Estimation de la volatilité σ à l'aide d'un problème inverse de l'équation de Black-Scholes.

Ader Thibaud, Cordier Swann, Wahoub Ismail.

1 Présentation du problème

Dans le contexte de la finance de marché, l'un des objectifs fondamentaux est d'évaluer correctement le prix d'instruments financiers dérivés, tels que les options. Parmi ces options, l'option d'achat européenne (appelée *call*) permet à son détenteur d'acheter un actif à un prix fixé K à l'avance (le *strike*) à une date donnée T (la *maturité*).

Le prix théorique d'un call dépend notamment de la volatilité σ de l'actif sous-jacent, paramètre difficilement observable directement. Une méthode courante consiste donc à résoudre un *problème inverse* : en observant les prix du call sur le marché, on cherche à en déduire la valeur de σ.

Le modèle classique utilisé est celui de **Black–Scholes**, qui décrit l'évolution du prix d'un call européen comme solution d'une équation aux dérivées partielles (1) linéaire pour le prix de l'option. Le prix du call est noté $u(x,t)$, où t est le temps et x est le prix de l'actif sous-jacent (par exemple une action) .

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 u}{\partial x^2} + rx\frac{\partial u}{\partial x} - ru = 0, \quad t \in [0,T], x > 0. \tag{1}$$

Ce modèle repose sur plusieurs hypothèses économiques :

- les marchés sont parfaits (pas de coûts de transaction, liquidité infinie) ;
- le taux d'intérêt sans risque r est constant ;
- la dynamique du prix de l'actif suit un mouvement brownien géométrique ;
- la volatilité σ est constante dans le temps.

Le problème est posé sur le domaine temporel $[0,T]$ et spatial $x \in [0,L]$, où L est une borne suffisamment grande pour approximer l'infini.

- **Condition terminale (à la maturité)** : à l'échéance $t = T$, la valeur d'un call est donnée par le *payoff* :

$$u(x, T) = \max(x - K, 0),$$

ce qui reflète le gain d'un investisseur à la date de maturité.

- **Condition au bord** $x = 0$: si le prix de l'actif chute à zéro, le call ne vaut rien :

$$u(0, t) = 0 \quad \text{pour tout } t \in [0, T].$$

- **Condition asymptotique (ou bord** $x = L$**)** : quand le prix de l'actif devient très grand, la valeur du call se rapproche du gain immédiat obtenu en exerçant l'option et satisfait la parité call-put :

$$u(x, t) \sim x - Ke^{-r(T-t)} \quad \text{quand } x \to \infty,$$

ce qui peut être approché numériquement par

$$u(L, t) = L - Ke^{-r(T-t)}.$$

2 Problème direct

On considère le problème direct décrit dans la section 1, modélisant le prix $u(t, x)$ d'un call européen dans le modèle de Black–Scholes :

$$\begin{cases} \dfrac{\partial u}{\partial t} + \dfrac{1}{2}\sigma^2 x^2 \dfrac{\partial^2 u}{\partial x^2} + rx\dfrac{\partial u}{\partial x} - ru = 0, & t \in [0, T),\ x \in (0, L), \\ u(x, T) = \max(x - K, 0) = u_T(x), & x \in [0, L], \\ u(0, t) = 0, & t \in [0, T], \\ u(L, t) = L - Ke^{-r(T-t)} = u_L(t), & t \in [0, T]. \end{cases} \tag{2}$$

3 Problème de minimisation

On souhaite retrouver la volatilité σ qui minimise la fonction de coût J:

$$J(\sigma) = \frac{1}{2}||u_\sigma - u_{\text{obs}}||^2_{L^2([0;T]\times[0;L[)} = \frac{1}{2}\int_0^T \int_0^L \left(u_\sigma(x, t) - u_{\text{obs}}(x, t)\right)^2 \, dx \, dt. \tag{3}$$

avec u_{obs} une observation que l'on va générer à partir d'une solution de l'EDP.

On cherche à trouver σ tel que :

$$\frac{dJ}{d\sigma} = 0$$

4 Méthode du Lagrangien

On introduit un multiplicateur de Lagrange $p(x,t)$ et on forme le Lagrangien :

$$\begin{aligned}\mathcal{L}(u,p,\sigma) &= \text{fonction de coût + EDP + conditions initiales/aux bords} \\ &= \frac{1}{2}\int_0^T\int_0^\infty (u-u_{\text{obs}})^2\,dx\,dt \\ &+ \int_0^T\int_0^\infty p\left(\frac{\partial u}{\partial t}+\frac{1}{2}\sigma^2x^2\frac{\partial^2 u}{\partial x^2}+rx\frac{\partial u}{\partial x}-ru\right)dx\,dt \\ &+ \int_0^\infty \gamma(x)(u(x,T)-u_T(x))\,dx + \int_0^T \lambda(t)u(0,t)\,dt \\ &+ \int_0^T \theta(t)(u(L,t)-u_L(t))\,dt\end{aligned} \tag{4}$$

Rappelons ue notre but est minimiser la fonction de coût J définie dans (3). En explicitant la dérivée totale de J par rapport à σ.

$$\frac{dJ}{d\sigma} = \frac{\partial \mathcal{L}}{\partial \sigma} + \left(\frac{\partial \mathcal{L}}{\partial u}\right)\left(\frac{\partial u_\sigma}{\partial \sigma}\right) \tag{J}$$

Cependant, le terme $\frac{\partial u_\sigma}{\partial \sigma}$ est inconnue et ce n'est pas évident d'utiliser cette formule pour optimiser la fonction de coût J.

L'idée est de trouver un u^* (solution de l'équation de Black-Scholes directe) et p^* (solution de l'équation adjointe) qui annulent le terme $\frac{\partial \mathcal{L}}{\partial u}$. Dans ce cas, on a :

$$\frac{\partial J}{\partial \sigma}|_{u^*,p^*} = \frac{\partial \mathcal{L}}{\partial \sigma}|_{u^*,p^*}$$

On n'aura plus qu'à faire une descente de gradient sur J pour trouver le σ^* optimal.

NB: à chaque étape k de la descent, on aura besoin de calculer les solutions $u^{(k)}$ et $p^{(k)}$ pour mettre à jour la volatilité :

$$\sigma^{(k+1)} \longleftarrow \sigma^{(k)} - \eta \cdot \nabla J(\sigma^{(k)}),$$

avec $\nabla J(\sigma^k) = \frac{dJ}{d\sigma}|_{u^{(k)},p^{(k)},\sigma^{(k)}}$ et η le pas de la descente.

5 Formulation du problème inverse

5.1 Calcul de la variation

On effectue la variation par rapport à u dans la direction ϕ : $u \mapsto u + \varepsilon\phi$, pour toute fonction test ϕ.

On calcule :

$$\frac{\partial \mathcal{L}}{\partial u}[\phi] = \frac{d}{d\varepsilon}\mathcal{L}(u + \varepsilon\phi, p, \sigma)\Big|_{\varepsilon=0}.$$

$$\frac{\partial \mathcal{L}}{\partial u}[\phi] = \int_0^T \int_0^\infty (u - u_{\text{obs}})\ \phi \, dx \, dt \tag{5}$$

$$+ \int_0^T \int_0^\infty p\left(\frac{\partial \phi}{\partial t} + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 \phi}{\partial x^2} + rx\frac{\partial \phi}{\partial x} - r\phi\right) dx \, dt \tag{6}$$

$$+ \int_0^\infty \gamma(x)\phi(x)\, dx + \int_0^T \lambda(t)\phi(t)\, dt + \int_0^T \theta(t)\phi(t)\, dt \tag{7}$$

Détaillons le premier terme correspondant à la dérivation de la fonction de coût J:

$$J(u + \varepsilon\phi) = \frac{1}{2}\int_0^T \int_0^L \left[(u + \varepsilon\phi - u_{\text{obs}})^2\right] dx \, dt.$$

On développe le carré :

$$\begin{aligned}(u + \varepsilon\phi - u_{\text{obs}})^2 &= ((u - u_{\text{obs}}) + \varepsilon\phi)^2 \\ &= (u - u_{\text{obs}})^2 + 2\varepsilon(u - u_{\text{obs}})\phi + \varepsilon^2\phi^2.\end{aligned}$$

Donc :

$$J(u + \varepsilon\phi) = \frac{1}{2}\int_0^T \int_0^L \left[(u - u_{\text{obs}})^2 + 2\varepsilon(u - u_{\text{obs}})\phi + \varepsilon^2\phi^2\right] dx \, dt.$$

On dérive par rapport à ε et on évalue en $\varepsilon = 0$:

$$\frac{d}{d\varepsilon}J(u + \varepsilon\phi)\Big|_{\varepsilon=0} = \int_0^T \int_0^L (u - u_{\text{obs}})\, \phi \, dx \, dt.$$

Les autres termes sont immédiats par linéarité de l'EDP et des conditions au bord/initiales en u.

