Equation de Black and Scholes

Gautier Poursin, Théo Lazzaroni

17 janvier 2021

Introduction

Ce projet de C++ a pour but de calculer la valeur d'un call ou d'un put au fil de temps et de la quantité. L'objectif est de calculer cette valeur via différentes méthodes: la méthode de Crank Nicholson et la méthode des différences finies implicites. Dans un premier temps, nous allons nous intéresser à la partie théorique du projet (prise en main des différentes méthodes, résolution de l'equation, détermination des coefficients etc.). Ensuite, nous aborderons la partie code. Nous expliquerons nos choix mais aussi les difficultées rencontrées, comment nous les avons contournées puis quelles sont les améliorations possibles de notre code.

Sommaire

1. Partie théorique

1.1 Méthode de Crank-Nicholson

On pose ici $\frac{\partial C}{\partial t} + rS\frac{\partial C}{\partial S} + \frac{\sigma^2 S^2}{2}\frac{\partial^2 C}{\partial^2 S} = rC$

Pour la résolution explicite, on pose $\frac{\partial C(t_n,s_i)}{\partial S} = \frac{1}{\Delta s}(C(t_n,s_i) - C(t_n,s_{i-1}))$, on obtient après remplacement dans l'équation:

$$\boxed{rC(t_n,S_i) = \frac{C(t_{n+1},S_i)-C(t_n,s_i)}{\delta T} + rj\delta S\frac{(C(t_{n+1},S_{i+1})-C(t_{n+1},S_{i-1}))}{2\delta S} + \frac{(\sigma i\delta S)^2}{2}\frac{C(t_{n+1},S_{i+1})-2C(t_{n+1},S_i)+C(t_{n+1},S_{i-1})}{\delta^2 S}}$$

Pour la résolution implicite, on pose $\frac{\partial C(t_n,s_i)}{\partial S} = \frac{1}{\Delta s}(C(t_{n+1},s_i) - C(t_{n+1},s_{i-1}))$.
Cette fois ci, :

$$\boxed{rC(t_n,S_i) = \frac{C(t_{n+1},S_i)-C(t_n,s_i)}{\delta T} + rj\delta S\frac{(C(t_n,S_{i+1})-C(t_n,S_{i-1}))}{2\delta S} + \frac{(\sigma i\delta S)^2}{2}\frac{C(t_n,S_{i+1})-2C(t_n,S_i)+C(t_n,S_{i-1})}{\delta^2 S}}$$

La méthode de Crank-Nicholson consiste à faire la moyenne entre le méthode implicite et la méthode explicite. On obtient l'équation suivante:

$(\frac{1}{\Delta T} + \frac{rS}{2\Delta S} + \frac{\sigma^2 S^2}{4\Delta S^2})C(t_{n+1},s_i) - (\frac{rS}{2\Delta S} + \frac{\sigma^2 S^2}{4\Delta S^2})C(t_{n+1},s_{i+1}) + \frac{\sigma^2 S^2}{4\Delta S^2}C(t_{n+1},s_{i-1}) =$
$(-r - \frac{rS}{2\Delta S} + \frac{\sigma^2 S^2}{4\Delta S^2} + \frac{1}{\Delta T})C(t_n,s_i) + (\frac{rS}{2\Delta S} + \frac{\sigma^2 S^2}{4\Delta S^2})(C(t_n,s_{i+1}) - \frac{\sigma^2 S^2}{4\Delta S^2}C(t_n,s_{i-1})$

On peut écrire cette équation de cette forme, qui sera plus lisible:
$a_{n+1,i}C(t_{n+1},s_i) + a_{n+1,i+1}C(t_{n+1},s_{i+1}) + a_{n+1,i-1}C(t_{n+1},s_{i-1}) = b_{n,i}C(t_n,s_i) +$
$b_{n,i+1}C(t_n,s_{i+1}) + b_{n,i-1}C(t_n,s_{i-1})$

