RAPPORT DE PROJET C++

Pricing d'Options par Résolution d'EDP
Modèle de Black-Scholes & Schéma de Crank-Nicolson

Auteurs :
Yléo AUBRY
Yepi Sylvian Jean-Mederick BESSI
Ayman GRAZIANA

Date de rendu :
29 décembre 2025

Résumé

Ce projet implémente un solveur numérique en C++ pour l'évaluation d'options européennes (Call et Put) dans le cadre du modèle de Black-Scholes. L'approche retenue repose sur la résolution de l'Équation aux Dérivées Partielles (EDP) par la méthode des différences finies, utilisant un schéma θ-généralisé (incluant Crank-Nicolson) couplé à une transformation logarithmique du sous-jacent. Le projet met l'accent sur une architecture logicielle robuste, modulaire et conforme aux standards professionnels modernes (C++17, CMake, Tests Unitaires séparés), garantissant précision numérique et extensibilité.

Table des matières

1 Introduction

L'évaluation des produits dérivés est un pilier de la finance quantitative. Si des formules fermées existent pour les cas simples (formule de Black-Scholes pour les options européennes), la résolution numérique devient indispensable pour des produits plus complexes ou lorsque les hypothèses du modèle varient. Ce projet a pour objectif de construire un "Pricer" robuste basé sur la résolution de l'EDP de Black-Scholes par la méthode des différences finies.

Nous avons choisi une approche rigoureuse, tant sur le plan mathématique (stabilité du schéma de Crank-Nicolson, changement de variable) que logiciel (architecture orientée objet, séparation Interface/Implémentation, gestion de la mémoire, tests automatisés). Ce rapport détaille exhaustivement la modélisation mathématique, l'architecture du code, et fournit un guide d'utilisation complet.

2 Analyse mathématique détaillée

Cette section établit formellement les résultats mathématiques implémentés dans le moteur de calcul.

2.1 Du modèle à l'EDP

Définition : Dynamique du sous-jacent

Sous la mesure risque-neutre $\mathbb{Q}$, le prix de l'actif risqué S_t suit un Mouvement Brownien Géométrique décrit par l'équation différentielle stochastique (EDS) :

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{1}$$

où r est le taux sans risque constant, σ la volatilité constante, et W_t un mouvement brownien standard sous $\mathbb{Q}$.

En construisant un portefeuille de réplication $\Pi_t = V(S_t, t) - \Delta S_t$ et en imposant l'absence d'opportunité d'arbitrage ($d\Pi_t = r\Pi_t dt$), on aboutit par application du Lemme d'Itô à l'équation fondamentale.

Théorème : EDP de Black-Scholes

Soit $V(S, t)$ le prix d'une option européenne de maturité T. V satisfait l'équation aux dérivées partielles parabolique :

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} - rV = 0 \tag{2}$$

avec la condition terminale $V(S, T) = \text{Payoff}(S)$.

2.2 Transformation logarithmique (Heat Equation)

La résolution directe sur S est numériquement instable car les coefficients rS et $\sigma^2 S^2$ dépendent de l'espace. Nous effectuons le changement de variable $x = \ln(S)$. Soit $U(x, t) = V(e^x, t)$.

Démonstration

Calculons les dérivées partielles via la règle de la chaîne :

$$\frac{\partial V}{\partial S} = \frac{\partial U}{\partial x}\frac{\partial x}{\partial S} = \frac{1}{S}\frac{\partial U}{\partial x}$$

$$\frac{\partial^2 V}{\partial S^2} = \frac{\partial}{\partial S}\left(\frac{1}{S}\frac{\partial U}{\partial x}\right) = -\frac{1}{S^2}\frac{\partial U}{\partial x} + \frac{1}{S^2}\frac{\partial^2 U}{\partial x^2}$$

En injectant ces expressions dans l'EDP de Black-Scholes :

$$\frac{\partial U}{\partial t} + rS\left(\frac{1}{S}\frac{\partial U}{\partial x}\right) + \frac{1}{2}\sigma^2 S^2\left(\frac{1}{S^2}\frac{\partial^2 U}{\partial x^2} - \frac{1}{S^2}\frac{\partial U}{\partial x}\right) - rU = 0$$

En simplifiant les termes S et S^2, on obtient une EDP à coefficients constants (Convection-Diffusion) :

$$\frac{\partial U}{\partial t} + \left(r - \frac{\sigma^2}{2}\right)\frac{\partial U}{\partial x} + \frac{\sigma^2}{2}\frac{\partial^2 U}{\partial x^2} - rU = 0 \tag{3}$$

Cette transformation est cruciale pour notre implémentation car elle permet de **pré-calculer** les matrices du système linéaire une seule fois (dans `PDESolver::precomputeMatrices`), rendant le code performant.