5.2 Formulation de l'équation adjointe

On effectue des intégrations par parties et on prend les fonctions tests ϕ à support compact (s'annulent aux bords).

- En temps :

$$\int_0^T \int_0^\infty p\frac{\partial \phi}{\partial t}\, dx\, dt = -\int_0^T \int_0^\infty \frac{\partial p}{\partial t}\phi\, dx\, dt \quad (\text{car } \phi(0) = \phi(T) = 0).$$

- En espace (1ère dérivée) :

$$\int_0^T \int_0^\infty prx\frac{\partial \phi}{\partial x}\, dx\, dt = -\int_0^T \int_0^\infty \frac{\partial}{\partial x}(prx)\phi\, dx\, dt. \quad (\text{car } \phi(0) = \phi(L) = 0)$$

avec $\frac{\partial}{\partial x}(prx) = rx\frac{\partial p}{\partial x} + rp$.

- En espace (2e dérivée) :

$$\int_0^T \int_0^\infty p \cdot \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 \phi}{\partial x^2} dx\, dt = \int_0^T \int_0^\infty \frac{\partial^2}{\partial x^2}\left(\frac{1}{2}\sigma^2 x^2 p\right)\phi\, dx\, dt.$$

avec $\frac{\partial^2}{\partial x^2}\left(\frac{1}{2}\sigma^2 x^2 p\right) = \frac{1}{2}\sigma^2 \frac{\partial}{\partial x}(2xp + x^2\frac{\partial p}{\partial x}) = \frac{1}{2}\sigma^2(2p + 4x\frac{\partial p}{\partial x} + x^2\frac{\partial^2 p}{\partial x^2})$

On obtient alors :

$$\frac{\partial \mathcal{L}}{\partial u}[\phi] = \int_0^T \int_0^\infty \left[(u - u_{\text{obs}}) - \frac{\partial p}{\partial t} - \frac{\partial (prx)}{\partial x} + \frac{\partial^2}{\partial x^2}\left(\frac{1}{2}\sigma^2 x^2 p\right) - rp\right]\phi\, dx\, dt. \tag{8}$$

Pour que la variation s'annule pour tout ϕ, il faut :

$$-\frac{\partial p}{\partial t} - \frac{\partial (prx)}{\partial x} + \frac{\partial^2}{\partial x^2}\left(\frac{1}{2}\sigma^2 x^2 p\right) - rp = -(u - u_{\text{obs}}). \tag{9}$$

Ce qui donne l'équation adjointe :

$$-\frac{\partial p}{\partial t} + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 p}{\partial x^2} + (2\sigma^2 - r)x\frac{\partial p}{\partial x} + \left(\sigma^2 - 2r\right)p = u_{obs} - u. \tag{10}$$

5.2.1 Conditions aux bords

Pour déterminer les conditions aux bords et initiales de l'équation adjointe, on part de la variation du Lagrangien dans la direction ϕ :

$$\frac{\partial \mathcal{L}}{\partial u}[\phi] = \int_0^T \int_0^L (u - u_{\text{obs}})\, \phi\, dx\, dt \tag{11}$$

$$+ \int_0^T \int_0^L p\left(\frac{\partial \phi}{\partial t} + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 \phi}{\partial x^2} + rx\frac{\partial \phi}{\partial x} - r\phi\right) dx\, dt \tag{12}$$

$$+ \int_0^L \gamma(x)\phi(x)\, dx + \int_0^T \lambda(t)\phi(t,0)\, dt + \int_0^T \theta(t)\phi(t,L)\, dt \tag{13}$$

1. Terme en $\partial_t \phi$:

$$\int_0^T \int_0^\infty p \, \frac{\partial \phi}{\partial t} \, dx \, dt = \left[\int_0^\infty p(t,x)\phi(t,x) \, dx \right]_0^T - \int_0^T \int_0^\infty \frac{\partial p}{\partial t} \, \phi \, dx \, dt$$

2. Terme en $\partial_x \phi$:

$$\int_0^T \int_0^\infty p \cdot rx \frac{\partial \phi}{\partial x} \, dx \, dt = \left[\int_0^T p(t,x) \cdot rx\phi(t,x) \, dt \right]_0^L - \int_0^T \int_0^\infty \frac{\partial}{\partial x}(rxp) \, \phi \, dx \, dt$$

3. Terme en $\partial_x^2 \phi$:

$$\begin{aligned}
\int_0^T \int_0^\infty p \cdot \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 \phi}{\partial x^2} \, dx \, dt &= \left[\int_0^T \frac{1}{2}\sigma^2 x^2 p \frac{\partial \phi}{\partial x} \, dt \right]_0^L - \int_0^T \int_0^\infty \frac{\partial}{\partial x} \left(\frac{1}{2}\sigma^2 x^2 p \right) \frac{\partial \phi}{\partial x} \, dx \, dt \\
&= \text{termes de bord} + \int_0^T \int_0^\infty \frac{\partial^2}{\partial x^2} \left(\frac{1}{2}\sigma^2 x^2 p \right) \phi \, dx \, dt \\
&= \text{bords} - \int_0^T \left[\frac{1}{2}\sigma^2 x^2 p \frac{\partial \phi}{\partial x} \right]_0^\infty dt + \int_0^T \int_0^\infty \frac{\partial^2}{\partial x^2} \left(\frac{1}{2}\sigma^2 x^2 p \right) \phi \, dx \, dt
\end{aligned}$$

En rassemblant toutes les intégrations par parties :

$$\frac{\partial \mathcal{L}}{\partial u}[\phi] = \int_0^T \int_0^L \left[(u - u_{\text{obs}}) - \frac{\partial p}{\partial t} - \frac{\partial}{\partial x} \left(\frac{1}{2}\sigma^2 x^2 \frac{\partial p}{\partial x} \right) + \frac{\partial}{\partial x}(rxp) - rp \right] \phi \, dx \, dt \tag{14}$$

$$+ \underbrace{\int_0^L p(x,T)\phi(x,T) \, dx - \int_0^L p(x,0)\phi(x,0) \, dx}_{\text{conditions initiales en } t} \tag{15}$$

$$+ \underbrace{\int_0^T \left[\frac{1}{2}\sigma^2 x^2 p \, \frac{\partial \phi}{\partial x} + rxp \, \phi \right]_{x=0}^{x=L} dt}_{\text{conditions aux bords en } x} . \tag{16}$$

Pour que la dérivée variationnelle s'annulle pour tout ϕ et comme ϕ est arbitraire, cela impose l'équation adjointe :

$$\boxed{\begin{cases} -\partial_t p - \frac{1}{2}\sigma^2 x^2 \partial_{xx}^2 p - rx\partial_x p + rp = u_{\text{obs}} - u, \\ p(x,0) = 0, \\ p(0,t) = 0, \quad p(L,t) = 0. \end{cases}}$$

Système adjoint complet :

$$\begin{cases} -\dfrac{\partial p}{\partial t} + \dfrac{1}{2}\sigma^2 x^2 \dfrac{\partial^2 p}{\partial x^2} + (2\sigma^2 - r)x\dfrac{\partial p}{\partial x} + \left(\sigma^2 - 2r\right) p = u_{obs} - u. & (x,t) \in (0,L)\times(0,T), \\ p(x,0) = 0, & x \in (0,L), \\ p(0,t) = 0, & t \in (0,T), \\ p(x,t) \to 0 \quad (x \to \infty), \quad \text{ou } p(L,t) = 0 & t \in (0,T). \end{cases}$$

6 Résolution numérique de l'équation de Black–Scholes

Nous allons dans cette partie utiliser deux méthodes classiques pour résoudre l'équation directe de Black–Scholes. Enfin nous en choisirons une des deux pour résoudre l'équation adjointe qui est très similaire à l'équation directe.

