Option's pricer

Thomas Aujoux, Louis Geist, Thibaut Metz, Justin Ruelland

3 février 2023

Table des matières

1 Introduction

1.1 But du programme

Le pricing d'une option est une notion fondamentale de la finance de marché. L'évaluation d'une option permet, par exemple, sa mise sur le marché à un prix n'exposant pas son émetteur à une situation d'arbitrage. Il est important de différencier la valeur d'un instrument financier et son prix ; le programme "pricer" ici développé permet uniquement d'estimer la valeur [1] de l'instrument selon un certain modèle. La valeur correspond à une estimation, par un modèle, du montant potentiel que cet instrument peut générer. Alors que le prix est la résultante d'un accord entre deux parties un acheteur et un vendeur et se matérialise par la transaction. Le projet permet d'évaluer des produits dérivés optionnels construits autour d'un sous-jacent qui sera dans ce projet uniquement considéré comme un actif de type valeur mobilière (security).

Dans le cadre du modèle Black-Scholes, ce projet effectue :

- le calcul du prix d'options
  - européennes sur sous-jacent avec ou sans dividendes (utilisation des formules explicites du modèle de *Black-Scholes-Merton*),
  - américaines (par la méthode de *Longstaff and Schwartz*),
  - asiatiques (par méthode de *Monte-Carlo*),
- l'affichage de la stratégie de réplication d'options (européennes sur sous-jacent avec ou sans dividendes - utilisation de la parité call-put)

La première partie du rapport se concentre sur l'architecture du programme. Dans un second temps, nous décrivons les fonctionnalités implémentées. Finalement, nous aborderons les limites de notre programme ainsi que des tentatives de solutions.

Les formules mathématiques utilisées sont explicitées à la fin du rapport.

1.2 Conventions et hypothèses

Quelques hypothèses importantes optées pour notre programme :

- r : le taux sans risque est constant,
- aucun affichage du programme ne présente une unité pour les prix : le programme donne de bons résultats tant que tous les prix sont utilisés avec la même devise.

2 Architecture du programme

2.1 Vue d'ensemble de l'architecture : le diagramme de classes UML

Choix de représentation :

- les Getter et Setter pour chaque attribut des classes sont bien définis dans le code, nous ne les écrivons pas dans le diagramme de classe,
- Pareil pour le constructeur par défaut, par copie et le destructeur.
- Les fonctions spécifiques à la classe *asset* tels que Acutalisation et Estimation ne sont pas non plus représentées dans le diagramme