2.3 Discrétisation et schéma θ

Nous travaillons sur une grille spatio-temporelle uniforme en log-price :
— Temps : $t_m = m\Delta t$ pour $m \in \{0, \dots, M\}$.
— Espace : $x_n = x_{\min} + n\Delta x$ pour $n \in \{0, \dots, N-1\}$.
Nous utilisons le schéma θ-généralisé qui relie le pas de temps m au pas $m+1$:

$$\frac{U_n^{m+1} - U_n^m}{\Delta t} + \mathcal{L}_h(\theta U^{m+1} + (1-\theta)U^m) = 0$$

Propriété : Schéma de Crank-Nicolson

Pour $\theta = 0.5$, le schéma est appelé Crank-Nicolson. Il est inconditionnellement stable et convergent à l'ordre $O(\Delta t^2 + \Delta x^2)$. C'est le schéma par défaut de notre solveur.

Cela mène au système linéaire matriciel :

$$\mathbf{A}U^m = \mathbf{B}U^{m+1} + \text{Conditions limites} \tag{4}$$

Où $\mathbf{A}$ et $\mathbf{B}$ sont des matrices tridiagonales.

2.4 Algorithme de Thomas

La résolution du système $Ax = d$ est effectuée par l'algorithme de Thomas (TDMA). C'est une forme optimisée de l'élimination de Gauss pour les matrices tridiagonales.

Démonstration

Soit le système $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$. L'algorithme se déroule en deux étapes :

1. **Descente (Forward Elimination) :** On modifie les coefficients pour éliminer le terme sous-diagonal a_i.

$$c'_i = \frac{c_i}{b_i - a_i c'_{i-1}}, \quad d'_i = \frac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}}$$

2. **Remontée (Backward Substitution) :** On résout le système triangulaire supérieur obtenu.

$$x_N = d'_N, \quad x_i = d'_i - c'_i x_{i+1}$$

La complexité est linéaire $O(N)$, ce qui est indispensable pour la performance.

3 Architecture logicielle du projet

L'architecture a été pensée pour être modulaire, extensible et lisible. Nous avons strictement séparé la bibliothèque de calcul (Core), l'interface utilisateur (App) et les tests de validation.

3.1 Arborescence des Fichiers

Voici la structure exacte du dossier rendu, conforme aux standards CMake :

```
Projet_EDP/
|-- CMakeLists.txt              # Configuration globale du projet
|-- include/
|   |-- edp/                    # Headers publics (Namespace edp)
|       |-- LinearSolver.h      # Déclaration algo de Thomas
|       |-- Payoff.h            # Classe abstraite Payoff
|       |-- PDESolver.h         # Moteur de résolution
|       |-- Interface.h         # Gestion I/O Console
|-- src/                        # Code Source de la Librairie (EDP_Core)
|   |-- LinearSolver.cpp        # Implémentation mathématique
|   |-- PDESolver.cpp           # Implémentation numérique
|   |-- Interface.cpp           # Implémentation UI
|-- app/
|   |-- main.cpp                # Exécutable final (Pricer)
|   |-- CMakeLists.txt
|-- tests/                      # Tests de validation séparés
    |-- Test_LinearSolver.cpp   # Benchmark algo Thomas
    |-- Test_PDESolver.cpp      # Validation vs Black-Scholes
    |-- CMakeLists.txt
```

3.2 Explication du code

3.2.1 Le cœur mathématique : `LinearSolver`

Ce module ne dépend d'aucune notion financière. Il implémente l'algorithme de Thomas.

- **Optimisation :** Nous passons les vecteurs par référence constante (`const std::vector<double>`) pour éviter toute copie inutile en mémoire, crucial quand N est grand.
- **Robustesse :** Le code vérifie la taille des vecteurs et lance des exceptions (`std::invalid_argume`) si les dimensions sont incohérentes.

3.2.2 La Logique Financière : `Payoff`

Nous utilisons le polymorphisme pour gérer différents types d'options sans dupliquer le code du solveur.

```
// Extrait de include/edp/Payoff.h
class Payoff {
public:
    virtual ~Payoff() = default;
    virtual double operator()(double S) const = 0;
};
```

Le solveur prend une référence `const Payoff&`. Qu'il s'agisse d'un Call, d'un Put ou d'une option exotique, le solveur appellera simplement `payoff(S)`. C'est le design pattern **Strategy**.

3.2.3 Le Moteur : `PDESolver`

C'est la classe principale.

- **Constructeur :** Il calcule le pas de temps dt et le pas d'espace dx.
- **precomputeMatrices() :** Cette méthode privée construit les matrices tridiagonales A et B avant la boucle temporelle. Comme les coefficients sont constants (log-transform), cela évite de recalculer les matrices à chaque étape de temps, un gain de performance majeur.
- **Conditions aux limites :** Elles sont calculées dynamiquement en interrogeant l'objet `Payoff`.

3.2.4 4. L'application : `app/main.cpp`

Ce fichier est le point d'entrée. Il orchestre le tout en utilisant des pointeurs intelligents pour la gestion automatique de la mémoire.

```
// Gestion m moire moderne (C++17)
std::unique_ptr<edp::Payoff> payoff;
if (ui.getIsCall()) {
    payoff = std::make_unique<edp::PayoffCall>(ui.getK());
}
```

4 Stratégie de test et validation

La fiabilité d'un pricer est critique. Nous avons mis en place une suite de tests automatisés via des exécutables dédiés.