Commençons d'abord par expliciter le gradient de J dans les conditions qu'on a déjà citées dans les conditions citées dans la section 4.

6.1 Gradient de la fonction de coût J

On ne fait varier que le paramètre $\sigma \mapsto \sigma + \varepsilon$, le couple (u, p) satisfaisant déjà les équations direct et adjointe, de sorte que

$$\frac{\mathrm{d}J}{\mathrm{d}\sigma} = \frac{\partial \mathcal{L}}{\partial \sigma} = -\,\sigma \int_0^T \int_0^L x^2 \, \frac{\partial^2 u}{\partial x^2}(x,t)\, p(x,t)\; dx\, dt.$$

Ainsi, après avoir résolu une fois le problème direct, puis une fois l'adjoint, le gradient se calcule par l'intégrale ci-dessus.

6.2 Méthode des différences finies avec le schéma de Crank-Nicholson

Premièrement, nous avons utilisé le schéma de Crank-Nicholson pour résoudre l'équation directe avec les différences finies.

Partant de l'équation de Black-Scholes formulée comme suit :

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 u}{\partial x^2} + rx\frac{\partial u}{\partial x} - ru = 0,$$

nous appliquons une discrétisation temporelle et spatiale sur une grille régulière : $t^n = n\Delta t$, $x_i = i\Delta x$, et notons $u_i^n \approx u(x_i, t^n)$. Le schéma de Crank-Nicholson est une moyenne du schéma explicite et implicite qui se formule comme suit :

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + \frac{1}{2}\left(\mathcal{L}_i^{n+1} + \mathcal{L}_i^n\right) = 0,$$

où l'opérateur spatial $\mathcal{L}_i^n$ est :

$$\mathcal{L}_i^n = \frac{1}{2}\sigma^2 x_i^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} + r x_i \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} - r u_i^n.$$

On regroupe alors les termes selon les indices $i-1$, i et $i+1$, ce qui permet de réécrire cette équation comme une combinaison linéaire des termes u_{i-1}^{n+1}, u_i^{n+1}, u_{i+1}^{n+1}, et u_{i-1}^n, u_i^n, u_{i+1}^n.

En notant U^n le vecteur colonne des valeurs u_i^n pour $i = 1, \ldots, N$, on peut exprimer le système à chaque pas de temps sous forme matricielle :

$$AU^{n+1} = BU^n,$$

où :

- A est une matrice tridiagonale construite à partir des coefficients issus de l'opérateur $\mathcal{L}^{n+1}$;
- B est une matrice tridiagonale semblable, dérivée de $\mathcal{L}^n$.

$$A = I + \frac{\Delta t}{2} L, \qquad B = I - \frac{\Delta t}{2} L,$$

où L est la matrice représentant l'opérateur spatial $\mathcal{L}$, la même que pour les schémas explicite et implicite vu en TP1. On obtient alors à chaque pas de temps une équation linéaire que l'on résout avec python.

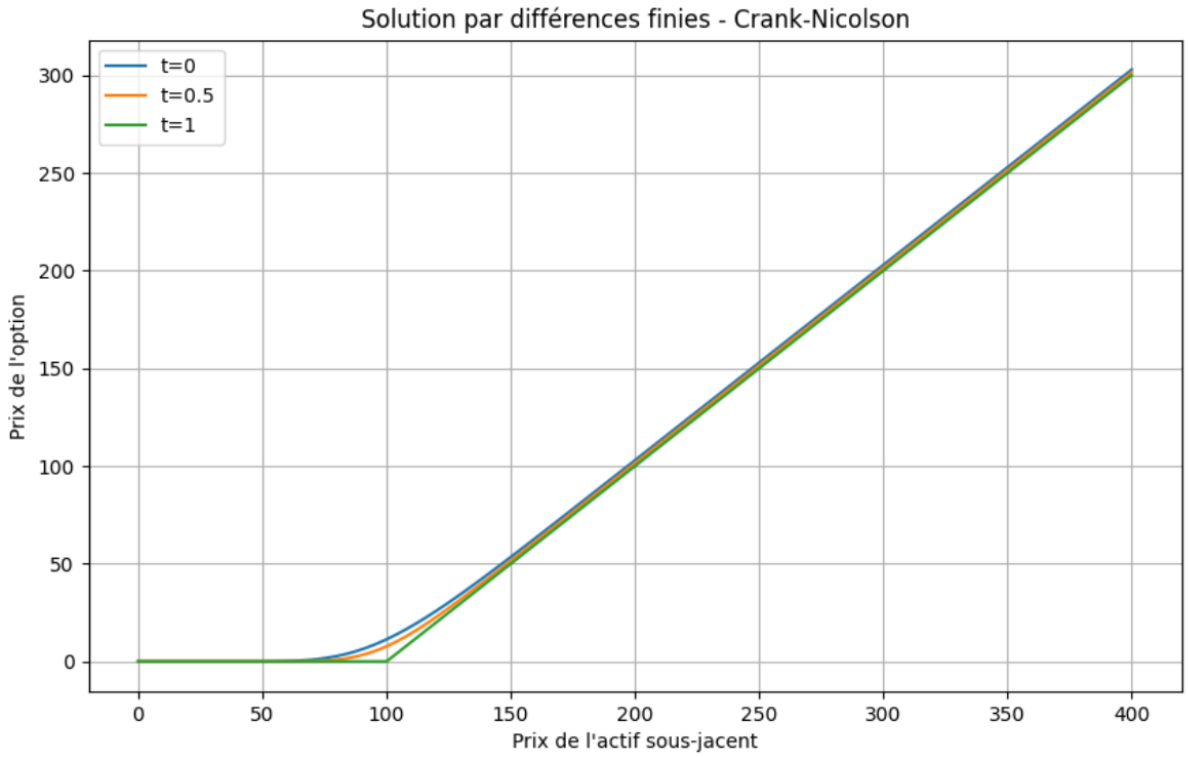

Figure 1: Solution u obtenu à l'aide du schéma Crank-Nicholson à 3 instants différents

6.2.1 Motivations du choix de ce schéma

Le schéma implicite pur est un choix naturel pour résoudre l'équation de Black-Scholes car il est inconditionnellement stable, ce qui permet d'utiliser des pas de temps relativement grands sans compromettre la convergence. Toutefois, il présente un inconvénient : il est seulement *d'ordre 1 en temps*, ce qui signifie que l'erreur commise à chaque pas de temps est en $\mathcal{O}(\Delta t)$. Cela peut devenir limitant si l'on souhaite obtenir une solution numérique plus précise.

À l'inverse, le schéma de Crank-Nicholson conserve la stabilité inconditionnelle du schéma implicite, tout en améliorant significativement la précision : il est *d'ordre 2 en temps*. Ce gain de précision vient du fait qu'il repose sur la moyenne arithmétique entre le schéma explicite et le schéma implicite, c'est-à-dire :

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + \frac{1}{2}\left(\mathcal{L}(u_i^n) + \mathcal{L}(u_i^{n+1})\right) = 0.$$

Ainsi, Crank-Nicholson combine :

- la stabilité du schéma implicite ;
- la précision temporelle grâce à la moyenne entre deux instants.

Dans le contexte de la résolution de l'équation de Black-Scholes pour une option européenne, où la fonction de valeur est régulière et le domaine borné, le schéma de Crank-Nicholson est donc particulièrement adapté. Il permet une meilleure précision sans nécessiter un pas de temps trop fin, ce qui rend les calculs plus efficaces tout en assurant une bonne qualité de l'approximation. De plus l'étude se fait à une dimension dans l'espace et dans un espace borné de géométrie "simple", la méthode des différences est en général efficace dans ce cas.

6.3 Analyse de l'erreur Crank-Nicholson

Nous disposons pour l'équation directe de Black and Scholes d'une solution exacte. Nous avons donc fait varier le pas d'espace pour observer l'évolution de l'erreur car c'est en espace que varie le plus l'erreur. Nous nous sommes intéressés à l'erreur L^2 et L^∞.