Cette équation est une équation vectorielle: $AC(t_{n+1}) + K_{n+1} = BC(t_n)$ avec A,B des matrices carrées de taille $M * M$, K un vecteur colonne déterminé en fonction des conditions initiales et $C(t_{n+1}), C(t_n)$ vecteur colonne possédant les valeurs du call ou du put à l'instant n et n+1. Au vu de la forme de l'équation précédente, on déduit assez rapidement que A et B sont **des matrices tridiagonales** avec , $\forall i \in [1, M]$:
- $a_{i,i} = 1 - \frac{(\sigma j\delta T)^2}{2}$
-$a_{i,i+1} = \frac{j\delta T(\sigma^2 j + r)}{4}$
-$a_{i,i-1} = \frac{j\delta T(\sigma^2 j - r)}{4}$
-$b_{i,i} = 1 + r\delta T + \frac{\sigma^2 j^2 \delta T}{2}$
-$b_{i,i+1} = -\frac{j\delta T(\sigma^2 j + r)}{4}$
-$b_{i,i-1} = -\frac{j\delta T(\sigma^2 j - r)}{4}$
Ces coefficients correspondent à la méthode de Crank-Nicholson, pour un put et un call. Tous les autres coefficients des matrices sont nuls.

Ce qui diffère dans entre un put et un call sont les conditions initiales et donc le vecteur K, qui est déterminé en fonction des conditions initiales.

Voici K pour un put:

$$\begin{pmatrix} a_1 K(e^{-r(T-(i-1)\delta T)} + e^{-r(T-i\delta T)}) \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

avec $a_1 = \frac{\delta T(\sigma^2 - r)}{4}$

Voici K pour un call:

$$\begin{pmatrix} 0 \\ \\ \vdots \\ c_M K(e^{-r(-T+(i-1)\delta T)} + e^{-r(-T+i\delta T)}) \end{pmatrix}$$

avec $c_M = -\frac{M\delta T(M\sigma^2 - r)}{4}$

Pour la résolution, nous allons donc commencer par effectuer le produit de A par C_{n+1}. On va donc obtenir un vecteur auquel on ajoute le vecteur K_{n+1} puis, à l'aide d'un algorithme de résolution pour les matrices triadiagonales, on résout $BC_n = C'_{n+1}$. On obtient donc C_n puis on recommence l'alogorithme pour obtenir C_{n-1},... jusqu'a obtenir C_0, qui est le résultat nécessaire pour l'affichage des courbes.

1.2 Méthode des différences finies

Avant tout, nous allons utilisé un nouveau **dT**. En effet, le chanegement de variable implique un nouveau interval de temps. On a $dT' = \frac{\sigma^2}{2}\delta T$ Pour cette méthode, nous avons utilisé déterminé de manière implicite cette équation: $\frac{\partial C}{\partial t'} = \mu \frac{\partial^2 C}{\partial^2 S}$.

Nous obtenons: $\boxed{\frac{C(t_{n+1},s_i) - C(t_n,s_i)}{\delta T'} = \frac{\mu}{\delta^2 s}(C(t_{n+1},s_{i+1}) - 2C(t_{n+1},s_i) + C(t_{n+1},s_{i-1}))}$

Ecrivons le de manière à avoir $AC_{n+1} + K_{n+1} = C_n$. Nous obtenons:

$\boxed{\frac{-\mu\delta T'}{\delta^2 s}C(t_{n+1},s_{i-1}) + (1 + \frac{2\mu\delta T'}{\delta^2 s})C(t_{n+1},s_i) - \frac{\mu\delta T'}{\delta^2 s}C(t_{n+1},s_{i+1}) = C(t_n,s_i)}$ Nous avons donc comme précédemment **une matrice tridiagonale**, avec comme coefficients:

- $a_{i,i} = 1 + 2\frac{\mu\delta T'}{\delta^2 s}$

-$a_{i,i+1} = -\frac{\mu\delta T'}{\delta^2 s}$

-$a_{i,i-1} = -\frac{\mu\delta T'}{\delta^2 s}$

Le reste des coefficients de la matrice étant nul. ce qui diffère est encore une fois le vecteur K entre un call et un put.

Voici K pour un put:

$$\begin{pmatrix} a_1 K(e^{-r(T-(i-1)\delta T')} + e^{-r(T-i\delta T')}) \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

avec $a_1 = -\frac{\mu \delta T'}{\delta^2 s}$

Voici K pour un call:

$$\begin{pmatrix} 0 \\ \vdots \\ c_M K(e^{-r(-T+(i-1)\delta T')} + e^{-r(-T+i\delta T')}) \end{pmatrix}$$

avec $c_M = -\frac{\mu \delta T'}{\delta^2 s}$

Pour la résolution, nous allons effectuer le produit matriciel AC_{n+1} avant d'ajouter le vecteur K_{n+1} au vecteur obtenu, ce qui donnera C_n. On réitère le calcul jusqu'à obtenir C_o.

