2})$ tel que :

- $P = \omega^* A + (1 - \omega^*) B$: portefeuille joint
- $R_P = \omega^* R_A + (1 - \omega^*) R_B$: Rendement du portefeuille joint
- $\sigma_P^2 = \omega^{*2} \sigma_A^2 + (1 - \omega^*)^2 \sigma_B^2 + \omega^*(1 - \omega^*) \rho$: Variance du portefeuille joint, où $\rho = \operatorname{cov}(A, B)$ est la covariance entre les deux portefeuilles.

On a également $\omega^* = \frac{N_A^*}{N_A^* + N_B^*}$, où

- N_A^* désigne le nombre d'actions A optimal à avoir dans le portefeuille P.
- N_A et N_B désignent respectivement le nombre d'actions présentes initialement dans le portefeuille A et dans le portefeuille B.

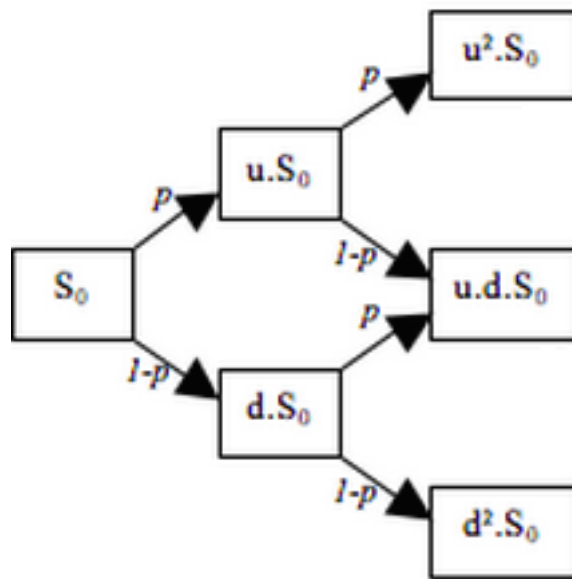
Par suite, on a $N_A^* = \frac{\omega^* N_B}{1 - \omega^*}$. Pour obtenir un tel portefeuille optimal P, il faut donc acheter ou vendre $(N_A^* - N_A)^+$ actions A, le sens de la vente étant déterminé par le signe de $(N_A^* - N_A)$.

2.2 Le marché options :

Les produits dérivés jouent un rôle primordial sur les marchés financiers, d'où l'importance de les implémenter dans notre projet. Nous nous sommes intéressés aux deux méthodes les plus courantes en ce qui concerne le pricing d'options vanilles : l'arbre binomial et les formules de Black-Scholes. Ainsi, nous avons créé deux agents utilisant ces deux méthodes.

2.2.1 L'agent Tree

L'arbre binomial permet de valoriser un produit dérivé en utilisant un "arbre" multi-période dans un contexte risque neutre. Nous avons choisi par souci de clarté et de simplicité de nous limiter à un arbre binomial recombinaison à deux périodes. Ce dernier est schématisé de la manière suivante :



Nous nous plaçons dans le cas du call européen dont le pay-off à maturité est $(S_T - K)^+$ où S désigne le prix spot du sous-jacent et S_0 désigne le prix du sous-jacent à l'instant initial. A chaque période, deux éventualités se présentent :

- avec une probabilité p^* , appelée probabilité risque-neutre, l'actif sous-jacent voit sa valeur multiplier par u ($u > 1$). Le pay-off associé est C_u .
- avec une probabilité $(1 - p^*)$, l'actif sous-jacent voit sa valeur multiplier par d ($0 < d < 1$). Le pay-off associé est C_d .
- le processus se répète et les pay-offs du dérivé deviennent C_{uu} , C_{dd} ou $C_{du} = C_{ud}$.

Pricer l'option se fait en deux étapes :

1. calculer l'espérance E^* du flux de cash-flow futur sous la probabilité risque neutre p^*
2. prendre la valeur actualisée en considérant le taux d'intérêt sans risque

L'agent choisit aléatoirement une option sur le marché, et la price avec ce modèle. Une fois le produit dérivé pricé, l'agent compare son prix avec le prix de l'option (le prix qui ressort de l'évaluation vendeur) : si le prix est avantageux (modulo sa marge), il fait une offre à ce prix.

2.2.2 L'agent Black-Scholes

La méthode de Black-Scholes est une généralisation en temps continu de la méthode de l'arbre binomial. Pour pricer un call ou un put, l'agent va donc regarder les paramètres suivants :

- S (prix spot du sous-jacent), K (prix d'exercice), σ (volatilité)
- r (taux d'intérêt sans risque), ρ (taux perçu au nom des dividendes)
- mat (maturité de l'option)

Une fois ces paramètres considérés, l'agent va pouvoir pricer le call. Pour cela, il va utiliser la formule de Black-Scholes qui donne le prix suivant :

$$C_t = S * \exp(-\rho * mat) * \mathcal{N}(d_1) - K * \exp(-r * mat) * \mathcal{N}(d_2) \quad (2.1)$$

où :

- \mathcal{N} représente la fonction de répartition d'une loi normale centrée réduite
- $d_1 = \frac{\log(\frac{S}{K}) + (r - \rho + \frac{\sigma^2}{2}) * mat}{\sigma * \sqrt{mat}}$
- $d_2 = d_1 - \sigma * \sqrt{mat}$

Afin de ne pas alourdir les calculs et pour optimiser le processus, le prix du put (P_t) est calculé à l'aide de la relation de parité call-put qui fait intervenir le prix du call calculé précédemment. La relation call-put est la suivante :

$$C_t - P_t = S * \exp(-\rho * mat) + K * \exp(-r * mat) \quad (2.2)$$

ainsi il vient que :

$$P_t = S * \exp(-\rho * mat) + K * \exp(-r * mat) - C_t \quad (2.3)$$

Un autre élément important est le calcul du Δ de couverture. En effet, ce dernier correspond à la variation de valeur de l'option pour un faible mouvement du sous-jacent et il s'interprète également comme la quantité d'actions à acheter / vendre pour insensibiliser le portefeuille de réplication à une petite variation du sous-jacent.