Voici ce que nous obtenons :

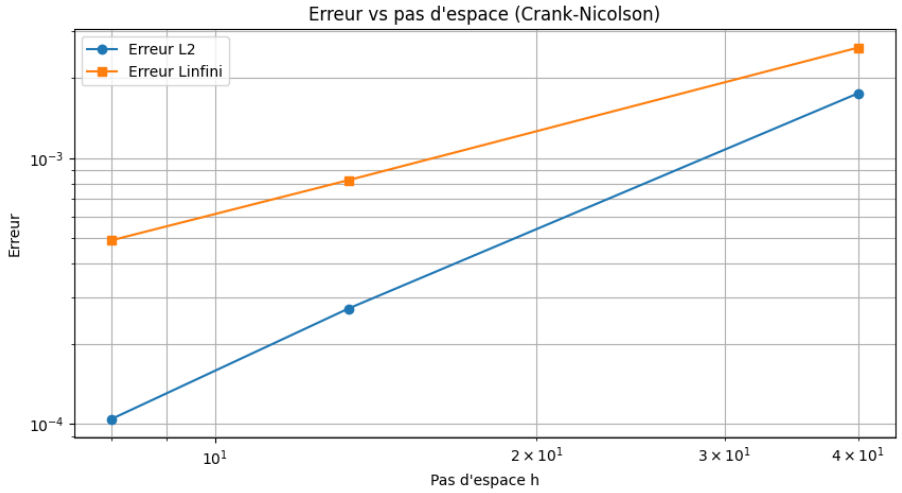

Figure 2: Courbe erreurs L^2 et L^∞

Ceci correspond à une erreur L^2 d'ordre 1.74 et 1.04 pour L^∞. Nous obtenons des erreurs un peu plus faibles que l'ordre 2.

6.4 Méthode des éléments finis (FEM)

L'autre méthode qui est moins utilisée est la méthode des éléments finis repose sur une formulation variationnelle de l'équation de Black-Scholes. On introduit une base de fonctions test $(\phi_i)_{i=0}^N$ et on multiplie l'équation par une fonction test $v = \phi_j$, puis on intègre sur le domaine spatial :

$$\int_0^L \frac{\partial u}{\partial t}\phi_j \, dx + \int_0^L \frac{1}{2}\sigma^2 x^2 \frac{\partial u}{\partial x}\frac{\partial \phi_j}{\partial x} \, dx + \int_0^L rx\frac{\partial u}{\partial x}\phi_j \, dx - \int_0^L ru\phi_j \, dx = 0.$$

En discrétisant u comme $u(x,t) \approx \sum_i u_i(t)\phi_i(x)$, on obtient un système d'équations différentielles en temps :

$$M\frac{du}{dt} + \left(\frac{1}{2}\sigma^2 S + rC - rM\right) u = 0,$$

où :

- M est la **matrice de masse**, avec $M_{ij} = \int_0^L \phi_i \phi_j \, dx$,
- S est la **matrice de rigidité pondérée**, avec $S_{ij} = \int_0^L x^2 \phi_i' \phi_j' \, dx$,
- C est la **matrice de convection**, avec $C_{ij} = \int_0^L x \phi_i' \phi_j \, dx$.

Dans l'implémentation, nous utilisons une base de fonctions linéaires par morceaux (éléments finis P^1). La matrice de masse M est tridiagonale, avec intégration analytique des produits $\phi_i \phi_j$. Elle s'écrit :

$$M = \frac{\Delta x}{6} \begin{bmatrix} 4 & 1 & & \\ 1 & 4 & 1 & \\ & \ddots & \ddots & \ddots \\ & & 1 & 4 \end{bmatrix}.$$

La matrice de rigidité S est construite en évaluant x^2 au centre de chaque élément. En notant $x_{i+1/2} = (x_i + x_{i+1})/2$, on approxime l'intégrale :

$$S_{ij} \approx \sum_e \left(x_{i+1/2}^2 \int_e \phi_i' \phi_j' \, dx \right),$$

ce qui donne une tridiagonale avec des poids dépendant de x^2.

La matrice de convection C repose également sur une évaluation en chaque élément :

$$C_{ij} \approx \sum_e \left(x_{i+1/2} \int_e \phi_i' \phi_j \, dx \right),$$

ce qui donne une matrice antisymétrique tridiagonale.

Le système linéaire à chaque pas de temps est donné par :

$$A = M + \Delta t \left(\frac{1}{2}\sigma^2 S + rC - rM\right), \quad B = M,$$

et la solution est propagée de manière rétrograde :

$$Au^n = Bu^{n+1}.$$

Les conditions de Dirichlet sont imposées fortement en modifiant les lignes et colonnes de A et B. La condition terminale $u(x,T) = \max(x-K, 0)$ est appliquée à $t = T$.

Cette méthode est plus flexible que les différences finies et permet un traitement plus précis des opérateurs différentiels, en particulier si le maillage est adapté. Elle est toutefois plus complexe à implémenter et nécessite l'assemblage de matrices dépendantes du problème.

6.5 Erreur éléments finis vs Crank-Nicholson

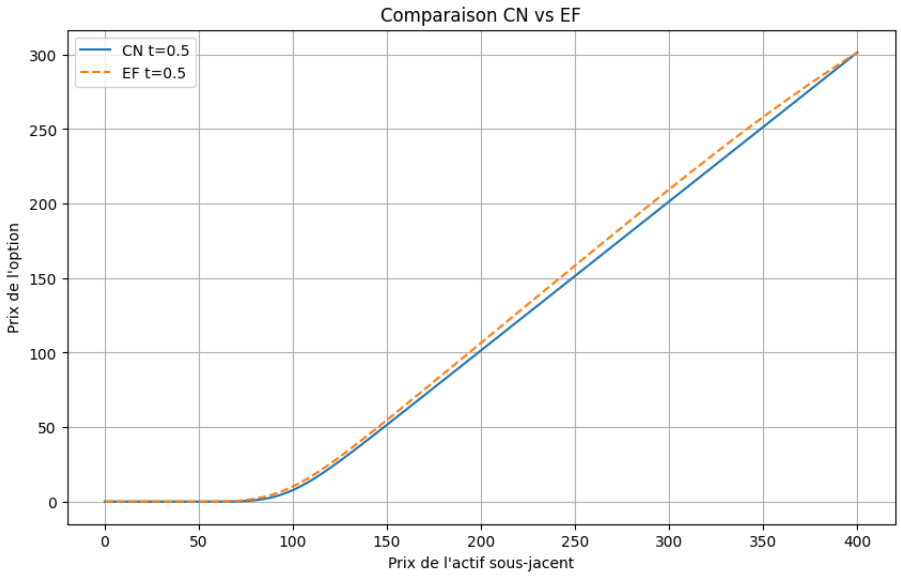

Figure 3: Solution EF vs Crank-Nicholson à t =0.5

Nous remarquons que la courbe obtenue avec les éléments finis s'éloignent de façon assez marquée de la solution obtenue avec le schéma de Crank-Nicholson. Ce dernier est beacoup plus précis ce qui montre que les éléments finis tels que nous les avons implémentés sont beaucoup moins précis.

