

PROJET

Valorisation et couverture d'options en C++

Manuel Griseri

Master 1 : Mathématiques et Interactions

Année 2024/2025

Table des matières

Introduction	2
1 Modèle CRR et options vanille	3
1.1 Modèle CRR	3
1.1.1 Modélisation discrète des marchés financiers	3
1.1.2 Modèle binomial	5
1.2 Valorisation et couverture d'options vanille dans le modèle CRR	7
1.2.1 Valorisation et couverture d'options	8
1.2.2 Valorisation et couverture CRR	9
1.2.3 Implémentation informatique	10
2 Modèle CRR et options exotiques	33
2.1 Options asiatiques	33
2.1.1 Valorisation et couverture	33
2.1.2 Implémentation informatique	34
2.2 Options américaines	41
2.2.1 Valorisation et couverture	41
2.2.2 Implémentation informatique	43
3 Convergence du modèle CRR	49
3.1 Convergence vers le modèle de Black-Scholes	49
3.1.1 Convergence	49
3.1.2 Vitesse de convergence	53
3.2 Convergence des options exotiques	55
3.2.1 Méthode Monte Carlo	55
3.2.2 Extrapolation répétée de Richardson	58
Conclusion	61
A D'autres options européennes	62
A.1 Options digitales	62
A.2 Bull et bear spread	66
A.3 Strangle et butterfly	70
B Modélisation en temps continu	75
B.1 Modèle Black–Scholes	75
B.2 Modèle à volatilité locale	92
Bibliographie	94

Introduction

La valorisation et la couverture des options jouent un rôle fondamental dans l'industrie financière contemporaine. Elles permettent non seulement d'évaluer avec précision le prix des produits dérivés, mais aussi de maîtriser les risques liés à la fluctuation des marchés. Face à la complexité croissante des instruments financiers, la modélisation mathématique, et notamment le calcul stochastique, offre des cadres rigoureux pour décrire l'évolution aléatoire des actifs. Cette description théorique impose de développer des méthodes numériques performantes, traduisibles en algorithmes optimisés, afin de garantir qu'un ordinateur puisse obtenir les valeurs recherchées en un temps minimal. Par ailleurs, la programmation assure la mise en œuvre, le test et l'automatisation de ces calculs, répondant ainsi aux exigences opérationnelles des établissements de gestion et de trading. D'où l'importance de savoir écrire un code propre, factorisé, abstrait, évolutif et reproductible.

Ce projet se concentre sur l'implémentation en C++ du modèle binomial de Cox-Ross-Rubinstein (CRR) pour la valorisation et la couverture d'options européennes, américaines et asiatiques. Les routines C++ sont compilées en DLL et exposées à Excel via VBA, offrant à l'utilisateur une interface conviviale et modulable. L'approche allie l'explicitation théorique du modèle, la présentation des équations et des démonstrations, et l'analyse expérimentale des résultats. Chaque section sera assortie d'extraits de code C++ et de captures d'écran Excel illustrant les performances obtenues.

Le chapitre 1 présente le cadre CRR appliqué aux options vanilles. Après avoir décrit la construction du modèle discret des marchés, nous détaillerons la valorisation itérative et la stratégie de couverture. Le chapitre 2 traite des options exotiques : asiatiques et américaines. Pour chacune, nous expliquerons les adaptations algorithmiques nécessaires, comme le traitement de l'exercice anticipé ou de la moyenne de prix. Le chapitre 3 étudie la convergence du modèle CRR vers le modèle de Black-Scholes. Nous analyserons la convergence et sa vitesse pour les options vanille et exotiques.

Une première annexe présente d'autres types d'options ou de stratégies d'options européennes (digitales, spreads, strangles et butterflies), complétant ainsi le panorama de la valorisation discrète des dérivés financiers. Une deuxième annexe prolongera le cadre discret analysé dans le projet principal en présentant les principaux modèles continus des marchés financiers : Black-Scholes et volatilité locale. Elle comprendra une étude approfondie des équations aux dérivées partielles sous-jacentes et de leur résolution numérique par schémas aux différences finies, entièrement implémentée en C++ et toujours accessible via une interface Excel.

Chapitre 1

Modèle CRR et options vanille

Dans la section 1.1, nous décrivons d'abord un cadre général de modélisation d'un marché financier à dates discrètes. Nous détaillons ensuite le cas particulier de l'arbre binomial, où le prix de l'actif ne peut qu'augmenter ou diminuer à chaque pas de temps. Enfin, nous présentons le modèle de Cox-Ross-Rubinstein.

La section 1.2 applique ce formalisme à la valorisation et à la couverture des options vanille, d'abord dans un modèle discret générique, puis de façon concrète sous le modèle CRR, grâce à des équations et à leur implémentation en C++ pour calculer prix et stratégie réplicante.

1.1 Modèle CRR

Soit d le nombre d'actifs risqués et un actif sans risque, et $N + 1$ le nombre de dates considérées. Pour borner les prix des actifs, on fixe une constante $M > 0$ (par exemple la richesse mondiale). On modélise alors l'ensemble des réalisations possibles par

$$\Omega = ([0, M] \cap \mathbb{N})^{(N+1)(d+1)} \quad (1.1)$$

où \mathbb{N} apparaît car les prix sont observés à la précision d'un tick (par exemple 0.0001 sur le Forex). Cette construction assure que Ω est un ensemble fini, évitant ainsi tout problème théorique d'intégrabilité.

On se donne une mesure de probabilité

$$P: \mathcal{P}(\Omega) \longrightarrow [0, 1] \quad (1.2)$$

appelée probabilité historique, qui attribue à chaque sous-ensemble d'issues une probabilité.

Enfin, pour formaliser l'information croissante dans le temps, on introduit une filtration

$$\mathbb{F} = (\mathcal{F}_n)_{0 \leq n \leq N} \quad (1.3)$$

où

$$\mathcal{F}_0 = \{\emptyset, \Omega\}, \quad \mathcal{F}_n \subseteq \mathcal{F}_{n+1}, \quad \mathcal{F}_N = \mathcal{P}(\Omega)$$

1.1.1 Modélisation discrète des marchés financiers

Définition 1.1 (Marché financier). Un marché financier est un espace de probabilité filtré (Ω, P, \mathbb{F}) non redondant, où Ω , P et \mathbb{F} sont ceux définis en (1.1), (1.2) et (1.3).

Par «espace non redondant», on entend que

$$\forall \omega \in \Omega, \quad P(\{\omega\}) > 0 \quad (1.4)$$

ce qui signifie qu'on exclut du modèle les trajectoires impossibles, toutes les égalités presque sûres deviennent alors de simples égalités. Ce modèle discret reproduit mieux la réalité qu'un modèle continu : la résolution temporelle la plus fine observée sur les marchés «ultra-low-latency» est de l'ordre de la nanoseconde, ce qui reste extrêmement petit mais toujours discret. Sur ce marché, on définit l'actif sans risque

$$S^0 = (S_n^0)_{0 \leq n \leq N}$$

par la relation

$$S_n^0 = (1+r)^n \quad (1.5)$$

où r est un taux d'intérêt constant, hypothèse justifiée par la tendance des banques centrales à ne pas modifier trop drastiquement leur taux directeur sur des périodes relativement courtes. Pour chaque $i = 1, \dots, d$, on considère un processus stochastique adapté à la filtration $\mathbb{F} = (\mathcal{F}_n)_{0 \leq n \leq N}$:

$$S^i = (S_n^i)_{0 \leq n \leq N}$$

c'est-à-dire que, pour chaque $n = 0, 1, \dots, N$, S_n^i est \mathcal{F}_n -mesurable. Chaque S_n^i modélise le prix à la date n de l'actif risqué i .

Pour la suite, introduisons quelques autres définitions. On appelle portefeuille un processus

$$\phi = (\phi_n)_{1 \leq n \leq N}$$

à valeurs dans \mathbb{R}^{d+1} , tel que pour tout $n \geq 1$, ϕ_n soit \mathcal{F}_{n-1} -mesurable (donc \mathbb{F} -prévisible). La composante ϕ_n^i (pour $i = 0, \dots, d$) modélise la quantité de l'actif i détenue sur l'intervalle $[n-1, n]$. La valeur du portefeuille ϕ à l'instant n est

$$V_n(\phi) = \sum_{i=0}^d \phi_n^i S_n^i = \langle \phi_n, S_n \rangle \quad (1.6)$$

Sa valeur actualisée est définie par

$$\tilde{V}_n(\phi) = \frac{V_n(\phi)}{S_n^0} \quad (1.7)$$

Un portefeuille ϕ est dit autofinancé si et seulement si

$$\forall n \in \llbracket 1, N-1 \rrbracket, \quad \langle \phi_n, S_n \rangle = \langle \phi_{n+1}, S_n \rangle \quad (1.8)$$

Définition 1.2 (Stratégie admissible). Le processus ϕ est une stratégie dite admissible si et seulement si

- i) ϕ est autofinancé,
- ii) $V_N(\phi) \geq 0$.

Définition 1.3 (Stratégie d'arbitrage). ϕ est une stratégie d'arbitrage si et seulement si

- i) ϕ est admissible,
- ii) $V_0(\phi) = 0$,
- iii) $P(V_N(\phi) > 0) > 0$.

Le concept d'arbitrage est fondamental en économie et en finance pour le processus de «price discovery» des actifs. De plus, comme on le verra, l'hypothèse d'absence d'arbitrage permet de valoriser correctement les options.

Définition 1.4 (Marché viable). Un marché est dit viable s'il n'existe aucune stratégie d'arbitrage.

Théorème 1.1 (Théorème fondamental de la finance). Un marché discret est viable si et seulement s'il existe une probabilité Q équivalente à P telle que le processus $\tilde{S} = (\tilde{S}_n)_{0 \leq n \leq N}$, avec $\tilde{S}_n = S_n / S_n^0$, soit une (\mathbb{F}, Q) -martingale.

Ce théorème revêt une importance fondamentale : il affirme que l'absence de stratégie d'arbitrage est équivalente à l'existence d'une probabilité équivalente sous laquelle les actifs actualisés sont des martingales (appelée «probabilité risque-neutre» en finance). On renvoie à Lamberton and Lapeyre (1997, pp. 17–19) pour la preuve complète.

1.1.2 Modèle binomial

Un cas concret et fondamental de modèle de marché en temps discret est le modèle binomial. Ce modèle est à la fois simple à comprendre et à implémenter et sert de référence standard pour comparer des modèles plus élaborés. Il reflète également le fonctionnement des carnets d'ordres des marchés financiers réels.

Définition 1.5 (Modèle binomial). Un marché financier (Ω, P, \mathbb{F}) suit un modèle binomial à $N + 1$ dates, de paramètres d, r, u avec $u \geq d > -1$, s'il comporte deux actifs :

- un actif sans risque défini comme en (1.5);
- un actif risqué S tel que, pour tout $n = 0, \dots, N - 1$,

$$S_{n+1} = S_n (1 + U_{n+1})$$

où $(U_n)_{1 \leq n \leq N}$ est une suite de variables aléatoires distribuées suivant une loi de Bernoulli à valeurs dans $\{d, u\}$.

On en déduit immédiatement qu'au noeud (n, k) de l'arbre (avec k montées et $n - k$ descentes),

$$S_n(k) = S_0 (1 + u)^k (1 + d)^{n-k}, \quad k = 0, 1, \dots, n \quad (1.9)$$

Théorème 1.2 (Viabilité du marché binomial). Le marché binomial est viable si et seulement si les accroissements $(U_n)_{1 \leq n \leq N}$ sont indépendants et identiquement distribués sous une probabilité martingale Q telle que

$$Q(U_n = u) = p = \frac{r - d}{u - d}$$

et, de ce fait, on a nécessairement

$$d < r < u$$

Démonstration :

\implies

Supposons qu'il existe une probabilité risque-neutre Q . Alors, pour tout $n \in \{0, \dots, N-1\}$,

$$\mathbb{E}_Q[\tilde{S}_{n+1} \mid \mathcal{F}_n] = \tilde{S}_n$$

Or $\tilde{S}_{n+1} = S_{n+1}/S_{n+1}^0$ et $S_{n+1}^0 = S_n^0(1+r)$, d'où

$$\mathbb{E}_Q[S_{n+1} \mid \mathcal{F}_n] = S_n(1+r)$$

Comme $S_{n+1} = S_n(1+U_{n+1})$ et que S_n est \mathcal{F}_n -mesurable, on obtient

$$\mathbb{E}_Q[U_{n+1} \mid \mathcal{F}_n] = r$$

et par déconditionnement,

$$\mathbb{E}_Q[U_{n+1}] = r$$

Posons $p_n = Q(U_n = u)$. Alors

$$\mathbb{E}_Q[U_{n+1}] = p_{n+1}u + (1-p_{n+1})d$$

d'où

$$p_{n+1} = \frac{r-d}{u-d} \quad \forall n$$

Notons $p = p_{n+1}$. Ainsi, les U_n sont identiquement distribués selon une loi de Bernoulli de paramètre p . De plus, on a

$$S_n = S_0 \prod_{i=1}^n (1+U_i) \implies \mathcal{F}_n = \sigma(U_1, \dots, U_n)$$

Comme $\mathbb{E}_Q[U_{n+1} \mid \sigma(U_1, \dots, U_n)] = r = \mathbb{E}_Q[U_{n+1}]$, les U_n sont également indépendants sous Q . Enfin, $0 < p < 1$ entraîne $d < r < u$.

\iff

La réciproque se démontre en construisant la probabilité martingale Q et en vérifiant que p est le seul paramètre possible.

■

Enfin, nous disposons de tous les éléments nécessaires pour introduire le modèle CRR. Ce modèle est un cas particulier du modèle binomial dont l'idée consiste à réduire le nombre de paramètres de quatre à trois. En effet, au lieu de spécifier (S_0, r, d, u) , on ne considère que (S_0, r, σ)

où $\sigma > 0$ est la volatilité de l'actif risqué. Cette volatilité est étroitement liée à la variance du logarithme naturel de la valeur de l'actif et est estimée statistiquement au moyen d'estimateurs; toutefois, l'objectif de ce chapitre n'est pas de détailler les méthodes d'estimation de la volatilité. Pour comprendre le modèle CRR, il convient de fixer une maturité T et un nombre de pas N , ce dernier représentant la date finale (choisie par le modélisateur souhaitant discréteriser le temps). Dans un cadre en temps continu, on dispose d'un taux d'intérêt continu annuel R et l'on impose que, à maturité, le facteur de capitalisation discret coïncide avec le facteur continu :

$$(1+r)^N = e^{RT} \quad (1.10)$$

On en déduit

$$r = e^{\frac{RT}{N}} - 1 = \frac{RT}{N} + o\left(\frac{1}{N}\right)$$

par approximation de Taylor au premier ordre. On choisit donc

$$r = \frac{RT}{N}, \quad \sigma_N = \sigma \sqrt{\frac{T}{N}} \quad (1.11)$$

Par ailleurs, on fixe essentiellement $p = \frac{1}{2}$ de sorte que

$$r = \frac{u+d}{2}, \quad \sigma_N = \frac{u-d}{2} \quad (1.12)$$

De ces relations, il suit

$$u = r + \sigma_N = \frac{RT}{N} + \sigma \sqrt{\frac{T}{N}}, \quad d = r - \sigma_N = \frac{RT}{N} - \sigma \sqrt{\frac{T}{N}} \quad (1.13)$$

Définition 1.6 (Modèle de Cox-Ross-Rubinstein). Étant donnée σ la volatilité de l'actif risqué, R le taux continu annualisé et T la maturité, un marché financier suit un modèle CRR à $N+1$ dates s'il suit un modèle binomial à $N+1$ dates avec

$$r = \frac{RT}{N}, \quad u = \frac{RT}{N} + \sigma \sqrt{\frac{T}{N}}, \quad d = \frac{RT}{N} - \sigma \sqrt{\frac{T}{N}}$$

1.2 Valorisation et couverture d'options vanille dans le modèle CRR

Dans cette section, nous commençons par définir ce qu'est un actif contingent européen et les options vanille.

Définition 1.7 (Actif contingent européen). Un actif contingent européen de maturité T est une variable aléatoire $H \geq 0$, \mathcal{F}_T -mesurable.

Une option vanille call de strike K et de maturité T est l'actif contingent européen défini par

$$H = (S_T - K)^+ \quad (1.14)$$

Une option vanille put de strike K et de maturité T est l'actif contingent européen défini par

$$H = (K - S_T)^+ \quad (1.15)$$

1.2.1 Valorisation et couverture d'options

Définition 1.8 (Actif contingent répliable). Un actif contingent H est dit répliable s'il existe une stratégie admissible ϕ telle que $V_N(\phi) = H$.

Théorème 1.3 (Prix d'une option européenne). Si H est répliable et le marché viable, alors un prix de couverture de H à l'instant $n \leq N$ est

$$P_n = \mathbb{E}_Q[\tilde{H} | \mathcal{F}_n], \quad \text{où} \quad \tilde{H} = \frac{S_n^0}{S_N^0} H$$

Démonstration :

Puisque H est répliable, il existe une stratégie admissible ϕ telle que $V_N(\phi) = H$. Le prix de couverture de H à l'instant n est $V_n(\phi)$. Or, le processus $(\tilde{V}_n(\phi))_{0 \leq n \leq N}$ est une martingale sous Q , d'où

$$\tilde{V}_n(\phi) = \mathbb{E}_Q[\tilde{V}_N(\phi) | \mathcal{F}_n]$$

■

En particulier, à l'instant initial $n = 0$ on obtient

$$P_0 = \mathbb{E}_Q[\tilde{H}] \tag{1.16}$$

Nous n'aborderons pas ici la question de la complétude du marché (le modèle CRR étant toujours complet), c'est-à-dire l'existence éventuelle de plusieurs probabilités martingales équivalentes. Dans ces situations, on peut soit recourir à la sur-couverture :

$$P_0 = \sup_{Q \in \mathcal{M}} \mathbb{E}_Q[\tilde{H}] \tag{1.17}$$

où \mathcal{M} est l'ensemble des probabilités risque-neutres, soit rendre le marché complet en ajoutant de nouveaux actifs risqués. De manière plus générale, pour valoriser une option européenne dans un modèle de marché discret quelconque (décrivant l'évolution des actifs risqués), on procède en deux étapes :

1. On établit le système linéaire issu des contraintes de martingale sur les prix actualisés des actifs, ce qui permet de déterminer la ou les probabilités risque-neutres Q .
2. Si le système n'admet aucune solution, le marché n'est pas viable, ce qui signale l'existence d'une opportunité d'arbitrage. Sinon, on calcule le prix de l'option en évaluant les espérances conditionnelles.

Nous avons ainsi obtenu un prix auquel le trader peut vendre l'option. Il reste à élaborer une stratégie de couverture dynamique, c'est-à-dire déterminer à chaque pas n les quantités ϕ_n^0 (actif sans risque) et ϕ_n (sous-jacent) à détenir pour garantir à maturité le paiement du payoff. On suppose ici un seul sous-jacent, la méthode étant le delta hedging. Le portefeuille de couverture est autofinancé, d'où, en notant $\tilde{V}_n = \mathbb{E}_Q[\tilde{H} | \mathcal{F}_n]$, le système :

$$\begin{cases} \tilde{V}_n = \phi_n^0 + \phi_n \tilde{S}_n \\ \tilde{V}_{n-1} = \phi_n^0 + \phi_n \tilde{S}_{n-1} \end{cases}$$

En calculant et projetant sur \mathcal{F}_{n-1} , on obtient la stratégie :

$$\begin{cases} \phi_n = \mathbb{E}_Q \left[\frac{\tilde{V}_n - \tilde{V}_{n-1}}{\tilde{S}_n - \tilde{S}_{n-1}} \mid \mathcal{F}_{n-1} \right] \\ \phi_n^0 = \tilde{V}_{n-1} - \phi_n \tilde{S}_{n-1} \end{cases} \quad (1.18)$$

1.2.2 Valorisation et couverture CRR

Nous passons maintenant à la valorisation CRR. Nous calculons l'espérance conditionnelle pour déterminer le prix d'une option vanille dans un modèle binomial. On pose

$$V_n = \mathbb{E}_Q[\tilde{H} \mid \mathcal{F}_n] = \mathbb{E}_Q \left[\frac{1}{(1+r)^{N-n}} f(S_N) \mid \mathcal{F}_n \right]$$

où f est la fonction de payoff définie par le contrat (par exemple (1.14) pour les calls et (1.15) pour les puts). Il suit que

$$(1+r)^{N-n} V_n = \mathbb{E}_Q \left[f \left(S_n \prod_{k=n+1}^N (1+U_k) \right) \mid \mathcal{F}_n \right]$$

On utilise la propriété suivante de l'espérance conditionnelle : si X et Y sont indépendants alors

$$\mathbb{E}[g(X, Y) \mid \sigma(X)] = \psi(X), \quad \psi(x) = \mathbb{E}[g(x, Y)]$$

Ici, prenons

$$X = S_n, \quad Y = (U_{n+1}, \dots, U_N)$$

qui sont indépendants de \mathcal{F}_n . Alors

$$(1+r)^{N-n} V_n = \psi(S_n), \quad \psi(x) = \mathbb{E}_Q[g(x, Y)]$$

avec

$$g(x, Y) = f \left(x \prod_{k=n+1}^N (1+U_k) \right)$$

On obtient ainsi

$$\begin{aligned} \psi(x) &= \sum_{y \in \{d,u\}^{N-n}} g(x, y) Q(Y = y) = \sum_{y \in \{d,u\}^{N-n}} g(x, y) \prod_{i=1}^{N-n} Q(U_{n+i} = y_i) \\ &= \sum_{y \in \{d,u\}^{N-n}} g(x, y) p^{\#\{i:y_i=u\}} (1-p)^{\#\{i:y_i=d\}} = \sum_{k=0}^{N-n} \binom{N-n}{k} f \left(x (1+u)^k (1+d)^{N-n-k} \right) p^k (1-p)^{N-n-k} \end{aligned}$$

Deux observations importantes :

- i) Dans le modèle CRR, on a $p = \frac{1}{2}$, ce qui simplifie la formule de valorisation.
- ii) La formule fermée avec les factorielles (dans les coefficients binomiaux) est peu adaptée à une implémentation rapide sur ordinateur. On lui préfère une programmation dynamique en deux phases :

- Phase d'initialisation des valeurs terminales :

$$\psi_N(x) = f(x) \quad (1.19)$$

où ψ_n désigne la fonction de valorisation au pas n et $x \in \mathbb{R}$.

- Phase de rétropropagation : Pour $n = N - 1, N - 2, \dots, 0$,

$$\psi_n(x) = \frac{1}{1+r} \frac{1}{2} \left(\psi_{n+1}(x(1+u)) + \psi_{n+1}(x(1+d)) \right) \quad (1.20)$$

En ce qui concerne la couverture dans un modèle CRR, on aboutit là aussi à une formule simple. À partir de (1.18), on obtient pour la position sur le sous-jacent au pas $n + 1$:

$$\phi_{n+1} = \mathbb{E}_Q \left[\frac{\psi_{n+1}(S_{n+1}) - (1+r)\psi_n(S_n)}{S_{n+1} - (1+r)S_n} \mid \mathcal{F}_n \right] = \gamma(S_n)$$

puisque S_n est \mathcal{F}_n -mesurable et U_{n+1} indépendant de \mathcal{F}_n , où

$$\gamma(x) = \mathbb{E}_Q \left[\frac{\psi_{n+1}(x(1+U_{n+1})) - (1+r)\psi_n(x)}{x(U_{n+1} - r)} \right]$$

En développant selon la loi de Bernoulli :

$$\gamma(x) = p \frac{\psi_{n+1}(x(1+u)) - (1+r)\psi_n(x)}{x(u-r)} + (1-p) \frac{\psi_{n+1}(x(1+d)) - (1+r)\psi_n(x)}{x(d-r)}$$

qui, dans le modèle CRR ($p = \frac{r-d}{u-d} = \frac{1}{2}$), se simplifie et on obtient donc, pour la stratégie de couverture :

$$\phi_{n+1} = \frac{\psi_{n+1}(S_n(1+u)) - \psi_{n+1}(S_n(1+d))}{S_n(u-d)} \quad (1.21)$$

qui est bien \mathcal{F}_n -mesurable.

1.2.3 Implémentation informatique

Passons maintenant à l'implémentation en C++ de ce qui a été expliqué jusqu'à présent. Tout d'abord, les classes représentant les payoffs pour différents types d'options ont été définies. Dans cette section, nous nous intéresserons uniquement aux payoffs des options vanille. Pour consulter les payoffs des autres options, se reporter à l'Annexe A.

Listing 1.1: Payoff.h

```

1 #ifndef PAYOFF_H
2 #define PAYOFF_H
3
4 #include <stdexcept>
5
6 namespace opt {
7
8     /**
9      * @brief Classe abstraite représentant le payoff d'une option.
10     */
11    class Payoff {

```

```

12 |     public:
13 |     /**
14 |      * @brief Calcule la valeur du payoff pour un prix  $S$  du sous-
15 |      * jacent.
16 |      * @param  $S$  Prix du sous-jacent à maturité.
17 |      * @return Valeur du payoff.
18 |     */
19 |     virtual double operator()(double S) const = 0;
20 |
21 | };
22 |
23 | /**
24 |  * @brief Payoff d'une option d'achat européenne (call).
25 |  */
26 | class PayoffCall : public Payoff {
27 | private:
28 |     double K_;
29 |
30 | public:
31 |     /**
32 |      * @param  $K$  Prix d'exercice de l'option.
33 |     */
34 |     PayoffCall(double K);
35 |
36 |     double operator()(double S) const override;
37 | };
38 |
39 | /**
40 |  * @brief Payoff d'une option de vente européenne (put).
41 |  */
42 | class PayoffPut : public Payoff {
43 | private:
44 |     double K_;
45 |
46 | public:
47 |     PayoffPut(double K);
48 |     double operator()(double S) const override;
49 | };
50 |
51 | /**
52 |  * @brief Payoff d'un call digital.
53 |  */
54 | class PayoffDigitCall : public Payoff {
55 | private:
56 |     double K_;
57 |
58 | public:
59 |     PayoffDigitCall(double K);
60 |     double operator()(double S) const override;
61 | };
62 |
63 | /**
64 |  * @brief Payoff d'un put digital.

```

```

63     */
64     class PayoffDigitPut : public Payoff {
65     private:
66         double K_;
67
68     public:
69         PayoffDigitPut(double K);
70         double operator()(double S) const override;
71     };
72
73 /**
74 * @brief Payoff d'une option double-digital.
75 */
76 class PayoffDoubleDigit : public Payoff {
77     private:
78         double K1_;
79         double K2_;
80
81     public:
82     /**
83     * @param K1 Prix d'exercice inférieur.
84     * @param K2 Prix d'exercice supérieur.
85     */
86     PayoffDoubleDigit(double K1, double K2);
87
88     double operator()(double S) const override;
89 };
90
91 /**
92 * @brief Payoff d'une option bull spread.
93 */
94 class PayoffBull : public Payoff {
95     private:
96         double K1_;
97         double K2_;
98
99     public:
100    PayoffBull(double K1, double K2);
101    double operator()(double S) const override;
102 };
103
104 /**
105 * @brief Payoff d'une option bear spread.
106 */
107 class PayoffBear : public Payoff {
108     private:
109         double K1_;
110         double K2_;
111
112     public:
113        PayoffBear(double K1, double K2);
114        double operator()(double S) const override;

```

```

115    };
116
117    /**
118     * @brief Payoff d'une option strangle.
119     */
120    class PayoffStrangle : public Payoff {
121    private:
122        double K1_;
123        double K2_;
124
125    public:
126        PayoffStrangle(double K1, double K2);
127        double operator()(double S) const override;
128    };
129
130    /**
131     * @brief Payoff d'une option butterfly.
132     */
133    class PayoffButterfly : public Payoff {
134    private:
135        double K1_;
136        double K2_;
137
138    public:
139        PayoffButterfly(double K1, double K2);
140        double operator()(double S) const override;
141    };
142
143 } // namespace opt
144
145 #endif // PAYOFF_H

```

Nous pouvons remarquer que la principale technique utilisée dans ce bloc de code est l'héritage, qui apporte d'importants avantages dans l'écriture du code. Dans ce cas, nous disposons d'une classe mère abstraite `Payoff`. Mère, car toutes les autres classes de payoff spécifiques héritent d'elle les méthodes; abstraite, car elle définit un opérateur () virtuel pur. Cela présente les avantages suivants :

- Créer des hiérarchies logiques : même si les payoffs sont différents, ils appartiennent tous au même ensemble et possèdent un opérateur () qui sera défini dans chaque classe dérivée.
- Permettre d'étendre la fonctionnalité de la classe mère.
- Factorisation : ce n'est pas le cas ici, mais nous verrons plus tard qu'on peut définir dans la classe mère des méthodes communes à toutes les classes filles, ce qui réduit la duplication et simplifie le code.
- Polymorphisme : on définit une interface commune sans implémentation concrète, de sorte que le code client peut traiter des objets de classes différentes comme s'ils étaient du même type, sans se soucier de leur implémentation spécifique.

Il est également intéressant de noter que nous représentons des payoffs génériques, mais toujours pour des options à un seul sous-jacent; c'est pourquoi, dans la classe mère, l'argument de l'opérateur

`()` est un unique `double`. En revanche, rien ne nous empêche de traiter des options multi-strike : les différents strikes sont alors passés en argument au constructeur spécifique de chaque payoff. Maintenant, nous insérons également le fichier `Payoff.cpp` dans lequel sont définies les méthodes déclarées dans `Payoff.h`. Nous retrouverons les formules (1.14) et (1.15) dans les définitions de l'opérateur `()` de `PayoffCall` et `PayoffPut`, respectivement.

Listing 1.2: `Payoff.cpp`

```

1 #include "pch.h"
2 #include "Payoff.h"
3
4 namespace opt {
5
6     PayoffCall::PayoffCall(double K)
7         : K_(K)
8     {
9         if (K_ < 0.0)
10             throw std::invalid_argument("Strike K doit être non-négatif");
11     }
12
13     double PayoffCall::operator()(double S) const {
14         return std::max<double>(S - K_, 0.0);
15     }
16
17     PayoffPut::PayoffPut(double K)
18         : K_(K)
19     {
20         if (K_ < 0.0)
21             throw std::invalid_argument("Strike K doit être non-négatif");
22     }
23
24     double PayoffPut::operator()(double S) const {
25         return std::max<double>(K_ - S, 0.0);
26     }
27
28     PayoffDigitCall::PayoffDigitCall(double K)
29         : K_(K)
30     {
31         if (K_ < 0.0)
32             throw std::invalid_argument("Strike K doit être non-négatif");
33     }
34
35     double PayoffDigitCall::operator()(double S) const {
36         return (S > K_) ? 1.0 : 0.0;
37     }
38
39     PayoffDigitPut::PayoffDigitPut(double K)
40         : K_(K)
41     {
42         if (K_ < 0.0)
```

```

43     throw std::invalid_argument("Strike K doit être non-négatif"
44         );
45 }
46
47 double PayoffDigitPut::operator()(double S) const {
48     return (S < K_) ? 1.0 : 0.0;
49 }
50
51 PayoffDoubleDigit::PayoffDoubleDigit(double K1, double K2)
52 : K1_(K1), K2_(K2)
53 {
54     if (K1_ < 0.0 || K2_ < 0.0 || K1_ > K2_)
55         throw std::invalid_argument("Strike incorrect");
56 }
57
58 double PayoffDoubleDigit::operator()(double S) const {
59     return (S > K1_ && S < K2_) ? 1.0 : 0.0;
60 }
61
62 PayoffBull::PayoffBull(double K1, double K2)
63 : K1_(K1), K2_(K2)
64 {
65     if (K1_ < 0.0 || K2_ < 0.0 || K1_ > K2_)
66         throw std::invalid_argument("Strike incorrect");
67 }
68
69 double PayoffBull::operator()(double S) const {
70     return (S < K1_) ? 0.0 : ((S > K2_) ? K2_ - K1_ : S - K1_);
71 }
72
73 PayoffBear::PayoffBear(double K1, double K2)
74 : K1_(K1), K2_(K2)
75 {
76     if (K1_ < 0.0 || K2_ < 0.0 || K1_ > K2_)
77         throw std::invalid_argument("Strike incorrect");
78 }
79
80 double PayoffBear::operator()(double S) const {
81     return (S < K1_) ? K2_ - K1_ : ((S > K2_) ? 0 : K2_ - S);
82 }
83
84 PayoffStrangle::PayoffStrangle(double K1, double K2)
85 : K1_(K1), K2_(K2)
86 {
87     if (K1_ < 0.0 || K2_ < 0.0 || K1_ > K2_)
88         throw std::invalid_argument("Strike incorrect");
89 }
90
91 double PayoffStrangle::operator()(double S) const {
92     return (S < K1_) ? K1_ - S : ((S > K2_) ? S - K2_ : 0);
93 }
```

```

94     PayoffButterfly::PayoffButterfly(double K1, double K2)
95         : K1_(K1), K2_(K2)
96     {
97         if (K1_ < 0.0 || K2_ < 0.0 || K1_ > K2_)
98             throw std::invalid_argument("Strike incorrect");
99     }
100
101    double PayoffButterfly::operator()(double S) const {
102        return (S > K1_ && S <= 0.5 * (K1_ + K2_)) ? S - K1_ : ((S > 0.5
103            * (K1_ + K2_)) && S < K2_) ? K2_ - S : 0);
104    }
105}

```

Nous allons maintenant exposer les fichiers dédiés aux options. Comme nous souhaitons essentiellement traiter trois types d'options (européennes, asiatiques et américaines), il est pertinent d'utiliser l'héritage pour définir une classe abstraite `Option` qui représente une option générique. Cette classe comportera des méthodes virtuelles pures pour :

- Construire l'arbre de valorisation.
- Calculer la stratégie de couverture (retournée sous la forme d'une `struct` contenant à la fois la position sur le sous-jacent et sur l'actif sans risque).