1. Nous parlerons abusivement de méthode de pricing ou d'estimation de spotprice au cours du projet

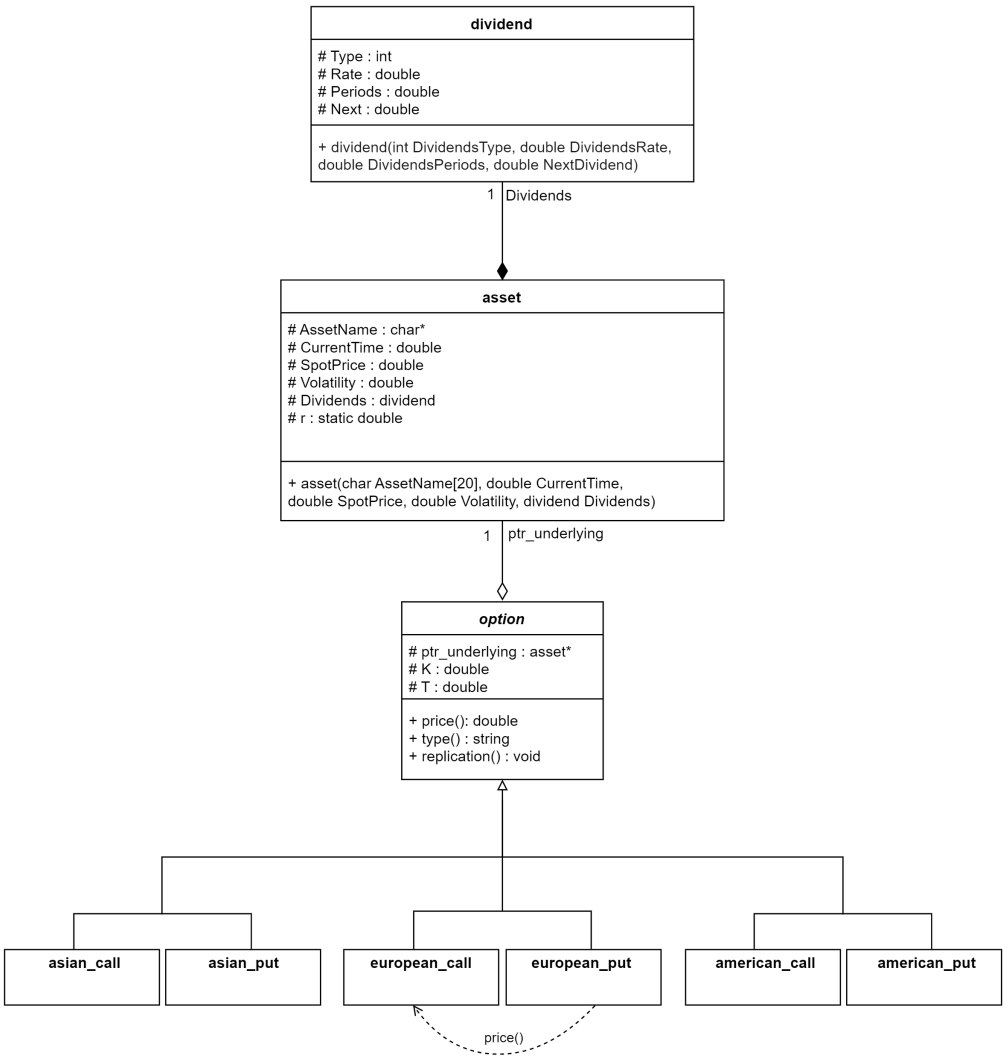

FIGURE 1 – Diagramme de classes UML

Légende.

La flèche indique un héritage. La classe mère de l'héritage est pointée par la flèche.

Les traits avec un losange indiquent :

— pour un losange noir : une relation de composition (Le losange est du côté de l'objet qui est composé par l'autre objet. Comme c'est une relation de composition, la destruction de l'objet composé détruit l'objet composant.)

— pour un losange creux : une relation d'agrégation (le losange est du côté de l'objet agrégat. Comme c'est une relation d'agrégation, la destruction de l'objet agrégat ne détruit pas l'objet agrégé.)

2.2 Classe asset et dividend

Nous avons une classe **asset** pour les actifs financiers, cette classe a pour but de représenter le sous-jacent des options et par simplicité uniquement des valeurs mobilières. Les variables membres de

la classe *asset* sont :

- AssetName : un pointeur vers une chaîne de caractères qui désigne le nom de l'actif,
- CurrentTime : la date à laquelle est évaluée l'actif,
- SpotPrice : le prix spot de l'actif,
- Volatility : la volatilité de l'action,
- r : le taux sans risque (c'est un membre statique de la classe, car il doit être le même pour tous les asset),
- Dividends : les dividendes de l'actif, qui est un objet de la classe *dividend*.

La classe **dividend** est une classe agrégée à la classe *asset*. Nous avons par choix déterminé que l'attribut relatif aux dividendes, Dividends, est un objet de la classe *dividend* et non pas un pointeur vers un objet de cette classe. En effet, il nous paraissait cohérent que les dividendes sont inhérents à l'actif auquel il est associé, et n'ont pas d'intérêt d'exister en dehors de ce dernier (contrairement à la relation entre l'option et son sous-jaccent). Ses variables membres sont :

- Type : un entier dans $\{0, 1, 2\}$ qui indique le type de dividendes :
  - 0 : signifie qu'il n'y a pas de dividendes,
  - 1 : désigne des dividendes à date de paiements discrets,
  - 2 : désigne des dividendes à paiements continus.
- Rate : le taux des dividendes,
- Periods : la période entre deux paiements (pertinent que pour un dividende de type 1),
- Next : durée jusqu'au prochain paiement (pertinent que pour un dividende de type 1).

2.3 Classe option

La classe **option** est une classe abstraite. Ses attributs sont :

- K : le strike de l'option,
- sigma : la volatilité du sous-jacent,
- ptr_underlying : pointeur vers un objet de la casse asset.