6.6 Erreurs CN, EF par rapport à la solution exacte

Rappelons que la solution analytique de (2) est définie par :

$$u_{call}(x,t) = x\mathcal{N}(d_1) - Se^{-r(T-t)}\mathcal{N}(d_2) \tag{C}$$

où

$$\mathcal{N}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-u^2/2} \tag{17}$$

et les deux quantités

$$d_1 := \frac{\log(\frac{x}{S}) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$

et

$$d_2 := d_1 - \sigma\sqrt{T-t}\ .$$

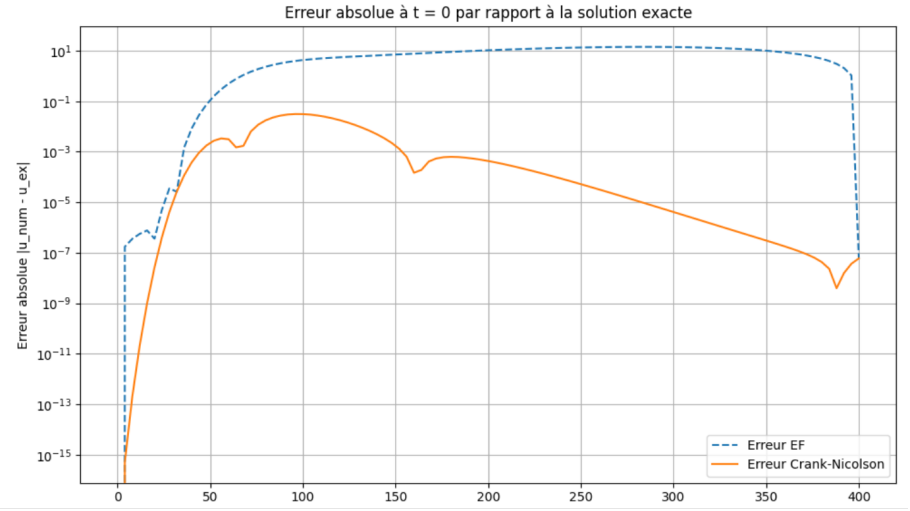

Figure 4: Étude d'erreurs des solutions CN, EF à $t = 0$ par rapport à la solution analytique

On voit que la méthode des différences finies donnent d'excellent résultats contrairement à la méthode des éléments finis qui n'est clairement pas adapté à ce type de problème $1D$.

- La méthode des DF donne à priori une bonne solution surtout pour les petits et grands x avec une erreur allant jusqu'à 10^{-7} environ.
- La méthode des EF est clairement inadaptée, pour $x > 50$ la solution s'éloigne énormément de la solution exacte.

On peut aussi voir ce phénomène en superposant les 3 solutions sur un même graphe :

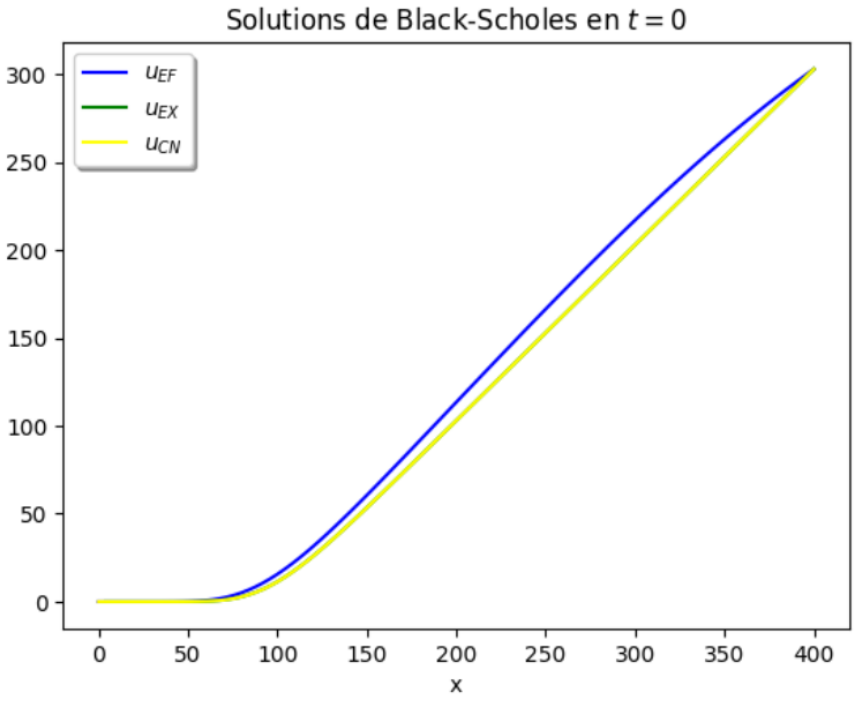

Figure 5: Les 3 solutions : analytique, CN et EF

La solution avec la méthode de Crank-Nicholson est tellement précise qu'elle embrasse la solution exacte. La solution u_{EF} avec la méthode des éléments finis est beaucoup moins précise et à partir d'un certain $x \approx 50$, elle s'éloigne énormément de la solution exacte.

6.7 Crank-Nicholson pour l'équation adjointe

A présent, comme nous avons vu que c'était la methode la plus efficace pour l'équation directe de Black and Scholes nous allons utiliser la méthode de Crank-Nicholson pour résoudre l'équation adjointe qui a une forme très similaire :

$$-\frac{\partial p}{\partial t} + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 p}{\partial x^2} + (2\sigma^2 - r)x\frac{\partial p}{\partial x} + \left(\sigma^2 - 2r\right)p = u_{obs} - u. \tag{18}$$

Les seules différences sont la formule de L, le terme source $u_{obs} - u$ et les conditions initiales et aux bords.

Comme pour l'équation directe, nous appliquons une discrétisation temporelle et spatiale sur une grille régulière : $t^n = n\Delta t$, $x_i = i\Delta x$, et notons $p_i^n \approx p(x_i, t^n)$.

Nous avons alors la formulation matricielle suivante :

$$AP^{n+1} = BP^n + \frac{1}{2}(b^n + b^{n+1}),$$

où P^n est le vecteur colonne des valeurs p_i^n pour $i = 1, \ldots, N$, $b^n = U_{obs} - U^n$ et :

$$A = I - \frac{\Delta t}{2}L, \qquad B = I + \frac{\Delta t}{2}L,$$

avec $L = \frac{\sigma^2}{2} \cdot \frac{\text{XXD}}{\Delta x^2} + (2\sigma^2 - r) \cdot \frac{\text{XD}}{\Delta x} + (\sigma^2 - 2r) \cdot I$

7 Algorithmes d'optimisation

Comme cela a été évoqué dans la section 4, nous chercherons à optimiser la fonction de coût J. Nous utiliserons deux algorithmes à savoir la *descente de gradient* et l'algorithme de *Newton-Raphson*.

7.1 Génération d'une observation

Afin d'effectuer nos algorithmes d'optimisation, il nous faut d'abord une observation u_{obs}, un choix naturel a été de résoudre l'équation (2) à l'aide du schéma de Crank-Nicholson, puis bruiter cette solution pour éviter un potentiel biais dans l'optimisation.

$$u_{obs} = u^* \cdot (1 + a \cdot \mathcal{N}_{t,x}(0,1)).$$

avec $\mathcal{N}_{t,x}(0,1)$ une loi normale standard dans la dimension du temps et l'espace.

7.2 Descente de gradient

1. Résoudre l'EDP directe (2) (temps arrière $T \to 0$) pour $u^{(k)}$.
2. Résoudre l'EDP adjointe (5.2.1) (temps avant $0 \to T$) pour $p^{(k)}$ que l'on approxime numériquement.
3. Gradient : $g^{(k)} = -\sigma^{(k)} \iint x^2 \dfrac{\partial^2 u^{(k)}}{\partial x^2} p^{(k)}$.
4. Mise à jour : $\sigma^{(k+1)} = \sigma^{(k)} - \eta \cdot g^{(k)}, \quad \eta > 0$ (recherche de pas).