Les méthodes concrètes fournies par `Option` incluent :

- Un accès au prix initial.
- Un accès au δ initial.

ce qui permet une factorisation du code. Le constructeur de `Option` vérifie que les paramètres du modèle CRR passés en entrée sont dans leur domaine de validité, puis calcule u , d et le facteur d'actualisation. Ces valeurs calculées sont alors accessibles et réutilisables par toutes les classes dérivées.

Listing 1.3: Option.h

```

1 #ifndef OPTION_H
2 #define OPTION_H
3
4 #include <vector>
5 #include <cmath>
6 #include <random>
7 #include <stdexcept>
8
9 namespace crr {
10
11 /**
12  * @brief Classe de base Option : calcul des paramètres du modèle.
13  */
14 class Option {
15 protected:
16     double S0_, R_, sigma_, T_; ///< Paramètres initiaux
17     int N_; ///< Nombre de pas

```

```

18     double u_, d_, discount_ ;      ///< Paramètres par pas
19
20 public:
21     /**
22      * @brief Constructeur : initialise les paramètres de l'option
23      * pour le modèle CRR.
24      * @param S0      Prix initial du sous-jacent.
25      * @param R       Taux d'intérêt sans risque continu.
26      * @param sigma   Volatilité annuelle du sous-jacent.
27      * @param T       Durée jusqu'à l'échéance en années.
28      * @param N       Nombre de pas de l'arbre binomial.
29      */
30     Option(double S0, double R, double sigma, double T, int N);
31
32     /**
33      * @brief Arbre des prix de l'option selon le modèle CRR.
34      * @return Matrice des valeurs aux noeuds.
35      */
36     virtual std::vector<std::vector<double>> treePrice() const = 0;
37
38     /**
39      * @brief Prix initial de l'option.
40      * @return Valeur de l'option à n = 0.
41      */
42     double price() const;
43
44     /**
45      * @brief Stratégie de couverture : delta et obligation.
46      */
47     struct HedgingStrategy {
48         std::vector<std::vector<double>> delta;    ///< Positions en
49                                     sous-jacent.
50         std::vector<std::vector<double>> bond;    ///< Positions en
51                                     actif sans risque.
52     };
53
54     /**
55      * @brief Calcule la couverture optimale (CRR).
56      * @return HedgingStrategy contenant delta et bond par noeud.
57      */
58     virtual HedgingStrategy hedgingStrategy() const = 0;
59
60     /**
61      * @brief Delta initial.
62      * @return Valeur du delta à n = 0.
63      */
64     double deltaZero() const;
65 };
66
67 } // namespace crr
68
69 #endif // OPTION_H

```

Listing 1.4: Option.cpp

```

1 #include "pch.h"
2 #include "Option.h"
3
4 namespace crr {
5
6     Option::Option(double S0, double R, double sigma, double T, int N)
7         : S0_(S0), R_(R), sigma_(sigma), T_(T), N_(N)
8     {
9         if (S0_ < 0.0)      throw std::invalid_argument("S0 doit être >= 0");
10        if (sigma_ < 0.0)  throw std::invalid_argument("Sigma doit être >= 0");
11        if (T_ < 0.0)       throw std::invalid_argument("T doit être >= 0");
12        if (N_ <= 0)        throw std::invalid_argument("N doit être > 0");
13        ;
14
15        double dt = T_ / N_;
16        double sq = sigma_ * std::sqrt(dt);
17        double rN = R_ * dt;
18        u_ = rN + sq;
19        d_ = rN - sq;
20        discount_ = 1.0 / (1.0 + rN);
21    }
22
23    double Option::price() const {
24        return treePrice()[0][0];
25    }
26
27    double Option::deltaZero() const {
28        return hedgingStrategy().delta[0][0];
29    }
30} // namespace crr

```

Nous exposerons le fichier de la classe des options européennes dans lequel nous définissons essentiellement les méthodes de valorisation et de couverture, qui ne sont autres que l'algorithme de rétropropagation de (1.20) et la formule de (1.21).

Listing 1.5: European.h

```

1 #ifndef EUROPEAN_H
2 #define EUROPEAN_H
3
4 #include "Option.h"
5
6 namespace crr {
7
8     /**
9      * @brief Option européenne CRR.
10     * @tparam TPayoff Type de payoff.
11     */

```

```

12  template<typename TPayoff>
13  class European : public Option {
14  private:
15      TPayoff payoff_;
16      std::vector<std::vector<double>> stockTree_; // < Arbre
17          binomiale recombinant des prix du sous-jacent
18  void buildStockTree(); // < Génère l'
19          arbre binomial recombinant des prix du sous-jacent
20
21  public:
22      European(double S0, double R, double sigma, double T, int N,
23          const TPayoff& payoff);
24      std::vector<std::vector<double>> treePrice() const override;
25      HedgingStrategy hedgingStrategy() const override;
26  };
27
28  template<typename TPayoff>
29  European<TPayoff>::European(double S0, double R, double sigma,
30      double T, int N, const TPayoff& payoff)
31      : Option(S0, R, sigma, T, N), payoff_(payoff)
32  {
33      buildStockTree();
34  }
35
36  template<typename TPayoff>
37  void European<TPayoff>::buildStockTree() {
38      stockTree_.assign(N_ + 1, {});
39      for (int n = 0; n <= N_; ++n) {
40          stockTree_[n].resize(n + 1);
41          for (int i = 0; i <= n; ++i) {
42              stockTree_[n][i] = S0_ * std::pow(1 + u_, i) * std::pow
43                  (1 + d_, n - i);
44          }
45      }
46  }
47
48  template<typename TPayoff>
49  std::vector<std::vector<double>> European<TPayoff>::treePrice()
50  const {
51      std::vector<std::vector<double>> V(N_ + 1);
52      V[N_].resize(N_ + 1);
53      // Valeurs terminales
54      for (int i = 0; i <= N_; ++i)
55          V[N_][i] = payoff_(stockTree_[N_][i]);
56
57      // Rétropropagation
58      for (int n = N_ - 1; n >= 0; --n) {
59          V[n].resize(n + 1);
60          for (int i = 0; i <= n; ++i) {
61              V[n][i] = discount_ * 0.5 * (V[n + 1][i + 1] + V[n + 1][
62                  i]);
63          }
64      }
65  }

```

```

58     }
59     return V;
60 }
61
62 template<typename TPayoff>
63 Option::HedgingStrategy European<TPayoff>::hedgingStrategy() const {
64     auto V = treePrice();
65     HedgingStrategy H;
66     H.delta.resize(N_);
67     H.bond.resize(N_);
68     for (int n = 0; n < N_; ++n) {
69         H.delta[n].resize(n + 1);
70         H.bond[n].resize(n + 1);
71         for (int i = 0; i <= n; ++i) {
72             double Vu = V[n + 1][i + 1];
73             double Vd = V[n + 1][i];
74             double Su = stockTree_[n + 1][i + 1];
75             double Sd = stockTree_[n + 1][i];
76             double dlt = (Vu - Vd) / (Su - Sd);
77             H.delta[n][i] = dlt;
78             H.bond[n][i] = V[n][i] - dlt * stockTree_[n][i];
79         }
80     }
81     return H;
82 }
83
84 } // namespace crr
85
86 #endif // EUROPEAN_H

```

Outre l'héritage, une autre technique très utile a été utilisée : le template. En C++, le template est un autre exemple de polymorphisme, permettant d'écrire un code très abstrait et générique auquel une classe spécifique est attribuée au moment de l'instanciation. Dans notre cas, nous définissons comme template le type du membre `payoff_`, et nous utilisons librement l'opérateur `()`. Le compilateur n'émet aucune erreur car `payoff_` n'est pas d'un type concret. C'est ensuite dans le `main` que nous donnerons un type précis à `TPayoff`, le transformant en l'une des classes spécifiques déclarées dans `Payoff.h`.

Pour disposer d'une interface Excel permettant de visualiser les arbres binomiaux, deux stratégies sont possibles. La première consiste à générer la sortie via la fonction `main` et à l'enregistrer dans un fichier CSV. Cependant, cette méthode est inefficace car elle ne permet pas l'utilisation directe des fonctions en Excel et oblige à relancer le code à chaque modification des données. La seconde approche, celle que nous avons retenue, repose sur l'utilisation d'une Dynamic Link Library (DLL) qui exporte les fonctions C++ vers VBA. Nous illustrerons cette méthode pour les fonctions liées aux options vanilles; nous l'omettrons dans la suite, car la procédure est toujours la même :

- Déclarer les fonctions dans `Exports.h`.
- Définir les fonctions dans `Exports.cpp`.

Il est nécessaire d'inclure des bibliothèques spécifiques pour gérer le transfert des matrices, qui requiert le type `VARIANT`. La première partie de `Exports.cpp` n'est pas réellement importante : elle se limite à factoriser des blocs de code similaires pour éviter les redondances et à résoudre un

problème d'affichage multiple des exceptions lors de l'exécution de plusieurs fonctions depuis Excel. Il est intéressant de noter que dans `Exports.cpp`, on assigne au template un type concret : dès lors, le membre `payoff_` de `European.h` est du type spécifié et l'opérateur () correspondant est correctement résolu.

Listing 1.6: `Exports.h`

```

1 #ifndef EXPORTS_H
2 #define EXPORTS_H
3
4 #include "Payoff.h"
5 #include "Option.h"
6 #include "European.h"
7 #include <windows.h> // MessageBoxA
8 #include <comdef.h> // VARIANT
9 #include <OleAuto.h> // SAFEARRAY
10
11 extern "C" {
12
13     //=====
14     // Vanilla Call
15     //=====
16
17     /**
18      * @brief Calcule le prix d'un call européen.
19      */
20     __declspec(dllexport) double __stdcall PriceEuCall(
21         double S0, double R, double sigma, double T, int N, double K
22     );
23
24     /**
25      * @brief Calcule le delta d'un call européen.
26      */
27     __declspec(dllexport) double __stdcall DeltaEuCall(
28         double S0, double R, double sigma, double T, int N, double K
29     );
30
31     /**
32      * @brief Renvoie la matrice 2D V[n][i] du call vanille en VARIANT
33      * SAFEARRAY.
34      */
35     __declspec(dllexport) VARIANT __stdcall TreeEuCall(
36         double S0, double R, double sigma, double T, int N, double K
37     );
38
39     /**
40      * @brief Renvoie la matrice des deltas du call vanille en VARIANT
41      * SAFEARRAY.
42      */
43     __declspec(dllexport) VARIANT __stdcall TreeDeltaEuCall(
44         double S0, double R, double sigma, double T, int N, double K
45     );
46
47 }
```

```

45  /**
46   * @brief Renvoie la matrice des bonds du call vanille en VARIANT
47   * SAFEARRAY.
48   */
49 __declspec(dllexport) VARIANT __stdcall TreeBondEuCall(
50     double S0, double R, double sigma, double T, int N, double K
51 );
52 //=====
53 // Vanilla Put
54 //=====
55 /**
56  * @brief Calcule le prix d'un put européen.
57  */
58 __declspec(dllexport) double __stdcall PriceEuPut(
59   double S0, double R, double sigma, double T, int N, double K
60 );
61 /**
62  * @brief Calcule le delta d'un put européen.
63  */
64 __declspec(dllexport) double __stdcall DeltaEuPut(
65   double S0, double R, double sigma, double T, int N, double K
66 );
67 /**
68  * @brief Renvoie la matrice de valorisation du put vanille.
69  */
70 __declspec(dllexport) VARIANT __stdcall TreeEuPut(
71   double S0, double R, double sigma, double T, int N, double K
72 );
73 /**
74  * @brief Renvoie la matrice des deltas du put vanille.
75  */
76 __declspec(dllexport) VARIANT __stdcall TreeDeltaEuPut(
77   double S0, double R, double sigma, double T, int N, double K
78 );
79 /**
80  * @brief Renvoie la matrice des bonds du put vanille.
81  */
82 __declspec(dllexport) VARIANT __stdcall TreeBondEuPut(
83   double S0, double R, double sigma, double T, int N, double K
84 );
85 /**
86  * @brief Renvoie la matrice des bonds du put vanille.
87  */
88 __declspec(dllexport) VARIANT __stdcall TreeBondEuPut(
89   double S0, double R, double sigma, double T, int N, double K
90 );
91 } // extern "C"
92
93 #endif // EXPORTS_H

```

Listing 1.7: Exports.cpp

```

1 #include "pch.h"
2 #include "Exports.h"
3
4 //=====
5 // Factorisation du code
6 //=====
7
8 /**
9  * @brief Déclare une fonction extern "C" __stdcall retournant un double
10 * protégée par try/catch.
11 * @param name Identifiant de la fonction exportée.
12 * @param args Signature (entre parenthèses) de la fonction.
13 * @param body Code à exécuter (doit inclure un return).
14 */
15 #define SAFE_DOUBLE(name, args, body) \
16     extern "C" double __stdcall name args { \
17         try { body; } \
18         catch(const std::exception& e) { \
19             return reportError(e.what(), #name); \
20         } \
21         catch(...) { \
22             return reportError("Erreur inconnue dans " #name, #name); \
23         } \
24     }
25 /**
26  * @brief Déclare une fonction extern "C" __stdcall retournant un
27  * VARIANT SAFEARRAY protégée par try/catch.
28  * @param name Identifiant de la fonction exportée.
29  * @param args Signature (entre parenthèses) de la fonction.
30  * @param body Bloc '{...}' remplissant et retournant un VARIANT.
31 */
32 #define SAFE_VARIANT(name, args, body) \
33     extern "C" VARIANT __stdcall name args { \
34         try { body; } \
35         catch(const std::exception& e) { \
36             reportError(e.what(), #name); \
37             return makeVariantFromArray(nullptr); \
38         } \
39         catch(...) { \
40             reportError("Erreur inconnue dans " #name, #name); \
41             return makeVariantFromArray(nullptr); \
42         } \
43     }
44 static bool g_errorDisplayed = false;
45
46 extern "C" __declspec(dllexport) void __stdcall ResetErrorFlag()
47 {
48     g_errorDisplayed = false;
49 }
```

```

50 /**
51 * @brief Affiche une boîte d'erreur une seule fois.
52 * @param msg Le message d'erreur à afficher.
53 * @param title Le titre de la boîte de dialogue.
54 * @return NaN pour signaler l'erreur au code appelant.
55 */
56
57 double reportError(const char* msg, const char* title) {
58     if (!g_errorDisplayed) {
59         wchar_t wmsg[256], wtitle[256];
60         MultiByteToWideChar(CP_UTF8, 0, msg, -1, wmsg, 256);
61         MultiByteToWideChar(CP_UTF8, 0, title, -1, wtitle, 256);
62         MessageBoxW(nullptr, wmsg, wtitle, MB_OK | MB_ICONERROR);
63         g_errorDisplayed = true;
64     }
65     return std::nan("");
66 }
67
68 /**
69 * @brief Emballe un SAFEARRAY* de doubles dans un VARIANT.
70 * @param psa Le pointeur vers le SAFEARRAY à encapsuler.
71 * @return Un VARIANT prêt à être renvoyé à VBA.
72 */
73 VARIANT makeVariantFromArray(SAFEARRAY* psa) {
74     VARIANT var;
75     VariantInit(&var);
76     var.vt = VT_ARRAY | VT_R8;
77     var.parray = psa;
78     return var;
79 }
80
81 /**
82 * @brief Convertit un conteneur matriciel en VARIANT COM pour l'interopérabilité.
83 * @tparam Mtx Type du conteneur à deux dimensions.
84 * @param matrix Référence constante à la matrice de valeurs.
85 * @return VARIANT contenant un SAFEARRAY bidimensionnel de doubles initialisé avec les éléments de la matrice.
86 */
87 template<typename Mtx>
88 VARIANT toVariant(const Mtx& matrix) {
89     int levels = int(matrix.size());
90     int maxWidth = 0;
91     for (int n = 0; n < levels; ++n)
92         maxWidth = std::max<int>(maxWidth, int(matrix[n].size()));
93
94     bool isExpo = maxWidth > levels;
95     int rows = isExpo ? maxWidth : levels;
96     int cols = levels;
97
98     SAFEARRAYBOUND sab[2];
99     sab[0].lLbound = 0; sab[0].cElements = rows;

```

```

100    sab[1].lLbound = 0; sab[1].cElements = cols;
101    SAFEARRAY* psa = SafeArrayCreate(VT_R8, 2, sab);
102
103    double* data = nullptr;
104    SafeArrayAccessData(psa, (void**)&data);
105    for (int n = 0; n < cols; ++n) {
106        int width = int(matrix[n].size());
107        for (int i = 0; i < width; ++i) {
108            data[i + n * rows] = matrix[n][i];
109        }
110    }
111    SafeArrayUnaccessData(psa);
112    return makeVariantFromArray(psa);
113}
114
115 //=====
116 // Vanilla Call
117 //=====
118
119 SAFE_DOUBLE(PriceEuCall,
120     (double S0, double R, double sigma, double T, int N, double K),
121     return crr::European<opt::PayoffCall>(S0, R, sigma, T, N, opt::
122         PayoffCall(K)).price();
123)
124
125 SAFE_DOUBLE(DeltaEuCall,
126     (double S0, double R, double sigma, double T, int N, double K),
127     return crr::European<opt::PayoffCall>(S0, R, sigma, T, N, opt::
128         PayoffCall(K)).deltaZero();
129)
130
131 SAFE_VARIANT(TreeEuCall,
132     (double S0, double R, double sigma, double T, int N, double K),
133     {
134         auto V = crr::European<opt::PayoffCall>(S0, R, sigma, T, N, opt::
135             ::PayoffCall(K)).treePrice();
136         return toVariant(V);
137     }
138 )
139
140 SAFE_VARIANT(TreeDeltaEuCall,
141     (double S0, double R, double sigma, double T, int N, double K),
142     {
143         auto H = crr::European<opt::PayoffCall>(S0, R, sigma, T, N, opt::
144             ::PayoffCall(K)).hedgingStrategy();
145         return toVariant(H.delta());
146     }
147 )
148
149 SAFE_VARIANT(TreeBondEuCall,
150     (double S0, double R, double sigma, double T, int N, double K),
151     {

```

```

148     auto H = crr::European<opt::PayoffCall>(S0, R, sigma, T, N, opt
149         ::PayoffCall(K)).hedgingStrategy();
150     return toVariant(H.bond);
151 }
152
153 //=====
154 // Vanilla Put
155 //=====
156
157 SAFE_DOUBLE(PriceEuPut,
158     (double S0, double R, double sigma, double T, int N, double K),
159     return crr::European<opt::PayoffPut>(S0, R, sigma, T, N, opt::
160         PayoffPut(K)).price();
161 )
162
163 SAFE_DOUBLE(DeltaEuPut,
164     (double S0, double R, double sigma, double T, int N, double K),
165     return crr::European<opt::PayoffPut>(S0, R, sigma, T, N, opt::
166         PayoffPut(K)).deltaZero();
167 )
168
169 SAFE_VARIANT(TreeEuPut,
170     (double S0, double R, double sigma, double T, int N, double K),
171     {
172         auto V = crr::European<opt::PayoffPut>(S0, R, sigma, T, N, opt::
173             PayoffPut(K)).treePrice();
174         return toVariant(V);
175     }
176 )
177
178 SAFE_VARIANT(TreeDeltaEuPut,
179     (double S0, double R, double sigma, double T, int N, double K),
180     {
181         auto H = crr::European<opt::PayoffPut>(S0, R, sigma, T, N, opt::
182             PayoffPut(K)).hedgingStrategy();
183         return toVariant(H.delta);
184     }
185 )
186
187 SAFE_VARIANT(TreeBondEuPut,
188     (double S0, double R, double sigma, double T, int N, double K),
189     {
190         auto H = crr::European<opt::PayoffPut>(S0, R, sigma, T, N, opt::
191             PayoffPut(K)).hedgingStrategy();
192         return toVariant(H.bond);
193     }
194 )

```

Passons maintenant à VBA où, chaque fois que l'on souhaite exporter une fonction, deux opérations doivent être réalisées : la déclaration des fonctions DLL dans **DLLModule** et les wrappers UDF correspondants dans **UDFModule**, pour pouvoir utiliser les fonctions sur la feuille Excel.

Listing 1.8: DLLModule

```
1' -----
2' Déclarations des fonctions DLL
3' -----
4'
5' -----
6' Vanilla Call
7' -----
8 Declare PtrSafe Function PriceEuCall Lib "src.dll" ( _
9     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
10    ByVal t As Double, ByVal N As Long, ByVal K As Double _
11 ) As Double
12
13 Declare PtrSafe Function DeltaEuCall Lib "src.dll" ( _
14     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
15    ByVal t As Double, ByVal N As Long, ByVal K As Double _
16 ) As Double
17
18 Declare PtrSafe Function TreeEuCall Lib "src.dll" ( _
19     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
20    ByVal t As Double, ByVal N As Long, ByVal K As Double _
21 ) As Variant
22
23 Declare PtrSafe Function TreeDeltaEuCall Lib "src.dll" ( _
24     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
25    ByVal t As Double, ByVal N As Long, ByVal K As Double _
26 ) As Variant
27
28 Declare PtrSafe Function TreeBondEuCall Lib "src.dll" ( _
29     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
30    ByVal t As Double, ByVal N As Long, ByVal K As Double _
31 ) As Variant
32
33' -----
34' Vanilla Put
35' -----
36 Declare PtrSafe Function PriceEuPut Lib "src.dll" ( _
37     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
38    ByVal t As Double, ByVal N As Long, ByVal K As Double _
39 ) As Double
40
41 Declare PtrSafe Function DeltaEuPut Lib "src.dll" ( _
42     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
43    ByVal t As Double, ByVal N As Long, ByVal K As Double _
44 ) As Double
45
46 Declare PtrSafe Function TreeEuPut Lib "src.dll" ( _
47     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
48    ByVal t As Double, ByVal N As Long, ByVal K As Double _
49 ) As Variant
50
51 Declare PtrSafe Function TreeDeltaEuPut Lib "src.dll" ( _
```

```

52     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
53     ByVal t As Double, ByVal N As Long, ByVal K As Double _
54 ) As Variant
55
56 Declare PtrSafe Function TreeBondEuPut Lib "src.dll" ( _
57     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
58     ByVal t As Double, ByVal N As Long, ByVal K As Double _
59 ) As Variant

```

Listing 1.9: UDFModule

```

1' -----
2' UDF wrappers
3' -----
4
5' -----
6' Vanilla Call
7' -----
8 Public Function PriceCallEu( _
9     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
10    ByVal t As Double, ByVal N As Long, ByVal K As Double _
11 ) As Double
12    PriceCallEu = PriceEuCall(S0, R, sigma, t, N, K)
13 End Function
14
15 Public Function DeltaCallEu( _
16     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
17     ByVal t As Double, ByVal N As Long, ByVal K As Double _
18 ) As Double
19    DeltaCallEu = DeltaEuCall(S0, R, sigma, t, N, K)
20 End Function
21
22 Public Function TreeCallEu( _
23     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
24     ByVal t As Double, ByVal N As Long, ByVal K As Double _
25 ) As Variant
26    TreeCallEu = TreeEuCall(S0, R, sigma, t, N, K)
27 End Function
28
29 Public Function TreeDeltaCallEu( _
30     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
31     ByVal t As Double, ByVal N As Long, ByVal K As Double _
32 ) As Variant
33    TreeDeltaCallEu = TreeDeltaEuCall(S0, R, sigma, t, N, K)
34 End Function
35
36 Public Function TreeBondCallEu( _
37     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
38     ByVal t As Double, ByVal N As Long, ByVal K As Double _
39 ) As Variant
40    TreeBondCallEu = TreeBondEuCall(S0, R, sigma, t, N, K)
41 End Function
42

```

```

43   ' -----
44   ' Vanilla Put
45   ' -----
46 Public Function PricePutEu( _
47     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
48     ByVal t As Double, ByVal N As Long, ByVal K As Double _
49 ) As Double
50   PricePutEu = PriceEuPut(S0, R, sigma, t, N, K)
51 End Function
52
53 Public Function DeltaPutEu( _
54     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
55     ByVal t As Double, ByVal N As Long, ByVal K As Double _
56 ) As Double
57   DeltaPutEu = DeltaEuPut(S0, R, sigma, t, N, K)
58 End Function
59
60 Public Function TreePutEu( _
61     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
62     ByVal t As Double, ByVal N As Long, ByVal K As Double _
63 ) As Variant
64   TreePutEu = TreeEuPut(S0, R, sigma, t, N, K)
65 End Function
66
67 Public Function TreeDeltaPutEu( _
68     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
69     ByVal t As Double, ByVal N As Long, ByVal K As Double _
70 ) As Variant
71   TreeDeltaPutEu = TreeDeltaEuPut(S0, R, sigma, t, N, K)
72 End Function
73
74 Public Function TreeBondPutEu( _
75     ByVal S0 As Double, ByVal R As Double, ByVal sigma As Double, _
76     ByVal t As Double, ByVal N As Long, ByVal K As Double _
77 ) As Variant
78   TreeBondPutEu = TreeBondEuPut(S0, R, sigma, t, N, K)
79 End Function

```

La partie VBA ne sera pas détaillée davantage : il faut simplement savoir que des macros exploitant ces UDF ont été implémentées pour créer une interface interactive dans la feuille Excel. Les deux exemples suivants illustrent le fonctionnement de cette interface.

Exemple 1 (Call vanille)

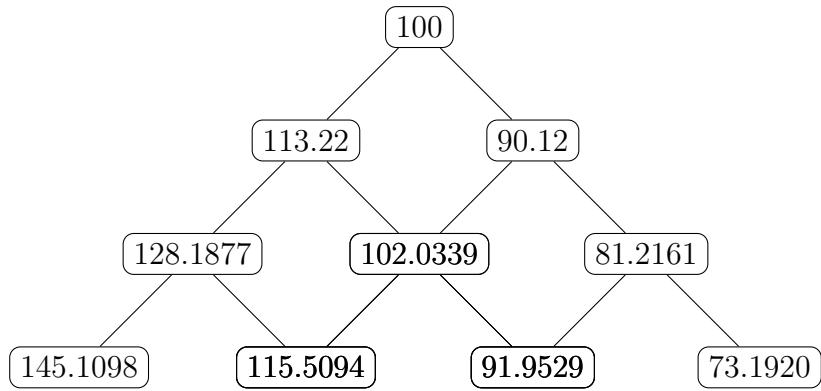
Nous nous plaçons sur un marché CRR avec les paramètres suivants :

$$S_0 = 100, \quad R = 0.05, \quad \sigma = 0.2, \quad N = 3, \quad T = 1$$

Nous souhaitons valoriser et couvrir une option call vanille de strike $K = 100$ et de maturité $T = 1$. On calcule :

$$r = \frac{RT}{N} = \frac{0.05}{3} = 0.0167, \quad \sigma_N = \sigma \sqrt{\frac{T}{N}} = 0.2 \sqrt{\frac{1}{3}} = 0.1155$$

$$u = r + \sigma_N = 0.1322, \quad d = r - \sigma_N = -0.0988$$



Pour résoudre automatiquement le problème avec notre code, nous utilisons l'interface Excel illustrée à la Figure 1.1.

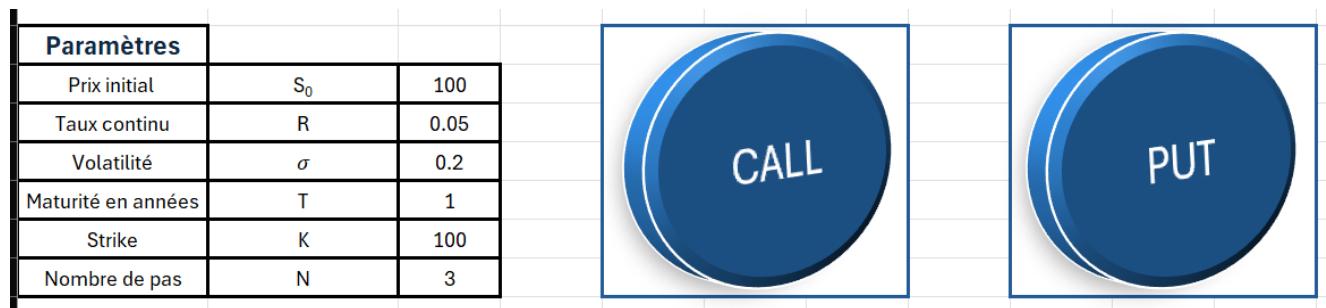


Figure 1.1: Tableau des paramètres et boutons pour les options vanilles

Il suffit désormais de saisir les valeurs des paramètres de notre modèle dans le tableau, puis de cliquer sur le bouton CALL. Les trois arbres correspondants seront alors générés : l'arbre de valorisation de la call, l'arbre des positions en sous-jacent pour la couverture et l'arbre des positions en actif sans risque.

	n=0	n=1	n=2	n=3
				45.1098
			29.8127	
		18.4132		15.5094
	10.9006		7.6276	
		3.7513		0
			0	
				0

Figure 1.2: Prix de la call vanille

n=0	n=1	n=2
	1	
0.8485		
0.6349	0.6582	
0.3665		0

Figure 1.3: Delta de la call vanille

n=0	n=1	n=2
	-77.6512	
-52.5877		-59.5301
	-29.2771	
		0

Figure 1.4: Bond de la call vanille

Exemple 2 (Put vanille)

Étant donné que l’arbre est recombinant, l’algorithme présente une complexité en $O(N^2)$. Nous pouvons donc augmenter N sans souci de performance. Pour des raisons de présentation, choisissons $N = 10$ tout en conservant les mêmes paramètres que dans l’exemple 1. Cette fois, nous souhaitons valoriser et couvrir une put vanille : il suffit de modifier N dans le tableau puis de cliquer sur le bouton PUT.

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
										0	
								0	0	0	
					0		0	0	0	0	
			0.1388		0		0	0	0	0	
		0.6336		0.279		0.5608		1.1271		0	
5.627	3.3387	1.6701	2.7234		2.0018			0		0	
		5.0407		4.3393		3.4629		2.2655		0	
	7.9716		7.4085		6.7202		5.8332		4.5537		
		10.9821		10.5518		10.0446		9.4592		9.153	
			14.6655		14.489		14.3565		14.4593		
				18.9259		19.0782		19.3973		19.9102	
					23.5521		23.9907		24.5293		
						28.2616		28.824		29.3937	
							32.8151		33.4069		
								37.1344		37.7542	
									41.2333		
										45.1247	

Figure 1.5: Prix de la put vanille

n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
								0	
						0	0	0	
				0	0	0	0	0	
			-0.0169		0				
		-0.0646		-0.0361		0			
	-0.1448		-0.1186		-0.0772		0		
-0.2494		-0.2357		-0.2122		-0.1647		0	
-0.3663	-0.3682		-0.3686		-0.3654		-0.3515		
	-0.4988	-0.5184		-0.5459		-0.5931		-0.7501	
		-0.6469		-0.6884		-0.7507		-0.8672	
			-0.7926		-0.8499		-0.9294		-1
				-0.9109		-0.9625		-1	
					-0.9801		-1		-1
						-1		-1	
							-1		-1
								-1	
									-1

Figure 1.6: Delta de la put vanille

n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
								0	
						0	0	0	
				0	0	0	0	0	
			2.3444		0				
		8.5071		4.7122		0			
	18.1914		14.7548		9.4716		0		
29.9857		28.0577		24.945		19.0378		0	
42.2527	42.0798		41.6411		40.6678		38.2661		
	54.9422	56.5228		58.7537		62.7044		76.9148	
		68.3541	71.9698		77.4271		87.7698		
			80.8688	85.9056		92.924		99.5025	
				90.5765	95.2432		99.0075		
					96.1532	98.5149		99.5025	
						98.0248		99.0075	
							98.5149		99.5025
								99.0075	
									99.5025

Figure 1.7: Bond de la put vanille

Chapitre 2

Modèle CRR et options exotiques

Dans ce deuxième chapitre, nous analyserons certaines options exotiques, que l'on définit comme tous les contrats qui ne sont pas vanilles et qui sont donc un peu plus complexes. Nous suivrons une approche similaire à celle de la section 1.2 : d'abord exposer, au niveau théorique, les étapes menant à un algorithme informatiquement implémentable pour la valorisation et la couverture, puis présenter cette implémentation en C++. La section 2.1 sera consacrée aux options asiatiques, tandis que la section 2.2 traitera des options américaines.

2.1 Options asiatiques

Une option asiatique est un actif contingent européen dont la fonction de payoff ne dépend pas seulement de la valeur terminale du sous-jacent S_N , mais de l'ensemble de la trajectoire (S_0, S_1, \dots, S_N) . Des exemples concrets sont par exemple une call sur moyenne arithmétique :

$$H = \left(\frac{1}{N+1} \sum_{i=0}^N S_i - K \right)^+ \quad (2.1)$$

ou une call lookback :

$$H = \left(\max_{0 \leq i \leq N} S_i - K \right)^+ \quad (2.2)$$

Dans la littérature, le terme «asiatique» est parfois réservé aux options sur moyenne arithmétique, mais dans ce mémoire nous l'utiliserons comme synonyme de path-dependent.

2.1.1 Valorisation et couverture

Quant à la valorisation et à la couverture, il n'y a pas grand-chose à ajouter par rapport aux options européennes : il suffit de considérer, dans la section 1.2.2, toute la trajectoire et non seulement S_N , puis les calculs sont identiques. On obtient donc les mêmes formules (1.19) et (1.20) pour la programmation dynamique, à la seule différence que $x \in \mathbb{R}^{N+1}$ cette fois. Pour la couverture, on retrouve également la même formule que dans (1.21).

Une chose importante à souligner est que la dépendance au parcours brise le caractère recombinatoire de l'arbre binomial. Cela entraîne que les formules CRR exactes deviennent un choix peu pertinent pour traiter les options asiatiques, car on obtient un algorithme de complexité exponentielle. Il a été vérifié empiriquement qu'avec déjà $N = 15$, on observe une forte dégradation des performances en termes de temps d'exécution. C'est pourquoi, pour les options asiatiques, on priviliege des méthodes mieux adaptées comme la méthode de Monte Carlo expliquée en section 3.2.1.