Les différentes étapes de la démarche faite par l'agent BlackScholes sont les suivantes :

- Dans un premier temps, l'agent regarde les options se trouvant dans son portefeuille. Si ces dernières ont été vendues, il va "hedger" sa position (d'où le calcul du Δ). Pour cela, il va remplir un vecteur (initialement vide), attribut de l'agent, δ contenant les deltas.
- Au tour suivant, le vendeur se rend compte que l'option a été vendue et il se couvre en achetant des obligations, "zero-coupon bonds".
- Dans un second temps, il va donc passer les ordres nécessaires pour obtenir ce vecteur δ 'optimal' le permettant de couvrir parfaitement les options qu'il a vendues.

2.3 Le marché obligataire :

L'agent Bond : Dans cette simulation, les agents Bond seront les premiers acteurs dans l'évolution du taux sans risque.

A chaque tour, ces derniers vont regarder parmi toutes les obligations qu'ils possèdent celles qu'ils vont vendre. Le prix est ici fixé par le Stock Exchange et expliqué dans la partie sur l'objet Bond. Pour les explications ci-dessous on notera $P = \text{price}$, t = temps écoulé depuis création de l'obligation, $T_m = \text{maturity}$, T_r période de référence, R = taux de l'obligation sur une période T_r , r = taux du Trésor pour ce tour-ci et enfin t temps écoulé depuis la création de l'obligation. Il va comparer ce prix au "rendement de l'obligation" η , i.e. les intérêts accumulés mais non encore perçus :

$$\eta = P(1 + R)^{\frac{t}{T_r}}$$

Pour que le Bond Agent vende, il faut que le prix de revente dépasse "le rendement de l'obligation" d'un certain taux margin , paramètre de l'agent.

En équation, il va regarder si cette inégalité est vraie :

$$P \frac{(1+R)^T}{(1+r)^{\frac{T_m-t}{T_r}}} > P(1 + R)^{\frac{t}{T_r}} \times (1 + \text{margin})$$

Pour simplifier, si le taux a assez baissé entre le moment où l'obligation a été souscrite et "aujourd'hui", l'agent va préférer encaisser tout de suite son gain pour le replacer directement dans une autre obligation. Puis dans un second temps, il va placer entre 15% et 25% de son cash dans une nouvelle obligation.

L'effet macroscopique observé est le suivant, une baisse de taux va entraîner un effet de vente des agents qui vont alors avoir plus de liquidité ce qui va alors créer une augmentation de la demande en obligation et accentuer la baisse. Ceci justifie le fait que les agents veulent placer le plus possible d'argent tout de suite plutôt que d'attendre. Au contraire, une augmentation du taux sans risque ne va pas créer cet effet de vente et la demande en obligation va rester faible permettant ainsi au taux d'augmenter.

Au long terme, ces agents voient leur capitaux augmenter et il sera normal de voir le taux baisser.

Grâce à cette modélisation, on va ainsi créer une certaine animation sur l'évolution du taux d'intérêt qui à son tour devrait permettre une animation sur le marché des actions grâce aux Companyagents.

Chapitre 3

Limites du projet et améliorations futures possibles

Tout d'abord, nous avons connu de grandes difficultés à utiliser Qt, ce qui nous a ralenti et qui explique la précarité de l'interface graphique. Par exemple, aucune fenêtre ne peut être fermée en cours d'exécution, et ces dernières sont toujours superposées de la même manière. Plus encore, le programme peut au démarrage peut ne pas fonctionner (ce dernier fonctionne tout de même relativement bien). Cependant, cela ne dépend pas de nous, Qt était l'interface graphique la plus conseillée pour programmer en C++. Malheureusement, sans déboguer, je n'ai pu résoudre les derniers bugs rencontrés et à cause de ça le marché option ne marche pas à 100%. De plus par le même bug, on ne peut incarner l'agent Tree et une partie du code de la fonction Do de l'agent BlackScholes est encore en commentaires.

Aussi, certains agents, dont l'agent Sharpe, réalisent de nombreux calculs, ce qui peut ralentir le programme à l'exécution. Nous avons essayé de rendre le programme le plus performant possible. En effet, avec seulement 5 agents Sharpe, le programme supporte la quantité de calculs sans aucun souci.

Pour aller plus loin Voici une liste non exhaustive d'idées qui permettraient de pousser un peu plus ce programme :

- Créer plus d'agents pour refléter plus précisément la multitude de techniques de pricing et de gestion de portefeuille qui existent dans le monde de la finance.
- Améliorer l'interface graphique : faire des sorties de graphiques des prix des actions sur le marché et mettre de la couleur (vert et rouge) lorsque les prix des actions montent ou descendent.
- De façon bien plus compliquée, il serait possible (sans trop de modification), de rajouter un ou plusieurs pays supplémentaires afin d'obtenir des actifs cotés en devise étrangère. On pourrait ainsi modéliser un autre marché, avec un autre Etat, et donc étudier les variations de taux de change.