On commence par une valeur de volatilité initiale arbitraire, puis on applique une descente de gradient pour trouver la vraie valeur de volatilité qu'on a définie au tout début de la résolution.

```
sigma = 0.25 # volatilité à trouver
```

On initialise $\sigma_0 = 0.85$ dans un premier temps et on a la courbe d'erreurs suivante:

```
sigma_opt, J_vals, sigma_vals = gradient_descent(sigma_init=0.85,
↪  u_obs=u_obs) #
```

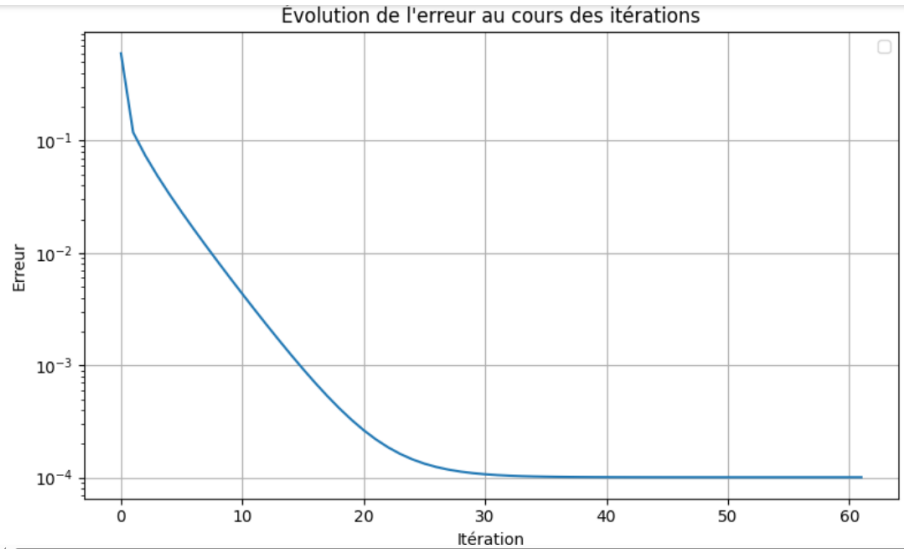

Figure 6: Évolution de l'erreur en fonction des itérations pour $\sigma_0 = 0.85$

On obtient une valeur de $\sigma = 0.25010$ au bout de 62 itérations en fixant le niveau de tolérance pour le gradient à 10^{-5} et un pas $\eta = 10^{-5}$.

Pour une initialisation $\sigma_0 = 0.05$, on obtient la courbe suivante :

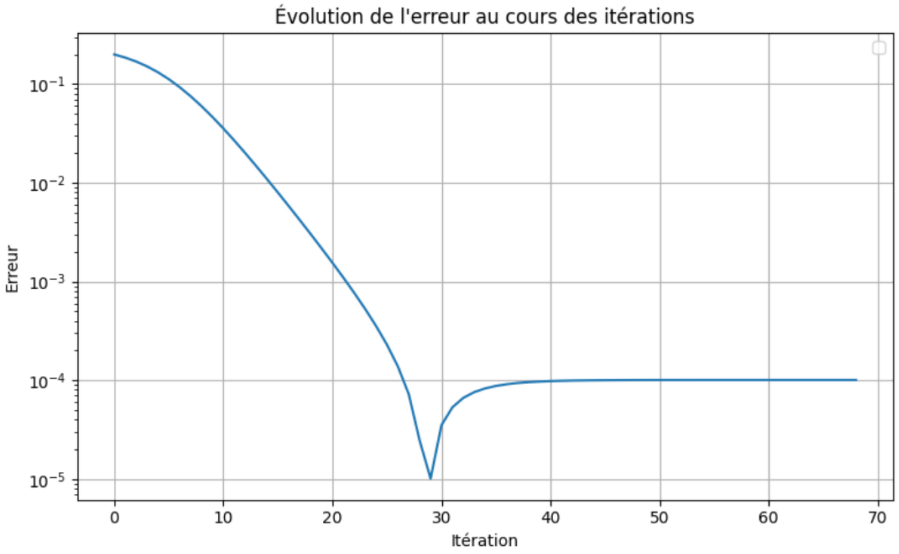

Figure 7: Évolution de l'erreur en fonction des itérations pour $\sigma_0 = 0.05$

On obtient encore une fois 0.25010 comme volatilité au bout de 69 itérations. La forme de la courbe peut être généralisée pour des initialisations $\sigma_0 < 0.25$.

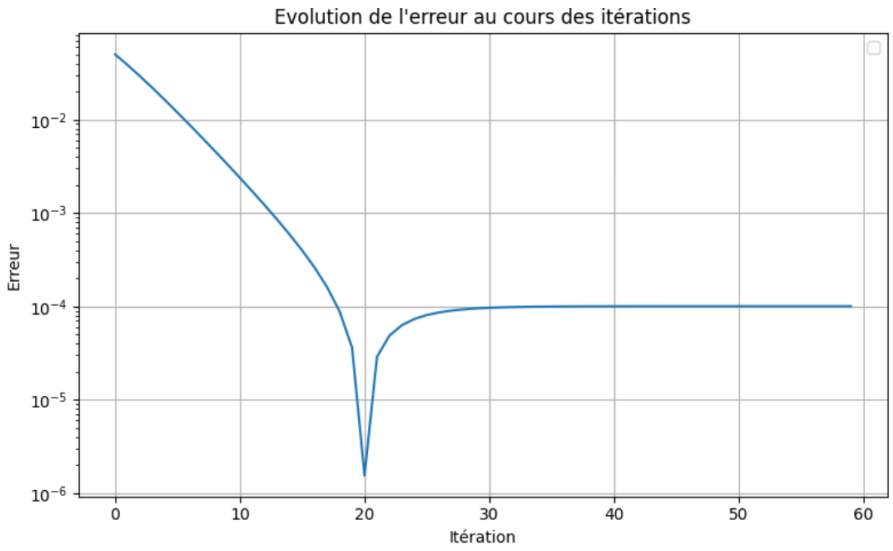

Figure 8: Évolution de l'erreur en fonction des itérations pour $\sigma_0 = 0.20$

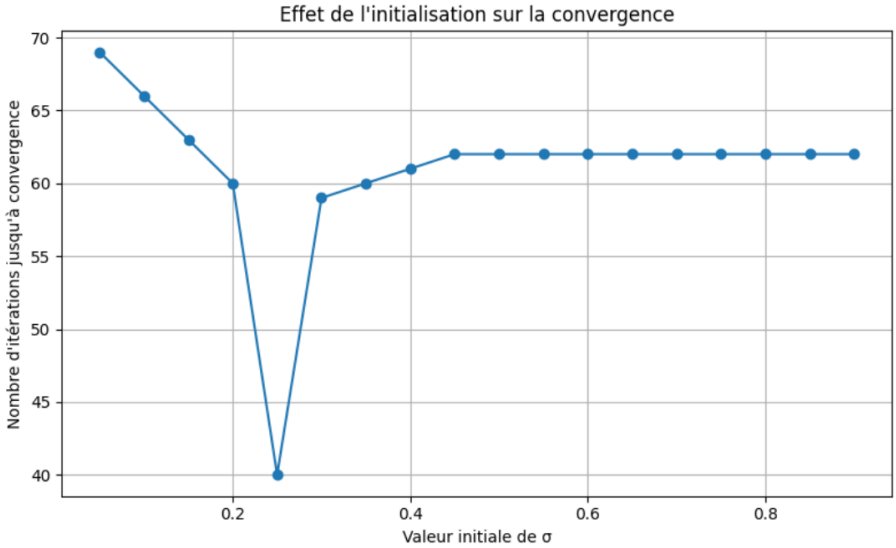

Figure 9: Effet de l'initialisation sur la convergence de la descente de gradient avec σ_0 allant de 0.05 à 0.95

Globalement, il nous faut environ 62 itérations dès que $\sigma_0 > 0.25$ et relativement plus (environ 65) quand $\sigma_0 < 0.25$. En moyenne on a besoin de 61 itérations pour trouver le σ optimal. On pourrait diminuer le nombre d'itérations en baissant la tolérance par rapport à la valeur du gradient puisqu'à l'exécution, on voit que la valeur de σ^* se stabilise vers la 35^e itération à environ 0.25010