2.1.2 Implémentation informatique

Dans ce cas, l'idée a été de concevoir un code très abstrait pour la classe des options asiatiques, en utilisant deux templates : l'un pour le payoff (comme pour les options européennes) et l'autre pour l'agrégateur, c'est-à-dire le type d'option asiatique. Ainsi, comme pour les payoffs, un fichier dédié a été créé contenant les classes de différents types d'agréateurs; en particulier, on a inclus : la moyenne arithmétique, la moyenne géométrique, le maximum (pour la call lookback) et le minimum (pour la put lookback). De cette manière, le code est extrêmement flexible et, au moment de l'instanciation des templates, on peut choisir toutes les combinaisons possibles entre payoffs et agrégateurs, offrant une vaste gamme d'options asiatiques. Enfin, des méthodes utilisant Monte Carlo sont également présentes dans la classe, que nous ignorerons pour l'instant.

Listing 2.1: Aggregator.h

```

1 #ifndef AGGREGATOR_H
2 #define AGGREGATOR_H
3
4 #include <cmath>
5 #include <stdexcept>
6
7 namespace crr {
8
9 /**
10 * @brief Interface abstraite pour l'agrégation des prix path-dépendants.
11 */
12 class Aggregator {
13 public:
14 /**
15 * @brief Calcule la nouvelle valeur agrégée.
16 * @param agg Valeur agrégée jusqu'à l'étape précédente.
17 * @param price Prix courant du sous-jacent.
18 * @param step Numéro de l'étape.
19 * @return Valeur agrégée mise à jour.
20 */
21 virtual double operator()(double agg, double price, double step)
22     const = 0;
23 };
24
25 /**
26 * @brief Agrégateur pour la moyenne arithmétique.
27 */
28 class Arithmetic : public Aggregator {
29 public:
30     double operator()(double agg, double price, double step) const
31         override {
32         return (agg * step + price) / (step + 1);
33     }
34 };
35
36 /**
37 * @brief Agrégateur pour la moyenne géométrique.
38 */

```

```

37 class Geometric : public Aggregator {
38 public:
39     double operator()(double agg, double price, double step) const
40         override {
41             return std::pow(std::pow(agg, step) * price, 1.0 / (step + 1));
42         }
43     };
44 
45 /**
46 * @brief Agrégateur pour le maximum.
47 */
48 class LookMax : public Aggregator {
49 public:
50     double operator()(double agg, double price, double step) const
51         override {
52             return std::max<double>(agg, price);
53         }
54     };
55 /**
56 * @brief Agrégateur pour le minimum.
57 */
58 class LookMin : public Aggregator {
59 public:
60     double operator()(double agg, double price, double step) const
61         override {
62             return std::min<double>(agg, price);
63         }
64     };
65 } // namespace crr
66 #endif // AGGREGATOR_H

```

Listing 2.2: Asian.h

```

1 #ifndef ASIAN_H
2 #define ASIAN_H
3 
4 #include "Option.h"
5 
6 namespace crr {
7 
8 /**
9 * @brief Options path-dépendante CRR.
10 * @tparam TPayoff Type de payoff.
11 * @tparam TAggregator Type d'agrégeur.
12 */
13 template<typename TPayoff, typename TAggregator>
14 class Asian : public Option {
15 private:
16     TPayoff payoff_;
17     TAggregator aggregator_;

```

```

18     std::vector<std::vector<double>> stockTreeNR_; ///< Arbre
19         binomiale non recombinant des prix du sous-jacent
20     void buildStockTreeNR(); ///< Génère l'
21         arbre binomial non recombinant des prix du sous-jacent
22
23 public:
24     Asian(double S0, double R, double sigma, double T, int N, const
25           TPayoff& payoff, const TAggregator& aggregator);
26     std::vector<std::vector<double>> treePrice() const override;
27     HedgingStrategy hedgingStrategy() const override;
28
29 /**
30 * @brief Valeurs terminales de l'option path-dépendante.
31 * @return Vecteur des payoffs à l'échéance pour chaque
32 *         trajectoire du sous-jacent.
33 */
34 std::vector<double> terminalValues() const;
35
36 /**
37 * @brief Prix asymptotique de l'option (méthode Monte Carlo).
38 * @return Valeur de l'option à n = 0.
39 */
40 double priceMC() const;
41
42 /**
43 * @brief Delta asymptotique (méthode bump-and-reprice).
44 * @return Valeur du delta à n = 0.
45 */
46 double deltaMC() const;
47
48 };
49
50 template<typename TPayoff, typename TAggregator>
51 Asian<TPayoff, TAggregator>::Asian(double S0, double R, double sigma
52 , double T, int N, const TPayoff& payoff, const TAggregator&
53 aggregator)
54 : Option(S0, R, sigma, T, N), payoff_(payoff), aggregator_(
55 aggregator)
56 {
57     buildStockTreeNR();
58 }
59
60 template<typename TPayoff, typename TAggregator>
61 void Asian<TPayoff, TAggregator>::buildStockTreeNR() {
62     stockTreeNR_.assign(N_ + 1, {});
63     stockTreeNR_[0] = { S0_ };
64     for (int n = 1; n <= N_; ++n) {
65         int sz = 1 << n; // 2^n
66         stockTreeNR_[n].resize(sz);
67         for (int j = 0; j < sz; ++j) {
68             bool up = (j & 1); // up = 1 si j impair, 0 sinon
69             double prev = stockTreeNR_[n - 1][j >> 1]; // j/2
70             stockTreeNR_[n][j] = prev * (1 + (up ? u_ : d_));

```

```

63         }
64     }
65 }
66
67 template<typename TPayoff, typename TAggregator>
68 std::vector<double> Asian<TPayoff, TAggregator>::terminalValues()
69     const {
70     int leafSz = 1 << N_;
71     std::vector<double> vals(leafSz);
72     for (int j = 0; j < leafSz; ++j) {
73         double agg = S0_;
74         for (int n = 1; n <= N_; ++n) {
75             int idx = j >> (N_ - n);
76             agg = aggregator_(agg, stockTreeNR_[n][idx], n);
77         }
78         vals[j] = payoff_(agg);
79     }
80     return vals;
81 }
82
83 template<typename TPayoff, typename TAggregator>
84 std::vector<std::vector<double>> Asian<TPayoff, TAggregator>::
85     treePrice() const {
86     auto terminal = terminalValues();
87     std::vector<std::vector<double>> V(N_ + 1);
88     int leafSz = 1 << N_;
89     V[N_].resize(leafSz);
90     for (int j = 0; j < leafSz; ++j)
91         V[N_][j] = terminal[j];
92
93     for (int n = N_ - 1; n >= 0; --n) {
94         int sz = 1 << n;
95         V[n].resize(sz);
96         for (int j = 0; j < sz; ++j)
97             V[n][j] = discount_ * 0.5 * (V[n + 1][2 * j + 1] + V[n +
98                                         1][2 * j]);
99     }
100    return V;
101 }
102
103 template<typename TPayoff, typename TAggregator>
104 Option::HedgingStrategy Asian<TPayoff, TAggregator>::hedgingStrategy
105     () const {
106     auto V = treePrice();
107     HedgingStrategy H;
108     H.delta.resize(N_);
109     H.bond.resize(N_);
110     for (int n = 0; n < N_; ++n) {
111         int sz = 1 << n;
112         H.delta[n].resize(sz);
113         H.bond[n].resize(sz);
114         for (int j = 0; j < sz; ++j) {

```

```

111     double Vu = V[n + 1][2 * j + 1];
112     double Vd = V[n + 1][2 * j];
113     double Su = stockTreeNR_[n + 1][2 * j + 1];
114     double Sd = stockTreeNR_[n + 1][2 * j];
115     double dlt = (Vu - Vd) / (Su - Sd);
116     H.delta[n][j] = dlt;
117     H.bond[n][j] = V[n][j] - dlt * stockTreeNR_[n][j];
118 }
119 }
120 return H;
121 }

123 template<typename TPayoff, typename TAggregator>
124 double Asian<TPayoff, TAggregator>::priceMC() const {
125     int steps = 100;
126     int paths = 10000;
127     std::mt19937_64 rng(42);
128     std::normal_distribution<double> nd(0.0, 1.0);
129     double dt = T_ / steps;
130     double discount = std::exp(-R_ * T_);
131     double sumPayoff = 0.0;
132     for (int i = 0; i < paths; ++i) {
133         double St = S0_;
134         double agg = S0_;
135         for (int j = 0; j < steps; ++j) {
136             double Z = nd(rng);
137             St *= std::exp((R_ - 0.5 * sigma_ * sigma_) * dt +
138                           sigma_ * std::sqrt(dt) * Z);
139             agg = aggregator_(agg, St, j + 1);
140         }
141         sumPayoff += payoff_(agg);
142     }
143     return discount * sumPayoff / paths;
144 }

145 template<typename TPayoff, typename TAggregator>
146 double Asian<TPayoff, TAggregator>::deltaMC() const {
147     double eps = 1e-4 * S0_;
148     Asian<TPayoff, TAggregator> up(S0_ + eps, R_, sigma_, T_, N_,
149                                     payoff_, aggregator_);
150     Asian<TPayoff, TAggregator> dn(S0_ - eps, R_, sigma_, T_, N_,
151                                     payoff_, aggregator_);
152     double priceUp = up.priceMC();
153     double priceDn = dn.priceMC();
154     return (priceUp - priceDn) / (2 * eps);
155 }

156 } // namespace crr
157 #endif // ASIAN_H

```

Comme déjà mentionné, la partie relative à la DLL et à VBA est omise car similaire à celle des options européennes. Passons directement aux exemples avec la nouvelle interface Excel.

Exemple 1 (Call arithmétique)

Nous nous trouvons toujours dans un modèle CRR avec les paramètres standards des autres exemples ($N = 3$) et nous souhaitons valoriser et couvrir une call sur moyenne arithmétique, toujours avec $K = 100$. Pour les options exotiques, l'interface Excel est un peu différente. Essentiellement, on choisit le type d'option nécessaire dans un menu déroulant, ce qui met automatiquement à jour la macro liée au bouton ARBRES, et en cliquant dessus, on obtient les trois arbres recherchés.

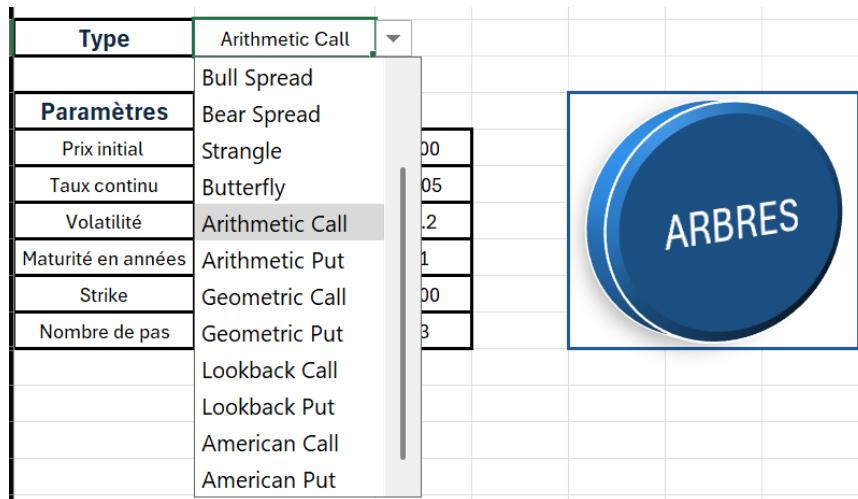


Figure 2.1: Menu déroulant pour les options exotiques

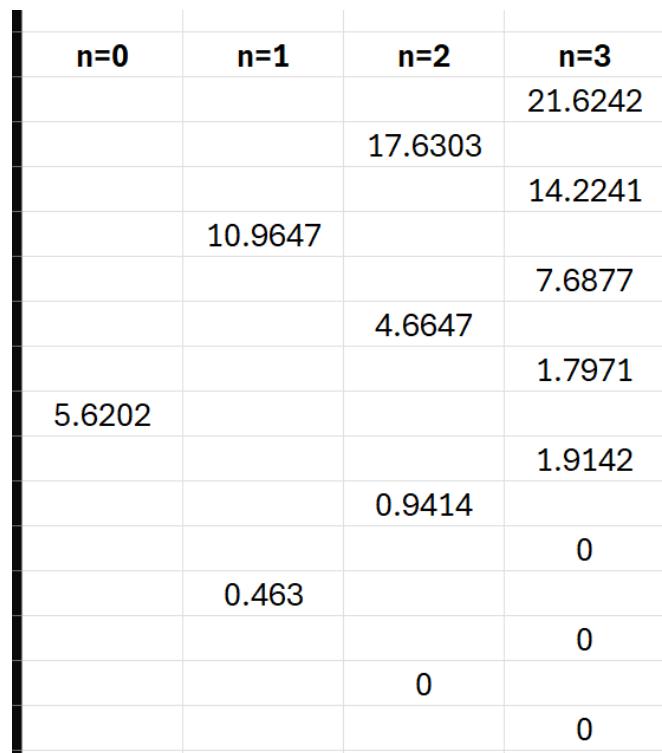


Figure 2.2: Prix de la call arithmétique

n=0	n=1	n=2
		0.25
	0.4959	
		0.25
0.4547		
	0.0812	
		0.0452
		0

Figure 2.3: Delta de la call arithmétique

n=0	n=1	n=2
		-14.413
		-45.1781
		-20.8423
	-39.8538	
		-7.3474
		-3.6135
		0

Figure 2.4: Bond de la call arithmétique

Exemple 2 (Call lookback)

Maintenant, nous faisons la même chose que dans l'exemple 1, mais en choisissant «Lookback Call» dans le menu déroulant.

n=0	n=1	n=2	n=3
			45.1098
		36.0409	
			28.1734
	24.6723		
			15.5094
		14.1261	
			13.2137
14.22			
			15.5094
		8.6248	
			2.0278
	4.2417		
			0
		0	
			0

Figure 2.5: Prix de la call lookback

n=0	n=1	n=2
		0.5722
	0.8382	
		0.0974
0.8847		
		0.5722
	0.4144	
		0

Figure 2.6: Delta de la call lookback

n=0	n=1	n=2
		-37.2959
		-70.2215
		4.1853
	-74.2469	
		-49.7523
		-33.1049
		0

Figure 2.7: Bond de la call lookback

2.2 Options américaines

Nous nous plaçons dorénavant sur un marché viable et complet. Une option américaine est un contrat qui, à la différence des options européennes, peut être exercé à n'importe quelle date avant l'échéance (y compris à l'échéance).

Définition 2.1 (Actif contingent américain). Un actif contingent américain est un processus $(H_n)_{0 \leq n \leq N}$, \mathbb{F} -adapté, positif.

2.2.1 Valorisation et couverture

Théorème 2.1 (Prix d'une option américaine). On note P_n le prix à l'instant n de l'option américaine $(H_n)_{0 \leq n \leq N}$. Alors P_n se calcule par programmation dynamique :

$$\begin{cases} P_N = H_N \\ P_n = \max\left(H_n, \mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right]\right), \quad n = 0, \dots, N-1 \end{cases}$$

Démonstration :

Il s'agit bien d'une relation d'absence d'arbitrage. Par l'absurde, si $P_N < H_N$, à la date N nous empruntons P_N (flux $+P_N$) et achetons l'option (flux $-P_N$), nous l'exerçons immédiatement (flux $+H_N$) puis remboursons le prêt (flux $-P_N$). On obtient $H_N - P_N > 0$, ce qui contredit la viabilité du marché. Si $P_N > H_N$, à la date N nous vendons l'option (flux $+P_N$), la contrepartie l'exerce aussitôt (flux $-H_N$), d'où $P_N - H_N > 0$, contradiction. Ainsi $P_N = H_N$.

Supposons à présent $P_n < \max\left(H_n, \mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right]\right)$. À la date n , nous empruntons P_n (flux $+P_n$) pour acheter l'option (flux $-P_n$). Si $H_n > P_n$, nous exerçons immédiatement (flux $+H_n$) et remboursons (flux $-P_n$), donc $H_n - P_n > 0$, contradiction. Si $H_n \leq P_n$, nous conservons l'option et, par hypothèse, $P_n < \mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right]$. Nous vendons alors le portefeuille de couverture

répliquant l'actif P_{n+1} (encaissement $\mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right]$, possible par complétude) et nous plaçons la différence $\mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right] - P_n$ sur l'actif sans risque. À la date $n+1$, nous vendons l'option détenue (flux $+P_{n+1}$), livrons $-P_{n+1}$ au titre du portefeuille répliquant (les deux se compensent) et récupérons $\frac{S_{n+1}^0}{S_n^0} (\mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right] - P_n) > 0$, contradiction.

Inversement, supposons $P_n > \max(H_n, \mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right])$. À la date n , nous vendons une option (flux $+P_n$). Si l'acheteur exerce immédiatement, nous payons $-H_n$ et $P_n - H_n > 0$, contradiction. S'il n'exerce pas, nous achetons le portefeuille répliquant P_{n+1} (décaissement $\mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right]$) et plaçons le reliquat $P_n - \mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right]$ sur l'actif sans risque. À la date $n+1$, le portefeuille vaut $+P_{n+1}$, nous rachetons l'option (flux $-P_{n+1}$), et nous encaissons $\frac{S_{n+1}^0}{S_n^0} (P_n - \mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right]) > 0$, contradiction. Donc

$$P_n = \max(H_n, \mathbb{E}_Q\left[\frac{S_n^0}{S_{n+1}^0} P_{n+1} \mid \mathcal{F}_n\right]), \quad n = 0, \dots, N-1$$

■

Ce procédé de valorisation est algorithmiquement efficace, c'est pourquoi il est celui utilisé en pratique pour valoriser les options américaines. Nous exposons toutefois aussi la méthode menant à sa formule générale. Commençons par rappeler la notion d'enveloppe de Snell et quelques-unes de ses propriétés.

Définition 2.2 (Enveloppe de Snell d'un processus). Soit $Z = (Z_n)_{0 \leq n \leq N}$ un processus \mathbb{F} -adapté. Son enveloppe de Snell $U = (U_n)_{0 \leq n \leq N}$ est définie par

$$\begin{cases} U_N = Z_N \\ U_n = \max(Z_n, \mathbb{E}[U_{n+1} \mid \mathcal{F}_n]), \quad n = 0, \dots, N-1 \end{cases}$$

On dispose en outre des propriétés suivantes. Le processus U est la plus petite surmartingale qui majore Z ; de plus,

$$U_n = \underset{\tau \in \mathcal{T}_{n,N}}{\text{ess sup}} \mathbb{E}[Z_\tau \mid \mathcal{F}_n] = \mathbb{E}[Z_{\tau_n^*} \mid \mathcal{F}_n] \quad (2.3)$$

où

$$\mathcal{T}_{n,N} := \{\tau \text{ temps d'arrêt } \mathbb{F}\text{-adaptés tels que } \tau(\omega) \in \llbracket n, N \rrbracket \text{ pour tout } \omega\} \quad (2.4)$$

et

$$\tau_n^* := \inf\{k \geq n : U_k = Z_k\} \quad (2.5)$$

En appliquant ceci aux options américaines et en posant $\tilde{P}_n := P_n/S_n^0$ et $\tilde{H}_n := H_n/S_n^0$, on obtient que \tilde{P} est l'enveloppe de Snell de \tilde{H} (sous Q). Par conséquent,

$$\tilde{P}_n = \underset{\tau \in \mathcal{T}_{n,N}}{\text{ess sup}} \mathbb{E}_Q[\tilde{H}_\tau \mid \mathcal{F}_n] = \mathbb{E}_Q[\tilde{H}_{\tau_n^*} \mid \mathcal{F}_n], \quad \tau_n^* := \inf\{k \geq n : \tilde{P}_k = \tilde{H}_k\} \quad (2.6)$$

Ainsi, τ_n^* représente le premier temps postérieur à n pour lequel il est optimal d'exercer l'option de manière anticipée. De plus, comme nous travaillons dans un espace de probabilité non redondant, l'ess sup coïncide avec le sup.

En ce qui concerne la couverture, on ne peut pas procéder comme pour les européennes puisque \tilde{P} est une surmartingale et non une martingale. On recourt donc à une couverture par excès fondée sur la décomposition de Doob.

Définition 2.3 (Décomposition de Doob). Soit $U = (U_n)_{0 \leq n \leq N}$ une surmartingale. Il existe une unique paire (M, A) telle que :

- i) $M = (M_n)_{0 \leq n \leq N}$ est une martingale,
- ii) $A = (A_n)_{0 \leq n \leq N}$ est \mathbb{F} -prévisible, croissante, et $A_0 = 0$,
- iii) pour tout n , $U_n = M_n - A_n$.

Ainsi, puisque \tilde{P} est une surmartingale et que A est croissant,

$$M_n \geq M_n - A_n = \tilde{P}_n \geq \tilde{H}_n \quad (2.7)$$

Autrement dit, si l'on couvre la martingale M (qui représente ici un actif fictif européen), on garantit à chaque date le paiement du payoff de l'option. La stratégie consiste à construire M via

$$M_n = \tilde{P}_n + A_n, \quad A_{n+1} = A_n + \tilde{P}_n - \mathbb{E}_Q[\tilde{P}_{n+1} | \mathcal{F}_n] \quad (2.8)$$

puis à le couvrir normalement comme un actif européen. Dans ce cas également, s'il existe des exercices anticipés possibles, l'arbre binomial de M sera non recombinant, rendant ainsi la valorisation CRR de l'option américaine très efficace, mais non la couverture CRR.

2.2.2 Implémentation informatique

Dans le code, nous avons implémenté exactement ce qui a été exposé ci-dessus, en développant simplement les espérances dans le cas spécifique du modèle CRR. En laissant de côté pour l'instant les méthodes liées à l'extrapolation répétée de Richardson (voir la section 3.2.2), les techniques utilisées sont similaires à celles présentées précédemment pour les autres options : il existe un héritage depuis la classe `Option` et nous employons un template pour le type de payoff. Il est également intéressant de noter que le constructeur construit l'arbre recombinant du sous-jacent afin d'alléger le calcul lorsque l'on n'appelle que les méthodes de valorisation et non la couverture; en revanche, lorsque l'on souhaite couvrir, il n'y a pas d'alternative et il faut aussi générer l'arbre exponentiel.

Listing 2.3: `American.h`

```

1 #ifndef AMERICAN_H
2 #define AMERICAN_H
3
4 #include "Option.h"
5
6 namespace crr {
7
8 /**
9 * @brief Option américaine CRR avec couverture via décomposition de
10 * Doob.
11 * @tparam TPayoff Type de payoff.

```

```

11  /*
12   * template<typename TPayoff>
13   * class American : public Option {
14   * private:
15   *     TPayoff payoff_;
16   *     std::vector<std::vector<double>> stockTree_; // < Arbre
17   *     recombiné du sous-jacent
18   *     void buildStockTree();
19
20   * public:
21   *     American(double S0, double R, double sigma, double T, int N,
22   *             const TPayoff& payoff);
23   *     std::vector<std::vector<double>> treePrice() const override;
24   *     HedgingStrategy hedgingStrategy() const override;
25
26   *     /**
27   *      * @brief Prix asymptotique de l'option (extrapolation répétée
28   *            de Richardson).
29   *      * @return Valeur de l'option à n = 0.
30   *      */
31   *     double priceRR() const;
32
33   *     /**
34   *      * @brief Delta asymptotique (bump-and-reprice).
35   *      * @return Valeur du delta à n = 0.
36   *      */
37   *     double deltaRR() const;
38   * };
39
40   template<typename TPayoff>
41   American<TPayoff>::American(double S0, double R, double sigma,
42   *                             double T, int N, const TPayoff& payoff)
43   * : Option(S0, R, sigma, T, N), payoff_(payoff)
44   {
45       buildStockTree();
46   }
47
48   template<typename TPayoff>
49   void American<TPayoff>::buildStockTree() {
50       stockTree_.assign(N_ + 1, {});
51       for (int n = 0; n <= N_; ++n) {
52           stockTree_[n].resize(n + 1);
53           for (int i = 0; i <= n; ++i) {
54               stockTree_[n][i] = S0_ * std::pow(1 + u_, i) * std::pow
55               (1 + d_, n - i);
56           }
57       }
58   }
59
60   template<typename TPayoff>
61   std::vector<std::vector<double>> American<TPayoff>::treePrice()
62   * const {

```

```

58     std::vector<std::vector<double>> V(N_ + 1);
59     V[N_].resize(N_ + 1);
60     for (int i = 0; i <= N_; ++i)
61         V[N_][i] = payoff_(stockTree_[N_][i]);
62
63     for (int n = N_ - 1; n >= 0; --n) {
64         V[n].resize(n + 1);
65         for (int i = 0; i <= n; ++i) {
66             double cont = discount_ * 0.5 * (V[n + 1][i + 1] + V[n +
67                 1][i]);
68             double exer = payoff_(stockTree_[n][i]);
69             V[n][i] = std::max<double>(exer, cont);
70         }
71     }
72     return V;
73 }
74
75 template<typename TPayoff>
76 Option::HedgingStrategy American<TPayoff>::hedgingStrategy() const {
77     // 0) Construction de l'arbre non recombinant
78     std::vector<std::vector<double>> stockTreeNR;
79     int N = N_;
80     stockTreeNR.assign(N + 1, {});
81     stockTreeNR[0] = { S0_ };
82     for (int n = 1; n <= N; ++n) {
83         int sz = 1 << n;
84         stockTreeNR[n].resize(sz);
85         for (int j = 0; j < sz; ++j) {
86             bool up = (j & 1);
87             double prev = stockTreeNR[n - 1][j >> 1];
88             stockTreeNR[n][j] = prev * (1 + (up ? u_ : d_));
89         }
90     }
91
92     // 1) Calcul de VNR sur arbre non recombinant
93     std::vector<std::vector<double>> VNR(N + 1);
94     VNR[N].resize(1 << N);
95     for (int j = 0; j < (1 << N); ++j)
96         VNR[N][j] = payoff_(stockTreeNR[N][j]);
97
98     for (int n = N - 1; n >= 0; --n) {
99         int sz = 1 << n;
100        VNR[n].resize(sz);
101        for (int j = 0; j < sz; ++j) {
102            double cont = discount_ * 0.5 * (VNR[n + 1][2 * j] + VNR
103                [n + 1][2 * j + 1]);
104            VNR[n][j] = std::max<double>(payoff_(stockTreeNR[n][j]),
105                cont);
106        }
107    }
108
109    // 2) Calcul des incrément de compensateur

```

```

107     std::vector<std::vector<double>> incr(N);
108     for (int n = 0; n < N; ++n) {
109         int sz = 1 << n;
110         incr[n].resize(sz);
111         for (int j = 0; j < sz; ++j) {
112             double cont = discount_ * 0.5 * (VNR[n + 1][2 * j] + VNR
113                 [n + 1][2 * j + 1]);
114             incr[n][j] = VNR[n][j] - cont;
115         }
116     }
117
118     // 3) Propagation forward pour A et calcul de M
119     std::vector<std::vector<double>> A(N + 1), M(N + 1);
120     A[0].resize(1);
121     M[0].resize(1);
122     A[0][0] = 0;
123     M[0][0] = VNR[0][0];
124     for (int n = 1; n <= N; ++n) {
125         int sz = 1 << n;
126         A[n].resize(sz);
127         M[n].resize(sz);
128         for (int j = 0; j < sz; ++j) {
129             double prevA = A[n - 1][j >> 1]; // j/2 pour la Fn-1
130             mesurabilité de An
131             double inc = incr[n - 1][j >> 1];
132             A[n][j] = (prevA + inc) / discount_; // faut confronter
133             avec Vn, valeurs de An-1 e incr dans n-1
134             M[n][j] = VNR[n][j] + A[n][j];
135         }
136     }
137
138     // 4) Couverture sur la martingale M
139     HedgingStrategy H;
140     H.delta.resize(N);
141     H.bond.resize(N);
142     for (int n = 0; n < N; ++n) {
143         int sz = 1 << n;
144         H.delta[n].resize(sz);
145         H.bond[n].resize(sz);
146         for (int j = 0; j < sz; ++j) {
147             double Mu = M[n + 1][2 * j + 1];
148             double Md = M[n + 1][2 * j];
149             double Su = stockTreeNR[n + 1][2 * j + 1];
150             double Sd = stockTreeNR[n + 1][2 * j];
151             double dlt = (Mu - Md) / (Su - Sd);
152             H.delta[n][j] = dlt;
153             H.bond[n][j] = M[n][j] - dlt * stockTreeNR[n][j];
154         }
155     }
156     return H;
157 }
```

```

156     template<typename TPayoff>
157     double American<TPayoff>::priceRR() const {
158         std::vector<int> Ns = { 100, 200, 400 };
159         int n = Ns.size(), m = 2;
160         std::vector<std::vector<double>> A(n, std::vector<double>(m + 1)
161             );
162         for (int i = 0; i < n; ++i) {
163             American<TPayoff> opt(S0_, R_, sigma_, T_, Ns[i], payoff_);
164             A[i][0] = opt.price();
165         }
166
167         for (int k = 1; k <= m; ++k) {
168             for (int i = 0; i + k < n; ++i) {
169                 double ratio = Ns[i + k] / Ns[i];
170                 A[i][k] = A[i + 1][k - 1] + (A[i + 1][k - 1] - A[i][k - 1]) / (ratio - 1.0);
171             }
172         }
173         return A[0][m];
174     }
175
176     template<typename TPayoff>
177     double American<TPayoff>::deltaRR() const {
178         double eps = 1e-4 * S0_;
179         American<TPayoff> up(S0_ + eps, R_, sigma_, T_, N_, payoff_);
180         American<TPayoff> dn(S0_ - eps, R_, sigma_, T_, N_, payoff_);
181         double priceUp = up.priceRR();
182         double priceDn = dn.priceRR();
183         return (priceUp - priceDn) / (2 * eps);
184     }
185 } // namespace crr
186
187 #endif // AMERICAN_H

```

Exemple 1 (Call américaine)

Si l'on n'introduit pas de dividendes dans la modélisation, une relation d'absence d'arbitrage (il n'est jamais optimal d'exercer une call sans dividendes) implique qu'à toute date n le prix d'une call américaine coïncide avec celui de la call européenne correspondante : $P_n^{\text{Am}} = P_n^{\text{Eu}}$. Les fonctions dédiées à la call américaine sont donc redondantes et, en pratique, inutiles. Elles ont néanmoins été implémentées afin d'être immédiatement disponibles si l'on souhaite ultérieurement intégrer des dividendes.

Exemple 2 (Put américaine)

Nous reportons les résultats avec les paramètres habituels ($N = 4$) en utilisant l'interface Excel pour les options exotiques introduite dans l'exemple 1 de la section 2.1.2. Nous avons également prévu qu'un indicateur rouge signale la présence d'exercices anticipés. Dans ce cas, il y en a deux. Pour la quasi-totalité des trajectoires (numérotées de haut en bas) $i = 1, \dots, 12$ et pour tout n , on

a $\tau_n^*(i) = 4$. Pour $i \in \{13, 14\}$,

$$\tau_n^*(i) = \begin{cases} 2, & n \in \{0, 1, 2\} \\ 4, & n \in \{3, 4\} \end{cases}$$

Pour $i \in \{15, 16\}$,

$$\tau_n^*(i) = \begin{cases} 2, & n \in \{0, 1, 2\} \\ 3, & n = 3 \\ 4, & n = 4 \end{cases}$$

n=0	n=1	n=2	n=3	n=4
			0	0
		0	0	0
1.8633			0	
5.9212		3.7732		0
	10.1272		7.6407	
		16.7344		15.4724
			24.0201	
				30.6684

Figure 2.8: Prix de la put américaine

n=0	n=1	n=2	n=3
		0	
		0	
-0.1696			
		0	
	-0.3763		-0.8351
-0.4132			
		0	
	-0.3763		-0.8351
-0.7102			
		-0.8351	
	-0.9836		
		-1	

Figure 2.9: Delta de la put américaine

n=0	n=1	n=2	n=3
		0	
		0	
20.7292			
		0	
	41.9766		85.0026
47.2407			
		0	
	41.9766		85.0026
74.9332			
		0	
	86.1157		
98.6316			
		0	
	101.1132		

Figure 2.10: Bond de la put américaine

Chapitre 3

Convergence du modèle CRR

Dans la section 3.1, nous étudierons la convergence du modèle CRR, lorsque $N \rightarrow \infty$, vers le modèle en temps continu le plus célèbre en finance, à savoir le modèle de Black-Scholes. Nous présenterons le prix et la stratégie de couverture asymptotiques pour les options vanille, ainsi que la vitesse de convergence.

Dans la section 3.2, nous aborderons les difficultés liées à la convergence des options exotiques et quelques méthodes permettant d'obtenir des valeurs asymptotiques approchées pour le prix et le δ .

3.1 Convergence vers le modèle de Black-Scholes

Obtenir les formules de Black-Scholes comme limites de celles du modèle CRR est l'une des voies possibles pour dériver le modèle de Black-Scholes (BS). Il en existe en réalité plusieurs; par exemple, dans l'Annexe B, nous présentons une autre approche, plus classique, fondée sur le calcul stochastique. Rappelons que, historiquement, les modèles CRR et BS ont été élaborés séparément et indépendamment; on a découvert ensuite que le CRR converge vers BS.

