Université Denis-Diderot

M2MO: Modélisation Aléatoire

Finite Difference Method for HJB Equations

Projet EDP et Méthodes Numériques

Étudiants:
Rathea Uth
Nouredine Larbi

Responsable du cours:
M. Olivier Bokanowski

22 Février 2022

Contents

1 Introduction

The aim is to test different schemes for HJB equations. A first part concerns first order HJB equations, related to deterministic control, where it is more easy to study the numerical schemes, the stability and monotonicity of the schemes. A second part concerns the approximation of a second order HJB equation (uncertain volatility model) including an example (for development) from the article [Bokanowski et al., 2018].

2 An Eikonal equation

We look for a numerical approximation of $v = v(t,x)$ solution of the following equation, for $t \in (0,T), x \in \Omega := (S_{\min}, S_{\max})$:

$$\begin{cases} v_t(t,x) + c\,|v_x(t,x)| = 0, \quad t \in (0,T), x \in (S_{\min}, \\ v\,(t,S_{\min}) = v_\ell(t) \quad t \in (0,T) \\ v\,(t,S_{\max}) = v_r(t) \quad t \in (0,T) \\ v(0,x) = v_0(x) \quad x \in (S_{\min}, S_{\max}) \end{cases} \tag{1}$$

This PDE is called an **"eikonal equation"**. It is a particular case of Hamilton-Jacobi-Bellman (HJB) equations. We will consider the following parameters:

$$c = 1, \quad T = 1$$

and the following initial data

$$v_0(x) = -\left(\max\left(1 - x^2, 0\right)\right)^2 \tag{2}$$

and the following boundary

$$(S_{\min}, S_{\max}) = (-3, 3)$$

with zero Dirichlet boundary conditions: $v_r(t) = v_\ell(t) = 0$. We notice that other boundary conditions could be used for different data.

In particular, we aim at computing $v(t,x)$ at final time $t = T$. We introduce a discrete mesh as usual: $h := \frac{S_{\max} - S_{\min}}{I+1}$, $\quad \Delta t := \frac{T}{N}$ and

$$\begin{aligned} &x_j := S_{\min} + jh, \quad j = 0, \ldots, I+1 \quad \text{(mesh points)} \\ &t_n = n\Delta t, \quad n = 0, \ldots, N \quad \text{(time mesh)} \end{aligned}$$

We are looking for U_j^n, an approximation of $v\,(t_n, x_j)$.

2.1 Euler Explicit Scheme (EE)

We remark that $c\,|v_x| = \max\,(cv_x, -cv_x)$. We assume in this case $c \geq 0$. Hence a first possible **"Euler Forward scheme"** (or Explicit Euler scheme), using the upwind and downwind approximations, for stability reasons, is as follows:

$$\frac{U_j^{n+1}-U_j^n}{\Delta t}+\max\left(c\frac{U_j^n-U_{j-1}^n}{h},-c\frac{U_{j+1}^n-U_j^n}{h}\right)=0 \tag{3a}$$

$$n=0,\dots,N-1; j=1,\dots,I$$

$$U_0^n=v_\ell(t)\equiv 0,\quad U_{I+1}^n=v_r(t)\equiv 0 \qquad n=0,\dots,N \tag{3b}$$

$$U_j^0=v_0(x_j) \qquad j=1,\dots,I \tag{3c}$$

1. Compute the consistency error of the scheme 3

In order to obtain the consistency error of this scheme, we denote $V_j^n=\varphi(t_n,x_j)$ where φ is a sufficiently regular function.

We write then the scheme in abstract form as follows:

$$\begin{aligned}\mathcal{S}(t_{n+1},x_j,V_j^{n+1}) &= \frac{V_j^{n+1}-V_j^n}{\Delta t}+\max(c\frac{V_j^n-V_{j-1}^n}{h},-c\frac{V_{j+1}^n-V_j^n}{h})\\ &= \frac{\varphi(t_n,x_j)+\Delta t\varphi_t(t_n,x_j)+\frac{1}{2}\Delta t^2\varphi_{tt}(t_n,x_j)+O(\Delta t^3)-\varphi(t_n,x_j)}{\Delta t}\\ &+\max\left(c\frac{\varphi(t_n,x_j)-(\varphi(t_n,x_j)-h\varphi_x(t_n,x_j)+\frac{1}{2}h^2\varphi_{xx}(t_n,x_j))+O(h^3)}{h},\right.\\ &\left.-c\frac{\varphi(t_n,x_j)+h\varphi(t_n,x_j)+\frac{1}{2}h^2\varphi_{xx}(t_n,x_j)+O(h^3)-\varphi(t_n,x_j)}{h}\right)\end{aligned}$$

We obtain then:

$$\begin{aligned}\mathcal{S}(t_{n+1},x_j,V_j^{n+1}) &= \varphi_t(t_n,x_j)+\frac{1}{2}\Delta t\varphi_{tt}(t_n,x_j)\\ &+\max\left(c(\varphi_x(t_n,x_j)+\frac{1}{2}h\varphi_{xx}(t_n,x_j),\right.\\ &\left.-c(\varphi_x(t_n,x_j)+\frac{1}{2}h\varphi_{xx}(t_n,x_j))\right)+O(\Delta t^2)+O(h^2)=0\end{aligned}$$

Let $\mathcal{H}$ corresponds to the PDE equation where:

$$\mathcal{H}(\varphi)(t_n,x)=\varphi_t(t_n,x)+c|\varphi_x(t_n,x)|=\varphi_t(t_n,x)+\max(c\varphi_x(t_n,x),-c\varphi_x(t_n,x))$$

So that by definition, the PDE for φ reads $\mathcal{H}(\varphi)(t,x)=0$. Then a consistency error can be defined as ϵ_j^n such that:

$$\mathcal{S}\left(t_{n+1},x_j,V_j^{n+1}\right)=\mathcal{H}(\varphi)\left(t_{n+1},x_j\right)+\epsilon_j^n$$

We obtain finally:

$$\epsilon_j^n=\frac{1}{2}\Delta t\varphi_{tt}(t_n,x_j)+\frac{1}{2}h\varphi_{xx}(t_n,x_j)+O(\Delta t^2)+O(h^2)$$

Therefore, the consistency error satisfies:

$$\left|\epsilon_j^n\right|\le C(\Delta t+h)$$

This means that the consistency error is of order *one* in time and *one* in space.

2. Program the scheme

   In our code, we program the matrix D^- and the vector function $q^-(t)$ such that:

$$\left(D^- U^n + q^-(t)\right)_{1\leq i\leq I} = \left(c\frac{U_i^n - U_{i-1}^n}{h}\right)_{1\leq i\leq I}.$$

   and $D^+, q^+(t)$ in the same way:

$$\left(D^+ U^n + q^+(t)\right)_{1\leq i\leq I} = \left(c\frac{U_{i+1}^n - U_i^n}{h}\right)_{1\leq i\leq I}$$

   One can furthermore use the command `np.maximum(v1,v2)` (resp. `np.minimum(v1,v2)`) to maximize (resp. minimize) component-wise two vectors of type `numpy.array`.

   Remarque: In order to improve the efficiency of the code, it is possible to precompute $D^-$and D^+ as **sparse matrices**.

   The program we implemented (in python) is shown in the section Annex 1.

3. Intuitively guess the exact solution of the problem (the formula may change depending on the initial condition)

   The exact solution of the problem is:

$$v_{exact}(t,x) = \begin{cases} v_0(x+ct) & \text{if} \quad x \leq -ct \\ v_0(0) & \text{if} \quad x \in (-ct, ct) \\ v_0(x-ct) & \text{if} \quad x \geq ct \end{cases}$$

   where $v_0(x) = -(max(1-x^2, 0))^2$

4. Show that the scheme is stable and monotone for an appropriate (sufficient) CFL [1] condition on the mesh steps to be found

   The scheme is given by, for $c \geq 0$:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + \max(c\frac{U_j^n - U_{j-1}^n}{h}, -c\frac{U_{j+1}^n - U_j^n}{h}) = 0$$

   We have two cases to observe. The first case is given by:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + c\frac{U_j^n - U_{j-1}^n}{h} = 0$$

   We obtain then :

$$U_j^{n+1} = (1 - \frac{c\Delta t}{h})U_j^n + \frac{c\Delta t}{h}U_{j-1}^n$$

   If $\frac{c\Delta t}{h} \leq 1$ (with the CFL number equals to $\frac{c\Delta t}{h}$), we get then:

$$\|U^n\| \leq K \left\|U^0\right\|.$$

[1] Courant–Friedrichs–Lewy condition

So the scheme is indeed stable for an appropriate (sufficient) CFL condition.

In addition, the second case is:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} - c\frac{U_{j+1}^n - U_j^n}{h} = 0$$

We obtain then :

$$U_j^{n+1} = (1 - \frac{c\Delta t}{h})U_j^n + \frac{c\Delta t}{h}U_{j+1}^n$$

Similar to the first case, we observe that the scheme is stable for an appropriate (sufficient) CFL condition.

In order to show that the scheme is **monotone**, we recall that an explicit scheme of the form $U_j^{n+1} = F\left(U_{j-1}^n, U_j^n, U_{j+1}^n\right)$ is called "monotone" if F is an increasing function of all its arguments, i.e. $a \to F(a,b,c) \uparrow, b \to F(a,b,c) \uparrow$, and $c \to F(a,b,c) \uparrow$.

We observe that in these two cases as shown above, the function $F\left(U_{j-1}^n, U_j^n, U_{j+1}^n\right)$ where:

$$F\left(U_{j-1}^n, U_j^n, U_{j+1}^n\right) = U_j^n - \Delta t \max(c\frac{U_j^n - U_{j-1}^n}{h}, -c\frac{U_{j+1}^n - U_j^n}{h})$$

is an increasing function of all its arguments for an appropriate (sufficient) CFL condition, i.e if $\frac{c\Delta t}{h} \leq 1$ (with the CFL number equals to $\frac{c\Delta t}{h}$).

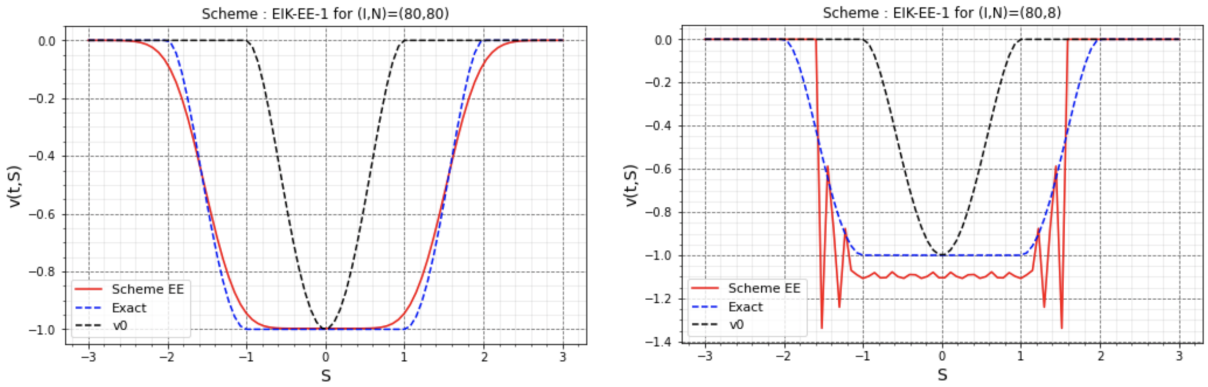

Figure 1: Graph of the Eikonal equation using Euler Forward scheme. (a) For $N = I = 80$. (b) For $I = 80, N = 8$.