7.3 Algorithme de Newton-Raphson

Il était également intéressant de tester d'autres algorithmes d'optimisation, cette fois avec l'algorithme de Newton-Raphson. Cependant, nous avons eu à évaluer le gradient ET la hessienne de J, ce qui peut être plus coûteux d'un point de vue numérique :

$$J''(\sigma) \approx \frac{J(\sigma + h) - 2J(\sigma) + J(\sigma - h)}{h^2}.$$

Puis appliquer le même algorithme (7.2) en changeant la mise à jour selon le schéma de Newton-Raphson :

$$\sigma_{k+1} \longleftarrow \sigma_k - J'(\sigma_k)/J''(\sigma_k)$$

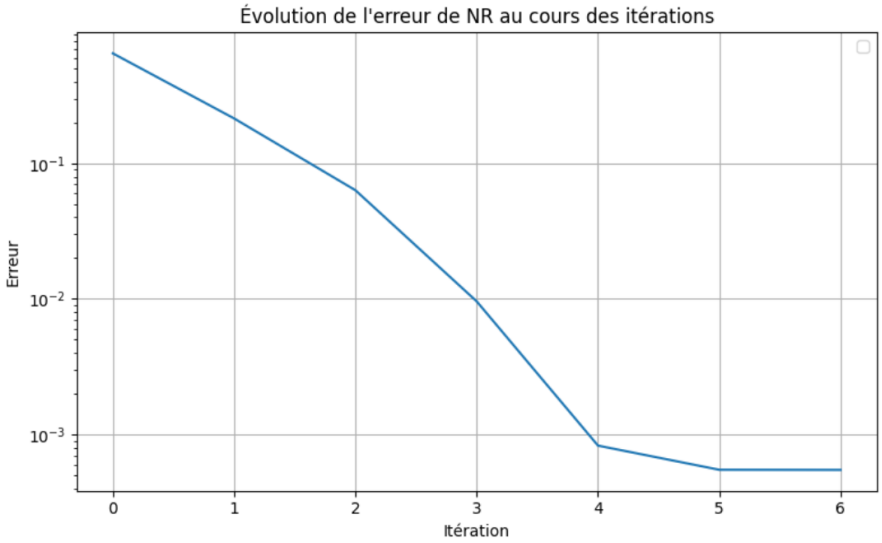

Figure 10: Erreur de la méthode Newton-Raphson avec une initialisation $\sigma_0 = 0.9$

Pour une initialisation de $\sigma_0 = 0.9$, on obtient $\sigma^* = 0.250547$ après uniquement 6 itérations. Il s'avère que l'algorithme de Newton-Raphson converge plus rapidement en général mais la précision n'est pas plus significatif par rapport à la méthode de la descente de gradient qui donne des résultats légèrement meilleurs sur cet aspect.

De plus, la méthode de *Newton-Raphson* est sensible à l'initialisation. En effet, pour $\sigma_0 < 0.1$, l'algorithme ne tourne pas car le pas est jugé trop petit et on ne réussit pas à trouver σ^*. On pourrait ajuster le pas h dans ce cas.

```
0  σ=0.080000   J=2.0766e+03   grad=-1.8303e+03
 pas trop petit, arrêt.
```

Figure 11: Initialisation pour $\sigma_0 = 0.08$

Annexes

Code Pour résoudre l'équation directe à l'aide d'un schéma par différences finis implicite :

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse as sp
import scipy.sparse.linalg as spla


sigma = 0.25 # volatilité
r     = 0.03 # taux sans risque (de 3%)
K     = 100  # le prix d'exercice (en EUR)
T     = 1    # l'échéance (1 an)

L = 400   # remplaçant oo

Nx = 100   # nombre d'intervalles en espace
N = 100   # nombre d'intervalles en temps

dx = L / Nx   # pas en espace
dt = T / N   # pas en temps

# condition terminale
def get_terminal_condition(x):
    return np.maximum(x-K,0)

# condition au bord de dirichlet en x = 0
def get_Dirichlet_boundary_condition_0(t,x):
    return np.zeros_like(x)

# condition au bord en x -> infini (ou L)
def get_Dirichlet_boundary_condition_oo(t,x):
    return x - K*np.exp(-r*(T - t))


x = np.linspace(0, L, Nx+1)    # points en espace
t = np.linspace(0, T, N+1)    # points en temps

# On réécrie ensuite sous forme matricielle pour trouver la solution
↪  rétrograde en partant de la condition terminale :

# Matrices pour différences finies
A = 2*np.eye(Nx+1) - np.eye(Nx+1, k=1) - np.eye(Nx+1, k=-1)
D = (np.eye(Nx+1, k=1) - np.eye(Nx+1, k=-1)) / 2
I = np.eye(Nx+1)
```

```
X = np.diag(x)

XD  = X @ D
XXA = X @ (X @ A)

#schéma de crank nicholson
def resol_crank_nicolson(sigma):
    u_cn = np.zeros((N+1, Nx+1))

    # Opérateur L de Black-Scholes
    L = -sigma**2/2 * XXA / dx**2 + r * XD / dx - r * I

    # Matrices du schéma de Crank-Nicolson
    B_cn = I - 0.5 * dt * L
    C_cn = I + 0.5 * dt * L

    # Conditions de Dirichlet sur la matrice B
    B_cn[0,:] = np.zeros(Nx+1)
    B_cn[0,0] = 1
    B_cn[-1,:] = np.zeros(Nx+1)
    B_cn[-1,-1] = 1

    # Condition terminale
    u_cn[N,:] = get_terminal_condition(x)

    # Résolution rétrograde
    for n in range(N-1, -1, -1):
        b = C_cn @ u_cn[n+1,:]
        # Conditions de bord
        b[0] = get_Dirichlet_boundary_condition_0(t[n], x[0])
        b[-1] = get_Dirichlet_boundary_condition_oo(t[n], x[-1])
        u_cn[n,:] = np.linalg.solve(B_cn, b)

    return u_cn

# Résolution
u_impl = resol_impl(sigma)
```

Code pour résoudre l'équation directe à l'aide d'un schéma par éléments finis:

```
main  = np.ones(Nx+1)
sub   = np.ones(Nx)

# masse M
M = (dx/6)*sp.diags(diagonals=[2*main, sub, sub],offsets=[0, -1, 1],
↪  format="csr")
```

```
# raideur pondérée S
x_mid = (x[:-1] + x[1:]) / 2          # centre de l'élément
wS    = x_mid**2 / dx                 # poids locale
S = sp.diags([ np.r_[wS, 0] + np.r_[0, wS] ,
               -wS , -wS],
             [0, -1, 1], shape=(Nx+1, Nx+1), format="csr")


# convection C
wC = x_mid / 2
lower = -wC
upper =  wC
C = sp.diags([ np.r_[lower, 0], np.zeros(Nx+1), np.r_[0, upper]],
             [-1, 0, 1], shape=(Nx+1, Nx+1), format="csr")


# matrices pas de temps
A = M + dt*( 0.5*sigma**2*S + r*C - r*M )
B = M.copy()                              # membre droit


# impose Dirichlet : lignes/colonnes → identité
for idx, val in [(0, 0.0), (Nx, L-K*np.exp(-r*T))]:
    A[idx, :] = 0.0;  A[idx, idx] = 1.0
    B[idx, :] = 0.0;  B[idx, idx] = 1.0


A_fact = spla.factorized(A)             # LU sparse


# condition terminale : pay-off
u = get_terminal_condition(x)                # u^{N}

u_all = np.zeros((N+1, Nx+1))
u_all[N] = get_terminal_condition(x)       # payoff
u = u_all[N].copy()

# marche arrière en temps
for n in range(N, 0, -1):               # N → 1
    t_n1 = (n-1)*dt
    # condition bord amont x=L
    u[-1] = L - K*np.exp(-r*(T - t_n1))
    u[0]  = 0.0
    u = A_fact(B @ u)
    u_all[n-1] = u
```

```
# u contient maintenant la solution au temps t = 0
print(f"Valeur du call pour S0={K} : {u[x==K][0]:.4f} €")

# Affichage
plt.figure(figsize=(10,6))
plt.plot(x, u_all[0], label='t=0')
plt.plot(x, u_all[N//2], label=f't={T/2}')
plt.plot(x, u_all[-1], label=f't={T}')
plt.title("Solution par éléments finis")
plt.xlabel("Prix de l'actif sous-jacent")
plt.ylabel("Prix de l'option")
plt.legend()
plt.grid()
plt.show()
main  = np.ones(Nx+1)
sub   = np.ones(Nx)

# masse M
M = (dx/6)*sp.diags(diagonals=[2*main, sub, sub],offsets=[0, -1, 1],
↪  format="csr")


# raideur pondérée S
x_mid = (x[:-1] + x[1:]) / 2          # centre de l'élément
wS    = x_mid**2 / dx                 # poids locale
S = sp.diags([ np.r_[wS, 0] + np.r_[0, wS] ,
               -wS , -wS],
             [0, -1, 1], shape=(Nx+1, Nx+1), format="csr")