3.1.1 Convergence

Commençons par établir la loi asymptotique de

$$S_N = x \prod_{k=0}^N (1 + U_k)$$

où x représente S_0 . Posons

$$X_N = \ln S_N = \ln x + \sum_{k=0}^N \ln (1 + U_k)$$

Sa fonction caractéristique vérifie

$$\varphi_{X_N}(t) = \mathbb{E}_Q \left[e^{it \ln x} \prod_{k=0}^N e^{it \ln(1+U_k)} \right] = e^{it \ln x} \left(\varphi_{\ln(1+U_1)}(t) \right)^N$$

Or, dans le modèle CRR (avec $p = \frac{1}{2}$, $u = \frac{rT}{N} + \sigma \sqrt{\frac{T}{N}}$, $d = \frac{rT}{N} - \sigma \sqrt{\frac{T}{N}}$),

$$\varphi_{\ln(1+U_1)}(t) = \mathbb{E}_Q \left[e^{it \ln(1+U_1)} \right] = \frac{1}{2} \left(e^{it \ln(1+d)} + e^{it \ln(1+u)} \right)$$

En utilisant $\ln(1+z) = z - \frac{z^2}{2} + o(z^2)$, on obtient

$$\ln(1+d) = \frac{rT}{N} - \sigma\sqrt{\frac{T}{N}} - \frac{\sigma^2 T}{2N} + o\left(\frac{1}{N}\right), \quad \ln(1+u) = \frac{rT}{N} + \sigma\sqrt{\frac{T}{N}} - \frac{\sigma^2 T}{2N} + o\left(\frac{1}{N}\right)$$

d'où

$$\begin{aligned} \varphi_{\ln(1+U_1)}(t) &= \frac{1}{2} \left(e^{it\left(\frac{rT}{N} - \sigma\sqrt{\frac{T}{N}} - \frac{\sigma^2 T}{2N} + o\left(\frac{1}{N}\right)\right)} + e^{it\left(\frac{rT}{N} + \sigma\sqrt{\frac{T}{N}} - \frac{\sigma^2 T}{2N} + o\left(\frac{1}{N}\right)\right)} \right) \\ &= e^{it\left(\left(r - \frac{\sigma^2}{2}\right)\frac{T}{N} + o\left(\frac{1}{N}\right)\right)} \cos\left(\sigma t\sqrt{\frac{T}{N}}\right) = e^{it\left(\left(r - \frac{\sigma^2}{2}\right)\frac{T}{N} + o\left(\frac{1}{N}\right)\right)} \left(1 - \frac{1}{2}\sigma^2 t^2 \frac{T}{N} + o\left(\frac{1}{N}\right)\right) \end{aligned}$$

On a donc

$$\varphi_{X_N}(t) = e^{it\ln x} \left(e^{it\left(\left(r - \frac{\sigma^2}{2}\right)\frac{T}{N} + o\left(\frac{1}{N}\right)\right)} \left(1 - \frac{\sigma^2 T}{2N} t^2 + o\left(\frac{1}{N}\right)\right) \right)^N$$

Par conséquent,

$$\lim_{N \rightarrow \infty} \varphi_{X_N}(t) = e^{it\ln x} e^{it\left(r - \frac{\sigma^2}{2}\right)T} e^{-\frac{1}{2}\sigma^2 T t^2}$$

On reconnaît la fonction caractéristique d'une loi normale; par le théorème de continuité de Lévy,

$$X_N \xrightarrow{\mathcal{L}} \mathcal{N}\left(\ln x + \left(r - \frac{\sigma^2}{2}\right)T, \sigma^2 T\right)$$

c'est-à-dire

$$S_N \xrightarrow{\mathcal{L}} S_T = x \exp\left(\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}G\right), \quad G \sim \mathcal{N}(0, 1) \quad (3.1)$$

Prenons maintenant la limite de la fonction de valorisation d'une option (nous conservons la notation des sections précédentes) :

$$\lim_{N \rightarrow \infty} \psi_0(x) = \lim_{N \rightarrow \infty} \mathbb{E}_Q \left[\frac{1}{\left(1 + \frac{rT}{N}\right)^N} f(S_N) \right] = e^{-rT} \mathbb{E}_L[f(S_T)] \quad (3.2)$$

où L désigne la probabilité limite. Calculons à présent le prix initial asymptotique d'un put vanille de strike K (x représente S_0) :

$$\begin{aligned} P_0^\infty &= \lim_{N \rightarrow \infty} \psi_0(x) = e^{-rT} \mathbb{E}_L \left[\left(K - x e^{\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}y} \right)^+ \right] \\ &= \frac{e^{-rT}}{\sqrt{2\pi}} \int_{\mathbb{R}} \left(K - x e^{\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}y} \right)^+ e^{-\frac{y^2}{2}} dy = \frac{e^{-rT}}{\sqrt{2\pi}} \int_{-\infty}^{d_0} \left(K - x e^{\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}y} \right) e^{-\frac{y^2}{2}} dy \end{aligned}$$

où

$$d_0 = \frac{1}{\sigma\sqrt{T}} \left(\ln\left(\frac{K}{x}\right) - \left(r - \frac{\sigma^2}{2}\right)T \right)$$

Donc

$$P_0^\infty = e^{-rT} K N(d_0) - \int_{-\infty}^{d_0} x e^{\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}y} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} dy$$

$$= e^{-rT} K N(d_0) - x \int_{-\infty}^{d_0} \frac{1}{\sqrt{2\pi}} e^{-\frac{(y-\sigma\sqrt{T})^2}{2}} dy$$

où N est la fonction de répartition de la loi normale standard. On a alors que la formule BS du prix est

$$P_0^\infty = e^{-rT} K N(d_0) - S_0 N(d_0 - \sigma\sqrt{T}), \quad d_0 = \frac{1}{\sigma\sqrt{T}} \left(\ln\left(\frac{K}{S_0}\right) - \left(r - \frac{\sigma^2}{2}\right) T \right) \quad (3.3)$$

Pour obtenir le prix à un instant $t \in [0, T]$, il suffit de remplacer T par $T - t$ et S_0 par S_t . En procédant de façon analogue, ou bien via la parité put–call $C_0 - P_0 = S_0 - K e^{-rT}$, on retrouve le prix de la call vanille (strike K) :

$$C_0^\infty = S_0 N(d_1) - e^{-rT} K N(d_1 - \sigma\sqrt{T}), \quad d_1 = \frac{1}{\sigma\sqrt{T}} \left(\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right) T \right) \quad (3.4)$$

Pour illustrer numériquement la convergence, nous avons utilisé dans Excel la fonction de pricing importée via DLL depuis le C++ afin de calculer les prix de l'option pour N allant de 2 à 100 (courbe bleue); on observe des oscillations dues à la discréétisation, mais la suite tend asymptotiquement vers la valeur de Black-Scholes (ligne rouge). Nous ne présentons le graphique que pour la call vanille afin d'éviter les redondances : par la parité put–call, le tracé du put aurait la même forme, simplement décalée.

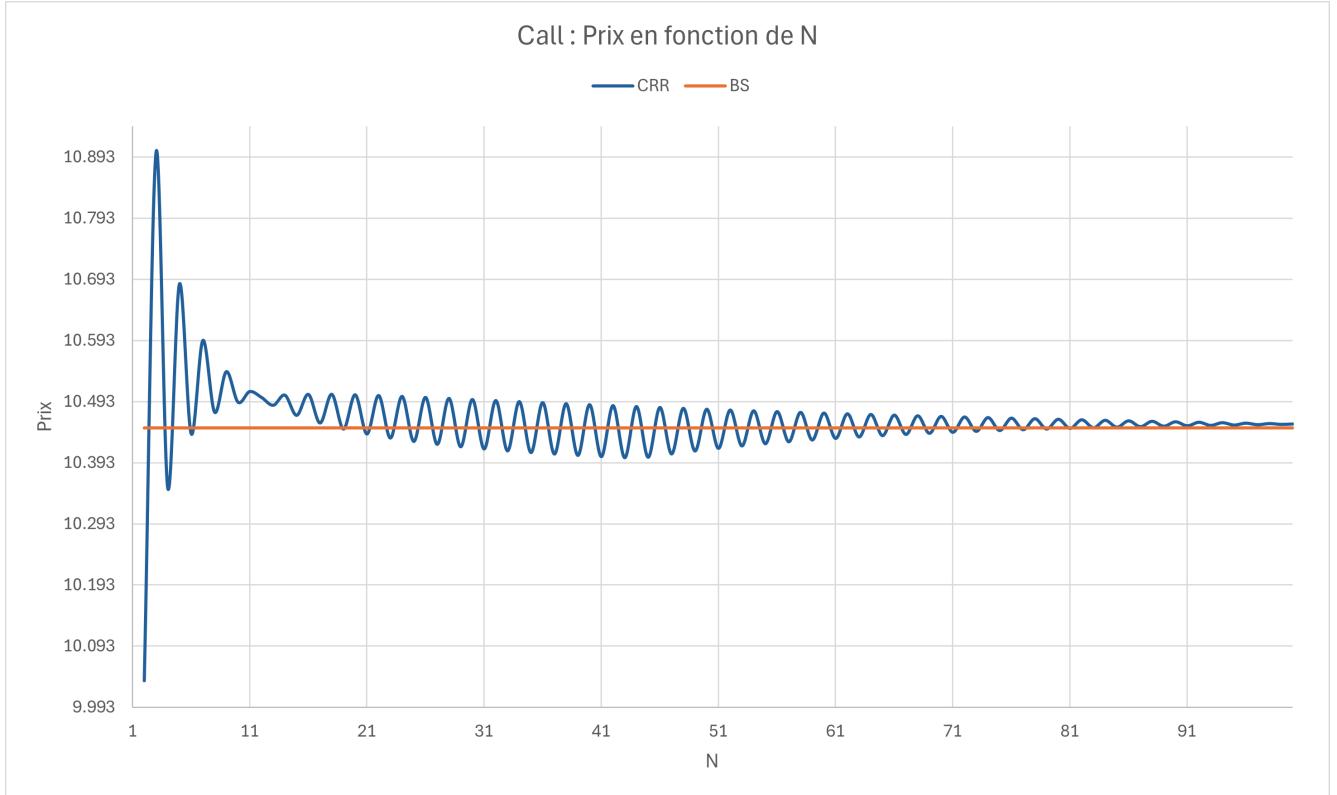


Figure 3.1: Convergence du prix de la call vanille

S'agissant de la stratégie de couverture asymptotique, le portefeuille répliquant de la put vérifie

$$\phi_1 x + \phi_0 = P_0^\infty(x, r, \sigma, K, T) \quad (3.5)$$

En dérivant des deux côtés par rapport à x , on obtient

$$\phi_1 = \frac{\partial P_0^\infty}{\partial x} \quad (3.6)$$

qui est la grecque la plus connue : le delta. Calculons donc cette dérivée,

$$\frac{\partial}{\partial x} P_0^\infty = \widetilde{K} \frac{\partial}{\partial x} \int_{-\infty}^{d_0(x)} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} dy - \left(N(d_0(x) - \sigma\sqrt{T}) + x \frac{\partial}{\partial x} \int_{-\infty}^{d_0(x)-\sigma\sqrt{T}} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} dy \right)$$

Pour la première dérivée, effectuons le changement de variable à l'intérieur de l'intégrale

$$z = d_0^{-1}(y) = K e^{-\left(\sigma\sqrt{T}y + \left(r - \frac{\sigma^2}{2}\right)T\right)}$$

et l'on obtient

$$\begin{aligned} \widetilde{K} \frac{\partial}{\partial x} \int_x^{+\infty} \frac{1}{\sigma z \sqrt{2\pi T}} \exp\left(-\frac{1}{2} \left(\frac{\ln\left(\frac{K}{z}\right) - \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}\right)^2\right) dz \\ = -\widetilde{K} \frac{1}{\sigma x \sqrt{2\pi T}} \exp\left(-\frac{1}{2} \left(\frac{\ln\left(\frac{K}{x}\right) - \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}\right)^2\right) =: v \end{aligned}$$

par le théorème fondamental de l'analyse. Pour la seconde dérivée, effectuons aussi un changement de variable dans l'intégrale

$$z = d_0^{-1}(y + \sigma\sqrt{T}) = e^{-\sigma^2 T} K e^{-\left(\sigma\sqrt{T}y + \left(r - \frac{\sigma^2}{2}\right)T\right)}$$

et l'on obtient donc

$$\begin{aligned} x \frac{\partial}{\partial x} \int_x^{+\infty} \frac{1}{\sigma z \sqrt{2\pi T}} \exp\left(-\frac{1}{2} \left(\frac{\ln\left(\frac{K}{z}\right) - \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} - \sigma\sqrt{T}\right)^2\right) dz \\ = -\frac{1}{\sigma\sqrt{2\pi T}} \exp\left(-\frac{1}{2} \left(\frac{\ln\left(\frac{K}{x}\right) - \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} - \sigma\sqrt{T}\right)^2\right) \end{aligned}$$

Ainsi, au total,

$$\begin{aligned} \frac{\partial}{\partial x} P_0^\infty &= -N(d_0 - \sigma\sqrt{T}) + v \left(-\widetilde{K} + x e^{\ln\left(\frac{K}{x}\right) - \left(r - \frac{\sigma^2}{2}\right)T - \frac{1}{2}\sigma^2 T} \right) \\ &= -N(d_0 - \sigma\sqrt{T}) + v \left(-\widetilde{K} + x \frac{K}{x} e^{-rT} \right) = -N(d_0 - \sigma\sqrt{T}) \end{aligned} \quad (3.7)$$

D'autre part, pour la call, on obtient

$$\frac{\partial}{\partial x} C_0^\infty = N(d_1) \quad (3.8)$$

Encore une fois, grâce à Excel, sur l'image suivante, on observe également la convergence du δ CRR vers le δ de Black-Scholes.

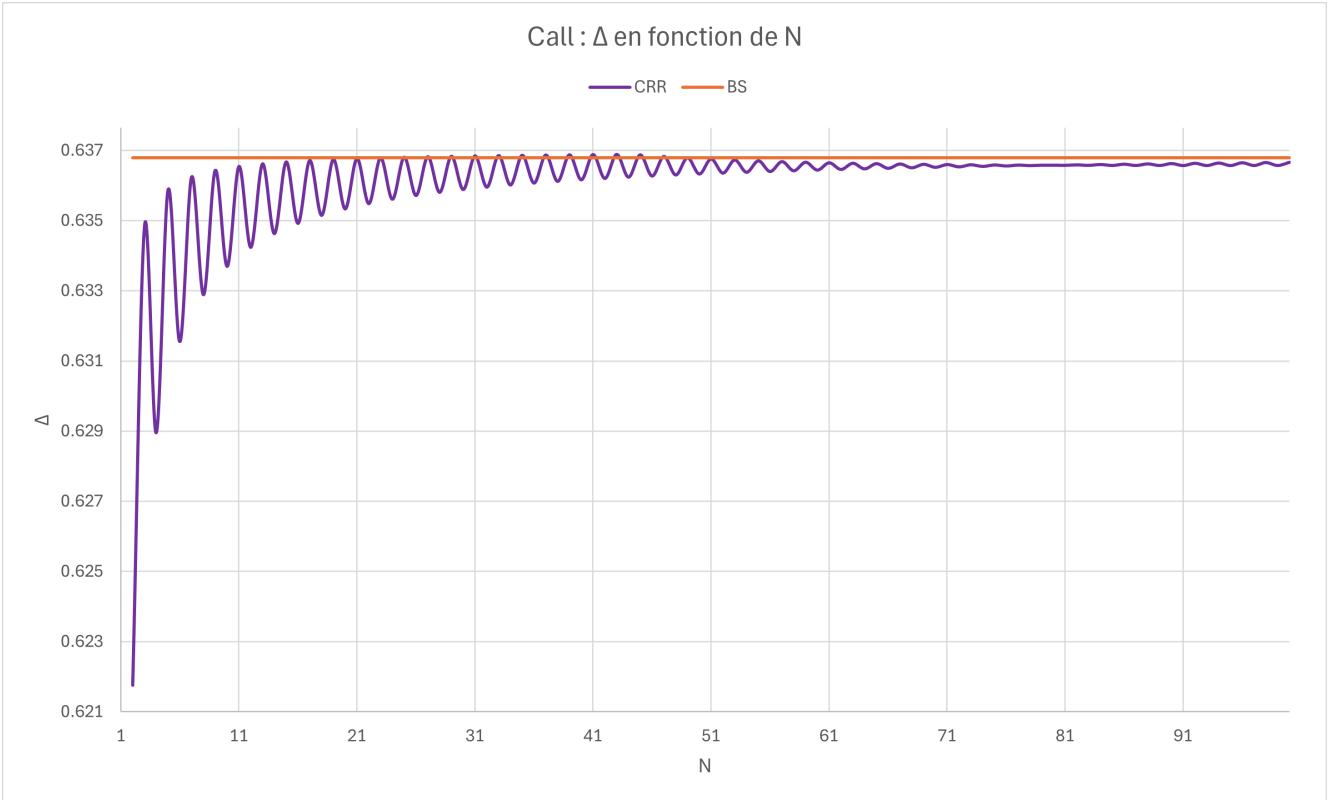


Figure 3.2: Convergence du delta de la call vanille

3.1.2 Vitesse de convergence

Comme on l'a vu dans la démonstration du chapitre précédent, le prix CRR converge vers le prix de Black-Scholes avec un ordre $O(N^{-1})$. La vérification numérique de cette vitesse s'effectue en traçant, pour $N = 2, \dots, 100$, en ordonnée $\ln(|P_N^{\text{CRR}} - P^{\text{BS}}|)$ et en abscisse $\ln(N)$, puis en contrôlant que la pente de la droite de régression est proche de -1 . En effet, si l'erreur décroît comme $1/N$, on peut écrire

$$|P_N^{\text{CRR}} - P^{\text{BS}}| = \frac{k}{N} \quad (k > 0) \quad (3.9)$$

d'où, en prenant le logarithme des deux côtés,

$$\ln(|P_N^{\text{CRR}} - P^{\text{BS}}|) = \ln k - \ln N \quad (3.10)$$

C'est ce que l'on observe sur la première figure ci-dessous.

La seconde montre une vitesse de convergence analogue pour le δ . En effet, si l'on fait aussi tendre à la limite la stratégie de couverture,

$$\lim_{N \rightarrow \infty} \mathbb{E}_Q \left[\frac{\tilde{P}_{n+1} - \tilde{P}_n}{\tilde{S}_{n+1} - \tilde{S}_n} \mid \mathcal{F}_n \right] \quad (3.11)$$

on obtient, au moyen de développements limités analogues à ceux de la preuve du prix asymptotique, que

$$\mathbb{E}_Q \left[\frac{\tilde{P}_{n+1} - \tilde{P}_n}{\tilde{S}_{n+1} - \tilde{S}_n} \mid \mathcal{F}_n \right] = \delta^{\text{BS}}(t_n, S_n) + O(N^{-1}), \quad t_n = \frac{nT}{N} \quad (3.12)$$

Autrement dit, la stratégie CRR converge vers le delta de Black-Scholes avec la vitesse $O(N^{-1})$.

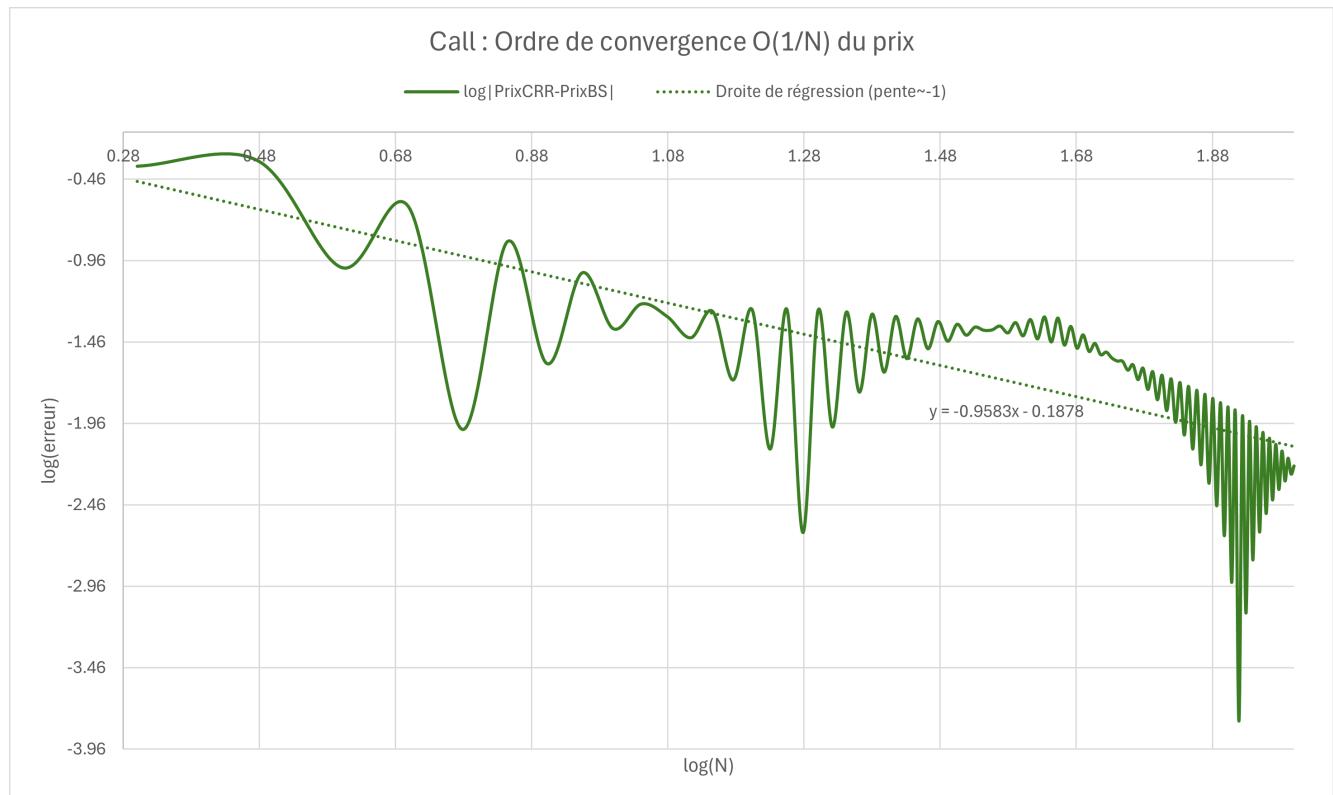


Figure 3.3: Vitesse de convergence du prix de la call vanille

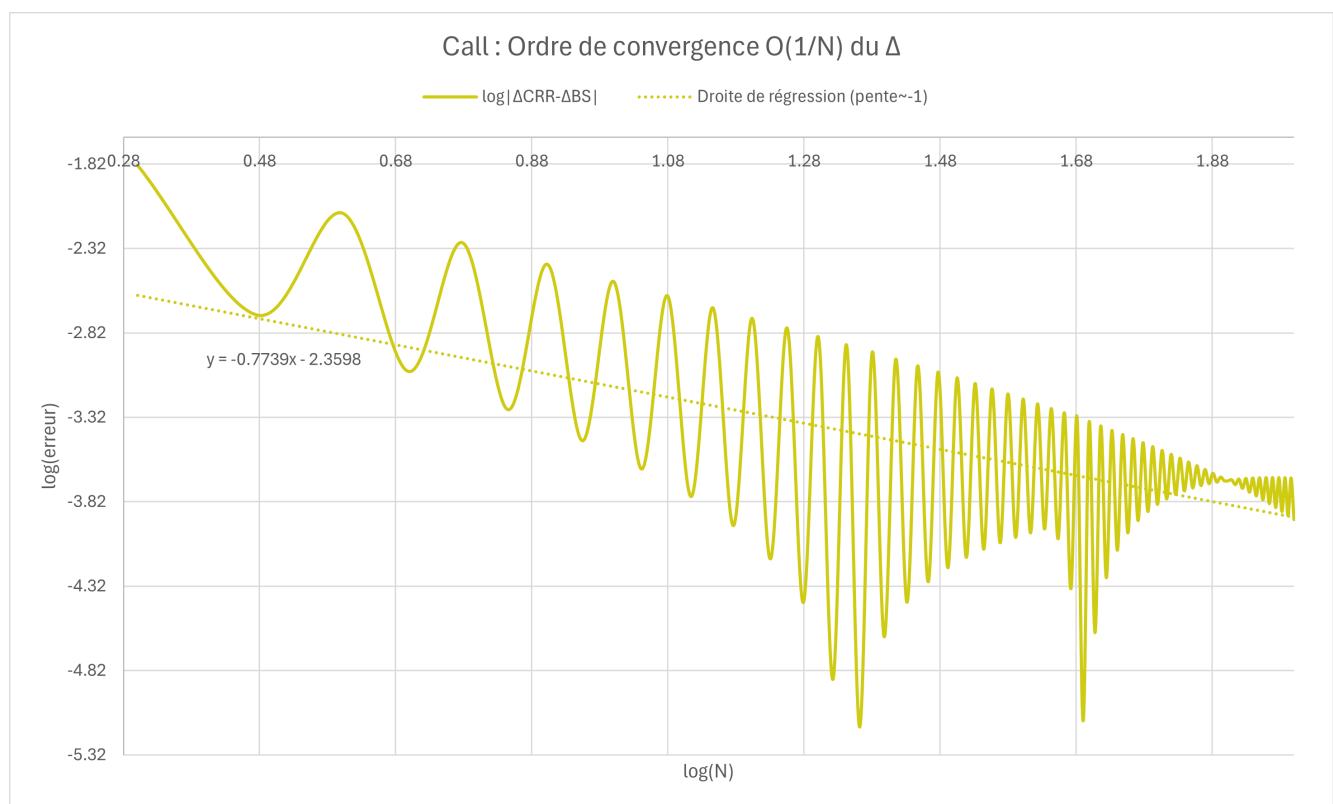


Figure 3.4: Vitesse de convergence du delta de la call vanille

3.2 Convergence des options exotiques

La convergence des options exotiques est plus problématique que celle des options vanille. On ne peut pas aboutir à une démonstration conduisant à des formules fermées exactes à la Black-Scholes, mais cela ne signifie pas qu'il n'existe pas de convergence vers un prix et un δ limites. Il existe en effet de nombreuses méthodes pour obtenir des formules ou des valeurs approchées. Dans cette section, nous en présenterons quelques-unes.

3.2.1 Méthode Monte Carlo

Nous avons déjà vu comment les formules CRR peuvent, en théorie, conduire à un prix exact pour les options asiatiques, mais pas en pratique. D'autres méthodes de pricing ont donc été élaborées pour ces options. Certaines s'appuient encore sur un modèle discret et proposent une solution binomiale approchée, mais nous ne les détaillerons pas ici; pour plus d'informations, voir par exemple Wang (2025, chap. 10).

La méthode Monte Carlo peut être utilisée pour tous les types d'options, toutefois, comme nous disposons d'autres méthodes très efficaces pour les produits vanille et américains, elle se révèle particulièrement utile pour traiter les options path-dependent. La méthode repose sur le fait que, dans le modèle de Black-Scholes, l'actif risqué suit un mouvement brownien géométrique (c'est la même loi asymptotique obtenue dans la démonstration, voir (3.1)) :

$$S_t = S_0 \exp\left(\left(r - \frac{\sigma^2}{2}\right)t + \sigma W(t)\right) \quad (3.13)$$

où $W(t)$ est un processus de Wiener sous la probabilité risque-neutre. Si l'on choisit un nombre de pas N pour discrétiser le temps et si l'on note toujours f la fonction de payoff, alors

$$P_0 = e^{-rT} \mathbb{E}[f(S_{t_1}, \dots, S_{t_N})], \quad t_k = \frac{kT}{N} \quad (3.14)$$

L'idée est d'approximer cette espérance par une moyenne sur M simulations. Comme les accroissements $W(t) - W(s) \sim \mathcal{N}(0, t - s)$ sont indépendants, on a, avec $\Delta t = T/N$ et des variables i.i.d. $Z_k \sim \mathcal{N}(0, 1)$,

$$S_{t_k} = S_{t_{k-1}} \exp\left(\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t} Z_k\right) \quad (3.15)$$

Pour une réalisation z_1, \dots, z_N , la trajectoire réalisée vérifie

$$\begin{aligned} s_{t_1} &= S_0 \exp\left(\left(r - \frac{\sigma^2}{2}\right)t_1 + \sigma\sqrt{t_1} z_1\right) \\ s_{t_k} &= s_{t_{k-1}} \exp\left(\left(r - \frac{\sigma^2}{2}\right)(t_k - t_{k-1}) + \sigma\sqrt{t_k - t_{k-1}} z_k\right), \quad k = 2, \dots, N \end{aligned} \quad (3.16)$$

Par la loi des grands nombres,

$$P_0 = e^{-rT} \lim_{M \rightarrow \infty} \frac{1}{M} \sum_{i=1}^M f(s_{t_1}^{(i)}, \dots, s_{t_N}^{(i)}) \quad (3.17)$$

où $s^{(i)}$ désigne la i -ème trajectoire simulée. En pratique, il faut donc générer un échantillon de $\mathcal{N}(0, 1)$. Une méthode usuelle est la transformation de Box-Muller, couplée à un générateur pseudo-aléatoire uniforme (par exemple Mersenne Twister). Ainsi, pour des valeurs suffisamment grandes de N (pas temporels) et M (trajectoires), l'approximation est bonne. L'algorithme complet a été implémenté en C++ dans la classe **Asian**, voir le Listing 2.2. Pour obtenir un δ approché, nous avons implémenté la méthode bump-and-reprice, un procédé numérique très intuitif pour approximer une dérivée : on choisit un ε petit devant S_0 et

$$\delta \approx \frac{P(S_0 + \varepsilon) - P(S_0 - \varepsilon)}{2\varepsilon} \quad (3.18)$$