By observing the Fig 1, the results are indeed coherent with the CFL condition.

5. Estimate the error at the point $S_{val} = 1.5$ and $T = 1$, and the order of convergence.

   To do so, we use for instance $N = I$ in a list of the form $10 \times 2^k, k = 0, 1, 2, \ldots, 6$, as shown in the Table 2 below, knowing that the exact value is equal to -0.5625.

	I	N	U(s)	error	order alpha
0	10	10	-0.458256	0.104244	NaN
1	20	20	-0.499857	0.062643	0.787587
2	40	40	-0.521524	0.040976	0.634454
3	80	80	-0.537191	0.025309	0.707628
4	160	160	-0.548287	0.014213	0.839949
5	320	320	-0.555030	0.007470	0.932146

Figure 2: Error Table of EE scheme

6. Test also the scheme on the following initial data

$$v_0(x) = \left(\max\left(1 - x^2, 0\right)\right)^2 \quad \text{and with} \quad T = 0.4 \tag{4}$$

We obtain the following graph:

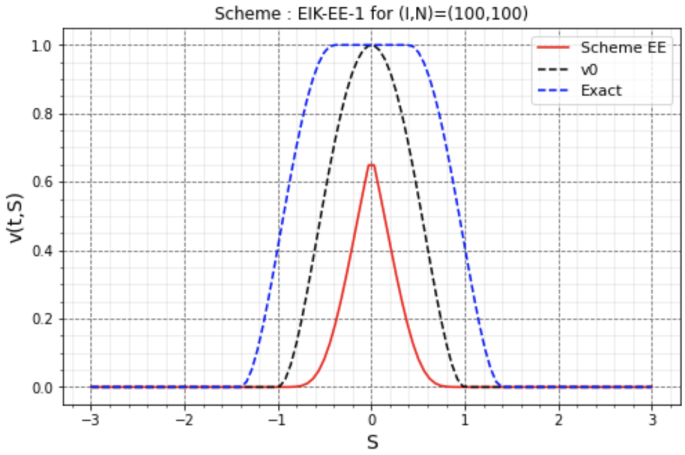

Figure 3: Graph of the Eikonal equation using Euler Forward scheme for $N = I = 100$

2.2 Improving the order of consistency

1. Show that the basic attempt to improve the accuracy with the following scheme is not working (take for instance $N = I = 400, N = I = 800$)

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + c\left|\frac{U_{j+1}^n - U_{j-1}^n}{2h}\right| = 0 \tag{5a}$$

$$n = 0, \ldots, N-1; \quad j = 1, \ldots, I$$

$$U_0^n = v_\ell\left(t_n\right) \equiv 0, \quad U_{I+1}^n = v_r\left(t_n\right) \equiv 0 \quad n = 0, \ldots, N \tag{5b}$$

$$U_j^0 = v_0\left(x_j\right) \quad j = 1, \ldots, I \tag{5c}$$

The previous scheme could be rewritten like this :

$$U_j^{n+1} = U_j^n + \Delta t.c \left| \frac{U_{j+1}^n - U_{j-1}^n}{2h} \right|$$

In matrix format, we have:

$$U^{n+1} = U^n + \Delta t.c \left| A' U^n + q'(t_n) \right|$$

with

$$A' = \frac{1}{2h} \begin{bmatrix} 0 & 1 & 0 & \dots & \dots & 0 & 0 \\ 1 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & 1 & 0 & 1 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & 1 \\ 0 & 0 & \dots & \dots & 0 & 1 & 0 \end{bmatrix}$$

and the vector function of boundary conditions :

$$q'(t_n) := \begin{pmatrix} \frac{1}{2h} U_0^n \\ 0 \\ \vdots \\ 0 \\ \frac{1}{2h} U_{I+1}^n \end{pmatrix} = \begin{pmatrix} \frac{1}{2h} v_l(t_n) \\ 0 \\ \vdots \\ 0 \\ \frac{1}{2h} v_r(t_n) \end{pmatrix}$$

The program we implemented (in python) is shown in the section Annex 2.

As expected, the scheme is not stable for large values of I and N, we can see in Fig. 4 that the scheme explodes for $N = I = 800$.

2. Look for a, b, c such that :

$$\phi_x(x_j) = \frac{a\phi(x_j) + b\phi(x_{j-1}) + c\phi(x_{j-2})}{h} + O\left(h^2\right)$$

where $x_i = S_{min} + ih$

By using the left-sided three points schemes for approximating $u'(x)$, we have that third order Taylor expansions of u at x yield that if $u \in \mathcal{C}^3(\mathbb{R})$, then

$$u(x-h) = u(x) - hu'(x) + \frac{h^2}{2}u''(x) + \frac{h^3}{6}u^{(3)}(x - \theta_1 h), \quad \text{for some } \theta_1 \in]0,1[,$$

$$u(x-2h) = u(x) - 2hu'(x) + 2h^2u''(x) + \frac{4h^3}{3}u^{(3)}(x - 2\theta_2 h), \quad \text{for some } \theta_2 \in]0,1[.$$

So, by multiplying the first line by 4 and then subtract the second line, we get that :

$$u'(x) \approx \frac{3u(x) - 4u(x-h) + u(x-2h)}{2h}$$

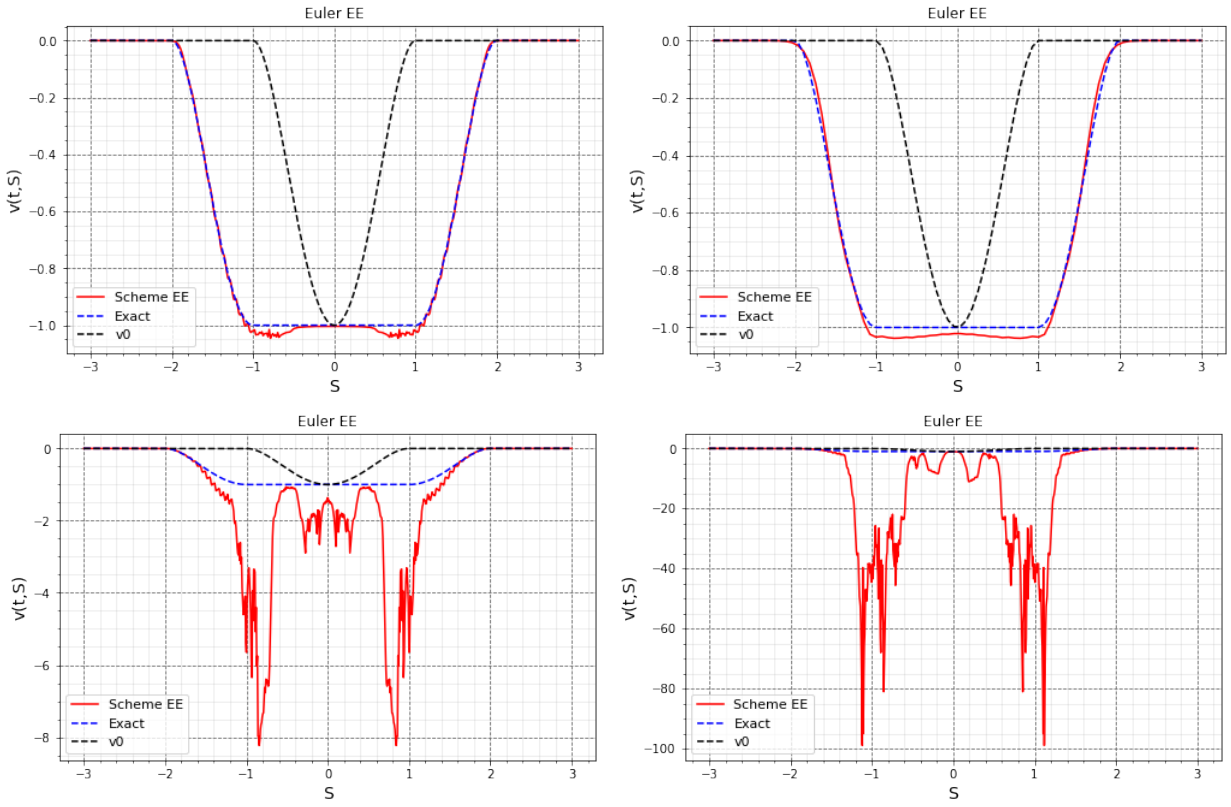

Figure 4: Graph of the Eikonal equation using Euler Forward scheme with absolute value. (a) For $N = I = 80$. (b) For $I = 400, N = 400$. (c) For $I = 800, N = 800$. (d) For $I = 1000, N = 1000$.

So, for $\phi_x(x)$ we get :

$$\phi_x(x_j) = \frac{\frac{3}{2}\phi(x_j) - \frac{4}{2}\phi(x_{j-1}) + \frac{1}{2}\phi(x_{j-2})}{h} + O\left(h^2\right)$$

3. Let

$$\tilde{D}^- U_j^n = \frac{3U_j^n - 4U_{j-1}^n + U_{j-2}^n}{2h} \quad \text{and} \quad \tilde{D}^+ U_j^n = -\frac{3U_j^n - 4U_{j+1}^n + U_{j+2}^n}{2h}$$

Program the following modified scheme:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + \max\left(c\tilde{D}^- U_j^n, -c\tilde{D}^+ U_j^n\right) = 0 \tag{6a}$$

$$n = 0, \ldots, N-1; \quad j = 1, \ldots, I$$

$$U_0^n = v_\ell(t_n) \equiv 0, \quad U_{I+1}^n = v_r(t_n) \equiv 0 \quad n = 0, \ldots, N \tag{6b}$$

$$U_j^0 = v_0(x_j) \quad j = 1, \ldots, I \tag{6c}$$

In matrix format, we get an explicit expression for U^{n+1} with respect to U^n :

$$U^{n+1} = U^n - \Delta t. \max\left[c(\tilde{D}^-_{mat}U^n + \tilde{q}^-(t_n)), -c(\tilde{D}^+_{mat}U^n + \tilde{q}^+(t_n))\right] \tag{7}$$

with :

$$\tilde{D}^-_{mat} = \frac{1}{2h}\begin{bmatrix} 3 & 0 & 0 & \dots & \dots & 0 & 0 \\ -4 & 3 & 0 & 0 & \dots & 0 & 0 \\ 1 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & -4 & 3 & 0 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \dots & 1 & -4 & 3 \end{bmatrix}$$

$$\tilde{D}^+_{mat} = \frac{1}{2h}\begin{bmatrix} -3 & 4 & -1 & \dots & \dots & 0 & 0 \\ 0 & -3 & 4 & -1 & \dots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & 0 & -3 & 4 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & -1 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & 4 \\ 0 & 0 & \dots & \dots & 0 & 0 & -3 \end{bmatrix}$$

and the vector function of boundary conditions :

$$\tilde{q}^-(t_n) := \begin{pmatrix} -\frac{4}{2h}U^n_0 + \frac{1}{2h}U^n_{-1} \\ \frac{1}{2h}U^n_0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -\frac{3}{2h}v_l(t_n) \\ \frac{1}{2h}v_l(t_n) \\ \vdots \\ 0 \\ 0 \end{pmatrix}$$

and by supposing that $U^n_{I+2} = U^n_{I+1} = v_l(t_n)$

$$\tilde{q}^+(t_n) := \begin{pmatrix} 0 \\ 0 \\ \vdots \\ -\frac{1}{2h}U^n_{I+1} \\ \frac{4}{2h}U^n_{I+1} - \frac{1}{2h}U^n_{I+2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ -\frac{1}{2h}v_r(t_n) \\ \frac{3}{2h}v_r(t_n) \end{pmatrix}$$

4. Is the error improved with this scheme ? Draw error tables for the scheme. Show that the scheme is not monotone. Observe that the scheme still converges numerically towards the correct solution

   We have implemented (in python) the program for this scheme, as shown in the section Annex 3. We also plot some graphs corresponding to this scheme for $(I, N) = (80, 80)$, $(I, N) = (400, 400)$ and $(I, N) = (800, 800)$ (as shown in Fig. 5).