# convection C
wC = x_mid / 2
lower = -wC
upper =  wC
C = sp.diags([ np.r_[lower, 0], np.zeros(Nx+1), np.r_[0, upper]],
             [-1, 0, 1], shape=(Nx+1, Nx+1), format="csr")


# matrices pas de temps
A = M + dt*( 0.5*sigma**2*S + r*C - r*M )
B = M.copy()                                # membre droit


# impose Dirichlet : lignes/colonnes → identité
for idx, val in [(0, 0.0), (Nx, L-K*np.exp(-r*T))]:
    A[idx, :] = 0.0;  A[idx, idx] = 1.0
    B[idx, :] = 0.0;  B[idx, idx] = 1.0
```

```
A_fact = spla.factorized(A)              # LU sparse


# condition terminale : pay-off
u = get_terminal_condition(x)                # u^{N}

u_all = np.zeros((N+1, Nx+1))
u_all[N] = get_terminal_condition(x)        # payoff
u = u_all[N].copy()

# marche arrière en temps
for n in range(N, 0, -1):                # N → 1
    t_n1 = (n-1)*dt
    # condition bord amont x=L
    u[-1] = L - K*np.exp(-r*(T - t_n1))
    u[0]  = 0.0
    u = A_fact(B @ u)
    u_all[n-1] = u


# u contient maintenant la solution au temps t = 0
print(f"Valeur du call pour S0={K} : {u[x==K][0]:.4f} €")
```

Code pour la résolution de l'équation adjointe par un schéma de différences finis implicite :

```
def resol_adjoint_crank_nicolson(sigma, u, u_obs):
    p = np.zeros((N+1, Nx+1))  # Initialisation
    F = u_obs - u              # Terme de source

    # Opérateur L de l'équation adjointe
    L_adj = -(sigma**2 / 2) * XXA / dx**2 + (2 * sigma**2 - r) * XD / dx
    ↪  + (sigma**2 - 2 * r) * I

    # Matrices du schéma de Crank-Nicolson
    B_adj = I - 0.5 * dt * L_adj
    C_adj = I + 0.5 * dt * L_adj

    # Conditions de Dirichlet sur B_adj
    B_adj[0, :] = 0.0
    B_adj[0, 0] = 1.0
    B_adj[-1, :] = 0.0
    B_adj[-1, -1] = 1.0

    # (3) Marche en avant : p^{n+1}
```

```
    for n in range(0, N):  # 0 ... N-1
        # Second membre avec Crank-Nicolson : moyenne temporelle du
        ↪  terme source
        b = C_adj @ p[n, :] + 0.5 * dt * (F[n+1, :] + F[n, :])

        # Conditions de Dirichlet sur le second membre
        b[0] = 0.0
        b[-1] = 0.0

        # Résolution du système
        p[n+1, :] = np.linalg.solve(B_adj, b)

    return p
```

Code pour la génération de données u_{obs} en ajoutant un bruit blanc gaussien:

```
    def make_obs(sigma_ref, noise=0.02, seed=None):

    rng = np.random.default_rng(seed)
    u_obs = resol_impl(sigma_ref)

    # bruit multiplicatif intérieur
    u_obs[:, 1:-1] *= 1.0 + noise * rng.standard_normal(u_obs[:,
    ↪  1:-1].shape)

    # contraintes de bord exactes
    u_obs[:, 0]  = 0.0
    u_obs[:, -1] = get_Dirichlet_boundary_condition_oo(t, x[-1])

    # pas de valeurs négatives
    u_obs = np.maximum(u_obs, 0.0)
    return u_obs
```

Code pour la détermination du σ optimal par descente de gradient :

```
    def derivee_seconde(u, dx):  # approximation de la dérivée seconde
    ↪  de u
    # u : tableau (Nt+1, Nx+1)
    return (u[:, 2:] - 2*u[:, 1:-1] + u[:, :-2]) / dx**2

def compute_gradient(sigma, u, p):

    u_xx = derivee_seconde(u, dx)          # shape (N+1, Nx-1)
    x_aucarre = x[1:-1]**2                        # shape (Nx-1,)
```

```
    grad = -sigma * np.sum(x_aucarre * u_xx * p[:, 1:-1]) * dx * dt  # on
    ↪  exclut les bords
    return grad

# eta : pas de la descente
# sigma_init : l'initialisation
# u_obs : l'observation
# tol :  marge par rapport à la valeur du gradient

def gradient_descent(sigma_init, u_obs, eta=1e-5, max_iter=1000,
↪  tol=1e-5):
    sigma = sigma_init
    cost_list = []
    sigma_list = []

    for k in range(max_iter):
        u = resol_impl(sigma)
        p = resol_adjoint(sigma, u, u_obs)
        grad = compute_gradient(sigma, u, p)

        cost = 0.5 * np.sum((u - u_obs)**2) * dt * dx
        cost_list.append(cost)
        sigma_list.append(sigma)

        print(f"Iter {k+1}: sigma = {sigma:.5f}, J = {cost:.5e}, grad =
        ↪  {grad:.5e}")

        if abs(grad) < tol:
            break

        sigma -= eta * grad

    return sigma, cost_list, sigma_list

# Génération des observations
u_obs = make_obs(sigma, noise=0.02, seed=40)

# Descente de gradient
sigma_opt, J_vals, sigma_vals = gradient_descent(sigma_init=0.4,
↪  u_obs=u_obs)
```

Code pour la détermination du σ optimal par la méthode de Newton-Raphson:

```
def cost(u, u_obs):
  # la fonction de coût qu'on cherche à minimiser
    return 0.5 * np.sum((u - u_obs)**2) * dx * dt
```

```
def derivee_seconde_J(sigma, u_obs, h=1e-4):

    h = max(h, 1e-8)      # évite h trop petit
    J0 = cost(resol_impl(sigma),     u_obs)
    Jp = cost(resol_impl(sigma + h), u_obs)
    Jm = cost(resol_impl(sigma - h), u_obs)
    return (Jp - 2*J0 + Jm) / h**2

def newton_raphson_sigma(sigma_init, u_obs,max_iter = 30, tol_grad =
↪  1e-8, tol_step = 1e-8, h_fd = 1e-4, verbose = True):
    sigma = max(sigma_init, 1e-8)
    sigma_list = []
    for k in range(max_iter):

        # calcul de la fonction de cout + gradient
        u = resol_impl(sigma)
        p = resol_adjoint(sigma, u, u_obs)
        grad = compute_gradient(sigma, u, p)        # calcul de  J'()
        cost_ = cost(u, u_obs)

        if verbose:
            print(f"{k:2d}  ={sigma:.6f}   J={cost_:.4e}
            ↪  grad={grad:.4e}")
            sigma_list.append(sigma)

        # critère d'arrêt sur le gradient
        if abs(grad) < tol_grad:
            break

        # dérivée seconde de J
        h = h_fd * max(sigma, 1.0)        # pas adaptatif
        hessian = derivee_seconde_J(sigma, u_obs, h=h)

        # si Hessian  0, on tombe sur un pas de gradient
        if abs(hessian) < 1e-12:
            step = -grad        # direction de descente simple
        else:
            step = - grad / hessian  # schéma de N-R

        # assurer la baisse de J
        eta = 1.0
        J_ref = cost_
        while True:
            sigma_trial = max(sigma + eta*step, 1e-8)
            J_trial = cost(resol_impl(sigma_trial), u_obs)
            if J_trial < J_ref:          # condition d'Armijo basique
                break
            eta *= 0.5                   # réduit le pas
```

```
            # on s'arrête si le pas devient trop petit
            if eta * abs(step) < tol_step:
                if verbose:
                    print("  pas trop petit, arrêt.")
                return sigma, sigma_list

        sigma = sigma_trial

        # critère d'arrêt sur l'incrément
        if abs(eta*step) < tol_step:
            break

    return sigma, sigma_list

u_obs = make_obs(sigma, noise=0.02, seed=40)

# Inversion par Newton{Raphson
sigma0 = 0.2
sigma_star, sigma_list= newton_raphson_sigma(sigma0,
↪  u_obs,max_iter=20,h_fd=1e-4,verbose=True)
print(f"\n* estimé : {sigma_star:.6f}  (vrai = {sigma})")
```