Nous avons défini trois paires de classes qui héritent de cette interface *option* :

- **Options européennes**
  - une classe *european_call*,
  - une classe *european_put* ;
- **Options américaines**
  - une classe *american_call*,
  - une classe *american_put* ;
- **Options asiatiques**
  - une classe *asian_call*,
  - une classe *asian_put*.

3 Fonctionnalités du programme

3.1 Fonctions classiques de classe

Les fonctions suivantes ont été implémentées pour toutes les classes du programme.

Pour chaque classe, les constructeurs par défaut, par copie et avec arguments ont été réécrits. A contrario, les destructeurs sont ceux par défaut, à l'exception de la classe *asset*. L'attribut AssetName étant un pointeur de caractères, et la création de cet attribut se faisant par allocation dynamique (cf setter AssetName).

Les fonctions membres classiques que sont les getters et setters ont été implémentées pour tous les attributs de classe.

Nous avons surchargé les opérateurs de iostream " >> " et " << " pour les classes *option* et *asset*. Comme une option est composée d'un pointeur vers un asset, les opérateurs iostream de *option* appellent ceux de *asset*.

L'opérateur "=" a aussi été surchargé pour les classes *asset* et *dividend*, afin de respecter la règle des trois.

3.2 Fonctions spécifiques aux classes

3.2.1 Classe asset

La classe *asset* comporte deux fonctions particulières spécifiques :

asset_actulalization : Cette fonction a pour but d'actualiser l'asset à un nouveau temps, elle considère comme connu le nouveau SpotPrice de l'actif. Elle permet au délà de regrouper les setters des attributs SpotPrice et CurrentTime d'actualiser correctement les dividendes et les futurs dates de paiement de dividendes (notamment si ils sont forfaitaires).

asset_estimation : Cette fonction est quasiment identique à la précédente, à la différence qu'elle ne considère pas comme connu le nouveau spotprice de l'actif. Elle fixe donc la valeur du spotprice comme l'espérance du spotprice au nouveau temps en calculant cette espérance par absence d'arbitrage suivant le taux sans risque et le type de dividendes émis. Cette fonction sert notamment dans les stratégies de réplication, permettant d'estimer les payoffs futurs du portfolio. Elle permettrai aussi le pricing très simple de produits dérivés linéaires du sous-jacent.

3.2.2 Classe option

Les fonctions membres spécifiques à la classe option sont toutes des fonctions purement virtuelles :
— type() : qui retourne (en string) le type de l'option (european put, asian call, etc... : c'est-à-dire le nom de la classe fille),
— price() [2] : qui retourne le prix actuel de l'option ainsi définie,
— replication() : qui affiche comment répliquer l'option (Nous ne savons pas répliquer des options autres qu'européennes, donc nous avons renvoyé un message d'avertissement pour les options non européennes.)

3.2.3 Intérêt informatique des fonctions de pricing

Comme détaillé dans l'introduction, les options sont évaluées en se plaçant sous le modèle de Black-Scholes. Les options européennes sont pricées par les formules analytiques dérivant de ce modèle. En revanche, les options asiatiques et américaines sont évaluées par des méthodes Monte Carlo.

Le **pricing d'options asiatiques** a été l'occasion d'utiliser la méthode de Monte-Carlo. En effet, l'évaluation est réalisée par simulation d'un grand nombre de trajectoires de prix du sous-jacent et par le calcul du prix a posteriori des options asiatiques à maturité pour chacune des trajectoires. En vertu de la loi forte des grands nombres, le moyennage de prix des options et l'actualisation à la date 0 permet d'obtenir la valeur de l'option asiatique.

Le calcul du prix a posteriori des options asiatiques nécessite la valeur moyenne du sous-jacent durant la période de détention de l'option. C'est pourquoi nous avons discrétisé le temps pour simuler des $\mathcal{N}(0,1)$ et ainsi simuler un processus de Wiener et donc obtenir une trajectoire de prix.

Le **pricing des options américaines** se fait par méthode *Least Squares Monte Carlo* (LSM introduite par Longstaff et Schwartz en 2001). Cette méthode nécessite entre autres l'implémentation de régression polynomiale et de simulation de mouvement brownien. La discrétisation du pas de temps couplée à la simulation de multiples trajectoires est stockée sous forme d'une matrice. L'implémentation de la méthode de LSM passe donc par l'utilisation de matrices de trajectoires et de valuation. Cette mise en forme a été choisie afin d'optimiser la gestion mémoire et simplifier les régressions polynomiales, on a pour cela eu besoin d'importer la librairie *eigen*. Les fonctions contiguës à cette matrice et à ces vecteurs de grandes tailles ne prennent que des pointeurs en argument afin de ne pas encombrer la

2. Nous détaillons l'intérêt informatique des méthodes de pricing par la suite

mémoire vive lors des appels fonctions. La simulation se fait de manière analogue à celle du pricing d'options asiatiques. La méthode de pricing nécessitant un certain nombre de fonctions connexes, nous avons crée le header tools.h et son fichier de définition tools.cpp.