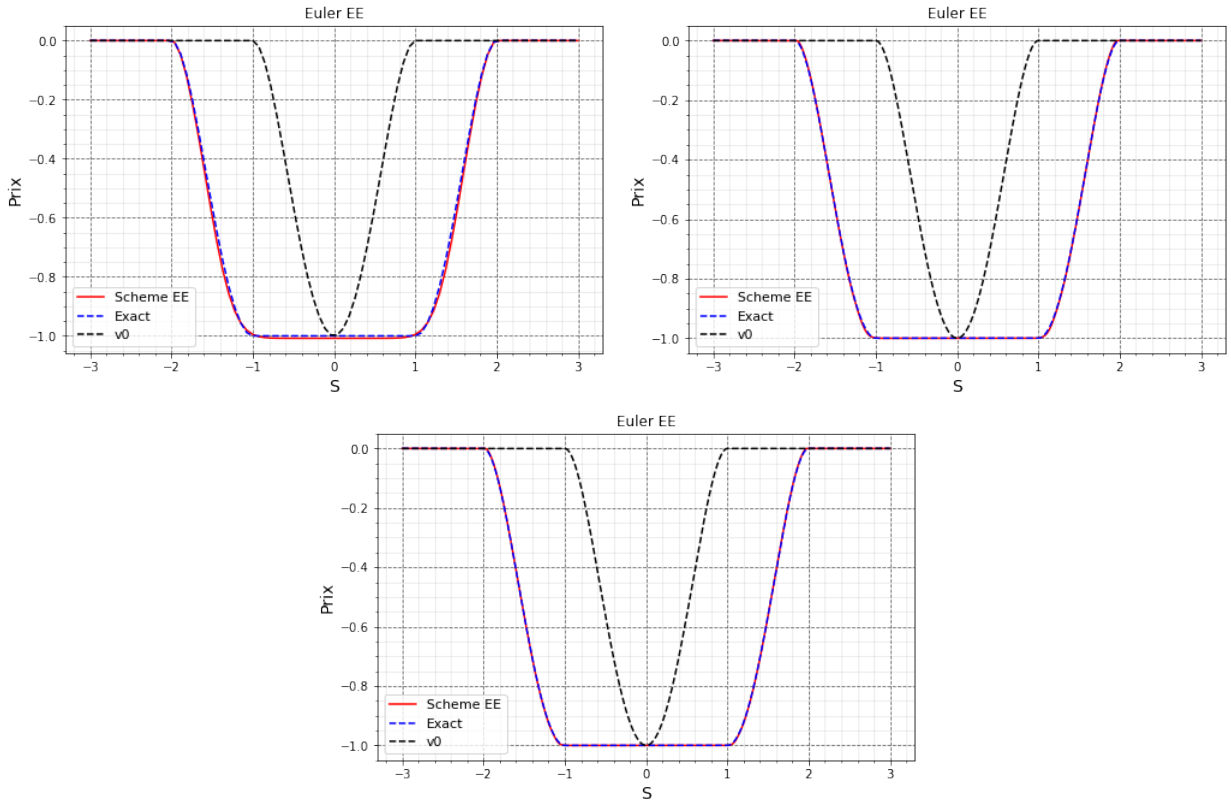

Figure 5: Graph of the Eikonal equation using Euler Forward scheme modified. (a) For $N = I = 80$. (b) For $I = 400, N = 400$. (c) For $I = 800, N = 800$.

Thanks to this graphics, we remark that the scheme does not explode, for large I and N. We also notice that the error has been improved with this scheme. To illustrate that fact, we draw error tables for this scheme (as shown in Tab. 6) and compared with the one using EE scheme (as shown in Tab. 7).

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	10	-0.458256	NaN	NaN	NaN	0.008158
1	20	20	-0.499857	-0.041600	NaN	NaN	0.008129
2	40	40	-0.521524	-0.021668	0.941049	0.974944	0.009496
3	80	80	-0.537191	-0.015666	0.467879	0.476310	0.016004
4	160	160	-0.548287	-0.011096	0.497611	0.502097	0.080705
5	320	320	-0.555030	-0.006743	0.718667	0.721906	0.394498
6	640	640	-0.558679	-0.003649	0.885769	0.887766	3.785729

Figure 6: Error Table of EE scheme

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	10	-0.698748	-0.136248	NaN	NaN	0.000000
1	20	20	-0.653443	-0.090943	0.583193	0.625149	0.008221
2	40	40	-0.628261	-0.065761	0.467731	0.484578	0.000000
3	80	80	-0.594177	-0.031677	1.053781	1.072771	0.008010
4	160	160	-0.569726	-0.007226	2.132274	2.151493	0.043710
5	320	320	-0.565275	-0.002775	1.380872	1.387096	0.139304
6	640	640	-0.563614	-0.001114	1.316906	1.319874	2.142348

Figure 7: Error Table of EE modified scheme

So we can deduce that the scheme still converges numerically towards the correct solution.

Show that the scheme is not monotone

Let's show that the scheme is not monotone. To do so, let us recall that an explicit scheme of the form $U_j^{n+1} = F\left(U_{j-1}^n, U_j^n, U_{j+1}^n\right)$ is called "monotone" if F is an increasing function of all its arguments, i.e. $a \to F(a,b,c) \uparrow, b \to F(a,b,c) \uparrow$, and $c \to F(a,b,c) \uparrow$.

In our case we have the following,

$$U^{n+1} = U^n - \Delta t. \max\left[c(\tilde{D}_{mat}^- U^n + \tilde{q}^-(t_n)), -c(\tilde{D}_{mat}^+ U^n + \tilde{q}^+(t_n))\right]$$

Let's suppose that, the max in the previous equation is equal to $c(\tilde{D}_{mat}^- U^n + \tilde{q}^-(t_n))$, so we have at step space j :

$$\begin{aligned} U_j^{n+1} &= U_j^n - \Delta t.c\tilde{D}^- U_j^n \\ &= U_j^n - \frac{\Delta t.c}{2h}\left(3U_j^n - 4U_{j-1}^n + U_{j-2}^n\right) \\ &= U_j^n \underbrace{\left(1 - \frac{3\Delta tc}{2h}\right)}_{\geq 0} + \underbrace{\frac{2\Delta t.c}{h}}_{\geq 0} U_{j-1}^n \underbrace{-\frac{\Delta t.c}{2h}}_{\leq 0} U_{j-2}^n \end{aligned}$$

Thanks to the CFL condition we can see that the scheme is not monotone, indeed the scheme is non-increasing with respect to U_{j-2}^n.

5. Show that the scheme corresponds to an RK2 scheme. Observe that the scheme is numerically of order $(2,2)$, and is subject to some CFL condition for stability

We recall that, for the approximation of $u_t = L(u)$, the **RK2 scheme** is such that $u^{n,1} = u^n n + \Delta t L\left(t_n, u^n\right)$ and $u^{n+1} = u^n + \frac{\Delta t}{2}\left(L\left(t_n, u^n\right) + L\left(t_{n+1}, u^{n,1}\right)\right)$. It is consistent of order 2 in time.

Let us denote $U^{n+1} = S^1(U^n)$ in the form of the scheme 7. We have then :

$$S^1 : U \mapsto U - \Delta t. \max\left[c(\tilde{D}_{mat}^- U + \tilde{q}^-(t_n)), -c(\tilde{D}_{mat}^+ U + \tilde{q}^+(t_n))\right]$$

Consider now the following scheme (RK2):

$$U^{n+1} = \frac{1}{2}\left(U^n + S^1\left(S^1\left(U^n\right)\right)\right)$$

We have implemented (in python) the program for this scheme, as shown in the section Annex 4.

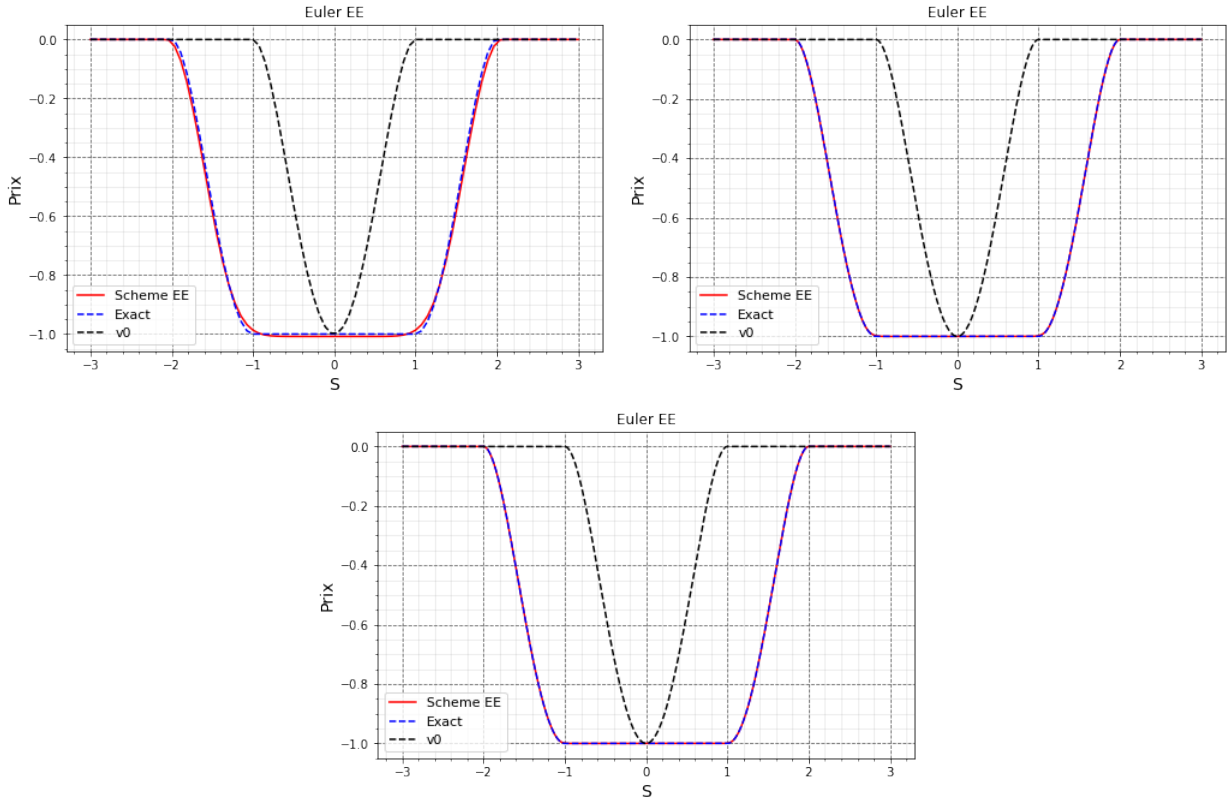

Figure 8: Graph of the Eikonal equation using RK2 scheme. (a) For $N = I = 80$. (b) For $I = 400, N = 400$. (c) For $I = 800, N = 800$..