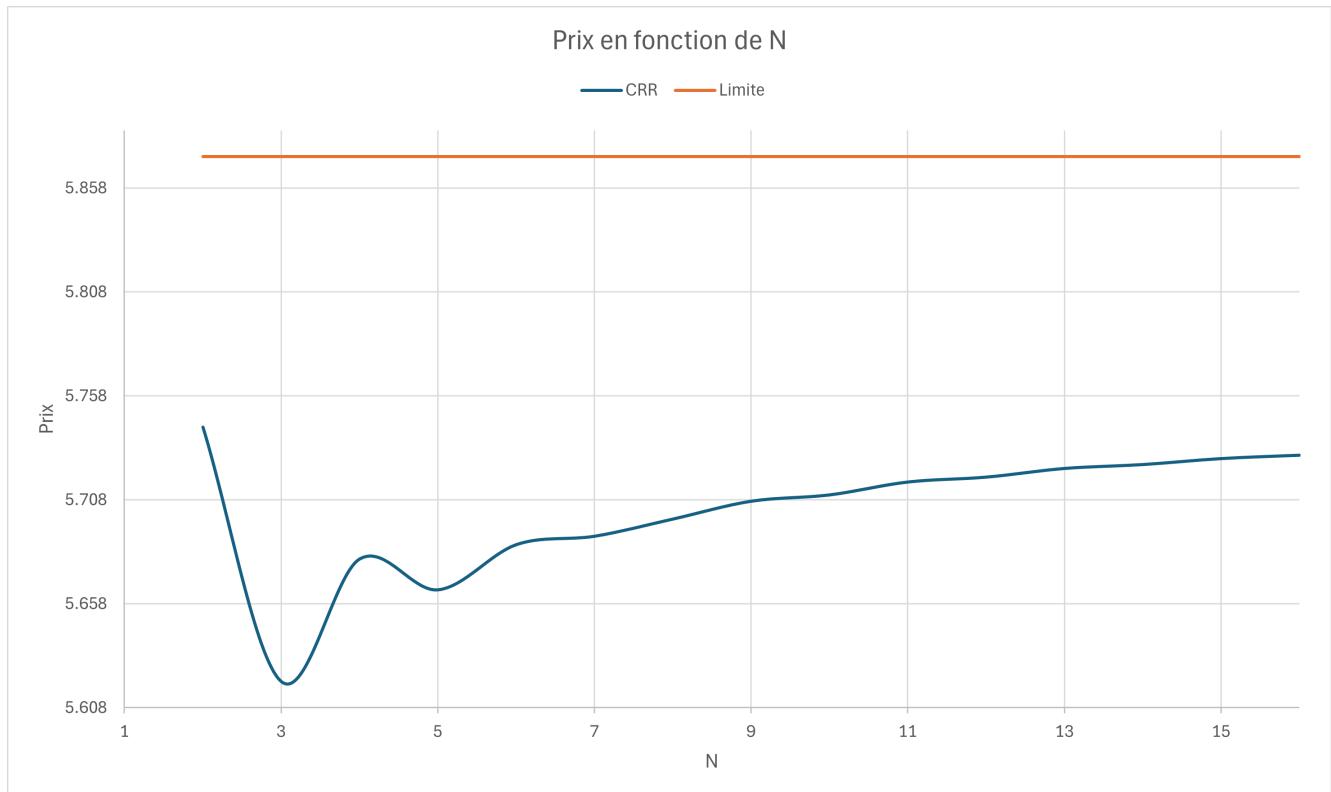


Figure 3.5: Convergence du prix de la call arithmétique

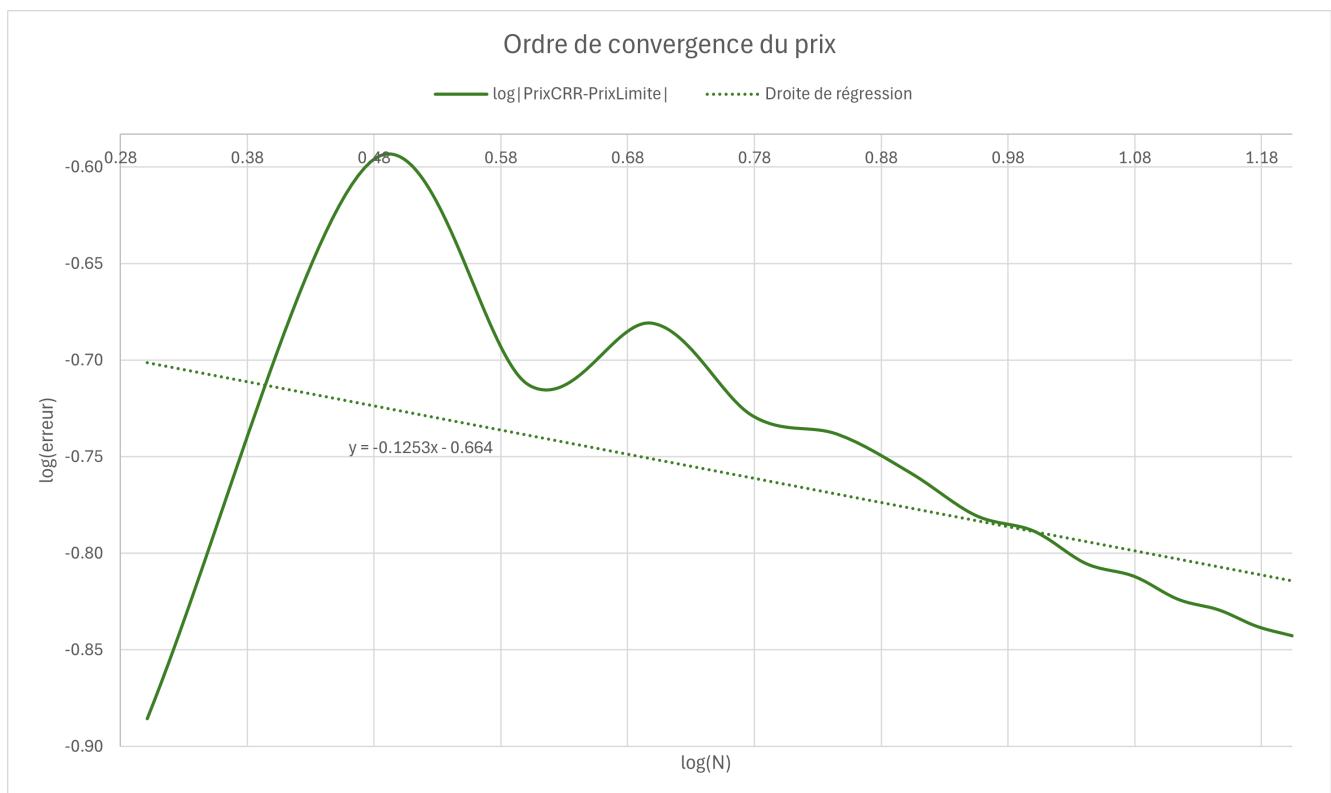


Figure 3.6: Vitesse de convergence du prix de la call arithmétique

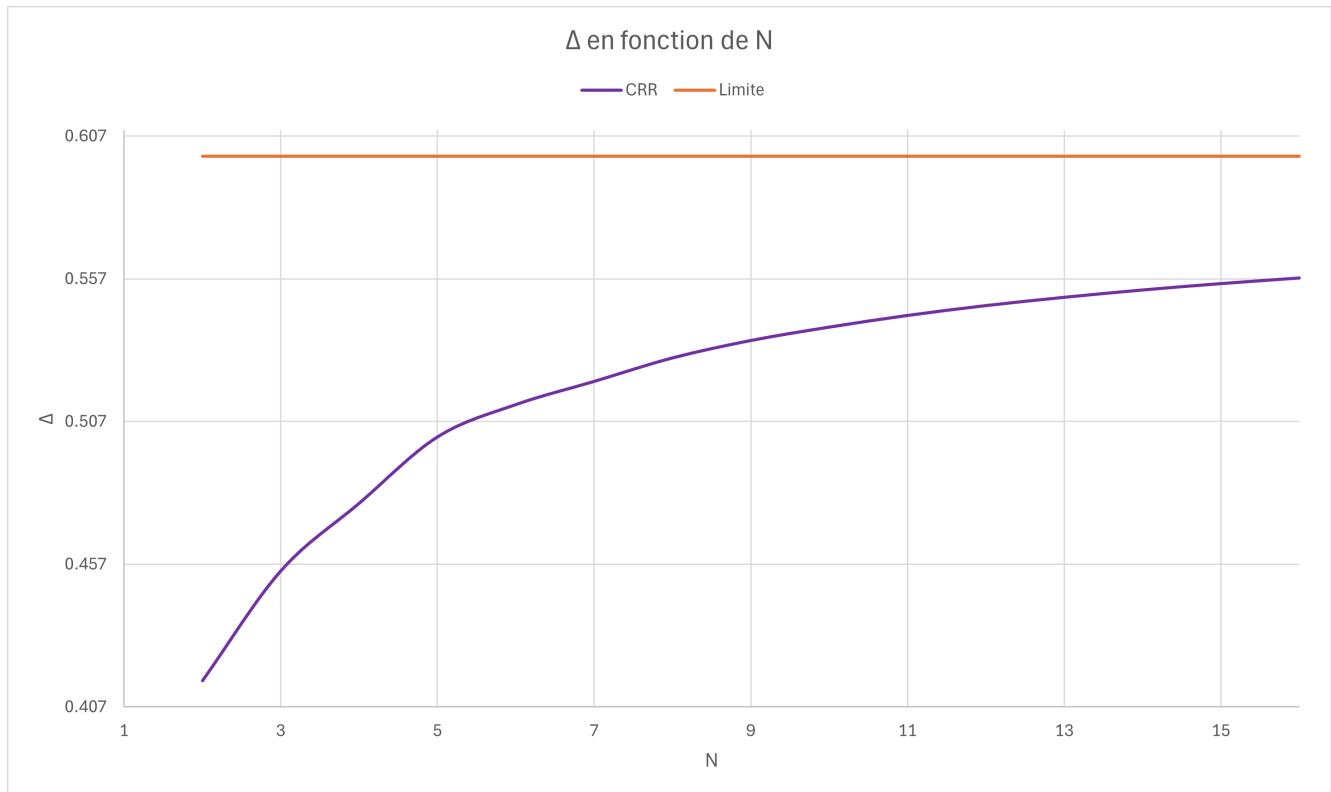


Figure 3.7: Convergence du delta de la call arithmétique

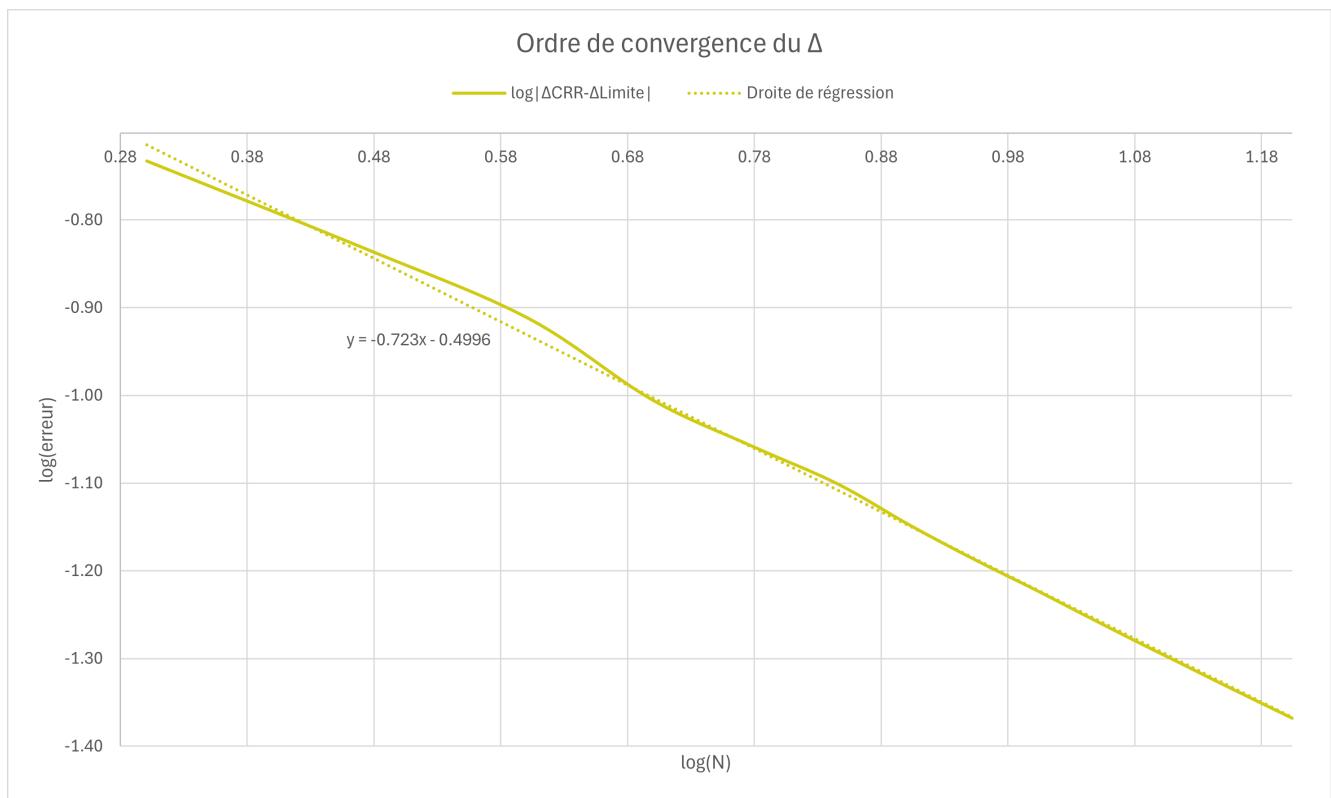


Figure 3.8: Vitesse de convergence du delta de la call arithmétique

Nous présentons la visualisation numérique de la convergence dans Excel pour la call à moyenne arithmétique, bien que le résultat soit peu instructif puisque, dans le cadre CRR, nous sommes contraints de rester à des régimes pré-asymptotiques.

3.2.2 Extrapolation répétée de Richardson

Le Monte Carlo standard appliqué aux options américaines ne fonctionne pas en présence d'exercices anticipés. Il existe donc une version adaptée et plus complexe, l'algorithme de Longstaff-Schwartz. D'autres méthodes, dans un cadre de Black-Scholes, cherchent des formules fermées, voir par exemple Bjerksund and Stensland (2002).

Cependant, en nous appuyant sur Chang, Chung, and Stapleton (2001), nous avons préféré implémenter l'algorithme d'extrapolation répétée de Richardson, à la fois simple à coder et performant. L'idée générale est qu'on approxime le prix américain a_0 par une suite $F(h)$ dépendant d'un pas $h > 0$ ($h = T/N$ si l'on discrétise en N pas). Sous des hypothèses usuelles,

$$F(h) = a_0 + c_1 h^\alpha + c_2 h^{2\alpha} + \dots, \quad \text{avec } \alpha = 1 \text{ dans la pratique (CRR)} \quad (3.19)$$

L'extrapolation de Richardson combine des évaluations à plusieurs pas pour éliminer les termes d'erreur dominants et accélérer la convergence vers a_0 . Les points clés à suivre sont :

1. Utiliser des pas géométriques (par exemple $N, 2N, 4N$) afin d'éviter les problèmes de non-convergence uniforme.
2. Calculer pour chaque N le prix $P(N)$ (valeur américaine via arbre binomial à N dates d'exercice).
3. Construire le tableau de Richardson répété et lire l'approximation raffinée.

Soient $N_1 < N_2 < N_3$ (par exemple 100, 200, 400) et

$$A_{i,0} = P(N_i), \quad r_{i,k} = \frac{N_{i+k}}{N_i} \quad (3.20)$$

Pour $k = 1, 2$ (extrapolation répétée) et $i = 1, \dots, 3 - k$,

$$A_{i,k} = A_{i+1,k-1} + \frac{A_{i+1,k-1} - A_{i,k-1}}{r_{i,k}^\alpha - 1} = A_{i+1,k-1} + \frac{A_{i+1,k-1} - A_{i,k-1}}{\frac{N_{i+k}}{N_i} - 1} \quad (3.21)$$

L'estimateur final est $A_{1,2}$ (trois niveaux, $m = 2$). Le code C++ correspondant a été écrit dans la classe `American` (voir le Listing 2.3). Quelques remarques pratiques :

- i) Trois évaluations $P(N)$ suffisent; l'algorithme est peu coûteux et simple à coder.
- ii) Un contrôle d'erreur peu onéreux est donné par l'inégalité de Schmidt :

$$|A_{i,k} - a_0| \leq |A_{i,k} - A_{i,k-1}| \quad (3.22)$$

pour i assez grand, ce qui fournit un intervalle prédictif autour de $A_{i,k}$.

Enfin, pour le δ , nous avons également utilisé la méthode bump-and-reprice. Nous présentons ci-dessous les résultats de convergence dans Excel. Nous obtenons un bon résultat pour le prix; toutefois, pour le δ , nous sommes contraints de rester en régime pré-asymptotique pour la couverture au moyen de la décomposition de Doob.

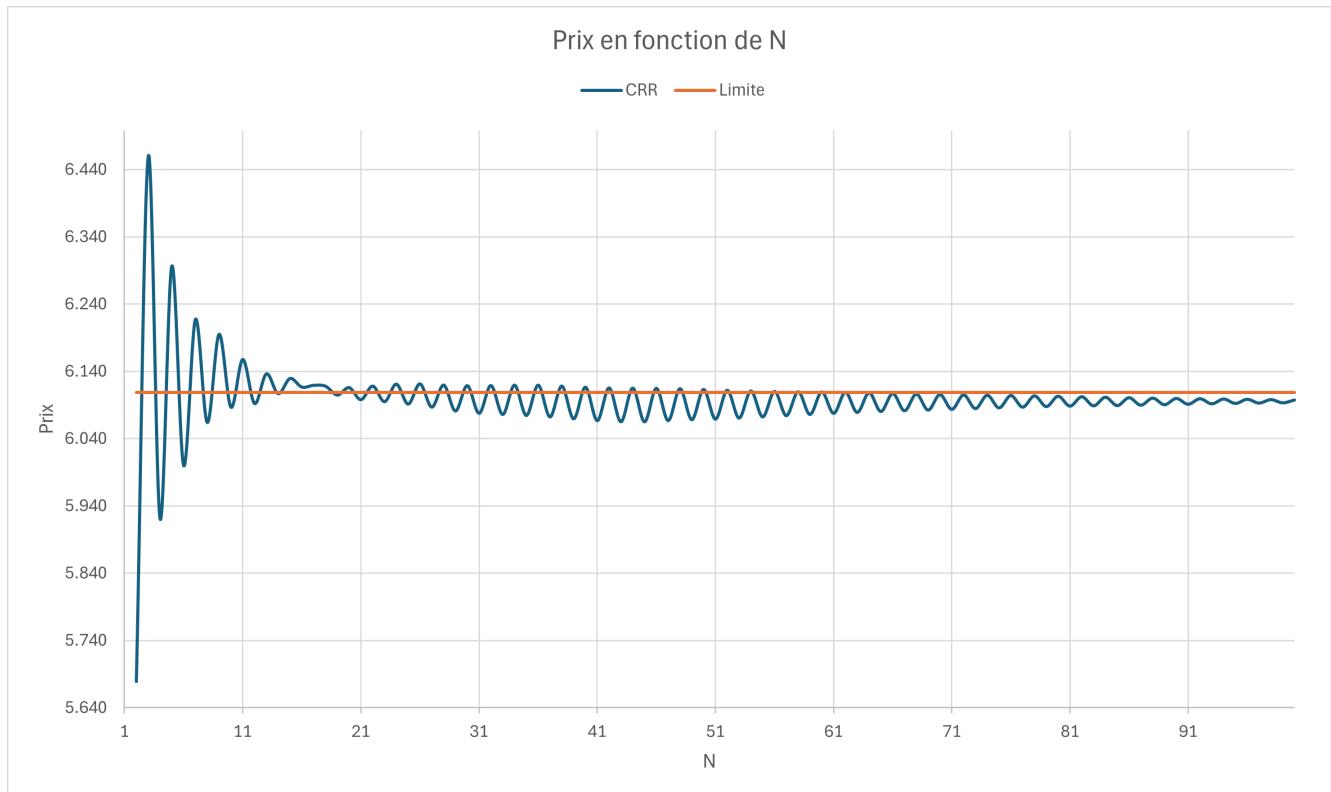


Figure 3.9: Convergence du prix de la put américaine

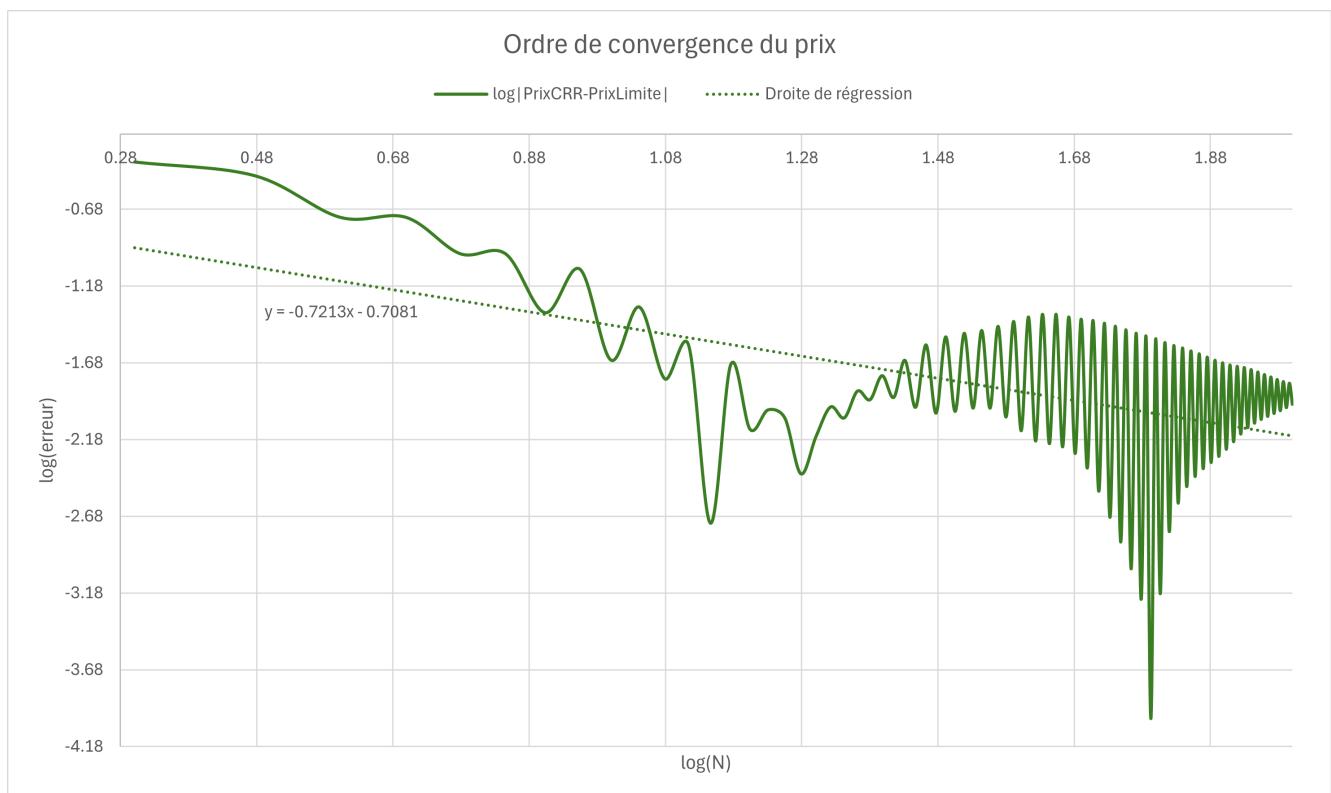


Figure 3.10: Vitesse de convergence du prix de la put américaine

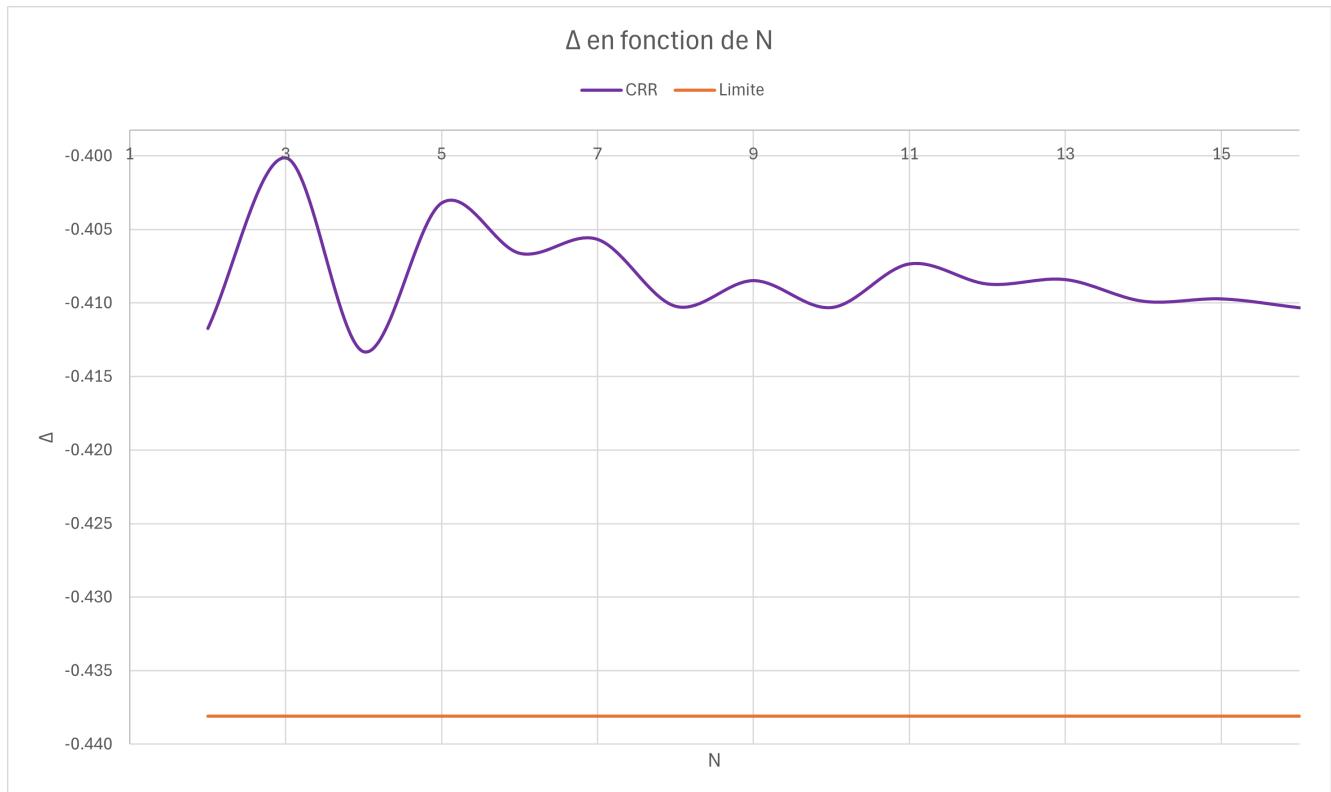


Figure 3.11: Convergence du delta de la put américaine



Figure 3.12: Vitesse de convergence du delta de la put américaine

Conclusion

Ce travail a suivi une trajectoire volontairement structurée : en partant de rien, nous avons progressivement bâti un cadre cohérent, depuis la modélisation discrète d'un marché générique jusqu'au modèle de Cox-Ross-Rubinstein. Nous avons montré comment valoriser et couvrir des options dans ce cadre, discuté des choix algorithmiques efficaces à implémenter, et mené cette réflexion jusqu'à l'écriture concrète d'un code C++ exposé à Excel au moyen d'une DLL. Enfin, nous avons mis en évidence le passage du discret au continu, qui ouvre l'accès à toute une famille de méthodes puissantes.

Nous nous sommes concentrés sur le modèle CRR, modèle discret de référence pour sa simplicité et sa clarté pédagogique. Au fil des chapitres, nous en avons souligné les forces et les limites, ainsi que des voies pour les contourner. Par exemple, les options asiatiques illustrent qu'aucun schéma n'est universel : le CRR est très performant pour la valorisation et la couverture des européennes, et pour la valorisation des américaines, tandis que Monte Carlo se révèle particulièrement adapté aux européennes et aux asiatiques, mais pas aux américaines. Nous verrons dans l'Annexe B que les méthodes aux EDP sont également très efficaces pour les européennes (et, avec des aménagements, pour les américaines), mais moins naturelles pour les asiatiques. Ainsi, nous espérons avoir fait passer l'idée générale de la complexité du problème et de l'absence de «règle d'or» unique en matière de trading d'options.

Dans la perspective d'une salle de marché dédiée aux options, ce projet constitue une pièce substantielle, mais seulement une pièce du puzzle. Nous avons joué à la fois le rôle de modélisateur probabiliste, de numéricien-algorithmicien et de programmeur; nos résultats outillent ensuite le trader et le vendeur, qui disposent de chiffres précis de prix et de stratégies de couverture. Il manque toutefois l'indispensable volet «données» : validation empirique du modèle, vérification de sa robustesse hors papier et calibration des paramètres sur données de marché via des estimateurs statistiques. De plus, l'évolution du cadre (par exemple l'ajout de dividendes dans CRR) ainsi que des développements logiciels (prise en charge d'options multi-strike, à strike flottant et/ou multi-actifs) constituent des pistes naturelles d'extension.

En conclusion, nous avons réalisé un travail fondamental pour une salle de marché, quoique partiel et naturellement perfectible. Qu'il s'agisse du CRR, de Black-Scholes ou des techniques de programmation en C++, ces compétences constituent aujourd'hui des notions cardinales de la finance quantitative et un socle de connaissances incontournable pour tout analyste quantitatif souhaitant évoluer dans l'écosystème économique et financier contemporain.

Annexe A

D'autres options européennes

Dans cette première annexe, nous présenterons les arbres de prix et de couverture et leur convergence vers les formules de Black-Scholes pour toute une série d'options européennes moins conventionnelles que les vanille. Aucun code supplémentaire n'est à écrire (seules les fonctions à exporter en DLL), ce qui montre que le code abstrait que nous avons écrit est efficace et permet d'analyser des options très différentes en n'ajoutant que quelques lignes de code. La déclaration des différents types de payoff étudiés se trouve dans `Payoff.h` et la définition des méthodes dans `Payoff.cpp` (voir Listing 1.1 et Listing 1.2), tandis que l'interface Excel utilisée est celle des options exotiques (voir Figure 2.1).

A.1 Options digitales

Nous avons implémenté trois types d'options dites digitales. La call et put digitale ont le payoff

$$H_{DC} = \mathbf{1}_{\{S_N > K\}}, \quad H_{DP} = \mathbf{1}_{\{S_N < K\}} \quad (\text{A.1})$$

Nous introduisons enfin un premier exemple d'option multi-strike, la double-digital, qui correspond à une stratégie longue une call digitale de strike K_1 et courte une call digitale de strike K_2 (avec $K_1 < K_2$). Son payoff est

$$H_{DD} = \mathbf{1}_{\{K_1 < S_N < K_2\}} \quad (\text{A.2})$$

Pour la convergence, au moyen de démonstrations très proches de celle de la section 3.1, on obtient les formules de Black-Scholes suivantes pour le prix et le delta asymptotique (où φ désigne la densité de la normale standard) :

$$P_{DC} = e^{-rT} N(d_2), \quad \delta_{DC} = \frac{e^{-rT} \varphi(d_2)}{S_0 \sigma \sqrt{T}} \quad (\text{A.3})$$

$$P_{DP} = e^{-rT} N(-d_2), \quad \delta_{DP} = -\frac{e^{-rT} \varphi(d_2)}{S_0 \sigma \sqrt{T}} \quad (\text{A.4})$$

où $d_2 = \frac{\ln(S_0/K) + (r - \sigma^2/2) T}{\sigma \sqrt{T}}$. Pour le prix et le delta de la double-digital, on effectue les calculs habituels ou l'on se rappelle sa construction, et l'on obtient

$$P_{DD} = e^{-rT} [N(d_2(K_1)) - N(d_2(K_2))], \quad \delta_{DD} = \frac{e^{-rT}}{S_0 \sigma \sqrt{T}} [\varphi(d_2(K_1)) - \varphi(d_2(K_2))] \quad (\text{A.5})$$

Nous ne montrerons qu'un exemple d'arbres et de convergence pour la double-digital, avec les paramètres CRR utilisés dans tous les exemples, $K_1 = 100$, $K_2 = 120$ et $N = 10$. On peut constater que la convergence est plus lente que pour les options vanille.

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
0.4292											0
									0	0	
							0		0	0	
						0.0613			0	0	
					0.1829		0.1231			0	
				0.3185		0.3063		0.2475			0
			0.4225		0.4572		0.4926		0.4975		
		0.4729		0.5308		0.6127		0.7426		0.995	
	0.4706		0.5281		0.6096		0.7389				1
	0.4729		0.5308		0.6127		0.7426				1
0.3922			0.4225		0.4572		0.4926		0.4975		
			0.3153		0.3185		0.3063		0.2475		0
			0.2112		0.1829		0.1231		0		
				0.1062		0.0613		0		0	
					0.0305		0		0		
						0		0		0	
							0		0		
								0		0	
									0		
										0	

Figure A.1: Prix de la double-digital

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	
0.0062										0	
									0	0	
						-0.0066			0	0	
					-0.0139			-0.014		0	
				-0.0167			-0.0223			-0.0298	
			-0.0138		-0.0197			-0.0317			-0.0637
		-0.0073		-0.0105			-0.0169			-0.0339	
	0		0		0			0		0	
	0.0083		0.0119			0.0191			0.0384		
	0.0132		0.0177		0.0254			0.0408		0.082	
0.0201		0.0188		0.0243			0.0325		0.0436		
			0.0201		0.0231			0.0231		0	
				0.0153			0.0123		0		
					0.0065			0		0	
						0		0		0	
							0		0		
								0		0	
									0		
										0	

Figure A.2: Delta de la double-digital

n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
								0	
							0	0	
					1.0348		0	0	
				2.1203		2.08			
			2.4872		3.2269		4.1807		
		2.1009		2.8789		4.4062		8.4032	
	1.308		1.7356		2.5597		4.6757		
0.4706		0.5281		0.6096		0.7389		0.995	
-0.1908	-0.3621		-0.6741		-1.3344		-3.1906		
-0.8541		-1.2559		-1.9645		-3.421		-7.4082	
-1.3547			-1.8503		-2.6143		-3.6857		
		-1.4671		-1.7545		-1.8337		0	
			-1.0987		-0.9123		0	0	
				-0.4539		0		0	
					0		0	0	
								0	
									0