2. Partie Code

2.1 Classe mère abstraite et ses filles

2.1.1 Présentation des classes

Pour la réalisation de ce projet, il nous est demandé de travailler avec une classe mère abstraite, dont les classes filles seront les classes associées à un Put et à un Call. Au sein de ces classes filles, les 2 méthodes y seront implémentées. La classe mère est abstraite: il faut donc au minimum une fonction membre virtuelle pure. C'est à dire qu'elle est suivie de **= 0** dans le fichier d'en tête. Elle ne sera donc pas définie dans la classe mère mais dans les classes filles distinctivement.

La classe mère se nomme **Resolv**. Nous allons y répertorier dedans la majorité des fonctions qui seront présentes dans les classes filles. Il y a des fonctions qui permettent d'obtenir **dT et dL**, des fonctions de calculs du maximum et du minimum entre 2 floats, le fonction puissance... De plus, on va y répertorier les fonctions qui permettent de créer les matrices (**computeMatrix**) et de les remplir avec les coefficients calculés par la fonction **computeCoef**. Ces 2 fonctions sont aussi définies pour la méthode de Crank Nicholson et celles des différences finies implicites (**computeMatrixDiffFinies** et **computeCoefDiffFinies**). De plus, on peut y retrouver des fonctions permettant la résolution

de systèmes matricielles (**decompLu** permet de résoudre Ax=B avec A une matrice tridiagonale, x et B deux vecteurs) ou une fonction permettant de réaliser le produit entre une matrice et un vecteur (**product**).

Comme dans toutes classes, il est nécessaire d'avoir un constructeur et un destructeur. Ici, le constructeur de la classe permet de donner une valeur à **dT** et **dL**. Ainsi, aucuns vecteurs ou matrices ne sont crées dans le constructeur. Il n'y aura donc rien dans le destructeur. Vecteurs et matrices seront, **par choix**, crées en dehors de la classe. Il est donc nécessaire d'avoir une fonction **deleted**, qui permet de supprimer l'espace mémoire alloué à tous nos vecteurs et matrices à la fin du programme, car ceux ci sont en dehors de la class eet donc le destructeur ne peut les supprimer.

La classe **Resolv** est abstraite donc il se doit d'y avoir des fonctions membres virtuelles pures. Il y en a 6:
- **initialize** et **initializeDiffFinies** : permet d'initialiser le vecteur de départ avec les conditions aux limites pour la méthode Crank Nicholson ou des différences finies implicites
- **print**: permet d'afficher le temps et la valeur S pour un put ou un call
- **get_id**: renvoie l'ID d'un payoff (0 pour un put, 1 pour un call). Ces ID sont important pour la fonction **print**
- **initializeDiffFinies**: permet d'initialiser le vecteur de départ avec les conditions aux limites pour la méthode des différences finies implicites.
-**calculate** et **calculateDiffFinies**: permet de calculer le nouveau vecteur V_n en connaissant V_{n+1} pour la méthode de Crank Nicholson ou celle des différences finies implicites.

Cette classe mère possède 2 classes filles: **Put** et **Call**. Ces classes utilisent des fonctions issuent de la classe mère, ainsi que les fonctions membres virtuelles pures de la classes mères, qui sont codés différement en fonction de la classe fille dans laquelle on se situe. Ces classes filles possèdent en plus le membre privé **id**, qui permet de les identifier et donc cela permet le code de certains fonctions. Nous verrons, plus loin dans ce rapport, l'implémentation de ces fonctions.