4 Critique des problèmes rencontrés et des solution adoptées

4.1 Organisation d'un programme de grande ambition

La classe *option* (et ce qui en hérite), d'un côté, et la classe *asset* (et la classe dividend), d'un autre, ont été codé par deux personnes. Nous avions réalisé des rendez-vous réguliers pour se mettre d'accord comment coder et aborder le programme, mais lorsqu'il était venu d'utiliser la classe *asset* dans la classe *option* (afin de valoriser des options sur sous-jacents avec dividendes), nous nous sommes rendu compte de petites spécificités à nos deux manières de coder que nous n'avions pas anticipées.

Forts de cette expérience, nous en concluons que nous devrons pour un prochain projet de programmation, davantage nous mettre d'accord sur les fonctionnalités de chacune des classes, notamment sur les types des arguments et le type renvoyé des fonctions. Une idée aussi serait d'essayer de réaliser un diagramme de classes avant le début du codage en ajoutant les appels de fonctions entre classes.

Par exemple, nous trouvons une limite importante de notre projet dans le manque de dynamisme temporel du pricing d'options. Nous entendons par là que si le prix d'un asset est actualisé à une nouvelle date (avec la fonction assset_actualization(t,S)), la méthode .price() de la classe option va prendre en compte le nouveau prix de l'asset, mais pas du changement de la date actuelle ("Current"). Par exemple, nous considérons une option émise avec une maturité de deux ans. Nous apprenons à bout d'un an que le prix du sous-jacent est S' . Dans notre programme, nous entrons alors le code suivant :

Listing 1 – Exemple d'une "mauvaise" utilisation du programme

```
double new_current_time = 1;

mon_asset.asset_actualization(new_current_time,S');
mon_option.price();
```

On s'attendrait à obtenir la nouvelle valeur de l'option en $t = 1ans$ avec pour prix du sous-jacent S' . Cependant, ces deux lignes renvoient le prix de l'option avec prix du sous-jacent S' mais avec une maturité de 2 ans encore. Pour corriger cela, il faut alors entrer la syntaxe suivante :

Listing 2 – Gestion du problème

```
double new_current_time = 1;

mon_asset.asset_actualization(new_current_time,S');
double old_T = mon_option.get_T();
mon_option.set_T(old_T-new_current_time);
mon_option.price();
```

On remarque que cette solution n'est pas robuste à une deuxième actualisation du temps (il faudrait un peu l'adapter).

Par ailleurs, la classe asset ne concerne dans notre projet que les sous-jacents des options et se restreint à représenter des valeurs mobilières. Dans une idée d'amélioration du programme, nous pensons qu'il aurait été judicieux de considérer les options comme des actifs. Nous aurions alors par la suite dérivé chacune des classes du projet (hormis dividendes en l'état) héritant de cette classe originel, ne comportant par exemple en attribut qu'un prix et un attribut temporel [3].

3. type CurrentTime

4.2 Modification d'un attribut d'une classe attribut

L'attribut *dividend* de la classe *asset* n'étant pas un pointeur il y avait nécessité d'obtenir un alias ou un pointeur vers cet objet lors de l'actualisation de l'asset notamment.

Il aurait été possible de faire autrement, néanmoins cet implémentation permettait la manipulation d'alias.

Solution : création de la fonction membre de asset 'get_alias_dividend' un "getter" non standard. Le getter n'est pas de type 'const'. Cela permet alors depuis l'asset accéder à *dividend*.

4.3 Problème du destructeur "timide"

Nous avons fait face à un bug très étrange sur notre programme. L'exécution du programme affichait un "segmentation fault" et nous ne comprenions pas comment une telle erreur avait lieu. Le problème provenait d'un destructeur que nous nommerons "timide", car il n'était pas trouvé par notre compilateur ; le compilateur essayait donc de créer son destructeur, mais comme on manipulait un pointeur dont la mémoire était allouée dynamiquement, cela menait à une erreur.