Figure A.3: Bond de la double-digital

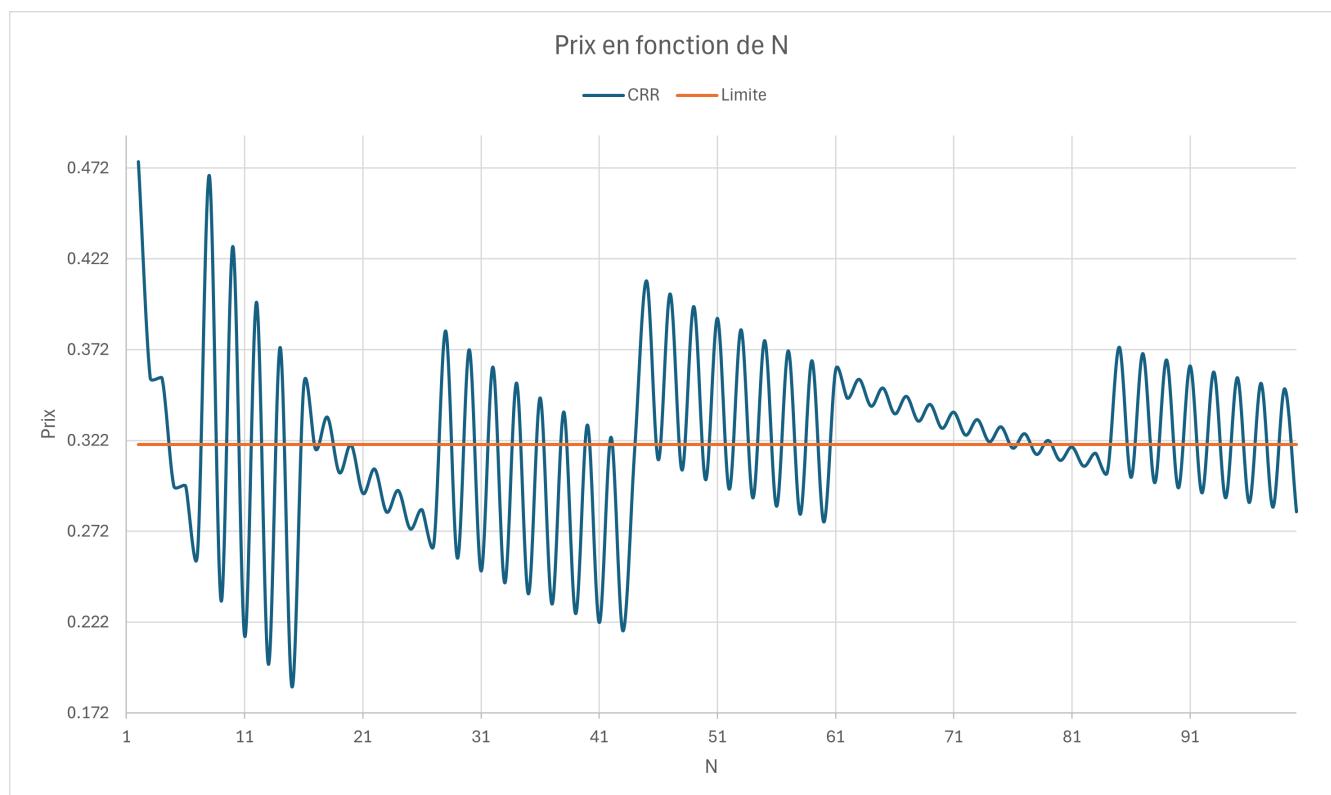


Figure A.4: Convergence du prix de la double-digital

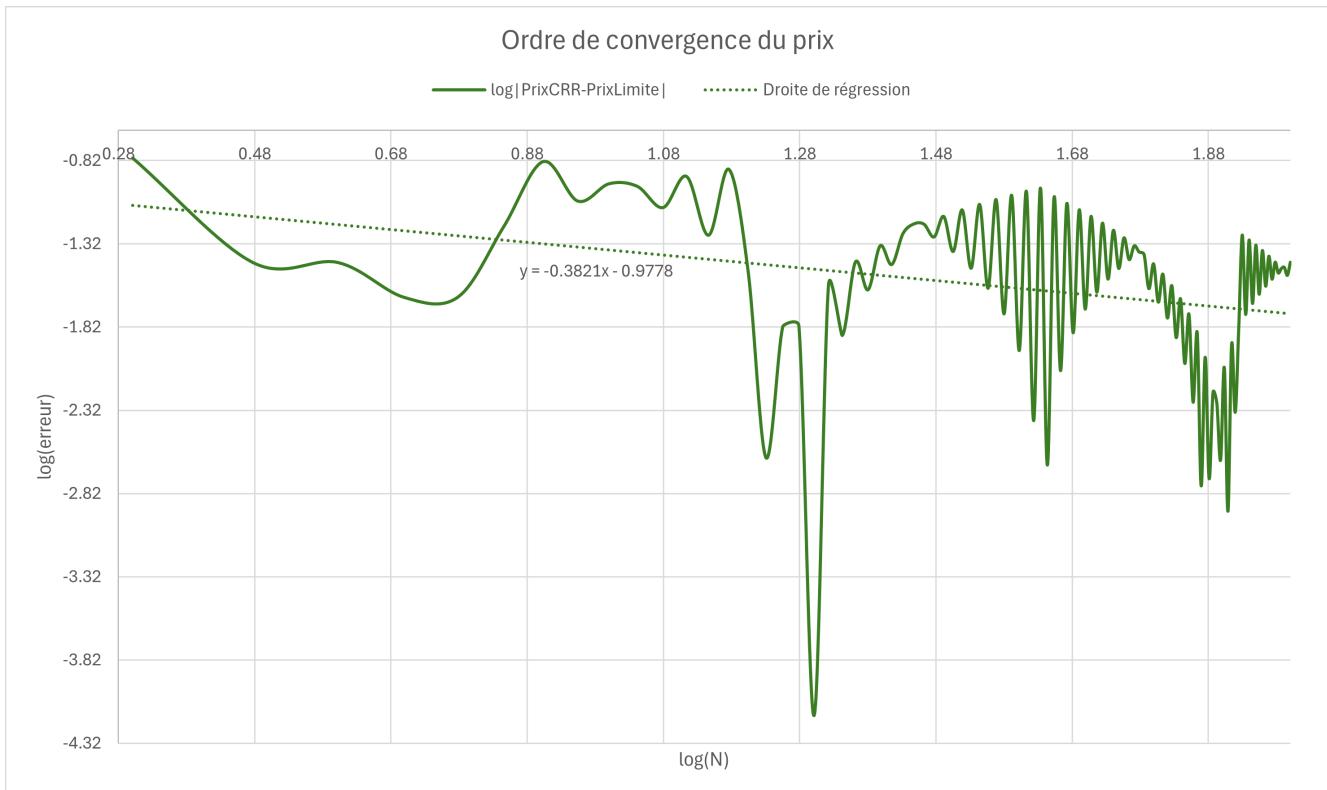


Figure A.5: Vitesse de convergence du prix de la double-digital

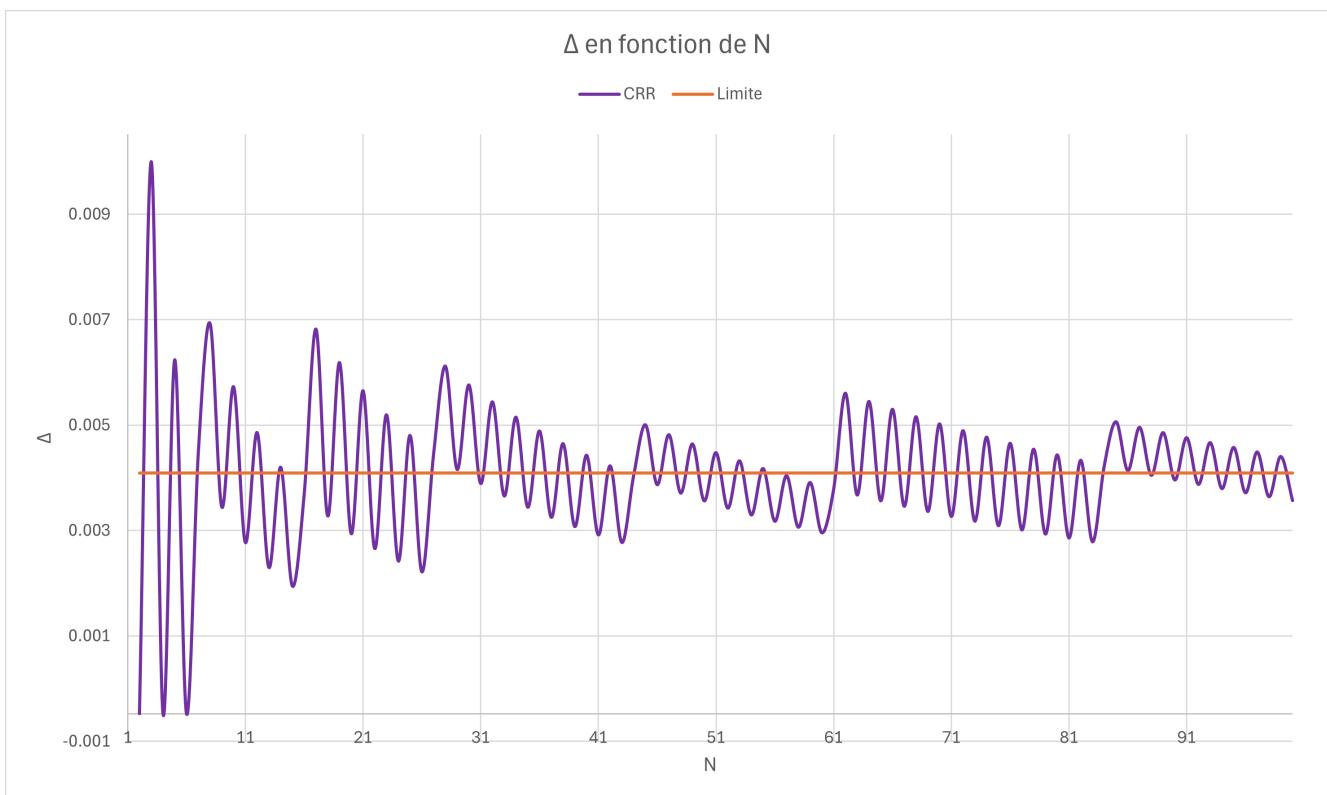


Figure A.6: Convergence du delta de la double-digital

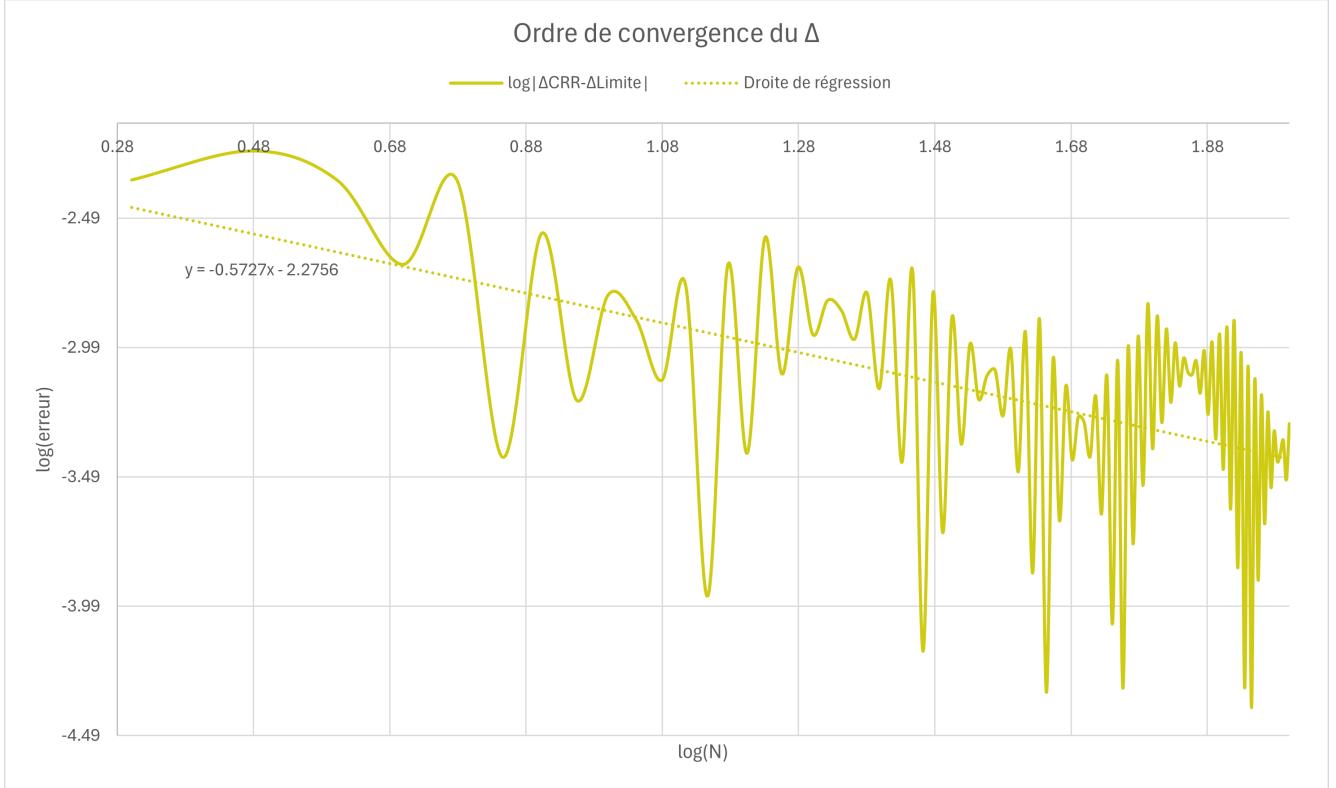


Figure A.7: Vitesse de convergence du delta de la double-digital

A.2 Bull et bear spread

Passons maintenant à des options qui sont des stratégies composées d'options vanille (elles convergeront donc à la même vitesse). La bull spread se construit en achetant une call de strike K_1 et en vendant une call de strike K_2 , tandis que la bear spread consiste à acheter une put de strike K_2 et à en vendre une de strike K_1 . Les payoffs sont donc

$$H_{\text{bull}}(S_N) = (S_N - K_1)^+ - (S_N - K_2)^+ = \begin{cases} 0, & S_N \leq K_1 \\ S_N - K_1, & K_1 < S_N < K_2 \\ K_2 - K_1, & S_N \geq K_2 \end{cases} \quad (\text{A.6})$$

$$H_{\text{bear}}(S_N) = (K_2 - S_N)^+ - (K_1 - S_N)^+ = \begin{cases} 0, & S_N \geq K_2 \\ K_2 - S_N, & K_1 < S_N < K_2 \\ K_2 - K_1, & S_N \leq K_1 \end{cases} \quad (\text{A.7})$$

Illustrons un exemple pour la bull spread, mais avant, en appliquant l'un des deux procédés présentés précédemment, on obtient les formules de Black-Scholes (avec $d_2(K) = d_1(K) - \sigma\sqrt{T}$) :

$$P_{\text{bull}} = S_0 [N(d_1(K_1)) - N(d_1(K_2))] - e^{-rT} [K_1 N(d_2(K_1)) - K_2 N(d_2(K_2))] \quad (\text{A.8})$$

$$\delta_{\text{bull}} = N(d_1(K_1)) - N(d_1(K_2)) \quad (\text{A.9})$$

$$P_{\text{bear}} = e^{-rT} [K_2 N(-d_2(K_2)) - K_1 N(-d_2(K_1))] - S_0 [N(-d_1(K_2)) - N(-d_1(K_1))] \quad (\text{A.10})$$

$$\delta_{\text{bear}} = N(d_1(K_2)) - N(d_1(K_1)) \quad (\text{A.11})$$

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
										20	
									19.9005		
								19.8015		20	
							19.703		19.9005		
						19.4144			19.8015		20
					18.5168			19.32		19.9005	
				16.8574			17.8043		19.0317		20
			14.5999			15.3665			16.4666		18.3532
		12.06			12.4884		13.0824		14.0662		16.89
9.5467			9.6408			9.7351			9.829		9.9199
7.2793		7.1288			6.8896		6.4852		5.69		3.049
	5.0848			4.6882		4.113			3.2063		1.5169
		3.0916			2.5336		1.782		0.7547		0
			1.5259			0.9795			0.3755		0
				0.5335			0.1868		0		0
					0.0929			0		0	
						0			0		0
							0			0	
								0		0	
									0		

Figure A.8: Prix de la bull spread

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
										0	
									0		
						0.0204			0		
					0.0915			0.0435			
				0.1913			0.1722			0.0928	
			0.2833			0.3044			0.3182		0.1981
		0.3436			0.3878			0.4544		0.5738	
0.3649			0.4119			0.4824			0.6088		1
0.3527		0.3892			0.4392			0.5142		0.6485	
	0.3389			0.3635			0.3901		0.4069		0.2499
		0.2819			0.2776			0.2493		0.1328	
			0.1893			0.1501			0.0706		0
				0.0891			0.0375			0	
					0.0199			0		0	
						0			0		0
							0			0	
								0		0	

Figure A.9: Delta de la bull spread

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
										19.9005
								19.703	19.8015	19.9005
						16.3867		13.2343	19.8015	19.9005
					5.7876		-4.7535		6.7994	
				-8.0477		-21.9635		-22.7889		-6.2337
			-19.94			-39.3931		-52.605		
		-27.1449		-32.0317		-42.4202		-56.3913		-99.5025
-27.9951		-29.4378		-34.6212		-37.5569		-45.8715		-60.7414
		-26.8324		-29.7493		-33.0692		-35.8105		-22.5877
			-21.908		-22.2392		-20.5977		-11.2377	
				-14.2858		-11.6314		-5.5909		0
					-6.4753		-2.7815		0	
						-1.3838		0		0
							0	0		0
								0		0
										0

Figure A.10: Bond de la bull spread

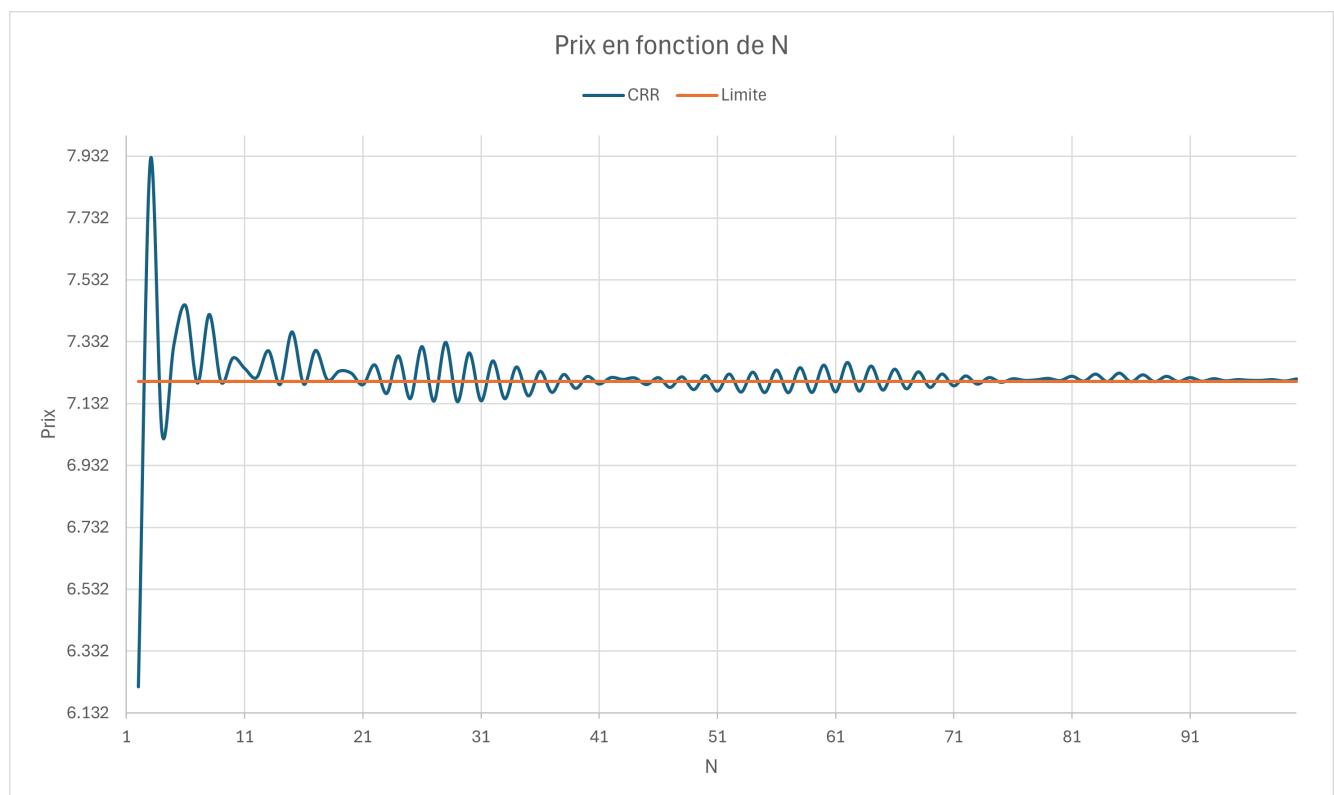


Figure A.11: Convergence du prix de la bull spread

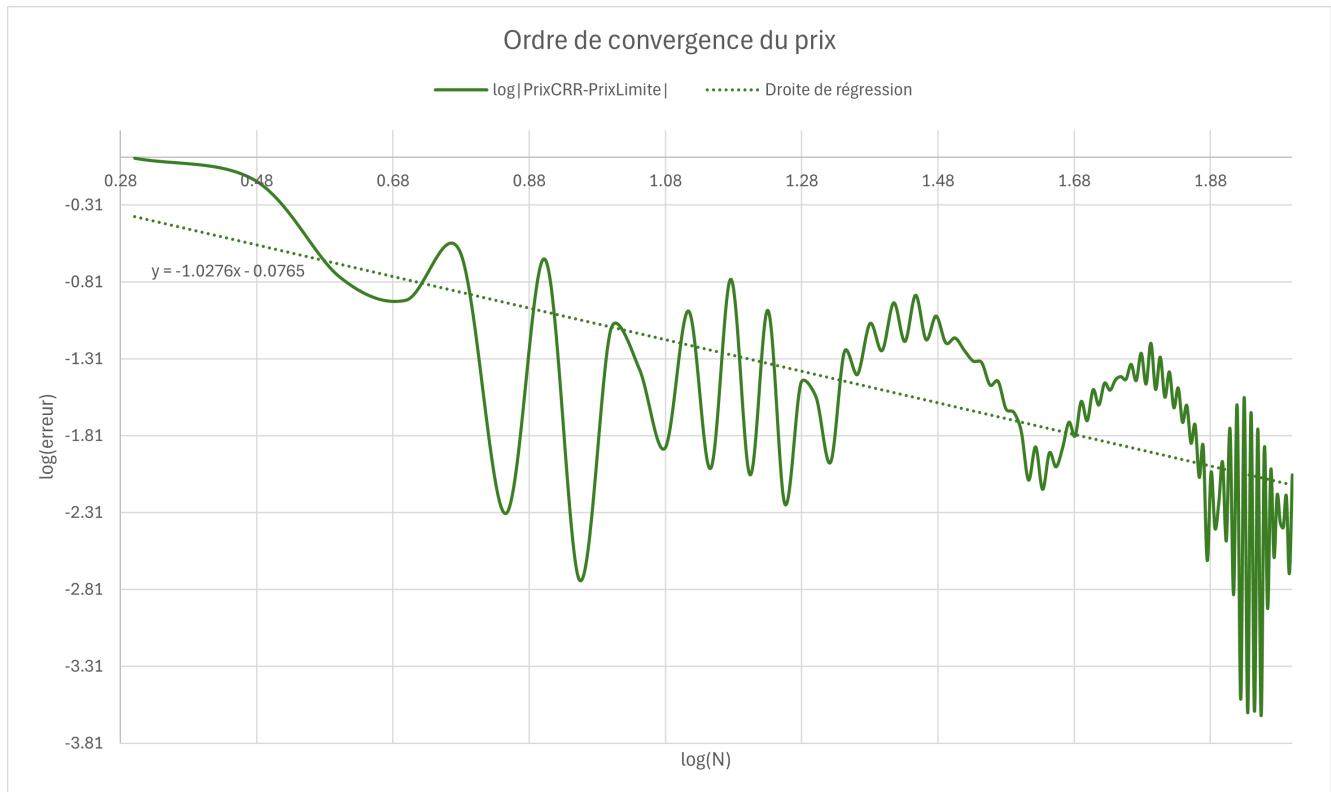


Figure A.12: Vitesse de convergence du prix de la bull spread

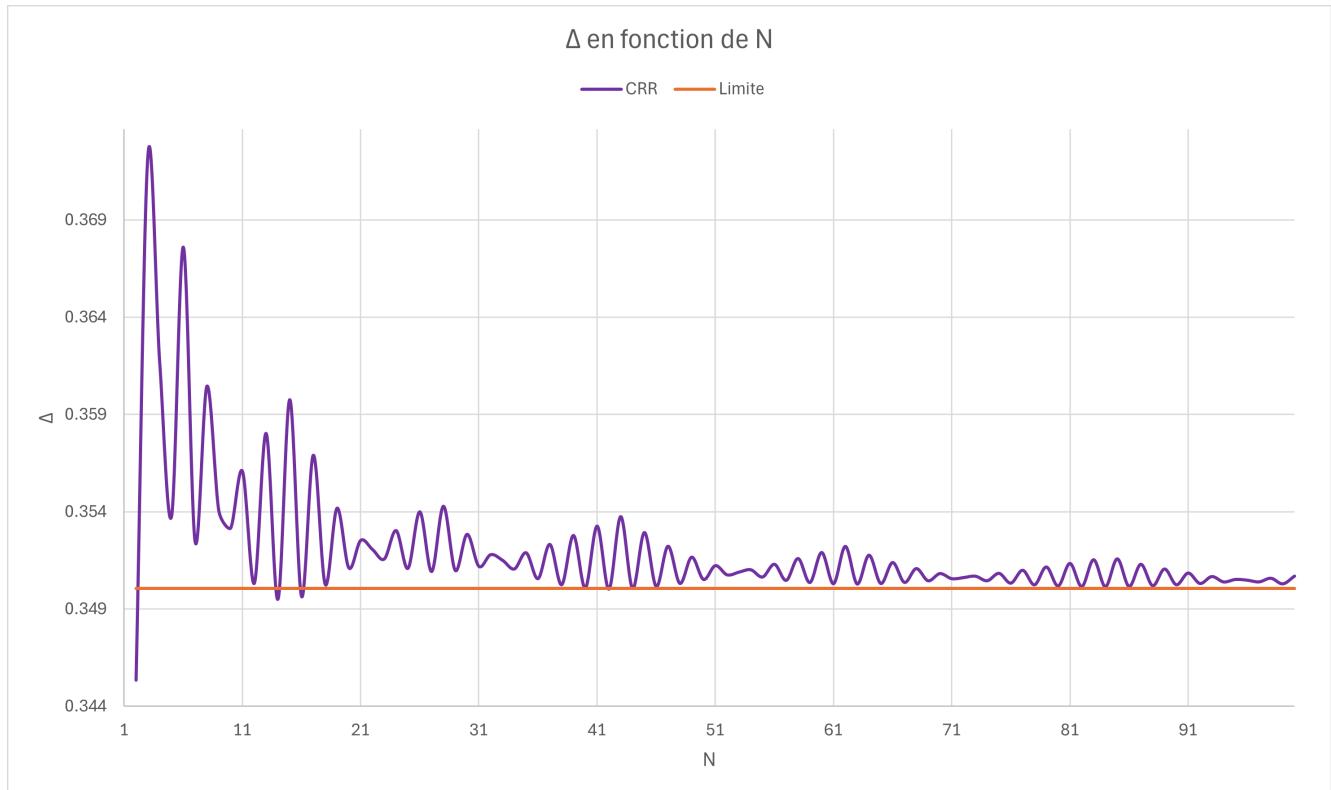


Figure A.13: Convergence du delta de la bull spread

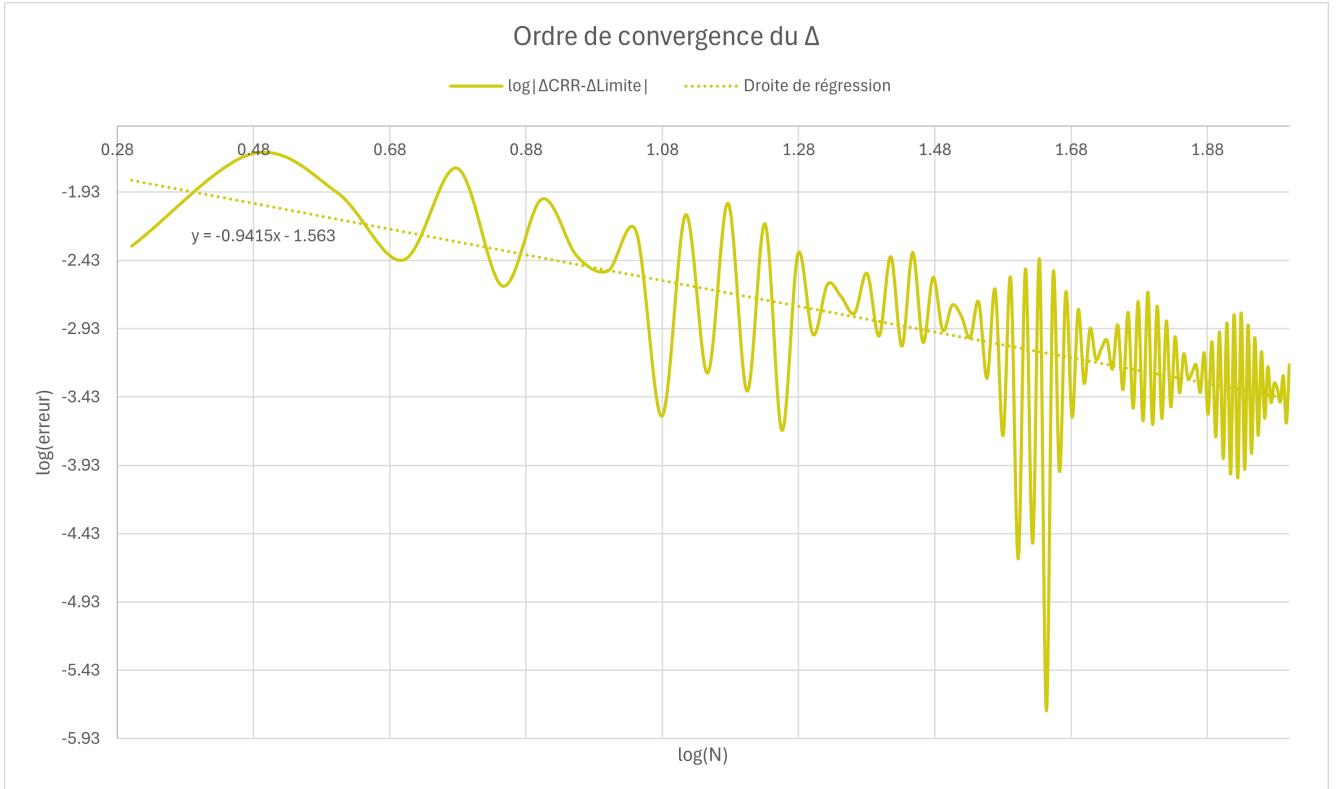


Figure A.14: Vitesse de convergence du delta de la bull spread

A.3 Strangle et butterfly

Enfin, nous présentons deux autres types de stratégies complémentaires. La strangle s'obtient en achetant une put de strike K_1 et une call de strike K_2 , elle prend de la valeur lorsque la volatilité est élevée. La butterfly s'obtient en achetant une call de strike K_1 et une call de strike K_2 et en vendant deux calls de strike $K_m = (K_1 + K_2)/2$, elle gagne, de façon complémentaire, lorsque la volatilité est faible. Les payoffs sont

$$H_s = (K_1 - S_N)^+ + (S_N - K_2)^+ = \begin{cases} K_1 - S_N, & S_N \leq K_1 \\ 0, & K_1 < S_N < K_2 \\ S_N - K_2, & S_N \geq K_2 \end{cases} \quad (\text{A.12})$$

$$H_b = (S_N - K_1)^+ - 2(S_N - K_m)^+ + (S_N - K_2)^+ = \begin{cases} S_N - K_1, & K_1 < S_N \leq K_m \\ K_2 - S_N, & K_m < S_N \leq K_2 \\ 0, & \text{sinon} \end{cases} \quad (\text{A.13})$$

Lorsque $K_1 = K_2$, la strangle prend le nom de straddle, une stratégie célèbre dont le payoff est toujours > 0 (sauf si $S_N = K$). Avant l'exemple, rappelons les formules de Black-Scholes :

$$P_s = S_0 [N(d_1(K_2)) - N(-d_1(K_1))] + e^{-rT} [K_1 N(-d_2(K_1)) - K_2 N(d_2(K_2))] \quad (\text{A.14})$$

$$\delta_s = N(d_1(K_1)) + N(d_1(K_2)) - 1 \quad (\text{A.15})$$

$$P_b = S_0 [N(d_1(K_1)) - 2N(d_1(K_m)) + N(d_1(K_2))] - e^{-rT} [K_1 N(d_2(K_1)) - 2K_m N(d_2(K_m)) + K_2 N(d_2(K_2))] \quad (\text{A.16})$$

$$\delta_b = N(d_1(K_1)) - 2N(d_1(K_m)) + N(d_1(K_2)) \quad (\text{A.17})$$

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
8.8399	8.3449	9.4233	9.3066	8.3449	7.4666	11.4743	14.7605	18.9259	23.5521	28.2616	32.8151
											73.5134
										61.7476	
										50.7688	50.5994
										40.5263	40.2976
										31.1635	30.3987
										23.0552	21.3874
										16.5904	12.59
										12.0011	6.2637
										9.3066	0
										7.5317	
										4.8471	
										5.5085	
										4.2342	
										2.6775	
										2.2655	
										1.3.7568	
										1.163	
										0	

n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
								-119.403	
							-118.8089		
						-118.2179		-119.403	
					-114.4114		-118.8089		
				-103.3247		-111.7492		-119.403	
			-84.3153		-93.2712		-105.8069		
		-59.6148		-66.1491		-75.726		-93.2688	
	-32.5608		-35.5105		-39.6885		-46.4024		
-6.2013		-5.8324		-5.227		-4.0479		0	
17.3658		20.0963		23.7873		29.1823		38.2661	
	41.1064		46.226		53.0395		62.7044		76.9148
		62.5276		69.1269		77.4271		87.7698	
			79.4545		85.9056		92.924		99.5025
				90.5765		95.2432		99.0075	
					96.1532		98.5149		99.5025
						98.0248		99.0075	
							98.5149		99.5025
								99.0075	
									99.5025