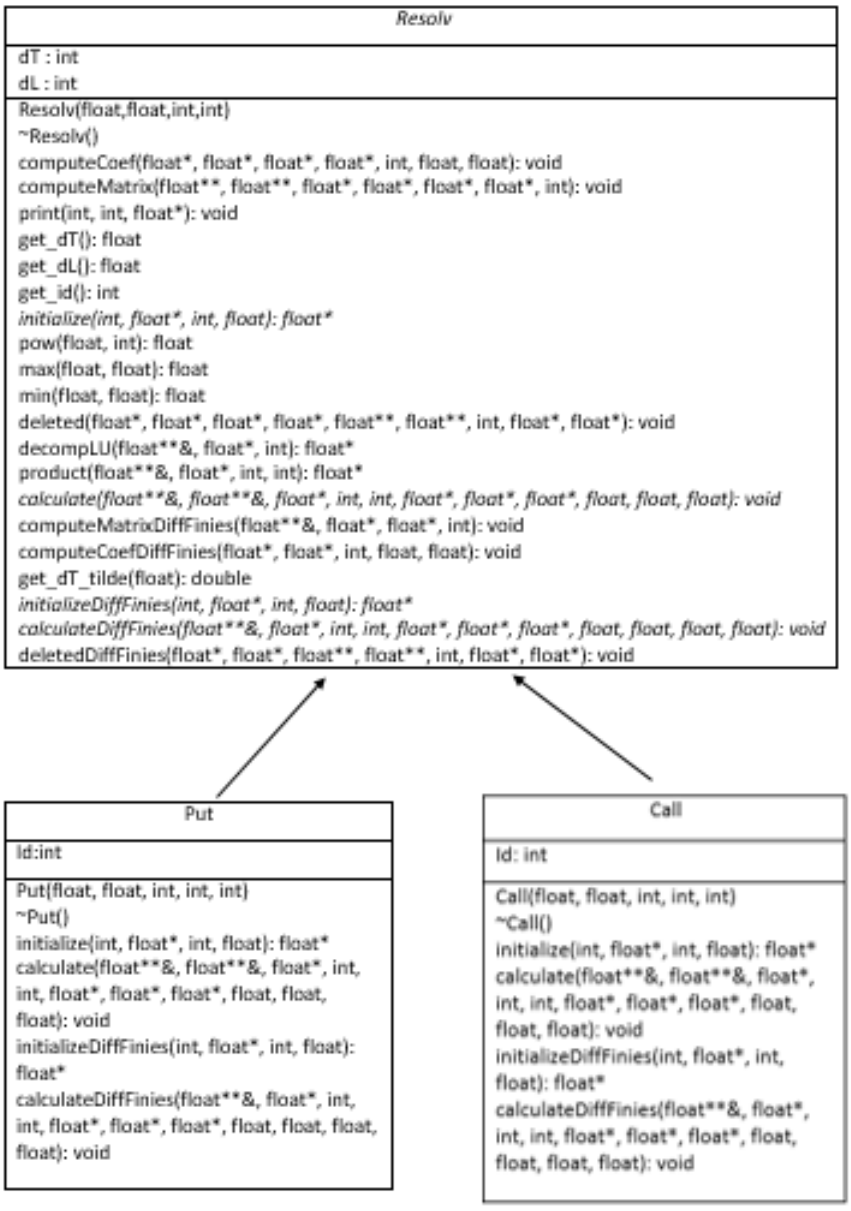

Figure 1: Diagramme UML des classes

2.1.2 Difficultés rencontrées

Nous n'avons pas rencontré de réelles difficultés dans cette partie. Le plus dur a été de bien voir quelles fonctions allaient changées en fonction des payoffs et donc quelles fonctions membres de la classe mère allaient être virtuelles pures. Ensuite, la deuxième difficulté a été le constructeur des différentes classes. Nous avons fait le choix de ne pas mettre vecteur et matrices dans les membres des classes. Avec du recul, cela aurait peut être été un meilleur choix de les inclure dedans et ainsi, cela aurait évité de codé une fonction **deleted** et tout aurait été inclu dans le destructeur de la classe.