Thanks to these graphics (Fig. 8), we remark that the scheme does not explode, for large I and N.

Moreover we observe that the scheme is order $(2, 2)$, thanks to the following Table 9.

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	10	-0.662104	-0.099604	NaN	NaN	0.000814
1	20	20	-0.622980	-0.060480	0.719730	0.771510	0.004465
2	40	40	-0.604358	-0.041858	0.530955	0.550079	0.008472
3	80	80	-0.584620	-0.022120	0.920183	0.936765	0.021941
4	160	160	-0.567385	-0.004885	2.178948	2.198588	0.113173
5	320	320	-0.563836	-0.001336	1.870059	1.878488	0.429523
6	640	640	-0.562842	-0.000342	1.965302	1.969732	1.870528

Figure 9: Error Table of RK2 scheme

To show that the scheme is subject to some CFL condition for stability. We have tested the scheme for different time step N and different space step I (with the value of I ten times larger than N).

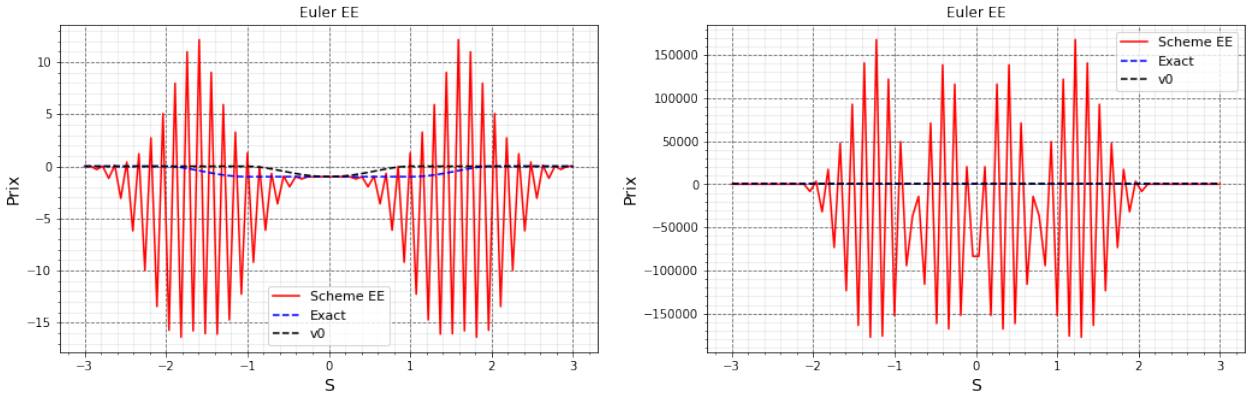

Figure 10: Graph of the Eikonal equation using RK2 scheme. (a) For $I = 80$and $N = 20$. (b) For $I = 80$ and $N = 8$.

We can also see that in the error table (Table 11), which displays enormous error for the scheme when I is ten times bigger than N.

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	1	-2.446657e+00	-1.884157e+00	NaN	NaN	0.000000
1	20	2	-3.043823e+00	-2.481323e+00	-0.397191	-0.425765	0.000000
2	40	4	-2.596379e+01	-2.540129e+01	-3.355720	-3.476585	0.000000
3	80	8	2.858584e+04	2.858641e+04	-10.136212	-10.318876	0.009564
4	160	16	-4.458903e+13	-4.458903e+13	-30.538712	-30.813978	0.034840
5	320	32	3.240610e+30	3.240610e+30	-56.012354	-56.264836	0.209870
6	640	64	-2.264472e+68	-2.264472e+68	-125.716177	-125.999543	0.872838

Figure 11: Error Table of RK2 scheme

So we can deduce that the scheme is subject to some CFL condition for stability.

2.3 IE scheme

The following implicit schemes are not to be programmed in this session. They are presented as examples of implicit non-linear schemes. They will be studied in the next programming session.

In order to get rid of the CFL condition, a natural implicit scheme is the following:

$$\frac{U_j^{n+1}-U_j^n}{\Delta t}+\max\left(c\frac{U_j^{n+1}-U_{j-1}^{n+1}}{h},-c\frac{U_{j+1}^{n+1}-U_j^{n+1}}{h}\right)=0 \tag{8a}$$

$$n=0,\dots,N-1; j=1,\dots,I$$

$$U_0^n=v_\ell(t)\equiv 0,\quad U_{I+1}^n=v_r(t)\equiv 0 \qquad n=0,\dots,N \tag{8b}$$

$$U_j^0=v_0(x_j) \qquad j=1,\dots,I \tag{8c}$$

What is the consistency error of this scheme?

Again, in order to obtain the consistency error of this scheme, we denote $V_j^n=\varphi(t_n,x_j)$ where φ is a sufficiently regular function.

We write then the scheme in abstract form as follows:

$$\begin{aligned}\mathcal{S}(t_{n+1},x_j,V_j^{n+1}) &= \frac{V_j^{n+1}-V_j^n}{\Delta t}+\max(c\frac{V_j^{n+1}-V_{j-1}^{n+1}}{h};-c\frac{V_{j+1}^{n+1}-V_j^{n+1}}{h})\\ &= \frac{\varphi(t_n,x_j)+\Delta t\varphi_t(t_n,x_j)+\frac{1}{2}\Delta t^2\varphi_{tt}(t_n,x_j)+O(\Delta t^3)-\varphi(t_n,x_j)}{\Delta t}\\ &+\max\left(c\frac{\varphi(t_{n+1},x_j)-(\varphi(t_{n+1},x_j)-h\varphi_x(t_{n+1},x_j)+\frac{1}{2}h^2\varphi_{xx}(t_{n+1},x_j))+O(h^3)}{h};\right.\\ &\left.-c\frac{\varphi(t_{n+1},x_j)+h\varphi(t_{n+1},x_j)+\frac{1}{2}h^2\varphi_{xx}(t_{n+1},x_j)+O(h^3)-\varphi(t_{n+1},x_j)}{h}\right)\end{aligned}$$

We obtain then:

$$\begin{aligned}\mathcal{S}(t_{n+1},x_j,V_j^{n+1}) &= \varphi_t(t_n,x_j)+\frac{1}{2}\Delta t\varphi_{tt}(t_n,x_j)\\ &+\max\left(c(\varphi_x(t_{n+1},x_j)+\frac{1}{2}h\varphi_{xx}(t_{n+1},x_j);\right.\\ &\left.-c(\varphi_x(t_{n+1},x_j)+\frac{1}{2}h\varphi_{xx}(t_{n+1},x_j))\right)+O(\Delta t^2)+O(h^2)=0\end{aligned}$$

Let $\mathcal{H}$ corresponds to the PDE equation where:

$$\mathcal{H}(\varphi)(t_n,x)=\varphi_t(t_n,x)+c|\varphi_x(t_n,x)|=\varphi_t(t_n,x)+\max(c\varphi_x(t_n,x),-c\varphi_x(t_n,x))$$

So that by definition, the PDE for φ reads $\mathcal{H}(\varphi)(t,x)=0$. Then a consistency error can be defined as ϵ_j^n such that:

$$\mathcal{S}\left(t_{n+1},x_j,V_j^{n+1}\right)=\mathcal{H}(\varphi)\left(t_{n+1},x_j\right)+\epsilon_j^n$$

We obtain finally:

$$\epsilon_j^n = \frac{1}{2}\Delta t \varphi_{tt}(t_n, x_j) + \frac{1}{2} h \varphi_{xx}(t_{n+1}, x_j) + O(\Delta t^2) + O(h^2)$$

Therefore, the consistency error satisfies:

$$\left|\epsilon_j^n\right| \leq C(\Delta t + h)$$

This means that the consistency error is of order *one* in time and *one* in space.

In order to program the scheme, we use an equivalent matrix formulation of the scheme in the following form:

Let $A^- := \frac{c}{h} \times \text{tridiag}(-1, 1, 0)$ and $A^+ := \frac{c}{h} \times \text{tridiag}(0, 1, -1)$ (where $T = \text{tridiag}\,(a_i, b_i, c_i)$ denotes the tridiagonal matrix with $T_{i,i-1} = a_i, T_{i,i} = b_i$ and $T_{i,i+1} = c_i$). We have:

$$A^- = \frac{c}{h}\begin{bmatrix} 1 & 0 & 0 & \dots & \dots & 0 & 0 \\ -1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & -1 & 1 & 0 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \dots & 0 & -1 & 1 \end{bmatrix} \qquad A^+ = \frac{c}{h}\begin{bmatrix} 1 & -1 & 0 & \dots & \dots & 0 & 0 \\ 0 & 1 & -1 & 0 & \dots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & 0 & 1 & -1 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & -1 \\ 0 & 0 & \dots & \dots & 0 & 0 & 1 \end{bmatrix}$$

The scheme can be written in vector form as follows:

$$\max_{\pm}\left(\frac{U^{n+1} - U^n}{\Delta t} + A^{\pm} U^{n+1} + q^{\pm}\left(t_{n+1}\right)\right) = 0, \qquad \text{in } \mathbb{R}^I$$

where $q^{\pm}$ are vectors taking into account the boundary conditions (here for the chosen boundary conditions we have $q^{\pm} \equiv 0$, so we won't have any problems with that). We denote:

$$B^{\pm} := I + \Delta t A^{\pm}, \quad b^{\pm} := U^n - \Delta t q^{\pm}$$

Notice that the scheme is also equivalent to find $x = U^{n+1} \in \mathbb{R}^I$ solution of

$$\max_{\pm}\left(B^{\pm} x - b^{\pm}\right) = 0, \quad \text{in} \quad \mathbb{R}^I$$

or, more precisely:

$$\max\left(B^{+} x - b^{+}, B^{-} x - b^{-}\right) = 0, \quad \text{in} \quad \mathbb{R}^I$$

1. Implement a **("semi-smooth")** Newton's method for finding $x \in \mathbb{R}^I$ solution of

   $$F(x) = \max(Bx - b, Cx - c) \equiv 0 \quad \text{for} \quad x \in \mathbb{R}^I$$

   for given vectors $b, c \in \mathbb{R}^I$ and matrices $B, C \in \mathbb{R}^{I \times I}$

   We want to apply a Newton type algorithm for solving $F(x) = 0$. We consider the following algorithm:

- iterate over $k \geq 0$ (for a given x^0 starting point of $\mathbb{R}^I$, to be chosen)

$$x^{k+1} = x^k - F'\left(x^k\right)^{-1} F\left(x^k\right)$$

until $F\left(x^k\right) = 0$ (or, that $x^{k+1} = x^k$ i.e until convergence).

In this iteration, $F(x^k) := \max(Bx^k - b, Cx^k - c)$ and we will take the following definition for a square matrix $F'\left(x^k\right)$ (row by row derivative):

$$F'\left(x^k\right)_{i,j} := \begin{cases} B_{i,j} & \text{if } \left(Bx^k - b\right)_i \geq \left(Cx^k - c\right)_i \\ C_{i,j} & \text{otherwise} \end{cases}$$

(Note the specific choice $F'\left(x^k\right)_{i,j} = B_{i,j}$ even in the case when $\left(Bx^k - b\right)_i = \left(Cx^k - c\right)_i$. The other choice $F'\left(x^k\right)_{i,j} = C_{i,j}$ if $\left(Bx^k - b\right)_i = \left(Cx^k - c\right)_i$ works also but may be less efficient : i.e more iterations might be needed).