Nous avons rapidement identifié que le programme essayait de détruire deux fois le même pointeur. Nous avons localisé l'erreur à l'aide de deux *cout* avant après après le delete du Name dans le destructeur non trivial de *asset* (cf ligne 98 de asset.cpp) qui supprime la mémoire allouée au pointeur AssetName. La destruction (en fin de programme) d'un objet de la classe european_call appelait le destructeur de european_call dans un premier temps, puis appelait le destructeur de option. Cependant, le programme ne semblait pas trouver le destructeur de european_call que nous avons écrit et définissait donc le sien qui amenait à la double destruction du même pointeur.

Le destructeur non trouvé par notre programme était :

```
european_call::~european_call() {};
```

Le debugage consistait simplement à 1. modifier le destructeur de la façon suivante (ajout d'un retour à la ligne) :

```
european_call::~european_call() {
};
```

2. make run à nouveau le main ; nous n'avons alors plus le "segmentation fault". 3. Et nous pouvions supprimer le retour à la ligne, pour revenir à :

```
european_call::~european_call() {};
```

4. make run à nouveau le main qui n'affiche plus de "segementation fault" (alors qu'à l'oeil nu, le c'est exactement le même destructeur qu'au début...).

NB : puisque le bug avait lieu pour chacune des classes filles (european_call, european_put, asian_call, asian_put...), nous avons pu clairement identifier cette manipulation qui permet de debug (et l'avons même enregistré en vidéo...).

5 Conclusion

Ce premier projet de programmation C++ portait sur l'implémentation d'un pricer d'options.

Au cours de ce travail, nous nous sommes concentré sur la programmation orientée objet, qui est naturelle dans le langage C++. Nous avons également constaté l'intérêt de C++ pour cette tâche en raison de sa rapidité d'exécution et sa gestion d'objets qui présente un certaine robustesse et une intégrité du code. La rapidité d'exécution s'est faite ressentir lors de la simulation d'un très grand nombre de trajectoires pour l'utilisation de Monte Carlo (dans le pricing d'options asiatiques et américaines). Enfin, l'utilisation d'un fichier makefile nous a permis de mieux comprendre les étapes de compilation et d'exécution d'un code (compréhension pratique du mot clef #pragma once pour pour les headers, compréhension de la manière de récupération des .cpp qui sont à compiler dans le

makefile – ce que je n'avais pas compris en utilisant d'abord une solution toute faite dans Microsoft Visual Studio).

En résumé, notre programme permet notamment de calculer les valeurs d'options européennes pour un sous-jacent avec et sans dividendes, d'options asiatiques pour un sous-jacent sans dividendes et d'options américains pour un sous-jacent sans dividendes.

6 Annexe : formule de pricing

6.1 European options

European sans dividendes

- call : formule de Black-Scholes : $c = SN(d_1) - e^{-rT}N(d_2)$; où : $N(.)$ est la fonction de répartition de $\mathcal{N}(0,1)$
- put : call-put parity : $c - p = S_T - e^{-rT}K$

European lump payment dividend :

- call : $c_{lump\_payments} = c((1-\delta)^n S_0, \sigma, T, K)$
- put : $c_{lump\_payments} - p_{lump\_payments} = (1-\delta)^n S_0 - e^{-rT}K$
- Remarque : dans le code, j'ai nommé S_hat $= e^{-\delta T}S_0$

European continuous dividend :

- call : $c_{continuous} = c(e^{-\delta T}S_0, \sigma, T, K)$
- put : $c_{continuous} - p_{continuous} = e^{-\delta T}S_0 - e^{-rT}K$
- Remarque : dans le code, j'ai nommé S_hat $= e^{-\delta T}S_0$

6.2 Asian options

On note $\bar{S} = \frac{1}{T}\int_0^T S_t dt$

- call : $V_T = (\bar{S} - K)^+$
- put : $V_T = (K - \bar{S})^+$

6.3 American options

American sans dividendes (émise à $t = 0$ de maturité T)

- call : payoff si choix d'exercer $Vt = (S_t - K)^+ \ \forall t \in [0;T]$
- put : payoff si choix d'exercer $Vt = (K - S_t)^+ \ \forall t \in [0;T]$