2.2 Implémentation de la méthode Crank Nicholson

2.2.1 Fonction calculate de la classe Put

Dans cette partie, nous allons parler principalement des fonctions **calculate** et **calculateDiffFinies**. Ceux sont les fonctions qui ont étés les plus difficiles à coder et elles doivent être expliquées. Tout d'abord, concernant la fonction **calculate**. Elle prend en arguments **2 matrices (float **), 3 vecteurs (float *), 2 entiers (int) et 3 floats**. En effet, on a vu précedemment qu'avec cette méthode, nous devons résoudre $Ax_{n+1} + K = Bx_n$. Nous avons donc un vecteur x_n, un vecteur k et le 3eme vecteur correspond au vecteur des coefficients situés sous la diagonale de la matrice A: le premier coefficient est pris en compte avec les conditions initiales! Les 2 entiers correspondent aux tailles des matrices carrées et les floats sont des données de l'énoncé.

```
//Permet de calculer à chaque étape le nouveau vecteur V_N en connaissant V_N+1
void Put::calculate(float** M1, float** M2, float* V, int M, int N, float* c1, float* k, float* vector1, float K, float r, float T) {
    std::ofstream fichier;
    fichier.open("result_put.txt", std::ios::out);
    for (int i = N; i > 0; i--) {
        vector1 = product(M1, vector1, M, M);
        //utilisation des conditions initiales pour un put
        k[0] = c1[0] * (K * exp(-r * (T - (i - 1) * get_dT())) + K * exp(-r * (T - i * get_dT())));
        for (int s = 1; s < M; s++) {
            k[s] = 0.0;
        }
        for (int j = 0; j < M; j++) vector1[j] = vector1[j] + k[j];
        vector1 = Put::decompLU(M2, vector1, M);
        std::cout << (i - 1) * get_dT() << "|";
        for (int j = 0; j < M; j++) {
            std::cout << vector1[j] << ";";
            fichier << vector1[j] << ";";
        }
        std::cout << "0.00 " << std::endl;
        fichier << "\n" << std::endl;
    }
    fichier.close();
}
```

Figure 2: Fonction Calculate

Nous devons écrire les solutions du système dans un fichier texte pour pouvoir plot les résultats donc il nous est nécessaire d'ouvrir un fichier (ligne 1

Algorithm 1 Resolution de $A * V_{n+1} + k = B * V_n$

for $N > 0$ **do**
  $vector1 \leftarrow A * V_{n+1}$
  $vector1 \leftarrow vector1 + k$
  $vector1 \leftarrow solution\ de\ B * V_n = vector1$
end for

et 2). Ensuite, note méthode de calcul est la suivante: nous calculons vecteur par vecteur et remplacant V_{n+1} par V_n à chaque itération. Ainsi, nous gardons en mémoire que le vecteur actuel et non les précédents. Donc, lors de l'appel de la fonction dans le **main.cpp**, **vector1** entré en fonction est initalisé précedemment avec la fonction **initialize**. Ce vecteur possède donc en mémoire V_{n+1}.
Nous allons donc calculer chaque vecteur V_n, d'où la boucle for. Dans cette boucle, on commence par calculer le produit $M1 * V_{n+1}$, avec la fonction **product**. Cette fonction renvoie un vecteur, donc vector1 prend cette valeur. On sais qu'on doit lui ajouter le vecteur k. On défini donc le vecteur k. Pour un put, il est nul sauf pour le premier coefficent, qui a pour valeur $a_0 * K * (e^{-r*(T-(i-1)*dT)} + e^{-r*(T-i*dT)})$ (voir explication dans la partie théorique). Une fois ce vecteur crée, on l'ajoute a vector1. On a donc le système $M2 * V_n = vector1$. Il suffit donc de résoudre ce système via une décomposition LU. Nous obtenons donc V_n, dont la valeur est enregistrée dans vector1. Nous écrivons les coefficients de vector1 dans le fichier texte puis nous réiterons via la boucle for.

2.2.2 Fonction calculate de la classe Call

Le schéma est identique à la fonction de la classe Put. La seul différence intervient au niveau des conditions initiales ie au niveau du vecteur k. Le vecteur k est nul sauf sur le dernier coefficient qui vaut $c_M * K * (e^{-r*(-T+(i-1)*dT)} + e^{-r*(-T+i*dT)})$

2.3 Implémentation de la méthode des différences finies

2.3.1 Fonction calculateDiffFinies de la classe Put

```cpp
void Put::calculateDiffFinies(float** M1, float* V, int M, int N, float* c1, float* k, float* vector1, float K, float r, float T, float sigma) {
    std::ofstream fichier;
    fichier.open("result_put_DF.txt", std::ios::out);
    for (int i = N; i > 0; i--) {
        vector1 = product(M1, vector1, M, M);
        //utilisation des conditions initiales pour un put
        k[0] = c1[0] * (K * exp(-r * (static_cast<float>(T) - (static_cast<float>(i) - static_cast<float>(1)) *
            static_cast<float>(get_dT_tilde(sigma)))) + K * exp(-r * (static_cast<float>(T) - static_cast<float>(i) * static_cast<float>(get_dT_tilde(sigma)))));
        for (int s = 1; s < M; s++) {
            k[s] = 0.0;
        }
        for (int j = 0; j < M; j++) vector1[j] = vector1[j] + k[j];
        std::cout << (i - 1) * get_dT() << "|";
        for (int j = 0; j < M; j++) {
            std::cout << vector1[j] << ";";
        }
        std::cout << std::endl;
        std::cout << std::endl;
        fichier << "\n" << std::endl;
    }
    for (int j = 0; j < M; j++) {
        fichier << vector1[j] << ";";
    }
    fichier << "\n" << std::endl;
    fichier.close();
}
```