We also notice that this Newton's method converges always if B and C are diagonally dominant M-matrices.

2. Program the EI scheme

We have implemented (in python) the program for this scheme, as shown in the section Annex 5. We plot some graphs as shown in Fig. 12 below using this EI scheme with the value of $(I, N) = (100, 100)$ and $(I, N) = (400, 400)$.

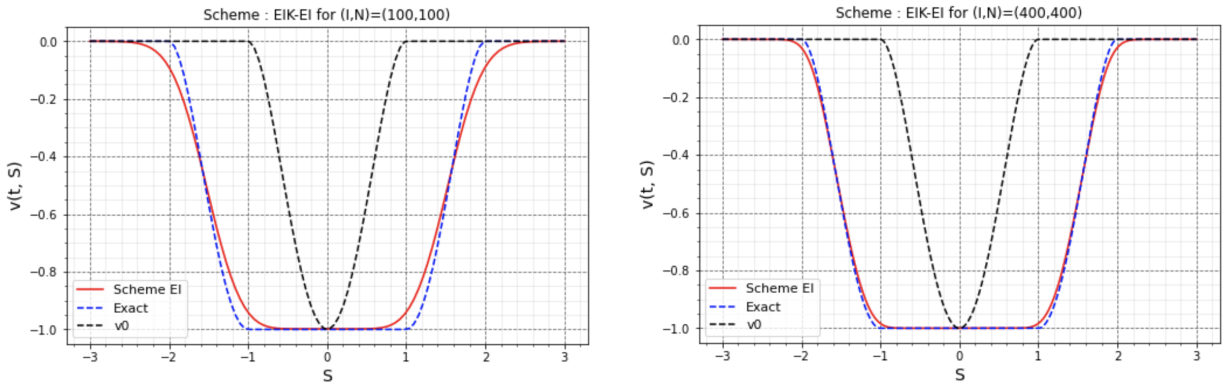

Figure 12: Graph of the Eikonal equation using Euler Implicit scheme. (a) For $N = I = 100$. (b) For $I = N = 400$.

3. Check the stability of the scheme unconditionally with respect to the mesh steps

We want to estimate the error at the point $S_{val} = 1.5$ and $T = 1$ and the order of convergence (with time and space). Therefore, we draw two error tables in order to compare the results between the Euler Explicit (EE) scheme (see Table 13) and the Euler Implicit (EI) scheme (see Table 14) by using different values of N and I with $I = 10 \times 2^k$ for $k = 0, 1, \ldots, 6$ and $N = I/10$. Indeed, we observe that the EI scheme is stable unconditionally with respect to the mesh steps i.e N and I.

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	1	-1.502630e-01	4.122370e-01	NaN	NaN	0.000968
1	20	2	-1.836735e-01	3.788265e-01	0.121937	0.130709	0.001261
2	40	4	-2.899588e-01	2.725412e-01	0.475063	0.492174	0.004820
3	80	8	-1.150614e+00	-5.881142e-01	-1.109622	-1.129618	0.008533
4	160	16	-8.030640e+00	-7.468140e+00	-3.666581	-3.699630	0.024019
5	320	32	-1.448551e+05	-1.448545e+05	-14.243496	-14.307701	0.105657
6	640	64	-1.584771e+13	-1.584771e+13	-26.705094	-26.765288	0.601581

Figure 13: Error Table of EE

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	1	-0.349821	0.212679	NaN	NaN	0.159191
1	20	2	-0.419287	0.143213	0.570510	0.611554	0.013164
2	40	4	-0.461938	0.100562	0.510083	0.528455	0.084507
3	80	8	-0.493825	0.068675	0.550226	0.560142	0.012591
4	160	16	-0.519190	0.043310	0.665095	0.671090	0.077032
5	320	32	-0.538097	0.024403	0.827630	0.831360	0.114709
6	640	64	-0.549933	0.012567	0.957431	0.959589	1.610012

Figure 14: Error Table of EI

We also plot two graphs for $(I, N) = (80, 80)$ and $(I, N) = (80, 8)$ by using EI scheme (see Fig. 15). We can clearly see that the scheme is stable even when N is small $(N = I/10)$.

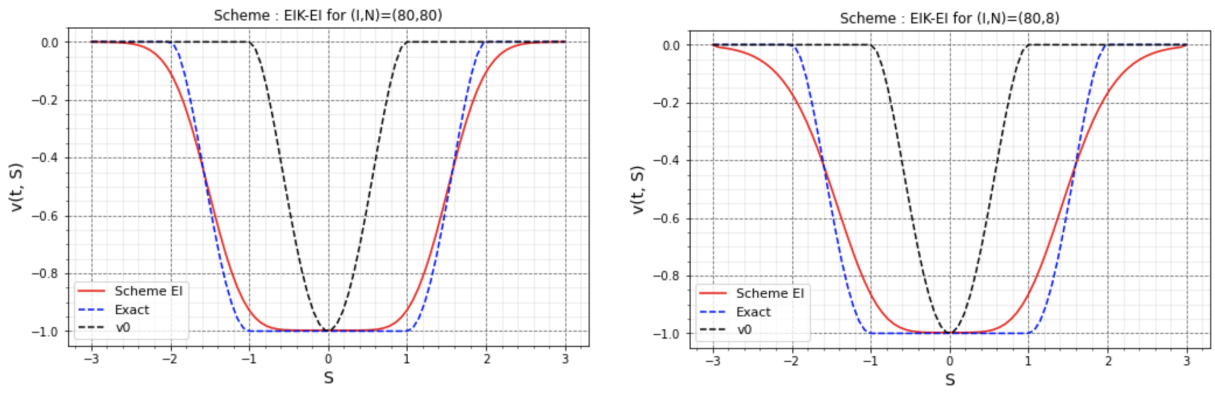

Figure 15: Graph of the Eikonal equation using Euler Implicit scheme. (a) For $N = I = 80$. (b) For $I = 80, N = 8$.

2.4 Implicit second order variant

We want to program the following scheme:

$$\frac{3U_j^{n+1} - 4U_j^n + U_j^{n-1}}{2\Delta t} + \max\left(c\tilde{D}^- U_j^{n+1}, -c\tilde{D}^+ U_j^{n+1}\right) = 0 \tag{9a}$$

$$n = 0, \dots, N-1; \quad j = 1, \dots, I$$

$$U_0^n = v_\ell\left(t_n\right) \equiv 0, \quad U_{I+1}^n = v_r\left(t_n\right) \equiv 0 \quad n = 0, \dots, N \tag{9b}$$

$$U_j^0 = v_0\left(x_j\right) \quad j = 1, \dots, I \tag{9c}$$

where:

$$\tilde{D}^- U_j^{n+1} = \frac{3U_j^{n+1} - 4U_{j-1}^{n+1} + U_{j-2}^{n+1}}{2h} \quad \text{and} \quad \tilde{D}^+ U_j^{n+1} = -\frac{3U_j^{n+1} - 4U_{j+1}^{n+1} + U_{j+2}^{n+1}}{2h}$$

The scheme can be written as the same form as previously, indeed, we have :

$$\frac{3U_j^{n+1} - 4U_j^n + U_j^{n-1}}{2\Delta t} + \max\left(c\tilde{D}^- U_j^{n+1}, -c\tilde{D}^+ U_j^{n+1}\right) = 0$$

which is equivalent to :

$$\max\left(\frac{3U_j^{n+1} - 4U_j^n + U_j^{n-1}}{2\Delta t} + c\tilde{D}^- U_j^{n+1}; \frac{3U_j^{n+1} - 4U_j^n + U_j^{n-1}}{2\Delta t} - c\tilde{D}^+ U_j^{n+1}\right) = 0$$

By replacing $\tilde{D}^-$ and $\tilde{D}^+$ by their expressions, we obtain (in matrix format) :

$$\max_{\pm}\left[\left(3Id + 2\Delta t\tilde{A}^{\pm}\right)U^{n+1} + 2\Delta t\tilde{q}^{\pm}(t_{n+1}) - 4U^n + U^{n-1}\right] = 0$$

with :

$$\tilde{A}^- = \frac{c}{h}\begin{bmatrix} 3 & 0 & 0 & \dots & \dots & 0 & 0 \\ -4 & 3 & 0 & 0 & \dots & 0 & 0 \\ 1 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & -4 & 3 & 0 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \dots & 1 & -4 & 3 \end{bmatrix}$$

$$\tilde{A}^+ = \frac{c}{h}\begin{bmatrix} 3 & -4 & 1 & \dots & \dots & 0 & 0 \\ 0 & 3 & -4 & 1 & \dots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & 0 & 3 & -4 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 1 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & -4 \\ 0 & 0 & \dots & \dots & 0 & 0 & 3 \end{bmatrix}$$

and the vector function of boundary conditions :

$$\tilde{q}^-(t_n) := \begin{pmatrix} -\frac{4c}{h}U_0^n + \frac{c}{h}U_{-1}^n \\ \frac{c}{h}U_0^n \\ \vdots \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -\frac{3c}{h}v_l(t_n) \\ \frac{c}{h}v_l(t_n) \\ \vdots \\ 0 \\ 0 \end{pmatrix}$$

and by supposing that $U_{I+2}^n = U_{I+1}^n = v_l(t_n)$

$$\tilde{q}^+(t_n) := \begin{pmatrix} 0 \\ 0 \\ \vdots \\ -\frac{c}{h}U_{I+1}^n \\ \frac{4c}{h}U_{I+1}^n - \frac{c}{h}U_{I+2}^n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ -\frac{c}{h}v_r(t_n) \\ \frac{3c}{h}v_r(t_n) \end{pmatrix}$$

This can be written :

$$\max_{\pm}\left(B^{\pm}x - b^{\pm}\right) = 0, \quad \text{in } \mathbb{R}^I$$

with :

$$B^{\pm} := 3I_d + 2\Delta t\tilde{A}^{\pm}, \quad b^{\pm} := 4U^n - U^{n-1} - 2\Delta t\tilde{q}^{\pm}$$

We have implemented (in python) the program for this scheme by using a **semi-smooth Newton's method**, as shown in the section Annex 6. We plot some graphs as shown in

Fig. 16 below using this second order EI scheme for Eikonal equation with the value of $(I, N) = (100, 100)$, $(I, N) = (400, 400)$, $(I, N) = (80, 8)$ and $(I, N) = (400, 40)$. Indeed, we notice that this EI scheme for Eikonal equation is unconditionally stable with respect to the mesh parameters.