Figure A.17: Bond de la strangle

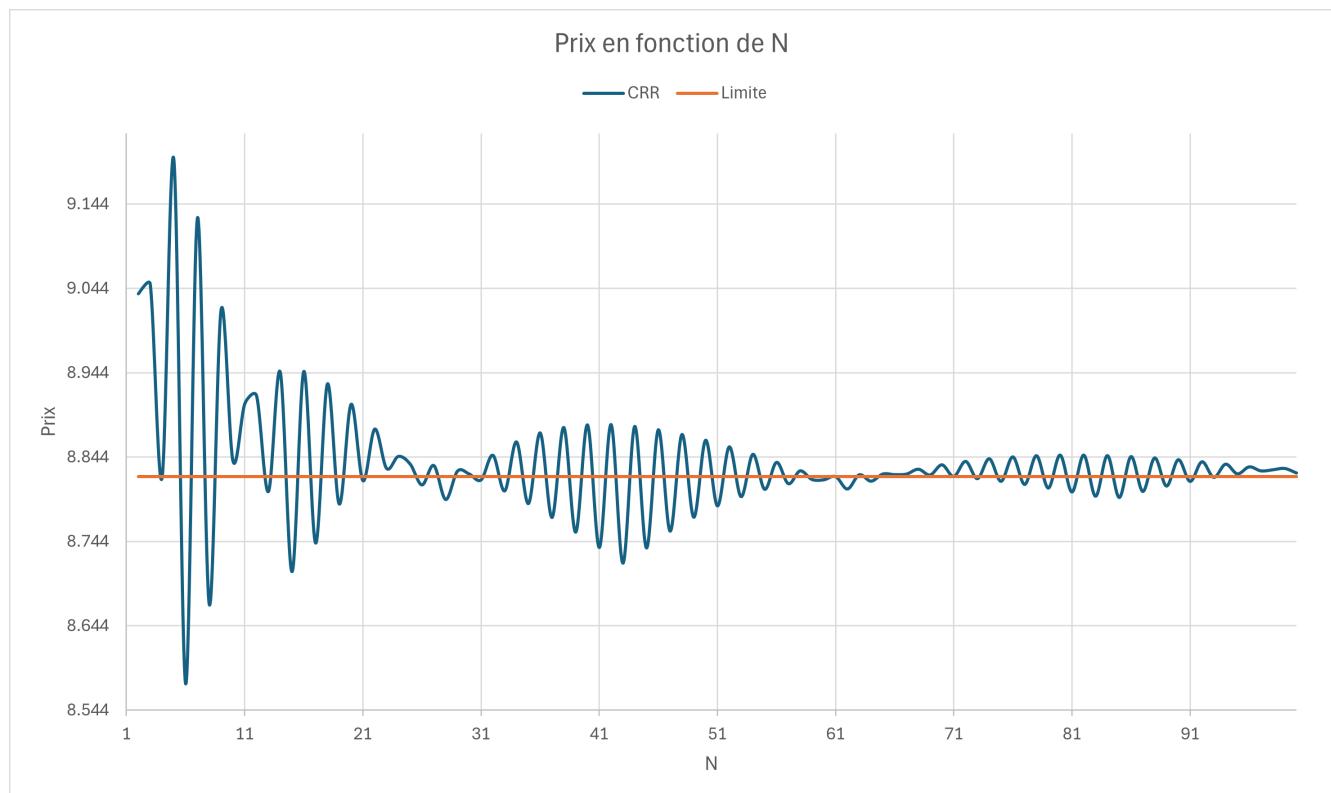


Figure A.18: Convergence du prix de la strangle

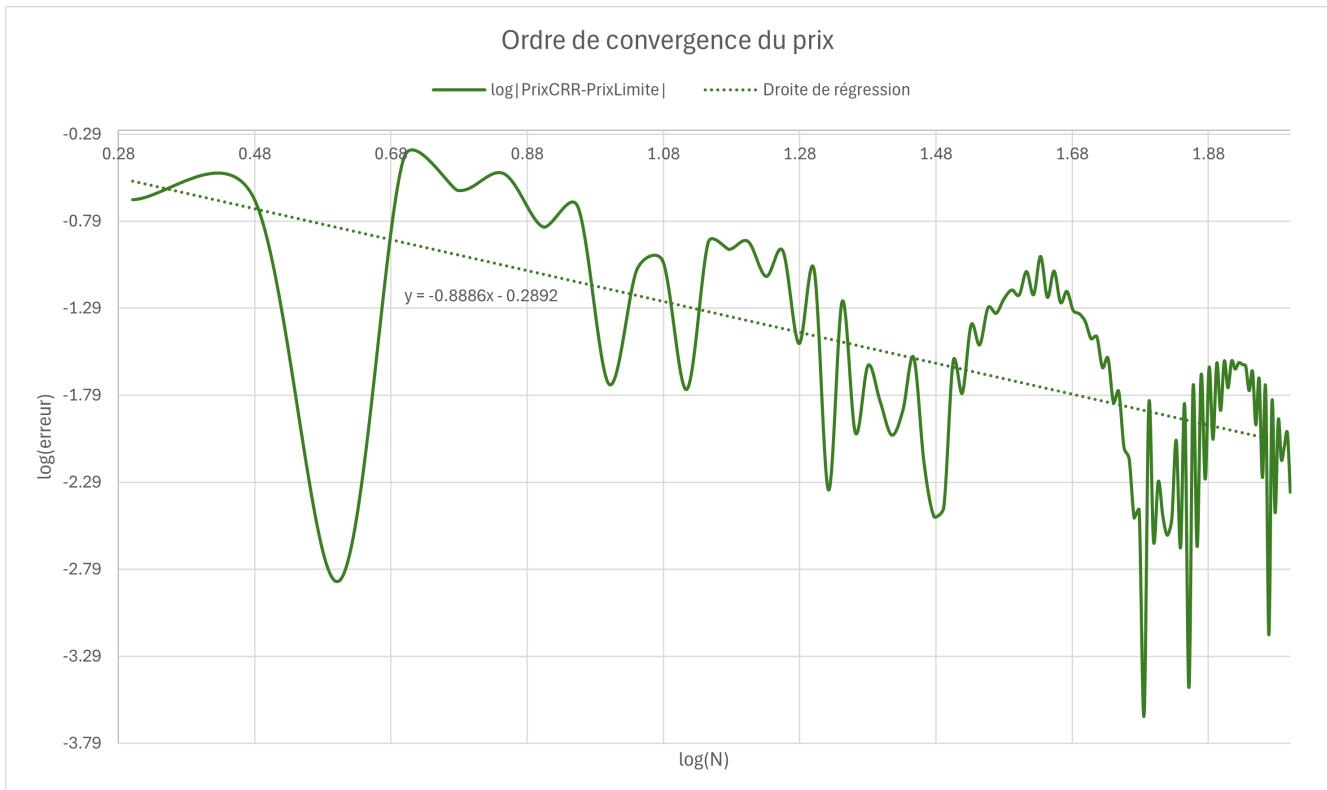


Figure A.19: Vitesse de convergence du prix de la strangle

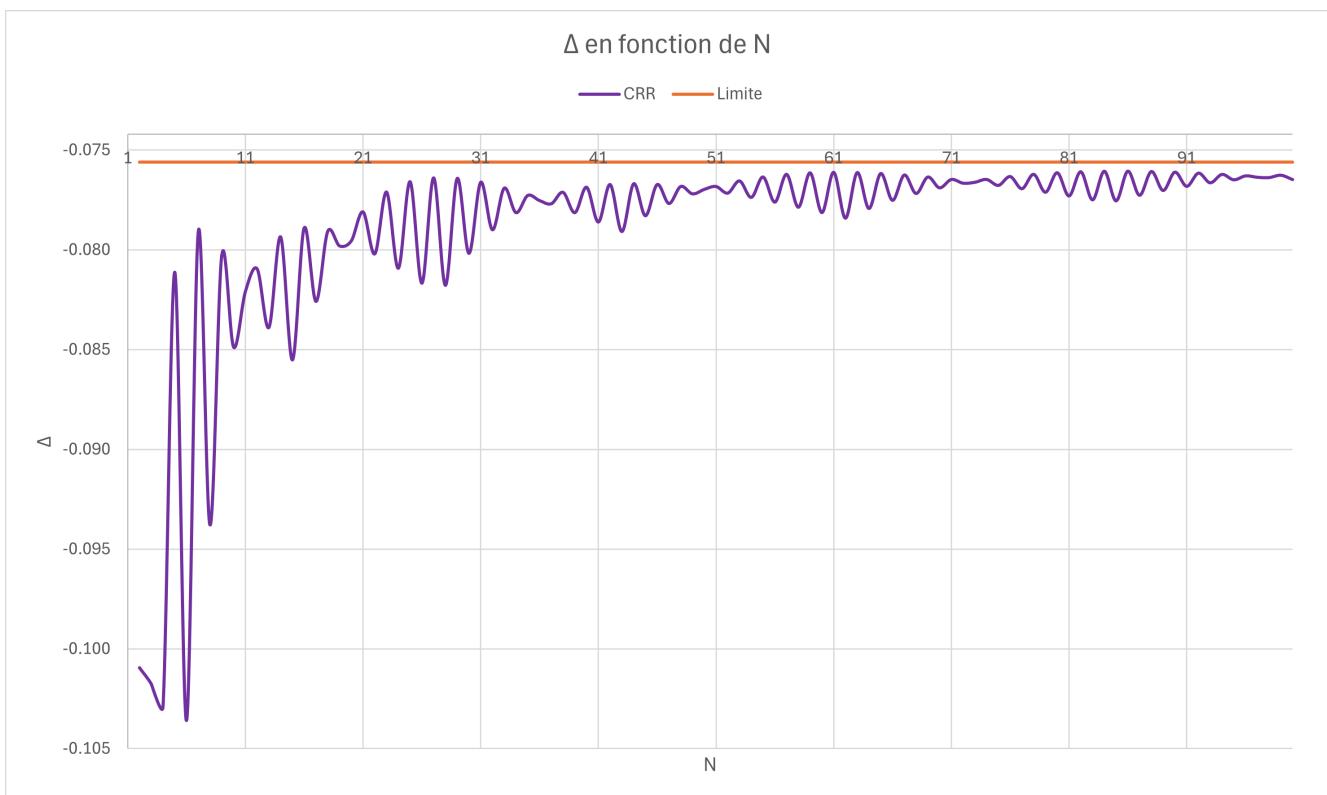


Figure A.20: Convergence du delta de la strangle

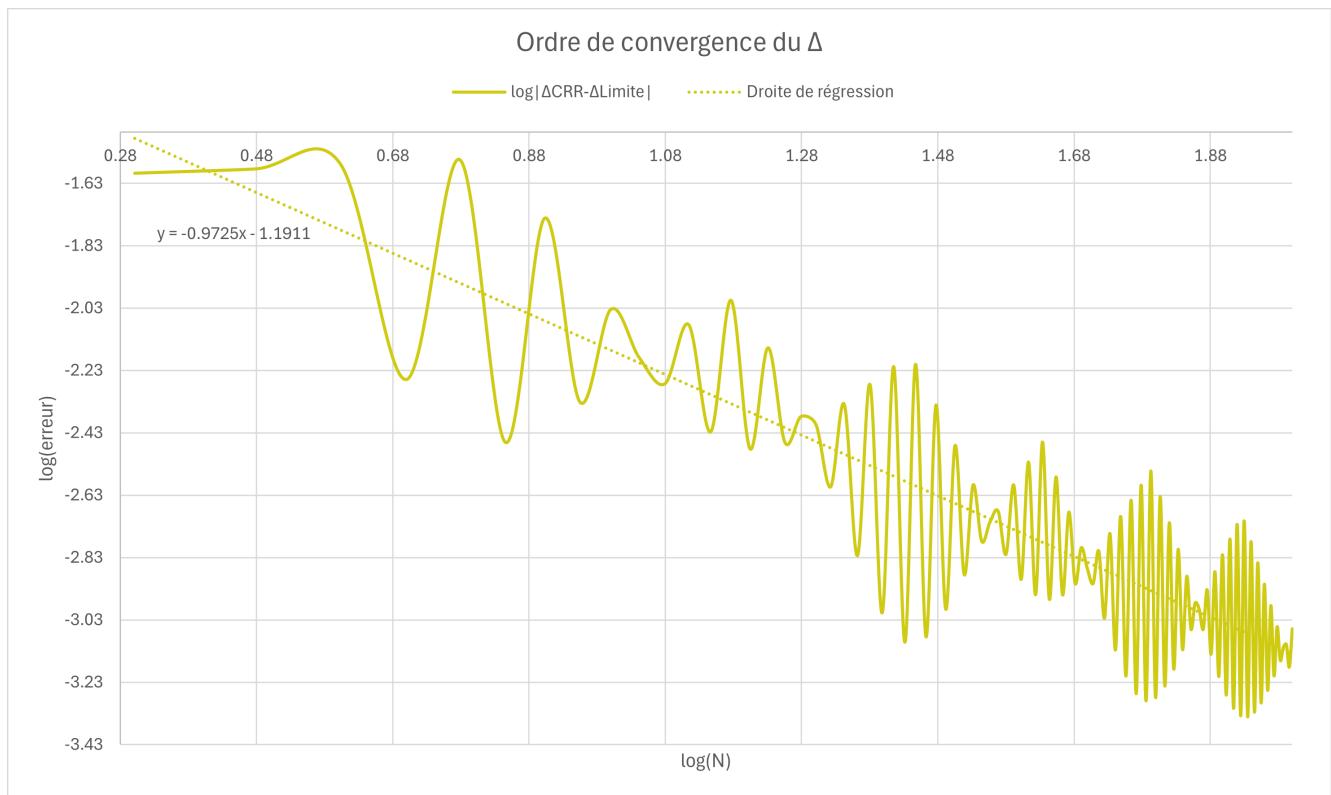


Figure A.21: Vitesse de convergence du delta de la strangle

Annexe B

Modélisation en temps continu

Dans cette annexe, nous nous concentrerons sur des modèles continus, d'abord à volatilité constante (Black-Scholes), puis à volatilité locale. Nous commencerons par la démonstration menant à l'équation aux dérivées partielles (EDP) parabolique de Black-Scholes, avant de la résoudre numériquement par la méthode des différences finies, selon le schéma implicite de Crank-Nicolson. Nous présenterons les codes C++, dont l'architecture des classes s'inspire de Capiński and Zastawniak (2012, chap. 6), avec les adaptations nécessaires via des templates. Enfin, nous illustrerons le fonctionnement de la nouvelle interface Excel développée, ainsi que des surfaces de prix et de delta pour plusieurs types d'options.

B.1 Modèle Black–Scholes

On suppose que le sous-jacent $(S_t)_{t \in [0, T]}$ suit un mouvement brownien géométrique

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \quad \mu \in \mathbb{R}, \sigma > 0 \quad (\text{B.1})$$

(3.13) est une solution de cette équation différentielle stochastique. Soit $V(t, S)$ le prix d'un dérivé européen de payoff $f(S_T)$, avec $V \in C^{1,2}([0, T] \times [S_{\min}, S_{\max}])$. Par la formule d'Itô appliquée à $V(t, S_t)$,

$$dV = \left(\frac{\partial V}{\partial t} + \mu S_t \frac{\partial V}{\partial S} + \frac{1}{2} \sigma^2 S_t^2 \frac{\partial^2 V}{\partial S^2} \right) dt + \sigma S_t \frac{\partial V}{\partial S} dW_t \quad (\text{B.2})$$

Considérons le portefeuille autofinancé

$$\Pi_t = V(t, S_t) - \Delta_t S_t, \quad \text{où } \Delta_t := \frac{\partial V}{\partial S}(t, S_t) \quad (\text{B.3})$$

Alors le terme stochastique est annulé et

$$d\Pi_t = dV - \Delta_t dS_t = \left(\frac{\partial V}{\partial t} + \frac{1}{2} \sigma^2 S_t^2 \frac{\partial^2 V}{\partial S^2} \right) dt \quad (\text{B.4})$$

En absence d'arbitrage, tout portefeuille sans risque croît au taux sans risque r :

$$d\Pi_t = r \Pi_t dt = r \left(V - S_t \frac{\partial V}{\partial S} \right) dt \quad (\text{B.5})$$

En identifiant les termes en dt , on obtient l'EDP de Black-Scholes :

$$\frac{\partial V}{\partial t}(t, S) + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 V}{\partial S^2}(t, S) + rS \frac{\partial V}{\partial S}(t, S) - rV(t, S) = 0 \quad (\text{B.6})$$

Nous avons donc la condition terminale

$$V(T, S) = f(S) \quad (\text{B.7})$$

tandis que les conditions de bord inférieure et supérieure sont

$$V(t, S_{\min}) = f(S_{\min}) e^{-r(T-t)}, \quad V(t, S_{\max}) = f(S_{\max}) e^{-r(T-t)} \quad (\text{B.8})$$

Passons maintenant à la première classe C++ représentant une EDP parabolique générale sur $[0, T] \times [x_{\min}, x_{\max}]$:

$$\frac{\partial u}{\partial t}(t, x) = a(t, x) \frac{\partial^2 u}{\partial x^2}(t, x) + b(t, x) \frac{\partial u}{\partial x}(t, x) + c(t, x) u(t, x) + d(t, x) \quad (\text{B.9})$$

Listing B.1: ParabPDE.h

```

1 #ifndef PARABPDE_H
2 #define PARABPDE_H
3
4 #include <stdexcept>
5 #include <cmath>
6
7 namespace pde {
8
9     /**
10      * @brief Interface pour EDP paraboliques de type général.
11      */
12     class ParabPDE {
13     protected:
14     double T_, Smin_, Smax_; // < Domaine de l'EDP
15
16     public:
17         ParabPDE(double T, double Smin, double Smax)
18             : T_(T), Smin_(Smin), Smax_(Smax)
19         {
20             if (T_ <= 0.0 || Smin_ >= Smax_ || Smin_ < 0)
21                 throw std::invalid_argument("Domaine EDP invalide");
22         }
23
24         /**
25          * @brief Coefficients de l'équation différentielle partielle.
26          */
27         virtual double a(double t, double S) const = 0;
28         virtual double b(double t, double S) const = 0;
29         virtual double c(double t, double S) const = 0;
30         virtual double d(double t, double S) const = 0;
31
32         /**
33          * @brief Terminal Boundary Condition.
34          * @return v(T, S).
35          */
36         virtual double Terminal(double S) const = 0;
37 }
```

```

38  /**
39   * @brief Lower Boundary Condition.
40   * @return  $v(t, S_{min})$ .
41   */
42   virtual double Lower(double t) const = 0;
43
44  /**
45   * @brief Upper Boundary Condition.
46   * @return  $v(t, S_{max})$ .
47   */
48   virtual double Upper(double t) const = 0;
49
50   double T() const { return T_; }
51   double Smin() const { return Smin_; }
52   double Smax() const { return Smax_; }
53 };
54
55 } // namespace pde
56
57 #endif // PARABPDE_H

```

Pour l'équation parabolique spécifique de Black-Scholes, nous avons défini une classe qui hérite de ParabPDE. Toutefois, cette classe modélise plus généralement des équations de diffusion de type Black-Scholes avec volatilité locale (thème développé dans la section suivante). En plus du template portant sur le payoff, on ajoute donc un second template pour la fonction de volatilité choisie, dans notre cas, la volatilité constante. Les différentes fonctions de volatilité disponibles sont implémentées sous forme de classes, déclarées dans Volatility.h et définies dans Volatility.cpp.

Listing B.2: Volatility.h

```

1 #include "pch.h"
2
3 #ifndef VOLATILITY_H
4 #define VOLATILITY_H
5
6 #include <stdexcept>
7
8 namespace pde {
9
10 /**
11  * @brief Classe abstraite représentant la volatilité du sous-jacent
12  *
13  */
14 class Volatility {
15 public:
16 /**
17  * @brief Calcule la valeur de la volatilité pour un prix  $S$  du
18  * sous-jacent à l'instant  $t$ .
19  * @param t Temps.
20  * @param S Prix du sous-jacent.
21  * @return Valeur de la volatilité.
22  */
23   virtual double operator()(double t, double S) const = 0;

```

```

22     };
23
24     /**
25      * @brief Volatilité dans le modèle de Black-Scholes.
26      */
27     class BSVol : public Volatility {
28     private:
29         double sigma_;
30
31     public:
32         /**
33          * @param sigma Valeur constante de la volatilité.
34          */
35         BSVol(double sigma);
36
37         double operator()(double t, double S) const override;
38     };
39
40     /**
41      * @brief Volatilité dans un modèle à volatilité locale.
42      */
43     class LocalVol : public Volatility {
44     private:
45         double alfa_, beta_;
46
47     public:
48         /**
49          * @param alfa Coefficient du temps.
50          * @param beta Coefficient du sous-jacent.
51          */
52         LocalVol(double alfa, double beta);
53
54         double operator()(double t, double S) const override;
55     };
56
57 } // namespace pde
58
59 #endif // VOLATILITY_H

```

Listing B.3: Volatility.cpp

```

1 #include "pch.h"
2 #include "Volatility.h"
3
4 namespace pde {
5
6     BSVol::BSVol(double sigma)
7         : sigma_(sigma)
8     {
9         if (sigma_ < 0.0)
10             throw std::invalid_argument("Sigma doit être >= 0");
11     }
12

```

```

13     double BSVol::operator()(double t, double S) const {
14         return sigma_;
15     }
16
17     LocalVol::LocalVol(double alfa, double beta)
18         : alfa_(alfa), beta_(beta)
19     {
20         if (alfa_ < 0.0 || beta_ < 0.0)
21             throw std::invalid_argument("Coefficient doit être >= 0");
22     }
23
24     double LocalVol::operator()(double t, double S) const {
25         return (alfa_ / (t + 1)) + (beta_ / (S + 1));
26     }
27
28 } // namespace pde

```

Listing B.4: Diffusion.h

```

1 #include "pch.h"
2
3 #ifndef DIFFUSION_H
4 #define DIFFUSION_H
5
6 #include "ParabPDE.h"
7 #include <cmath>
8
9 namespace pde {
10
11 /**
12 * @brief EDP d'un modèle de diffusion.
13 * @tparam TPayoff Type de payoff.
14 * @tparam TVol Type de volatilité.
15 */
16 template<typename TPayoff, typename TVol>
17 class Diffusion : public ParabPDE {
18 private:
19     TPayoff payoff_;
20     TVol vol_;
21     double R_;
22
23 public:
24     Diffusion(double T, double Smin, double Smax, double R, const
25               TPayoff& payoff, const TVol& vol)
26     : ParabPDE(T, Smin, Smax), R_(R), payoff_(payoff), vol_(vol)
27     {}
28
29     double a(double t, double S) const override {
30         return -0.5 * std::pow(vol_(t, S) * S, 2.0);
31     }
32
33     double b(double t, double S) const override {

```

```

34         return -R_ * S;
35     }
36
37     double c(double t, double S) const override {
38         return R_;
39     }
40
41     double d(double t, double S) const override {
42         return 0;
43     }
44
45     double Terminal(double S) const override {
46         return payoff_(S);
47     }
48
49     double Lower(double t) const override {
50         return payoff_(Smin_) * std::exp(-R_ * (T_ - t));
51     }
52
53     double Upper(double t) const override {
54         return payoff_(Smax_) * std::exp(-R_ * (T_ - t));
55     }
56 };
57
58 } // namespace pde
59
60 #endif // DIFFUSION_H

```

Dans la majorité des cas, on ne dispose pas d'une solution analytique de l'EDP (B.9), d'où le recours à une approche numérique. Nous utilisons ici la méthode des différences finies, fondée sur une discrétisation du domaine de la solution. L'indice de temps $i = 0, \dots, i_{\max}$ et l'indice d'espace (le sous-jacent) $j = 0, \dots, j_{\max}$.

$$t_i = i \Delta t, \quad \Delta t = \frac{T}{i_{\max}}, \quad x_j = x_{\min} + j \Delta x, \quad \Delta x = \frac{x_{\max} - x_{\min}}{j_{\max}} \quad (\text{B.10})$$

Abréviations (valeurs échantillonées) :

$$u_{i,j} = u(t_i, x_j), \quad a_{i,j} = a(t_i, x_j), \quad b_{i,j} = b(t_i, x_j), \quad c_{i,j} = c(t_i, x_j), \quad d_{i,j} = d(t_i, x_j) \quad (\text{B.11})$$

$$f_j = f(x_j), \quad f_{\ell,i} = f_{\ell}(t_i), \quad f_{u,i} = f_u(t_i) \quad (\text{B.12})$$

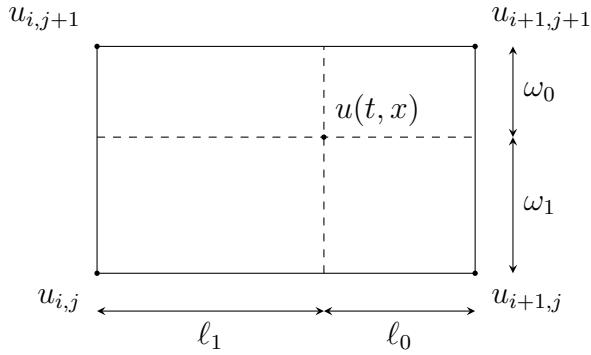
La classe `FDMethod` contient un membre qui représente une EDP (comme template, afin de pouvoir traiter une diffusion de type Black-Scholes ou une autre. De plus, à son tour, `Diffusion` est instanciée avec deux autres templates). Donc la première chose faite est de déclarer et de définir les méthodes qui construisent l'appareil de discrétisation exposé plus haut. En outre, nous trouverons trois autres méthodes :

- Interpolation bilinéaire : pour un point $(t, x) \in [t_i, t_{i+1}] \times [x_j, x_{j+1}]$, on pose

$$\ell_1 = \frac{t - t_i}{\Delta t}, \quad \ell_0 = 1 - \ell_1, \quad \omega_1 = \frac{x - x_j}{\Delta x}, \quad \omega_0 = 1 - \omega_1 \quad (\text{B.13})$$

et l'on approxime

$$u(t, x) \approx \ell_1 \omega_1 u_{i+1,j+1} + \ell_1 \omega_0 u_{i+1,j} + \ell_0 \omega_1 u_{i,j+1} + \ell_0 \omega_0 u_{i,j} \quad (\text{B.14})$$



- Delta par différences centrales : pour un temps fixé t_i , le δ discret (dérivée spatiale) en x_j est approché par

$$\delta_{i,j} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2 \Delta x} \quad (\text{B.15})$$

- Échantillonnage pour l'export Excel : un maillage régulier de $[0, T] \times [x_{\min}, x_{\max}]$ en $121 = 11 \times 11$ points est généré; deux matrices stockent les valeurs $\{u\}$ et $\{\delta\}$ en ces 121 nœuds (utilisés pour tracer les surfaces de prix et de delta dans Excel).

Listing B.5: FDMETHOD.H

```

1 #ifndef FDMETHOD_H
2 #define FDMETHOD_H
3
4 #include <vector>
5 #include <stdexcept>
6
7 namespace pde{
8
9 /**
10  * @brief Méthodes aux différences finies pour la résolution numérique d'EDP.
11  * @tparam TPDE Type d'EDP.
12  */
13 template<typename TPDE>
14 class FDMETHOD {
15 protected:
16     TPDE pde_;
17     int imax_, jmax_;                                ///< Nombre de pas en temps et en prix
18     double dt_, dS_;                                 ///< Pas en temps et en prix
19     std::vector<std::vector<double>> V;           ///< Matrice de solution
20
21 public:
22     FDMETHOD(const TPDE& pde, int imax, int jmax);
23
24     double t(double i) const { return dt_ * i; }
25     double S(int j) const { return pde_.Smin() + dS_ * j; }
26
27     double a(double i, int j) const { return pde_.a(t(i), S(j)); }

```

```

28     double b(double i, int j) const { return pde_.b(t(i), S(j)); }
29     double c(double i, int j) const { return pde_.c(t(i), S(j)); }
30     double d(double i, int j) const { return pde_.d(t(i), S(j)); }
31
32     double f(int j) const { return pde_.Terminal(S(j)); }
33     double fu(int i) const { return pde_.Upper(t(i)); }
34     double fl(int i) const { return pde_.Lower(t(i)); }
35
36 /**
37  * @brief Prix de l'option par interpolation bilinéaire.
38  * @return v(t, S).
39  */
40     double v(double t, double S) const;
41
42 /**
43  * @brief Delta de l'option par schéma aux différences centrales
44  *
45  * @return delta(t, S).
46  */
47     double delta(double t, double S) const;
48
49 /**
50  * @brief Grille : prix et delta.
51  */
52     struct Grid {
53         std::vector<std::vector<double>> val;    ///< Prix.
54         std::vector<std::vector<double>> del;    ///< Delta.
55     };
56
57 /**
58  * @brief Génère un maillage fixe de 11x11 points pour (t, S).
59  * @return Matrices 11x11 contenant v(t_i, S_j) et delta(t_i, S_j)
60  *
61  */
62     Grid grid() const;
63 };
64
65 template<typename TPDE>
66 FDMethod<TPDE>::FDMethod(const TPDE& pde, int imax, int jmax)
67 : pde_(pde), imax_(imax), jmax_(jmax)
68 {
69     if (imax_ <= 0) throw std::invalid_argument("Nt doit être >= 1");
70     ;
71     if (jmax_ <= 1) throw std::invalid_argument("NS doit être >= 2");
72     ;
73
74     dS_ = (pde_.Smax() - pde_.Smin()) / jmax_;
75     dt_ = pde_.T() / imax_;
76     V.resize(imax_ + 1);
77     for (int i = 0; i <= imax_; i++)
78         V[i].resize(jmax_ + 1);
79 }

```

```

76
77     template<typename TPDE>
78     double FDMETHOD<TPDE>::v(double t, double S) const {
79         if (t < 0 || t > pde_.T())
80             throw std::out_of_range("t hors du domaine");
81         if (S < pde_.Smin() || S > pde_.Smax())
82             throw std::invalid_argument("S hors du domaine");
83         if (S == pde_.Smax())
84             return pde_.Upper(t);
85         if (t == pde_.T())
86             return pde_.Terminal(S);
87
88         int i = (int)(t / dt_);
89         int j = (int)((S - pde_.Smin()) / dS_);
90         double l1 = (t - FDMETHOD<TPDE>::t(i)) / dt_, l0 = 1.0 - l1;
91         double w1 = (S - FDMETHOD<TPDE>::S(j)) / dS_, w0 = 1.0 - w1;
92         return l1 * w1 * V[i + 1][j + 1] + l1 * w0 * V[i + 1][j]
93             + l0 * w1 * V[i][j + 1] + l0 * w0 * V[i][j];
94     }
95
96     template<typename TPDE>
97     double FDMETHOD<TPDE>::delta(double t, double S) const {
98         if (t < 0 || t > pde_.T())
99             throw std::out_of_range("t hors du domaine");
100        if (S < pde_.Smin() || S > pde_.Smax())
101            throw std::invalid_argument("S hors du domaine");
102
103        double dlt;
104        int i = (int)(t / dt_);
105        int j = (int)((S - pde_.Smin()) / dS_);
106        if (j == 0)
107            dlt = (V[i][1] - V[i][0]) / dS_;
108        else if (j == jmax_)
109            dlt = (V[i][jmax_] - V[i][jmax_ - 1]) / dS_;
110        else
111            dlt = (V[i][j + 1] - V[i][j - 1]) / (2.0 * dS_);
112        return std::min<double>(1.0, std::max<double>(-1.0, dlt));
113    }
114
115    template<typename TPDE>
116    typename FDMETHOD<TPDE>::Grid FDMETHOD<TPDE>::grid() const {
117        int N = 10;
118        std::vector<double> tvals(N + 1), Svals(N + 1);
119        for (int i = 0; i <= N; ++i)
120            tvals[i] = pde_.T() * i / double(N);
121        for (int j = 0; j <= N; ++j)
122            Svals[j] = pde_.Smin() + (pde_.Smax() - pde_.Smin()) * j /
123                double(N);
124
125        Grid M;
126        M.val.assign(N + 1, std::vector<double>(N + 1));
127        M.del.assign(N + 1, std::vector<double>(N + 1));

```

```

127     for (int i = 0; i <= N; ++i) {
128         for (int j = 0; j <= N; ++j) {
129             M.val[i][j] = v(tvals[i], Svals[j]);
130             M.del[i][j] = delta(tvals[i], Svals[j]);
131         }
132     }
133     return M;
134 }
135
136 } // namespace pde
137
138 #endif // FDMETHOD_H

```

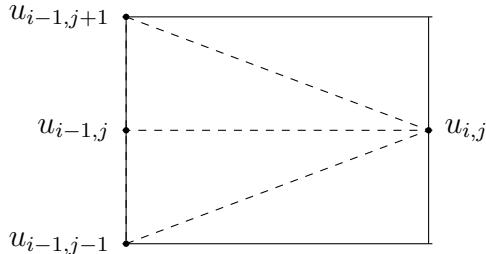
En pratique, les différences finies offrent deux approches : l'explicite, très simple mais peu stable, et l'implicite, que nous retenons pour sa meilleure stabilité au prix d'une mise en œuvre un peu plus lourde. On procède d'abord par un développement de Taylor au point (t_{i-1}, x_j) pour approximer les dérivées

$$\frac{\partial u}{\partial t}(t_{i-1}, x_j) \approx \frac{u_{i,j} - u_{i-1,j}}{\Delta t}, \quad \frac{\partial u}{\partial x}(t_{i-1}, x_j) \approx \frac{u_{i-1,j+1} - u_{i-1,j-1}}{2\Delta x} \quad (\text{B.16})$$

$$\frac{\partial^2 u}{\partial x^2}(t_{i-1}, x_j) \approx \frac{u_{i-1,j+1} - 2u_{i-1,j} + u_{i-1,j-1}}{\Delta x^2} \quad (\text{B.17})$$

En les substituant dans l'EDP $u_t = a u_{xx} + b u_x + c u + d$, on obtient l'équation aux différences dite «implicite» :

$$\frac{u_{i,j} - u_{i-1,j}}{\Delta t} = a_{i-1,j} \frac{u_{i-1,j+1} - 2u_{i-1,j} + u_{i-1,j-1}}{\Delta x^2} + b_{i-1,j} \frac{u_{i-1,j+1} - u_{i-1,j-1}}{2\Delta x} + c_{i-1,j} u_{i-1,j} + d_{i-1,j} \quad (\text{B.18})$$



Pour améliorer la précision temporelle, on remplace ensuite l'approximation précédente par une discréttisation centrée au milieu du pas de temps, c'est-à-dire au point $\left(t_i - \frac{\Delta t}{2}, x_j\right)$: on garde

$$\frac{\partial u}{\partial t}\left(t_i - \frac{\Delta t}{2}, x_j\right) \approx \frac{u_{i,j} - u_{i-1,j}}{\Delta t} \quad (\text{B.19})$$

et l'on moyenne les dérivées spatiales sur les deux instants t_{i-1} et t_i ,

$$\begin{aligned} \frac{\partial u}{\partial x}\left(t_i - \frac{\Delta t}{2}, x_j\right) &\approx \frac{1}{2}\left(\frac{u_{i,j+1} - u_{i,j-1}}{2\Delta x} + \frac{u_{i-1,j+1} - u_{i-1,j-1}}{2\Delta x}\right) \\ \frac{\partial^2 u}{\partial x^2}\left(t_i - \frac{\Delta t}{2}, x_j\right) &\approx \frac{1}{2}\left(\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta x^2} + \frac{u_{i-1,j+1} - 2u_{i-1,j} + u_{i-1,j-1}}{\Delta x^2}\right) \\ u\left(t_i - \frac{\Delta t}{2}, x_j\right) &\approx \frac{u_{i-1,j} + u_{i,j}}{2} \end{aligned} \quad (\text{B.20})$$

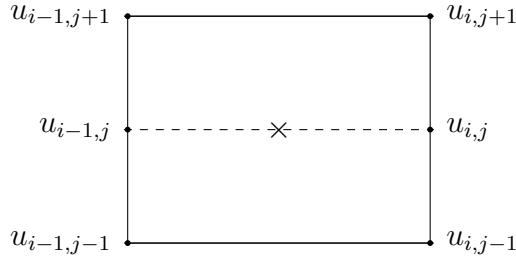
En notant $a_{i-\frac{1}{2},j} = a(t_i - \frac{\Delta t}{2}, x_j)$ et de même pour b, c, d , la substitution dans l'EDP conduit, après simple réorganisation, à

$$E_{i,j} u_{i-1,j-1} + F_{i,j} u_{i-1,j} + G_{i,j} u_{i-1,j+1} = A_{i,j} u_{i,j-1} + B_{i,j} u_{i,j} + C_{i,j} u_{i,j+1} + D_{i,j} \quad (\text{B.21})$$

avec

$$\begin{aligned} A_{i,j} &= \frac{\Delta t}{2} \left(\frac{b_{i-\frac{1}{2},j}}{2\Delta x} - \frac{a_{i-\frac{1}{2},j}}{\Delta x^2} \right), \quad B_{i,j} = 1 + \frac{\Delta t}{2} \left(\frac{2a_{i-\frac{1}{2},j}}{\Delta x^2} - c_{i-\frac{1}{2},j} \right), \quad C_{i,j} = \frac{\Delta t}{2} \left(-\frac{b_{i-\frac{1}{2},j}}{2\Delta x} - \frac{a_{i-\frac{1}{2},j}}{\Delta x^2} \right) \\ E_{i,j} &= -A_{i,j}, \quad F_{i,j} = 2 - B_{i,j}, \quad G_{i,j} = -C_{i,j}, \quad D_{i,j} = -\Delta t d_{i-\frac{1}{2},j} \end{aligned} \quad (\text{B.22})$$

Le schéma devient donc Crank-Nicolson puisque que l'on évalue les dérivées au milieu du pas de temps.