4.1 Mode 1 : Suite de tests (Benchmark)

Ce mode lance une batterie de tests unitaires destinée à valider la fiabilité et la cohérence des différents composants du moteur de pricing :

4.1.1 Validation du solveur linéaire tridiagonal

Le bon fonctionnement de l'algorithme de Thomas est vérifié sur une série de systèmes tridiagonaux, avec des coefficients légèrement variables pour tester la robustesse numérique. Pour chaque test, une solution exacte est construite analytiquement (méthode du problème inverse), puis comparée à la solution numérique obtenue par le solveur. Les résultats sont exportés au format CSV, incluant la solution exacte, la solution calculée et l'erreur associée.

4.1.2 Test de Pricing par EDP (Convergence)

Le pricer basé sur la résolution de l'équation de Black–Scholes par un schéma de Crank–Nicolson est validé en comparant le prix numérique d'options européennes avec la formule analytique fermée de Black–Scholes. Pour un ensemble de 20 cas de marché (Call/Put, ITM/OTM/ATM, haute/basse volatilité), le prix théorique, le prix obtenu par EDP ainsi que l'erreur absolue sont calculés.

Résultats observés : Nous obtenons systématiquement une erreur absolue inférieure à 10^{-2} (1 centime) pour une grille standard ($N = 250, M = 2500$). L'erreur diminue quadratiquement lorsque l'on raffine le maillage, confirmant l'ordre 2 du schéma de Crank-Nicolson.

5 Critique et Solutions

5.1 Problèmes rencontrés

1. Instabilité numérique (Oscillations) : Lors des premiers tests avec un schéma purement explicite, nous avons observé des oscillations pour certaines valeurs de volatilité. *Solution :* Implémentation du schéma θ-généralisé. L'utilisation par défaut de Crank-Nicolson ($\theta = 0.5$) offre le meilleur compromis stabilité/précision.

2. Gestion des conditions aux limites : Fixer arbitrairement 0 ou S aux bords empêchait de pricer correctement les Puts. *Solution :* Utilisation de la valeur intrinsèque actualisée du Payoff aux bornes, permettant de traiter Calls et Puts de manière unifiée.

3. Compilation sous Windows : L'environnement Windows ne fournit pas nativement les outils standards ('make'). *Solution :* Mise en place d'un système de build **CMake** robuste, capable de détecter et configurer les compilateurs (MinGW, MSVC) et de générer les exécutables de manière portable.

5.2 Optimisations réalisées

Nous avons implémenté le stockage des matrices tridiagonales sous forme de 3 vecteurs (diagonale, sous-diagonale, sur-diagonale) plutôt qu'une matrice dense $N \times N$, réduisant l'empreinte mémoire de $O(N^2)$ à $O(N)$.

6 Guide d'utilisation

Ce guide explique étape par étape comment compiler et utiliser le logiciel.

6.1 Pré-requis

Vous devez avoir installé sur votre machine :
— **CMake** (version > 3.10).
— Un compilateur C++ (G++ sous Linux/MinGW sous Windows, ou Clang).
— Un terminal (PowerShell, Bash ou Cmd).

6.2 Méthode 1 : Depuis un Terminal (Recommandée)

Cette méthode fonctionne universellement (Windows, Mac, Linux).

Étape 1 : Préparation Placez-vous à la racine du dossier `Projet_EDP`. Créez le dossier de compilation pour ne pas polluer les sources :

```
mkdir build
cd build
```

Étape 2 : Configuration du projet Demandez à CMake de générer les fichiers de construction (Makefiles).

— Sous Linux / MacOS :

```
cmake ..
```

— Sous Windows (avec MinGW installé dans msys64) :

```
cmake -G "MinGW Makefiles" \
-D CMAKE_MAKE_PROGRAM=C:/msys64/mingw64/bin/mingw32-make.exe ..
```

Étape 3 : Compilation Lancez la compilation de l'ensemble du projet (Librairie + App + Tests) :

```
cmake --build .
```

Si tout se passe bien, la barre de progression atteint 100%.

Étape 4 : Exécution Les exécutables sont générés dans le sous-dossier `bin/`.
— Pour lancer le Pricer : `./bin/PricerApp`
— Pour lancer les Tests : `./bin/Test_PDESolver`

6.3 Méthode 2 : Depuis Visual Studio Code

Si vous avez l'extension *CMake Tools* installée :

1. Ouvrez le dossier `Projet_EDP` dans VSCode.
2. Acceptez la configuration automatique du kit (choisissez GCC/G++).
3. Cliquez sur le bouton **Build** dans la barre bleue en bas de l'écran.
4. Pour exécuter, cliquez sur l'icône de lecture ("Play") en bas, après avoir sélectionné la cible `PricerApp` ou `Test_PDESolver`.

7 Conclusion

Dans ce projet, nous avons mis l'accent sur la rigueur mathématique ainsi que sur l'architecture du code. La séparation claire des responsabilités permet une maintenance aisée, et la couverture de tests garantit la fiabilité des résultats financiers produits.