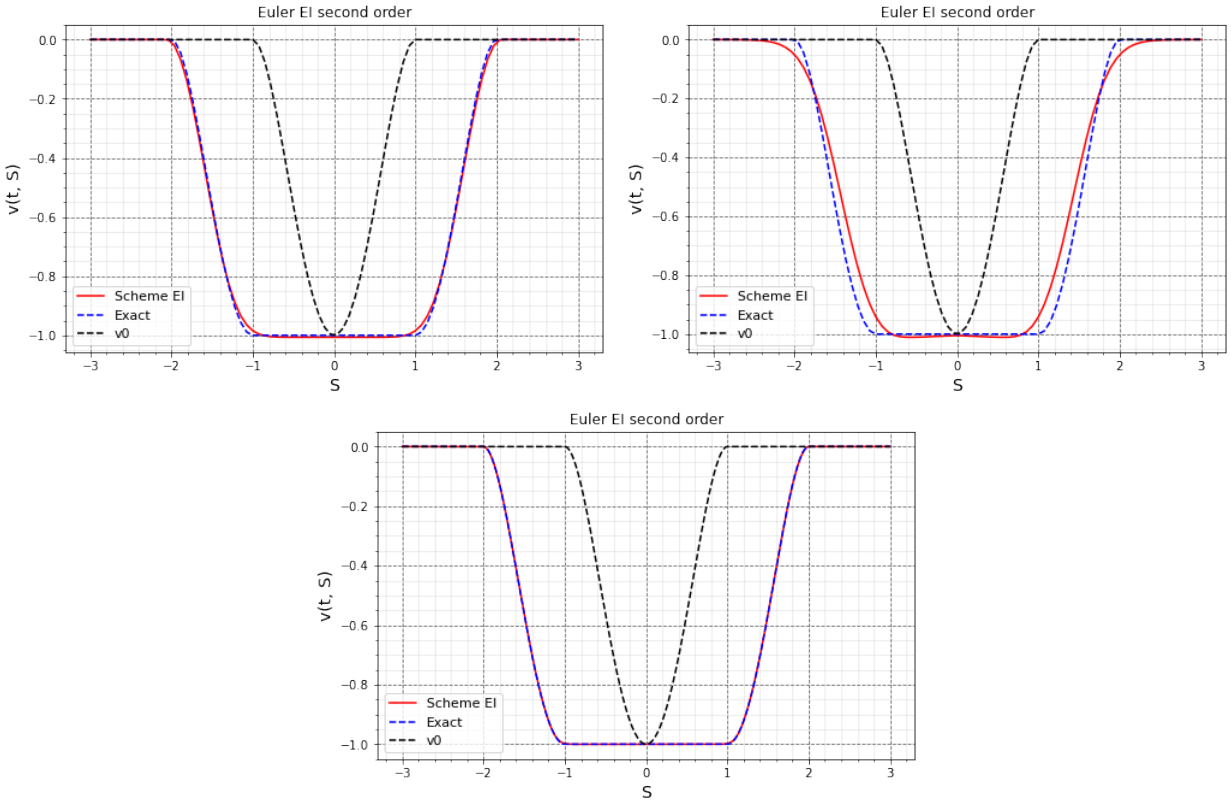

Figure 16: Graph of the Eikonal equation using second order Euler implicit scheme. (a) For $N = I = 80$. (b) For $I = 80, N = 8$. (c) For $I = 400, N = 400$..

As an implicit scheme, it is not subject to some CFL condition as it is shown in Fig. 16 (b). Therefore this scheme is stable. Moreover, this scheme is of order *two* both in time and space, as shown in the following table :

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	10	-0.567192	-0.004692	NaN	NaN	0.000000
1	20	20	-0.576305	-0.013805	1.556837	1.668840	0.028777
2	40	40	-0.581508	-0.019008	0.461420	0.478039	0.060757
3	80	80	-0.573040	-0.010540	0.850639	0.865968	0.184824
4	160	160	-0.562315	0.000185	5.832797	5.885372	0.775867
5	320	320	-0.561389	0.001111	2.586452	2.598111	3.325298

Figure 17: Error Table of second order EI for $N = I$

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	1	-0.349821	0.212679	NaN	NaN	0.000000
1	20	2	-0.352066	0.210434	0.015306	0.016407	0.000000
2	40	4	-0.387735	0.174765	0.267958	0.277609	0.069872
3	80	8	-0.442920	0.119580	0.547440	0.557305	0.039750
4	160	16	-0.501268	0.061232	0.965614	0.974318	0.149954
5	320	32	-0.535214	0.027286	1.166132	1.171388	0.539803
6	640	64	-0.549788	0.012712	1.101946	1.104430	2.740898

Figure 18: Error Table of second order EI for $N = I/10$

3 A simple uncertain volatility model

We consider an academic uncertain volatility model where the volatility of some asset can be controlled and can take any value $\sigma \in [0, 1]$ (i.e. $dS_\tau = \sigma_\tau dW_\tau$). The maximisation of the terminal price (with payoff $v_0(.)$) under such controlled asset leads to an HJB equation of the form:

$$v_t(t,x) + \min_{\sigma\in[0,1]} \left(-\frac{1}{2}\sigma^2 v_{xx}(t,x) \right) = 0, \quad t \in (0,T), x \in \mathbb{R}$$
$$v(T,x) = v_0(x)$$

After a time reversal, and imposing boundary conditions, we are led to the following second order HJB equation:

$$-v_t(t,x) + \min\left(0, -\frac{1}{2}v_{xx}(t,x)\right) = 0, \quad t \in (0,T), \quad x \in (S_{\min}, S_{\max}) \tag{10a}$$
$$v(0,x) = v_0(x) \quad x \in (S_{\min}, S_{\max}) \tag{10b}$$

with the following boundary conditions:

$$v\left(t, S_{\min}\right) = v_\ell \quad t \in (0,T) \tag{11a}$$
$$v\left(t, S_{\max}\right) = v_r \quad t \in (0,T) \tag{11b}$$

We will consider the following parameters:

$$S_{\min} = -3, \quad S_{\max} = 3, \quad \text{and} \quad T = 0.5$$

with smooth initial data

$$v_0(x) := \operatorname{sign}(x)\left((\max(1-|x|,0))^4 - 1\right)$$

and boundary conditions compatible the initial data: $v_l = 1$ and $v_r = -1$.

We introduce the second order approximation of $-u_{xx}$:

$$D^2 U_i = \frac{U_{i+1} - 2U_i + U_{i-1}}{h^2}$$

3.1 Euler Explicit scheme

A possible Euler Explicit (EE) scheme for 10 and 11 is therefore:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + \min\left(0, -\frac{1}{2} D^2 U_i^n\right) = 0 \tag{12a}$$

$$n = 0, \ldots, N-1; \quad j = 1, \ldots, I$$

$$U_0^n = v_\ell(t), \quad U_{I+1}^n = v_r(t) \qquad n = 0, \ldots, N \tag{12b}$$

$$U_j^0 = v_0(x_j) \qquad j = 1, \ldots, I \tag{12c}$$

What is the consistency error of the scheme?

We have:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + \min(0, -\frac{1}{2} D^2 U_j^n) = 0$$

$$\iff \frac{U_j^{n+1} - U_j^n}{\Delta t} + \min(0, -\frac{1}{2}\frac{U_{i-1}^n - 2U_i^n + U_{i+1}^n}{h^2}) = 0$$

We recall that the centered three points finite difference approximation of $u''(x)$ is:

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}$$

Fourth order Taylor expansions of u at x yield that if $u \in \mathcal{C}^4(\mathbb{R})$, then:

$$u(x+h) = u(x) + hu'(x) + \frac{h^2}{2}u''(x) + \frac{h^3}{6}u^{(3)}(x) + \frac{h^4}{24}u^{(4)}(x+\theta_1 h), \quad \text{for some } \theta_1 \in]0,1[,$$

$$u(x-h) = u(x) - hu'(x) + \frac{h^2}{2}u''(x) - \frac{h^3}{6}u^{(3)}(x) + \frac{h^4}{24}u^{(4)}(x-\theta_2 h), \quad \text{for some } \theta_2 \in]0,1[.$$

Summing up the two lines above,

$$u''(x) - \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} = \frac{h^2}{24}\left(u^{(4)}(x+\theta_1 h) + u^{(4)}(x-\theta_2 h)\right)$$

Since we know that, by using Taylor expansions:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = \varphi_t(t_n, x_j) + \frac{1}{2}\Delta t \varphi_{tt}(t_n, x_j) + O(\Delta t^2)$$

$$\frac{U_{i-1}^n - 2U_i^n + U_{i+1}^n}{h^2} = \varphi_{xx}(t_n, x_j) + \frac{1}{12}h^2 \varphi_{xxxx}(t_n, x_j) + O(h^3)$$

So we get the consistency error :

$$\epsilon_j^n = \frac{1}{2}\Delta t \varphi_{tt}(t_n, x_j) + O(\Delta t^2) + \min(0, -\frac{1}{24}h^2 \varphi_{xxxx}(t_n, x_j)) + O(h^3)$$

Therefore, the consistency error satisfies:

$$\left|\epsilon_j^n\right| \leq C(\Delta t + h^2)$$

This means that the consistency error is of order *one* in time and *two* in space.

+ Program and test the EE scheme

In our code, we program the matrix D and the vector function $q(t)$ such that:

$$(DU^n + q(t))_{1 \leq i \leq I} = \left(\frac{U_{i+1}^n - 2U_i^n + U_{i-1}^n}{h^2}\right)_{1 \leq i \leq I}.$$

where:

$$D = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & \dots & \dots & 0 & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & 1 & -2 & 1 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & 1 \\ 0 & 0 & \dots & \dots & 0 & 1 & -2 \end{bmatrix}$$

and the vector function of boundary conditions :

$$q\left(t_n\right) := \begin{pmatrix} \frac{1}{h^2}U_0^n \\ 0 \\ \vdots \\ 0 \\ \frac{1}{h^2}U_{I+1}^n \end{pmatrix} = \frac{1}{h^2}\begin{pmatrix} v_l(t_n) \\ 0 \\ \vdots \\ 0 \\ v_r(t_n) \end{pmatrix}$$

Furthermore, we use the command `np.minimum(v1,v2)` to minimize component-wise two vectors of type `numpy.array`. In order to improve the efficiency of the code, we also pre-compute the matrice D as a **sparse matrice**.

The program we implemented (in python) is shown in the section Annex 7. We plot some graphs as shown in Fig. 19 below using this EI scheme for volatility model with the value of $(I, N) = (50, 50)$, $(I, N) = (80, 80)$, $(I, N) = (80, 8)$ and $(I, N) = (300, 100)$. We can clearly see that the scheme is not stable for some values of (I, N) and it is stable only for an appropriate (suffcient) CFL condition on the mesh steps.