En rassemblant les termes de bord et en empilant les inconnues internes en vecteurs, on obtient la relation matricielle

$$B_i \mathbf{u}_{i-1} = A_i \mathbf{u}_i + \mathbf{w}_i, \quad \mathbf{u}_i = \begin{pmatrix} u_{i,1} \\ \vdots \\ u_{i,j_{\max}-1} \end{pmatrix} \quad (\text{B.23})$$

$$A_i = \begin{pmatrix} B_{i,1} & C_{i,1} & 0 & 0 & \cdots & 0 \\ A_{i,2} & B_{i,2} & C_{i,2} & 0 & \cdots & 0 \\ 0 & A_{i,3} & B_{i,3} & C_{i,3} & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \cdots & \ddots & \ddots & C_{i,j_{\max}-2} \\ 0 & 0 & \cdots & 0 & A_{i,j_{\max}-1} & B_{i,j_{\max}-1} \end{pmatrix} \quad (\text{B.24})$$

$$B_i = \begin{pmatrix} F_{i,1} & G_{i,1} & 0 & 0 & \cdots & 0 \\ E_{i,2} & F_{i,2} & G_{i,2} & 0 & \cdots & 0 \\ 0 & E_{i,3} & F_{i,3} & G_{i,3} & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \cdots & \ddots & \ddots & G_{i,j_{\max}-2} \\ 0 & 0 & \cdots & 0 & E_{i,j_{\max}-1} & F_{i,j_{\max}-1} \end{pmatrix} \quad (\text{B.25})$$

$$\mathbf{w}_i = \begin{pmatrix} D_{i,1} + A_{i,1} f_{\ell,i} - E_{i,1} f_{\ell,i-1} \\ D_{i,2} \\ \vdots \\ D_{i,j_{\max}-2} \\ D_{i,j_{\max}-1} + C_{i,j_{\max}-1} f_{u,i} - G_{i,j_{\max}-1} f_{u,i-1} \end{pmatrix} \quad (\text{B.26})$$

Les conditions sont imposées sous la forme $u_{i,0} = f_{\ell,i}$ et $u_{i,j_{\max}} = f_{u,i}$, tandis que la condition terminale donne $u_{i_{\max},j} = f_j$. Le calcul s'effectue par récurrence arrière via

$$\mathbf{u}_{i-1} = B_i^{-1} \left(A_i \mathbf{u}_i + \mathbf{w}_i \right) \quad (\text{B.27})$$

ce qui conduit à résoudre, à chaque i , un système tridiagonal. On applique la décomposition LU (méthode de Thomas) : en posant $B = (E_j, F_j, G_j)_{j=1..d}$ et $\mathbf{p} = B^{-1}\mathbf{q}$, on effectue une élimination avant

$$r_1 = F_1, \quad y_1 = q_1, \quad r_j = F_j - \frac{E_j}{r_{j-1}} G_{j-1}, \quad y_j = q_j - \frac{E_j}{r_{j-1}} y_{j-1} \quad (j = 2, \dots, d) \quad (\text{B.28})$$

puis une substitution arrière

$$p_d = \frac{y_d}{r_d}, \quad p_j = \frac{y_j - G_j p_{j+1}}{r_j} \quad (j = d-1, \dots, 1) \quad (\text{B.29})$$

Tout cela a été implémenté en C++ au moyen de deux classes. La première, `ImplicitScheme`, hérite de `FDMethod` et regroupe l'ensemble des routines de calcul; la plus importante est évidemment `SolvePDE`, appelée chaque fois que l'on souhaite résoudre l'équation pour accéder aux valeurs de la solution. La seconde, `CNMMethod`, hérite de `ImplicitScheme` et se contente de fixer les coefficients de l'équation discrétisée dans le cas où l'on emploie le schéma de Crank-Nicolson. Avec cette organisation du code, il devient aisément d'ajouter d'autres schémas implicites concrets au-delà de Crank-Nicolson.

Listing B.6: `ImplicitScheme.h`

```

1 #ifndef IMPLICITSCHHEME_H
2 #define IMPLICITSCHHEME_H
3
4 #include "FDMethod.h"
5
6 namespace pde {
7
8 /**
9 * @brief Schéma implicite aux différences finies.
10 * @tparam TPDE Type d'EDP.
11 */
12 template<typename TPDE>
13 class ImplicitScheme : public FDMethod<TPDE> {
14 public:
15     ImplicitScheme(const TPDE& pde, int imax, int jmax);
16
17 /**
18 * @brief Coefficients de l'équation discrétisée.
19 */
20     virtual double A(int i, int j) const = 0;
21     virtual double B(int i, int j) const = 0;
22     virtual double C(int i, int j) const = 0;
23     virtual double D(int i, int j) const = 0;
24     virtual double E(int i, int j) const = 0;
25     virtual double F(int i, int j) const = 0;
26     virtual double G(int i, int j) const = 0;
27 }
```

```

28 /**
29  * @brief Résolution d'un système tridiagonal par l'algorithme
30  * de Thomas.
31  * @return Vecteur solution du système.
32  */
33 std::vector<double> LUDecomposition(int i, std::vector<double> q
34 ) const;
35 /**
36  * @brief Calcul de la matrice de solution V.
37  */
38 void SolvePDE();
39
40 std::vector<double> w(int i) const;
41 std::vector<double> A(int i, std::vector<double> q) const;
42 };
43
44 template<typename TPDE>
45 ImplicitScheme<TPDE>::ImplicitScheme(const TPDE& pde, int imax, int
46 jmax)
47 : FDMethod<TPDE>(pde, imax, jmax)
48 {
49 }
50
51 template<typename TPDE>
52 std::vector<double> ImplicitScheme<TPDE>::w(int i) const {
53     std::vector<double> w(this->jmax_ + 1);
54     w[1] = D(i, 1) + A(i, 1) * this->f1(i) - E(i, 1) * this->f1(i -
55         1);
56     for (int j = 2; j < this->jmax_ - 1; j++)
57         w[j] = D(i, j);
58     w[this->jmax_ - 1] = D(i, this->jmax_ - 1) + C(i, this->jmax_ -
59         1) * this->fu(i) - G(i, this->jmax_ - 1) * this->fu(i - 1);
60     return w;
61 }
62
63 template<typename TPDE>
64 std::vector<double> ImplicitScheme<TPDE>::A(int i, std::vector<
65 double> q) const {
66     std::vector<double> p(this->jmax_ + 1);
67     p[1] = B(i, 1) * q[1] + C(i, 1) * q[2];
68     for (int j = 2; j < this->jmax_ - 1; j++)
69         p[j] = A(i, j) * q[j - 1] + B(i, j) * q[j] + C(i, j) * q[j +
70             1];
71     p[this->jmax_ - 1] = A(i, this->jmax_ - 1) * q[this->jmax_ - 2]
72         + B(i, this->jmax_ - 1) * q[this->jmax_ - 1];
73     return p;
74 }
75
76 template<typename TPDE>
77 std::vector<double> ImplicitScheme<TPDE>::LUDecomposition(int i, std
78 ::vector<double> q) const {

```

```

71     std::vector<double> p(this->jmax_ + 1), r(this->jmax_ + 1), y(
72         this->jmax_ + 1);
73     r[1] = F(i, 1);
74     y[1] = q[1];
75     for (int j = 2; j < this->jmax_; j++) {
76         r[j] = F(i, j) - E(i, j) * G(i, j - 1) / r[j - 1];
77         y[j] = q[j] - E(i, j) * y[j - 1] / r[j - 1];
78     }
79     p[this->jmax_ - 1] = y[this->jmax_ - 1] / r[this->jmax_ - 1];
80
81     for (int j = this->jmax_ - 2; j > 0; j--)
82         p[j] = (y[j] - G(i, j) * p[j + 1]) / r[j];
83     return p;
84 }
85
86 template<typename TPDE>
87 void ImplicitScheme<TPDE>::SolvePDE()
88 {
89     for (int j = 0; j <= this->jmax_; j++)
90         this->V[this->imax_][j] = this->f(j);
91
92     for (int i = this->imax_; i > 0; i--)
93     {
94         auto pvec = this->A(i, this->V[i]);
95         auto wvec = this->w(i);
96         for (int j = 0; j <= this->jmax_; j++)
97             pvec[j] += wvec[j];
98         this->V[i - 1] = LUDecomposition(i, pvec);
99         this->V[i - 1][0] = this->fl(i - 1);
100        this->V[i - 1][this->jmax_] = this->fu(i - 1);
101    }
102 }
103 } // namespace pde
104
105 #endif // IMPLICITScheme_H

```

Listing B.7: CNMethod.h

```

1 #include "pch.h"
2
3 #ifndef CNMETHOD_H
4 #define CNMETHOD_H
5
6 #include "ImplicitScheme.h"
7
8 namespace pde {
9
10 /**
11  * @brief Méthode de Crank-Nicolson pour le schéma implicite aux
12  * différences finies.
13  * @tparam TPDE Type d'EDP.
14  */

```

```

14 template<typename TPDE>
15     class CNMethod : public ImplicitScheme<TPDE> {
16     public:
17         CNMethod(const TPDE& pde, int imax, int jmax)
18             : ImplicitScheme<TPDE>(pde, imax, jmax)
19         {}
20
21
22         double A(int i, int j) const override {
23             return 0.5 * this->dt_ * (this->b(i - 0.5, j) / 2.0 - this->
24                 a(i - 0.5, j) / this->dS_) / this->dS_;
25         }
26
27         double B(int i, int j) const override {
28             return 1.0 + 0.5 * this->dt_ * (2.0 * this->a(i - 0.5, j) / 
29                 (this->dS_ * this->dS_) - this->c(i - 0.5, j));
30         }
31
32         double C(int i, int j) const override {
33             return -0.5 * this->dt_ * (this->b(i - 0.5, j) / 2.0 + this-
34                 ->a(i - 0.5, j) / this->dS_) / this->dS_;
35         }
36
37         double D(int i, int j) const override {
38             return -this->dt_ * this->d(i - 0.5, j);
39         }
40
41         double E(int i, int j) const override {
42             return -A(i, j);
43         }
44
45         double F(int i, int j) const override {
46             return 2.0 - B(i, j);
47         }
48
49         double G(int i, int j) const override {
50             return -C(i, j);
51         }
52     };
53 } // namespace pde
54
55 #endif // CNMETHOD_H

```

Comme d'habitude, nous ne présenterons pas l'exportation DLL complète de toutes les méthodes; nous montrons néanmoins comment cela fonctionne pour une seule fonction (celle qui renvoie le prix, aux coordonnées t et S , d'une call vanille) car la situation est ici un peu plus délicate que d'ordinaire.

Listing B.8: `Exports.cpp`

```

1 using DiffusionCallBS = pde::Diffusion<opt::PayoffCall, pde::BSVol>;
2

```

```

3  SAFE_DOUBLE(PriceEuCallBS ,
4      (double t, double S, double sigma, double T, double R, double K,
5       double Smin, double Smax, int imax, int jmax),
6      {
7          DiffusionCallBS eq(T, Smin, Smax, R, opt::PayoffCall(K), pde::
8              BSVol(sigma));
9          pde::CNMethod<DiffusionCallBS> solver(eq, imax, jmax);
10         solver.SolvePDE();
11         return solver.v(t, S);
12     }
13 )

```

En premier lieu, on doit fournir de nombreux arguments en entrée : les coordonnées (t, S) auxquelles on souhaite évaluer le prix, les paramètres de marché, du sous-jacent et de l'option (qui déterminent les coefficients de l'EDP), ainsi que la précision désirée du schéma aux différences finies en fixant i_{max} et j_{max} . En second lieu, trois templates imbriqués doivent être instanciés. On crée d'abord un objet représentant l'équation de diffusion en spécifiant le type de volatilité et le payoff (cet objet est le premier argument passé au constructeur de `CNMethod`). On crée ensuite un objet `CNMethod` paramétré par le type d'EDP. Enfin, grâce à l'héritage, on appelle `SolvePDE` (défini dans `ImplicitScheme`) puis on renvoie $v(t, S)$ (méthode de `FDMETHOD`).

Exemple (Call vanille dans un modèle Black-Scholes)

Passons maintenant à un exemple sur Excel. L'interface créée est présentée ci-dessous.

Type	Vanilla Call	
Paramètres		
Volatilité	σ	0.2
Taux continu	R	0.05
Maturité en années	T	1
Strike	K	100
Differences finies		
Sous-jacent min	S_{min}	0
Sous-jacent max	S_{max}	200
Nombre de pas en temps	N_t	100
Nombre de pas en prix	N_s	200
Coordonnées		
Temps	t	0
Sous-jacent	s	100
Prix	V(t,S)	10.4471
Delta	$\Delta(t,S)$	0.6366
Bond	B(t,S)	-53.2089

Figure B.1: Interface Excel du modèle Black-Scholes

Les tableaux aux en-têtes bleus sont ceux d'entrée, tandis que ceux aux en-têtes rouges sont de sortie. Dans la cellule *Type* se trouve une liste déroulante contenant les options vanille ainsi que les options européennes présentées dans l'Annexe A. On observe que les valeurs du prix et du delta sont très proches de celles obtenues à partir des formules exactes (3.4) et (3.8) (à savoir 10.4506 et 0.6368, respectivement). Cela signifie que notre approximation par différences finies est déjà très bonne avec $i_{\text{max}} = 100$ et $j_{\text{max}} = 200$. Lorsque l'on modifie un paramètre ou que l'on choisit un autre type d'option, la feuille met automatiquement à jour les sorties (cellules rouges) ainsi que les graphiques des surfaces de prix et de delta. Nous présentons ci-dessous les graphiques correspondant aux paramètres choisis dans la Figure B.1.

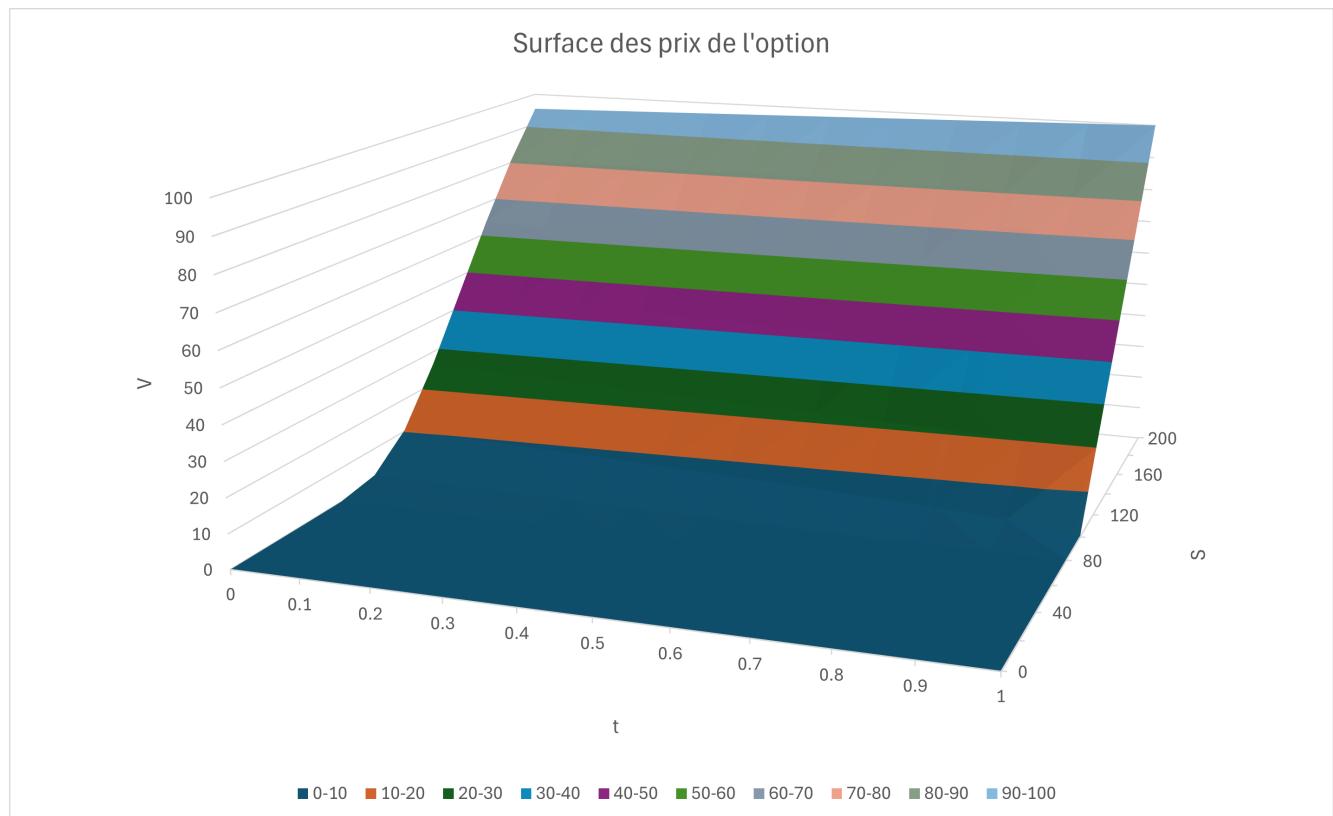


Figure B.2: Surface des prix de la call vanille

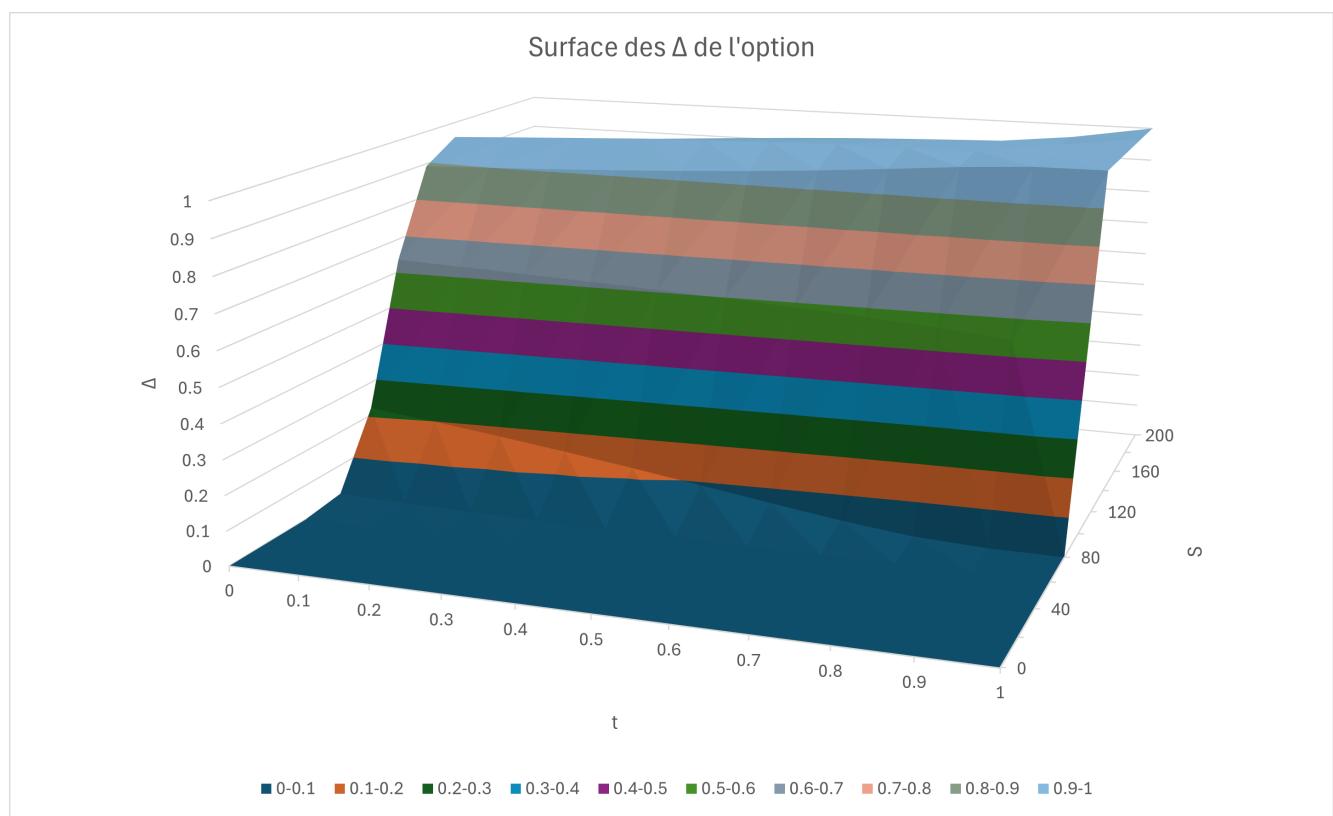


Figure B.3: Surface des deltas de la call vanille

B.2 Modèle à volatilité locale

Dans la réalité, l'hypothèse de volatilité constante est fausse : on observe que la volatilité varie à la fois avec le temps et avec le niveau du sous-jacent. On modélise alors le sous-jacent, sous la mesure risque-neutre, par

$$S_t = S_0 \exp \left(\int_0^t \left(r - \frac{1}{2} \sigma^2(u, S_u) \right) du + \int_0^t \sigma(u, B_u) dB_u \right) \quad (\text{B.30})$$

où B est un mouvement brownien et σ une fonction de $[0, +\infty)^2$ dans $(0, +\infty)$. L'EDP de valorisation associée devient

$$\frac{\partial V}{\partial t}(t, S) + \frac{1}{2} \sigma^2(t, S) S^2 \frac{\partial^2 V}{\partial S^2}(t, S) + rS \frac{\partial V}{\partial S}(t, S) - rV(t, S) = 0 \quad (\text{B.31})$$

Pour déterminer la forme de $\sigma(t, S)$ en pratique, on part des prix de marché (ou des volatilités implicites déduites des formules inverses de Black-Scholes) et l'on utilise la formule de Dupire, qui relie la volatilité locale à la surface des prix des calls $C(T, K)$. En l'absence de dividendes, elle s'écrit :

$$\sigma_{\text{loc}}^2(T, K) = \frac{\partial_T C(T, K) + rK \partial_K C(T, K)}{\frac{1}{2} K^2 \partial_{KK} C(T, K)} \quad (\text{B.32})$$

Dans ce projet, faute de données réelles, nous avons implémenté dans `Volatility.h` un exemple non rigoureux de volatilité locale, uniquement à titre illustratif, qui respecte l'observation (souvent) décroissante en t et en S :

$$\sigma(t, S) = \frac{\alpha}{t+1} + \frac{\beta}{S+1} \quad (\text{B.33})$$

où α et β sont deux coefficients choisis par l'utilisateur.

Exemple (Strangle dans un modèle à volatilité locale)

L'interface Excel pour le modèle à volatilité locale est pratiquement identique à celle de Black-Scholes; la seule différence est qu'il faut choisir les paramètres α et β pour le calcul de la volatilité locale, laquelle est affichée dans une cellule de sortie dédiée.

Type	Strangle	
Paramètres		
Coefficient temps	α	0.2
Coefficient prix	β	10
Taux continu	R	0.05
Maturité en années	T	1
Strike inf	K ₁	100
Strike sup	K ₂	130
Differences finies		
Sous-jacent min	S _{min}	0
Sous-jacent max	S _{max}	200
Nombre de pas en temps	N _t	100
Nombre de pas en prix	N _s	200
Coordonnées		
Temps	t	0
Sous-jacent	S	100
Volatilité		
Prix	V(t,S)	9.4446
Delta	$\Delta(t,S)$	-0.1949
Bond	B(t,S)	28.9325

Figure B.4: Interface Excel du modèle à volatilité locale

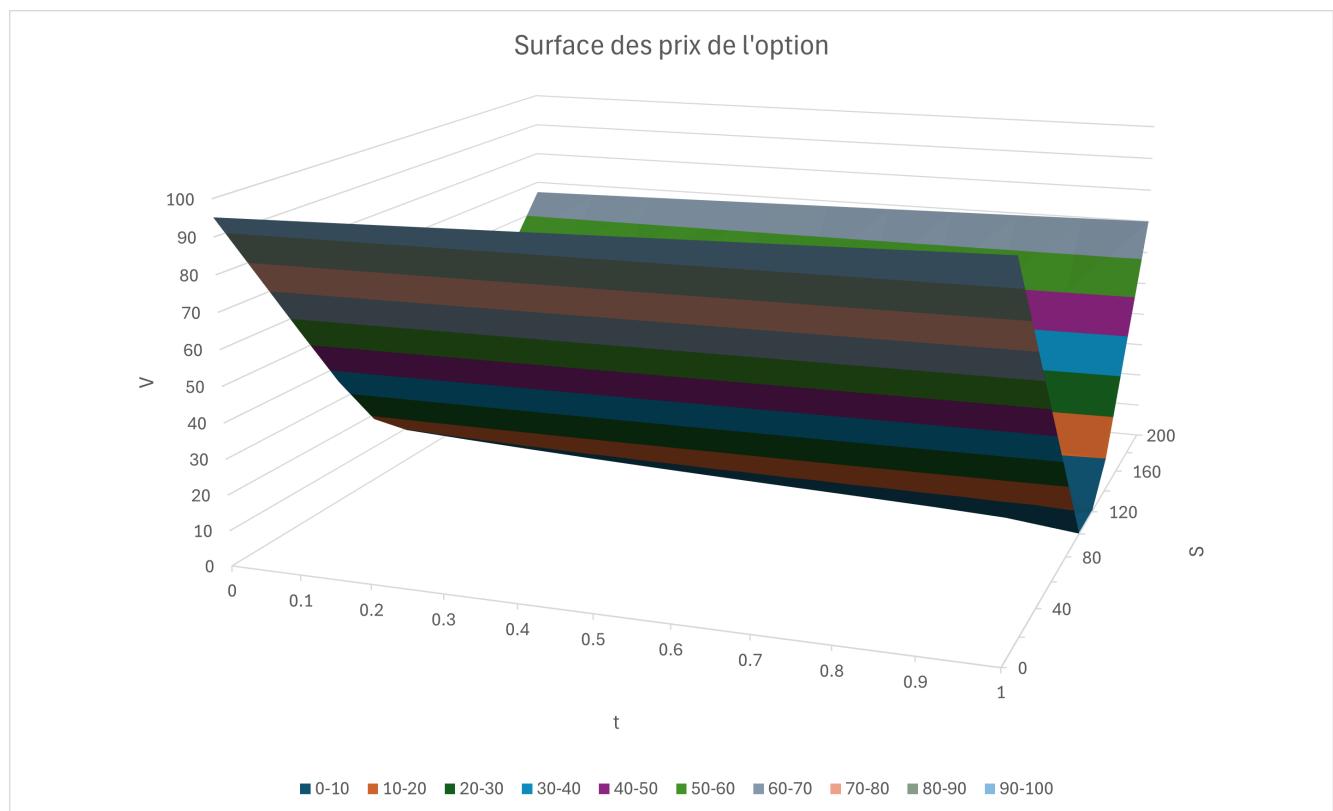


Figure B.5: Surface des prix de la strangle

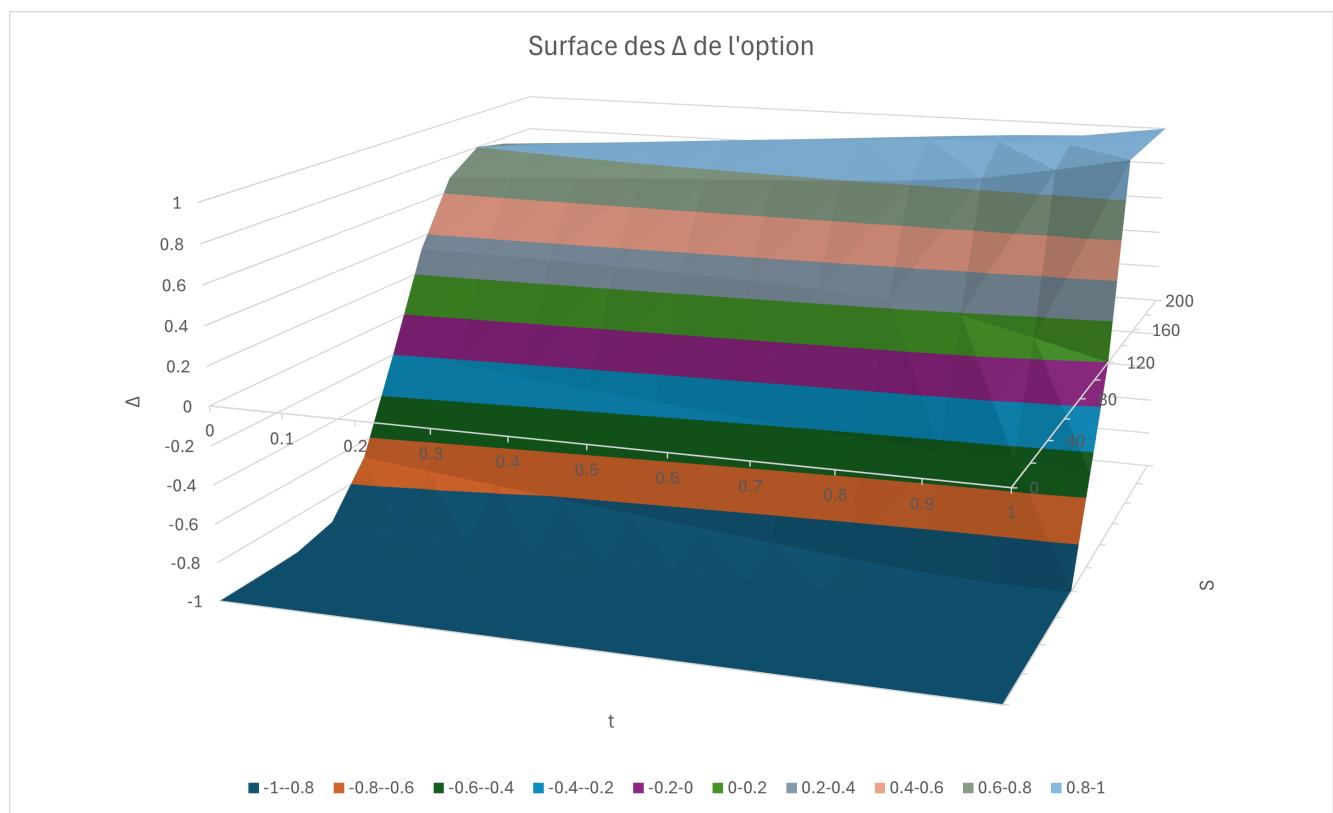


Figure B.6: Surface des deltas de la strangle

Bibliographie

C++ Reference. <https://cppreference.com/>.

CTAN: The Comprehensive TeX Archive Network. <https://ctan.org/>.

Doxxygen User Manual. <https://www.doxygen.nl/manual/>.

Graphviz Documentation. <https://graphviz.org/documentation/>.

Stack Overflow. <https://stackoverflow.com/>.

Bjerksund, P. and G. Stensland (2002). Closed form valuation of american options. NHH Norwegian School of Economics.

Capiński, M. J. and T. Zastawniak (2012). *Numerical Methods in Finance with C++*. Mastering Mathematical Finance. Cambridge: Cambridge University Press.

Chang, C.-C., S.-L. Chung, and R. C. Stapleton (2001). Richardson extrapolation technique for pricing american-style options. National Central University and University of Strathclyde.

Chevalier, É. (2025). Cours de Mathématiques Financières - notes. Université d'Évry.

Ferreira, P. (2025). Cours de VBA - notes, transparents, codes. Université d'Évry.

Gallucci, A. (2012). Modelli discreti per opzioni asiatiche. Master's thesis, Alma Mater Studiorum - Università di Bologna.

Lamberton, D. and B. Lapeyre (1997). *Introduction au calcul stochastique appliqué à la finance*. Ellipses.

Leisen, D. and M. Reimer (1995). Binomial models for option valuation: Examining and improving convergence. *Applied Mathematical Finance*.

Oliveira, C. (2023). *Options and Derivatives Programming in C++23* (3 ed.). Apress.

Ren, Z. (2025). Cours de Processus Stochastiques - notes. Université d'Évry.

Smolski, A. (2022). Crank-Nicolson Python Notebook. https://github.com/antek0308/Volatility_notebooks/blob/main/Medium/Crank_Nicholson.ipynb.

Tahar, I. B., J. Trashorras, and G. Turinici (2016). Éléments de calcul stochastique pour l'évaluation et la couverture des actifs dérivés.

Torri, V. (2025). Cours de C++ - notes, codes. Université d'Évry.

Walsh, J. B. (2003). The rate of convergence of the binomial tree scheme. *Finance and Stochastics*.

Wang, J.-Y. (2025). *Financial Computation or Financial Engineering (graduate level)*. Taipei, Taiwan: National Taiwan University.