Figure 3: CalculateDiffFinies pour un Put

Algorithm 2 Resolution de $A * V_{n+1} + k = V_n$

for $N > 0$ **do**
  $vector1 \leftarrow A * V_{n+1}$
  $vector1 \leftarrow vector1 + k$
end for

Concernant la fonction **calculateDiffFinies**. Elle prend en arguments **1 matrice (float **), 3 vecteurs (float *), 2 entiers (int) et 3 floats**. En effet, on a vu précedemment qu'avec cette méthode, nous devons résoudre $Ax_{n+1} + K = x_n$. Nous avons donc un vecteur x_n, un vecteur k et le 3eme vecteur correspond au vecteur des coefficients situés sous la diagonale de la matrice A: le premier coefficient est pris en compte avec les conditions initiales! Les 2 entiers correspondent aux tailles des matrices carrées et les floats sont des données de l'énoncé. Nous connaissons X_{n+1} donc pour avoir X_n, il suffit d'effectuer le produit matriciel puis d'ajouter les conditions aux limites! L'avantage de cette méthode est qu'elle est bien plus rapide que la précedente. En effet, il n'y a pas necessité de réaliser une décomposition LU. Le taux de complexité de l'algorithme est plus faible. La différence supplémentaire est l'interval de temps dT. Celui ci a été dilaté. On a désormais : $dT' = \frac{dT}{2} * sigma^2$, d'où la fonction **get_dT_tilde**. Les conditions initiales restent inchangées, hormis l'apparition de ce nouveau dT. Nous écrivons toujours les résultats dans un fichier texte pour pouvoir plot les résultats.

2.3.2 Fonction calculateDiffFinies de la classe Call

Comme pour la méthode de Crank Nicholson, l'implémentation de la fonction **calculateDiffFinies** ne change que très peu en fonction des payoff. La seul différence se situe au niveau du vecteur k (les conditions aux limites).