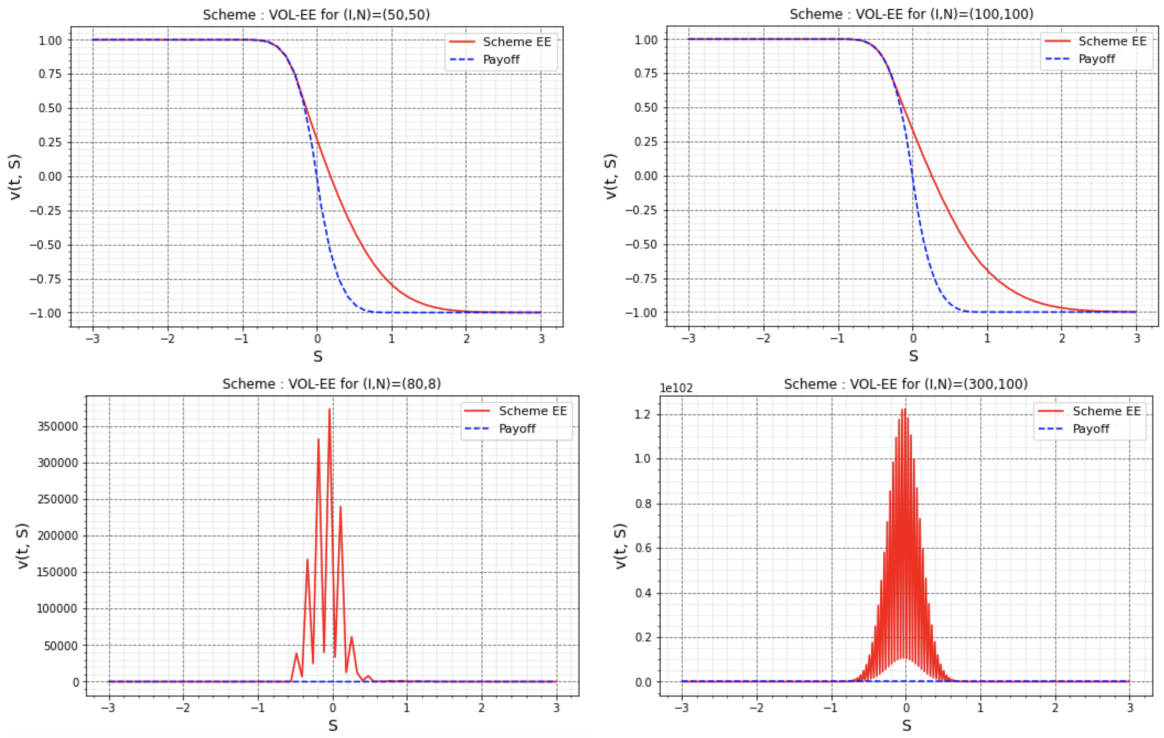

Figure 19: Graph of the uncertain volatility model using Euler Explicit scheme. (a) For $N = I = 50$. (b) For $I = N = 80$. (c) For $I = 80, N = 8$. (d) For $I = 300, N = 100$.

3.2 Euler Implicit scheme

We introduce the second order approximation of $-u_{xx}$ where :

$$D^2 U_i^{n+1} = \frac{U_{i+1}^{n+1} - 2U_i^{n+1} + U_{i-1}^{n+1}}{h^2}$$

Thus, a possible Euler Implicit (EI) scheme for 10 and 11 is given by:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + \min\left(0, -\frac{1}{2} D^2 U_j^{n+1}\right) = 0 \tag{13a}$$

$$n = 0, \ldots, N-1; \quad j = 1, \ldots, I$$

$$U_0^n = v_\ell(t), \quad U_{I+1}^n = v_r(t) \qquad n = 0, \ldots, N \tag{13b}$$

$$U_j^0 = v_0(x_j) \qquad j = 1, \ldots, I \tag{13c}$$

In order to program the scheme, we use an equivalent matrix formulation of the scheme in the following form:

Let A be a square matrix such that:

$$A = -\frac{1}{2h^2}\begin{bmatrix} -2 & 1 & 0 & \dots & \dots & 0 & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & 1 & -2 & 1 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & \ddots & \ddots & \ddots & 1 \\ 0 & 0 & \dots & \dots & 0 & 1 & -2 \end{bmatrix}$$

The scheme can be written in vector form as follows:

$$\min\left(\frac{U^{n+1}-U^n}{\Delta t}; \quad \frac{U^{n+1}-U^n}{\Delta t} + AU^{n+1} + q\left(t_{n+1}\right)\right) = 0, \quad \text{in } \mathbb{R}^I$$

where q are vectors taking into account the boundary conditions. The vector function of boundary conditions :

$$q\left(t_{n+1}\right) := \begin{pmatrix} -\frac{1}{2h^2}U_0^{n+1} \\ 0 \\ \vdots \\ 0 \\ -\frac{1}{2h^2}U_{I+1}^{n+1} \end{pmatrix} = -\frac{1}{2h^2}\begin{pmatrix} v_l(t_{n+1}) \\ 0 \\ \vdots \\ 0 \\ v_r(t_{n+1}) \end{pmatrix}$$

We denote:

$$B := I + \Delta t A, \quad b := U^n - \Delta t q(t_{n+1}), \quad c := U^n$$

Notice that the scheme is also equivalent to find $x = U^{n+1} \in \mathbb{R}^I$ solution of

$$\min\left(Bx - b, x - c\right) = 0, \quad \text{in} \quad \mathbb{R}^I$$

In the python code, we program this scheme by implementing a **("semi-smooth")** Newton's method for finding $x \in \mathbb{R}^I$ solution of

$$F(x) = \min(Bx - b, x - c) \equiv 0 \quad \text{for} \quad x \in \mathbb{R}^I$$

for given vectors $b, c \in \mathbb{R}^I$ and matrices $B \in \mathbb{R}^{I\times I}$

In order to apply the Newton type algorithm for solving $F(x) = 0$, we consider the following algorithm:

- iterate over $k \geq 0$ (for a given x^0 starting point of $\mathbb{R}^I$, to be chosen)

$$x^{k+1} = x^k - F'\left(x^k\right)^{-1} F\left(x^k\right)$$

  until $F\left(x^k\right) = 0$ (that is $x^{k+1} = x^k$ i.e until convergence).

In this iteration, $F(x^k) := \min(Bx^k - b, x^k - c)$ and we will take the following definition for a square matrix $F'\left(x^k\right)$:

$$F'\left(x^k\right)_{i,j} := \begin{cases} B_{i,j} & \text{if } \left(Bx^k - b\right)_i \leq \left(x^k - c\right)_i \\ \delta_{i,j} & \text{otherwise} \end{cases}$$

(Note the specific choice $F'\left(x^k\right)_{i,j} = B_{i,j}$ even in the case when $\left(Bx^k - b\right)_i = \left(x^k - c\right)_i$. The other choice $F'\left(x^k\right)_{i,j} = \delta_{i,j}$ if $\left(Bx^k - b\right)_i = \left(x^k - c\right)_i$ works also but may be less efficient : i.e more iterations might be needed).

We have implemented (in python) the program for this scheme, as shown in the section Annex 8. We plot some graphs as shown in Fig. 20 below using this EI scheme for volatility model with the value of $(I, N) = (100, 100)$, $(I, N) = (400, 400)$, $(I, N) = (80, 8)$ and $(I, N) = (400, 40)$. Indeed, we notice that this EI scheme for volatility model is unconditionally stable with respect to the mesh parameters.

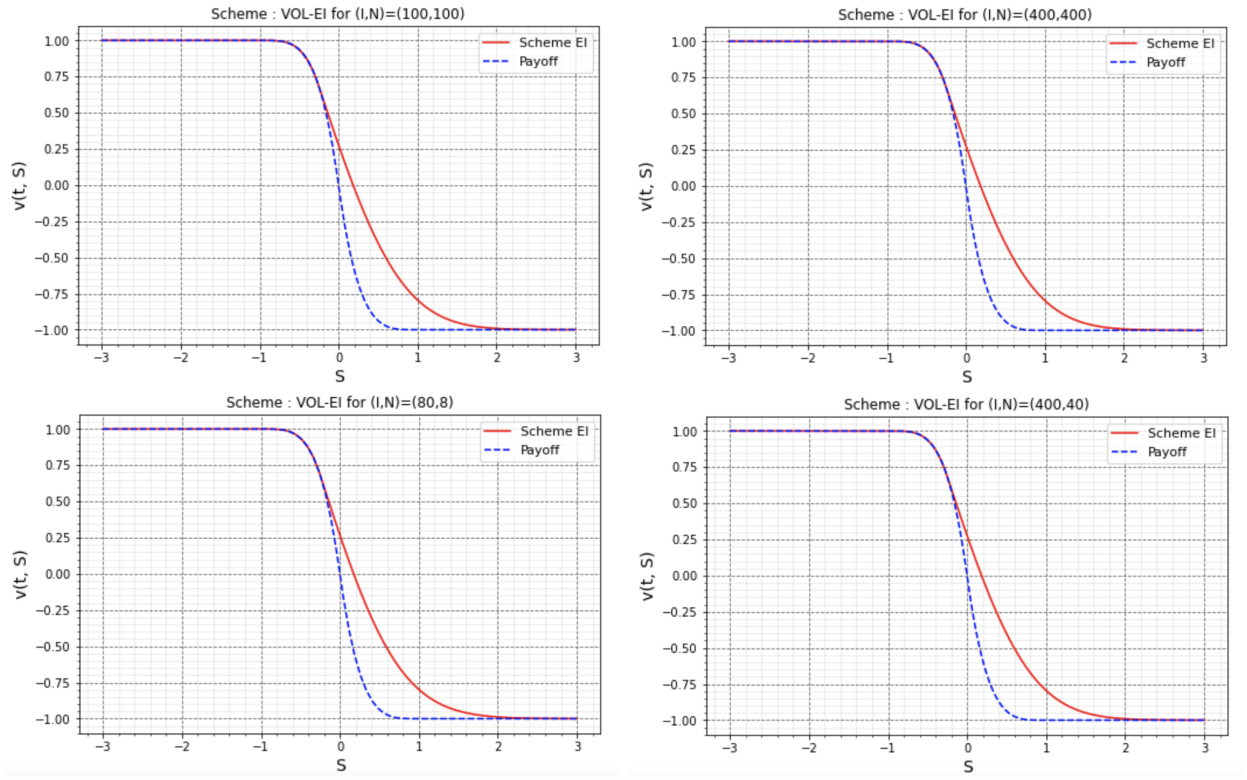

Figure 20: Graph of the uncertain volatility model using Euler Implicit scheme. (a) For $N = I = 100$. (b) For $I = N = 400$. (c) For $I = 80, N = 8$. (d) For $I = 400, N = 40$.

3.3 Example 2: Butterfly option pricing with uncertain volatility

We want to program the uncertain volatility model of the **Example 4.2, Section 4** of [Bokanowski et al., 2018].

We consider the price of a financial option under a Black-Scholes model with uncertain volatility, with the worst scenario for a "butterfly spread" with respect to all the possible values of the volatility σ in the interval $[\sigma_{\min}, \sigma_{\max}]$. This problem is associated with the following PDE in $\mathbb{R}_+ \times (0,T)$ for some expiry $T > 0$:

$$v_t + \sup_{\sigma \in \{\sigma_{\min}, \sigma_{\max}\}} \left(-\frac{\sigma^2}{2} x^2 v_{xx} \right) - r x v_x + r v = 0 \tag{14a}$$

$$v(0,x) = v_0(x) \tag{14b}$$

where

$$v_0(x) := \max(x - K_1, 0) - 2\max(x - (K_1 + K_2)/2, 0) + \max(x - K_2, 0)$$

After a time reversal, and imposing boundary conditions, we are led to the following second order HJB equation:

$$\begin{aligned} &- v_t(t,x) + \max\left(-\frac{1}{2}\sigma_{min}^2 x^2 v_{xx}(t,x); -\frac{1}{2}\sigma_{max}^2 x^2 v_{xx}(t,x) \right) \\ &\qquad - r x v_x(t,x) + r v(t,x) = 0, \quad t \in (0,T), \quad x \in (S_{\min}, S_{\max}) \end{aligned} \tag{15a}$$

$$v(0,x) = v_0(x) \quad x \in (S_{\min}, S_{\max}) \tag{15b}$$

with the following boundary conditions:

$$v(t, S_{\min}) = v_\ell \quad t \in (0,T) \tag{16a}$$

$$v(t, S_{\max}) = v_r \quad t \in (0,T) \tag{16b}$$

We will consider the following parameters:

$$S_{\min} = 50, S_{\max} = 150, K_1 = 90, K_2 = 110, r = 0.1, \sigma_{min} = 0.15, \sigma_{max} = 0.25 \quad \text{and} \quad T = 0.1$$

with smooth initial data

$$v_0(x) = \max(x - K_1, 0) - 2\max(x - (K_1 + K_2)/2, 0) + \max(x - K_2, 0)$$

and boundary conditions compatible the initial data: $v_l = 0$ and $v_r = 0$.