3. Affichage des résultats

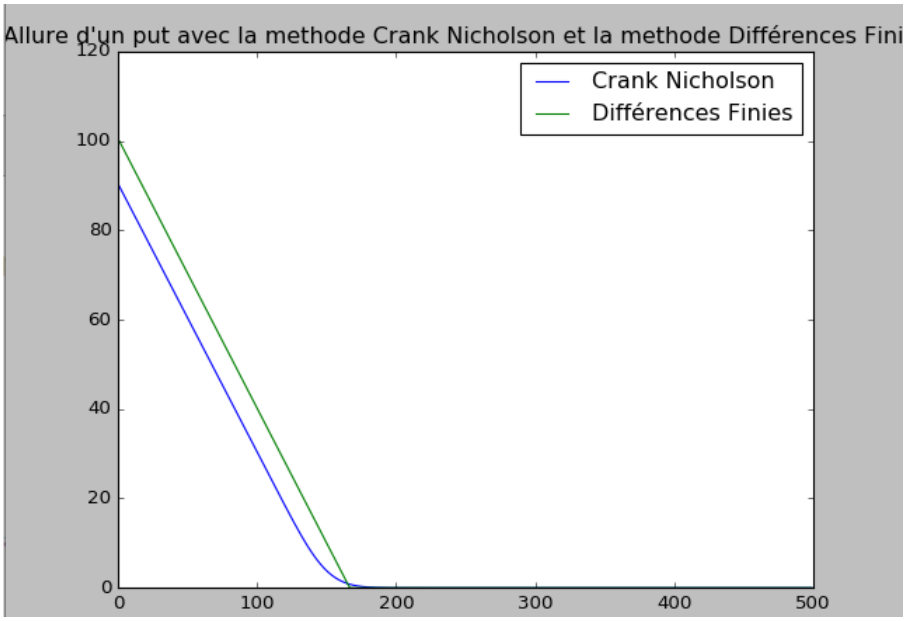

Figure 4: Allure pour un Put

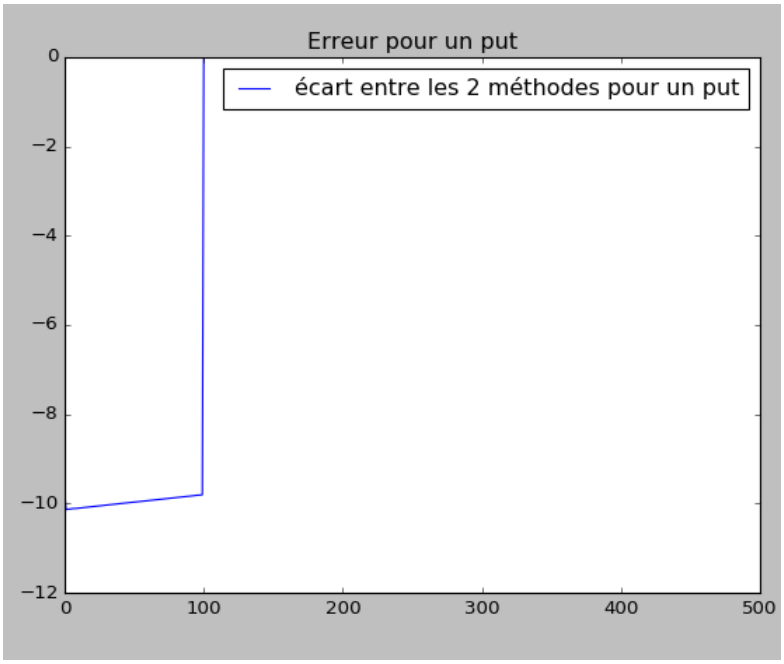

Figure 5: Erreur entre les 2 méthodes

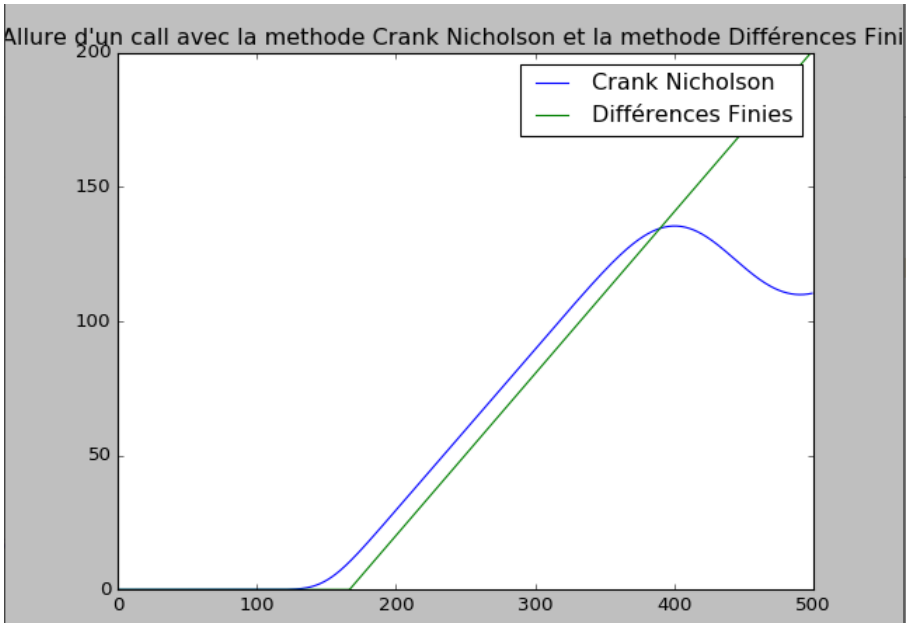

Figure 6: Allure pour un Call

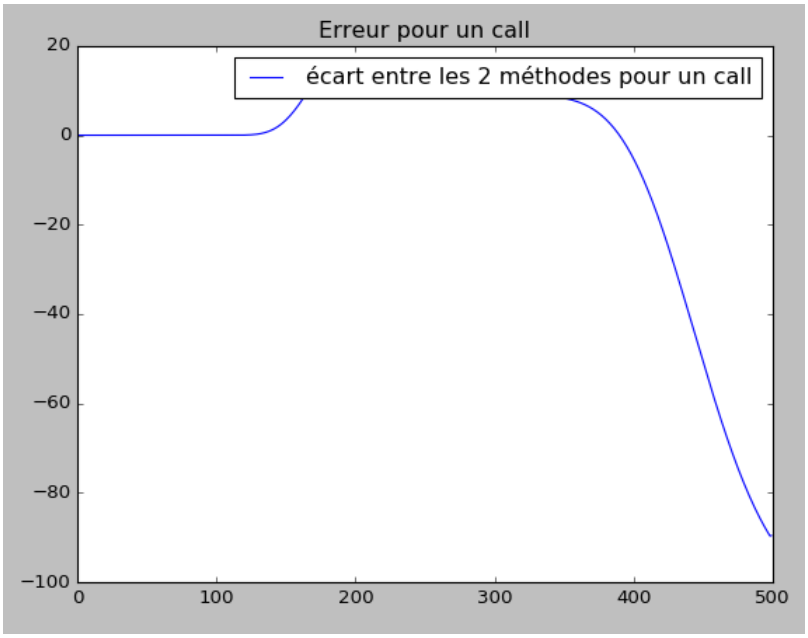

Figure 7: Erreur entre les 2 méthodes

A la vue des résultats, on remarque que les 2 méthodes sont assez similaires (à l'exception de la fin d'un call, nous ne savons pas comment expliquer cette "queue" pour la méthode de Crank Nicholson). Bien sur, la méthode de Crank Nicholson donne de meilleurs résultats mais, elle est bien plus complexe d'un point de vue algorithme. Il faut une décomposition LU, qui prend du temps à être calculée. On peut d'ailleurs facilement le voir lorsqu'on lance le programme pour N=M=500. Pour la méthode de Crank Nicholson, nous obtenons les résultats en environ 5min. Pour la méthode des différences finies, les résultats tombent en 1min environ. Pour privilégier une méthode ou non, il faut alors faire un compromis entre temps de calcul et cohérence des résultats.

4. Améliorations possibles et difficultés lors de ce projet

4.1 Difficultés rencontrées

Tout d'abord, évoquons les difficultés rencontrées lors de ce projet. Nous avons rencontré beaucoup de problèmes de mémoires (oublis de création d'espace ou oublies de remplissage des matrices). Ainsi, lors des premiers calculs, nous avions de "fausses valeurs" (**de l'ordre de e+34**). Cela est dû au fait que nos matrices n'étaient pas entièrement remplies et cela engendrait l'apparition ce genre de valeurs. Nous avons aussi eu des problèmes en ce qui concerne

les classes abstraites. En effet, certains points étaient encore un peu flou mais ce projet nous a permis de mieux comprendre et de maîtriser pleinement les classes abstraites en C++. Nous n'avons pas réussi à utiliser **std::chrono** ni **std::thread**. C'est un point sur lequel nous avons buté durant tout le long du projet. De plus, nous avons eu énormement de warnings concernant des possibles pertes de données: **'int' tranformsé en 'float',...**. Pour remédier à cela, nous avons utilisé **static_cast**. Nous avons ainsi transformé int en float et donc ces warnings ont étés fixés.

Concernant la partie théorique et les résultats, nous avons eu des difficultés sur la méthode des différences finies. En effet, nous ne savons pas si la méthode de calcul employée a été la bonne. Mais, les résultats des affichages semblent montrer que la méthode de calcul est assez fiable. Pour les résultats, nous sommes asseez surpris quand a l'allure de notre call. Nous ne trouvons pas d'explications à la fin de courbe, hormis peut être une "saturation": on ne peut pas aller à l'infini d'où cette chute. Mais alors, il y a une différence entre les deux méthodes: une tend vers l'infini tandis que l'autre stagne.

4.2 Améliorations possibles

En ce qui concerne les améliorations du projet, la majeure amélioration concerne le temps de calcul de l'algorithme. En effet, ce dernier dure plus d'une heure pour N=M=1000, ce qui est bien trop important. C'est d'ailleurs pourquoi nous avons travaillé avec **N=M=500**, le temps de calcul est réduit à une poigné de minutes (environ 5min pour la méthode de Crank Nicholson et environ une minute pour la méthode des différences finies). De plus, notre projet n'est codé que pour des matrices carrées i.e. **il est nécessaire d'avoir N=M** pour le bon fonctionnement du projet. C'est un point qui peut être intéressant d'améliorer car nous avons remarqué que la valeur de notre vecteur final dépend principalement de S et non de T. Donc, en travaillant avec le même dL mais un dT plus petit (i.e. N différent de M), les résultats seraient identiques et la durée de l'algorithme serait bien plus faible.