3.3.1 First order scheme: Euler Implicit (EI) scheme

We introduce the second order approximation of $-u_{xx}$ where :

$$D^2 U_i^{n+1} = \frac{U_{i+1}^{n+1} - 2U_i^{n+1} + U_{i-1}^{n+1}}{h^2}$$

Thus, a possible Euler Implicit (EI) scheme for 15 and 16 is given by:

$$\frac{U_j^{n+1}-U_j^n}{\Delta t}+\max\left(-\frac{1}{2}\sigma_{min}^2 s_j^2 D^2 U_j^{n+1};-\frac{1}{2}\sigma_{max}^2 s_j^2 D^2 U_j^{n+1}\right)$$
$$-rs_j\frac{U_{j+1}^{n+1}-U_{j-1}^{n+1}}{2h}+rU_j^{n+1}=0 \tag{17a}$$
$$n=0,\dots,N-1;\quad j=1,\dots,I$$
$$U_0^n=v_\ell(t),\quad U_{I+1}^n=v_r(t)\quad n=0,\dots,N \tag{17b}$$
$$U_j^0=v_0(x_j)\quad j=1,\dots,I \tag{17c}$$

The scheme can be written in vector form as follows:

$$\max_{\pm}\left(\frac{U^{n+1}-U^n}{\Delta t}+A^{\pm}U^{n+1}+q^{\pm}(t_{n+1})\right)=0,\quad \text{in } \mathbb{R}^I$$

where $q^{\pm}$ are vectors taking into account the boundary conditions (here for the chosen boundary conditions we have $q^{\pm}\equiv 0$, so we won't have any problems with that). Notice that $A^{\pm}$ is a square matrix of dimension $(I\times I)$ (in terms of the parameter $\sigma=\{\sigma_{min},\sigma_{max}\}$) where:

$$A^+=A(\sigma_{max}),\quad A^-=A(\sigma_{min})$$

Let us denote:

$$\alpha_j:=\frac{\sigma^2}{2}\frac{s_j^2}{h^2},\quad \beta_j:=r\frac{s_j}{2h}.$$

By identification we see that A is a tridiagonal matrix where :

$$A=\begin{bmatrix}
2\alpha_1+r & -\alpha_1-\beta_1 & 0 & \dots & \dots & 0 & 0\\
-\alpha_2+\beta_2 & 2\alpha_2+r & -\alpha_2-\beta_2 & 0 & \dots & 0 & 0\\
0 & \ddots & \ddots & \ddots & \ddots & & \vdots\\
\vdots & \ddots & -\alpha_i+\beta_i & 2\alpha_i+r & -\alpha_i-\beta_i & \ddots & \vdots\\
\vdots & & \ddots & \ddots & \ddots & \ddots & 0\\
0 & 0 & \dots & \ddots & \ddots & \ddots & -\alpha_{I-1}-\beta_{I-1}\\
0 & 0 & \dots & \dots & 0 & -\alpha_I+\beta_I & 2\alpha_I+r
\end{bmatrix}$$

We denote:

$$B^{\pm}:=I+\Delta t A^{\pm},\quad b^{\pm}:=U^n-\Delta t q^{\pm}$$

Notice that the scheme is also equivalent to find $x=U^{n+1}\in\mathbb{R}^I$ solution of

$$\max_{\pm}\left(B^{\pm}x-b^{\pm}\right)=0,\quad \text{in}\quad \mathbb{R}^I$$

or, more precisely:

$$\max\left(B^+x-b^+,B^-x-b^-\right)=0,\quad \text{in}\quad \mathbb{R}^I$$

In the python code, we program this scheme by implementing a **("semi-smooth")** Newton's method for finding $x\in\mathbb{R}^I$ solution of

$$F(x)=\max(Bx-b,Cx-c)\equiv 0\quad \text{for}\quad x\in\mathbb{R}^I$$

for given vectors $b, c \in \mathbb{R}^I$ and matrices $B, C \in \mathbb{R}^{I \times I}$

In order to apply a Newton type algorithm for solving $F(x) = 0$, we consider the following algorithm:

- iterate over $k \geq 0$ (for a given x^0 starting point of $\mathbb{R}^I$, to be chosen)

$$x^{k+1} = x^k - F'\left(x^k\right)^{-1} F\left(x^k\right)$$

  until $F\left(x^k\right) = 0$ (or, that $x^{k+1} = x^k$ i.e until convergence).

In this iteration, $F(x^k) := \max(Bx^k - b, Cx^k - c)$ and we will take the following definition for a square matrix $F'\left(x^k\right)$ (row by row derivative):

$$F'\left(x^k\right)_{i,j} := \begin{cases} B_{i,j} & \text{if } \left(Bx^k - b\right)_i \geq \left(Cx^k - c\right)_i \\ C_{i,j} & \text{otherwise} \end{cases}$$

We program this scheme using python and the implementation is given in the section Annex 9. We plot some graphs as shown in Fig. 21 below using this EI scheme for butterfly option pricing with uncertain volatility model with the value of $(I, N) = (400, 400)$ and $(I, N) = (80, 8)$. Indeed, we notice that this EI scheme for volatility model is unconditionally stable with respect to the mesh parameters.

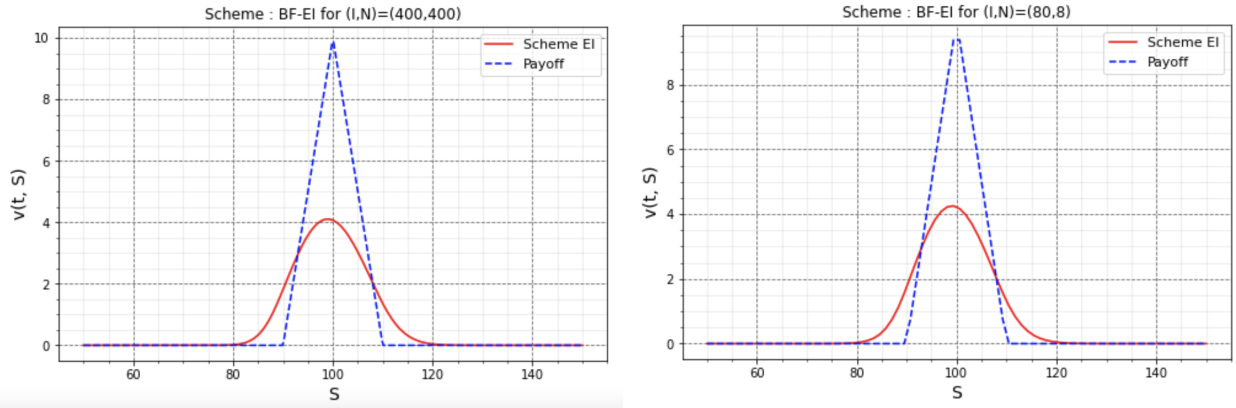

Figure 21: Graph of the Butterfly option pricing with uncertain volatility model using Euler Implicit scheme. (a) For $N = I = 400$. (b) For $I = 80, N = 8$.

In order to estimate the error, several methods are possible. Here we simply estimate the difference between to successive computations, let's say $U^{(k-1)}$ and $U^{(k)}$ (obtained with mesh (I_{k-1}, N_{k-1}) and (I_k, N_k), resp.), so that $error_k = U^{(k-1)} - U^{(k)}$ (and e_1 is not defined). The table 22 and 23 bellow illustrate the error of this scheme when $N = I$ and $N = I/10$ respectively with the value of $S_{val} = 100$.

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	10	3.817557	NaN	NaN	NaN	0.004031
1	20	20	4.090095	0.272538	NaN	NaN	0.008163
2	40	40	4.084462	-0.005633	5.596477	5.798048	0.030108
3	80	80	4.079545	-0.004917	0.196003	0.199535	0.075858
4	160	160	4.073923	-0.005621	-0.193083	-0.194824	0.449421
5	320	320	4.070611	-0.003312	0.763281	0.766721	1.973947
6	640	640	4.068823	-0.001789	0.888626	0.890629	13.683525

Figure 22: Error Table of EI scheme for $N = I$

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	1	4.035820	NaN	NaN	NaN	0.003111
1	20	2	4.458703	0.422883	NaN	NaN	0.001827
2	40	4	4.324904	-0.133799	1.660191	1.719987	0.003858
3	80	8	4.211041	-0.113863	0.232773	0.236968	0.010316
4	160	16	4.141649	-0.069392	0.714445	0.720885	0.048173
5	320	32	4.104878	-0.036771	0.916215	0.920345	0.272313
6	640	64	4.086052	-0.018826	0.965819	0.967996	2.088023

Figure 23: Error Table of EI scheme for $N = I/10$

3.3.2 Second order scheme: BDF2

We focus on this part, a second order Backward Difference Formula (BDF2). We introduce the second order approximation of $-u_{xx}$ where :

$$D^2 U_i^{n+1} = \frac{U_{i+1}^{n+1} - 2U_i^{n+1} + U_{i-1}^{n+1}}{h^2}$$

Thus, a possible second order Implicit BDF2 scheme for 15 and 16 is given by:

$$\frac{3U_j^{n+1} - 4U_j^n + U_j^{n-1}}{2\Delta t} + \max\left(-\frac{1}{2}\sigma_{min}^2 s_j^2 D^2 U_j^{n+1}; -\frac{1}{2}\sigma_{max}^2 s_j^2 D^2 U_j^{n+1}\right)$$

$$-r s_j \tilde{D} U_j^{n+1} + r U_j^{n+1} = 0 \qquad (18a)$$

$$n = 0, \ldots, N-1; \quad j = 1, \ldots, I$$

$$U_0^n = v_\ell(t), \quad U_{I+1}^n = v_r(t) \quad n = 0, \ldots, N \qquad (18b)$$

$$U_j^0 = v_0(x_j) \quad j = 1, \ldots, I \qquad (18c)$$

where:

$$\tilde{D}U_j^{n+1} = \frac{U_{j+1}^{n+1} - U_{j-1}^{n+1}}{2h}$$

The scheme can be written in vector form as follows:

$$\max_{\pm}\left(\frac{3U^{n+1} - 4U^n + U^{n-1}}{2\Delta t} + A^{\pm}U^{n+1} + q^{\pm}\left(t_{n+1}\right)\right) = 0, \quad \text{in } \mathbb{R}^I$$

where $q^{\pm}$ are vectors taking into account the boundary conditions (here for the chosen boundary conditions we have $q^{\pm} \equiv 0$, so we won't have any problems with that). Notice that $A^{\pm}$ is a square matrix of dimension $(I \times I)$ with the same definition as in the previous EI scheme.

We now can write the scheme in vector form as follows :

$$\max_{\pm}\left[\left(3I_d + 2\Delta t A^{\pm}\right)U^{n+1} + 2\Delta t q^{\pm}(t_{n+1}) - 4U^n + U^{n-1}\right] = 0$$

We denote then:

$$B^{\pm} := I + \Delta t A^{\pm}, \quad b^{\pm} := U^n - \Delta t q^{\pm}$$

Notice that the scheme is also equivalent to find $x = U^{n+1} \in \mathbb{R}^I$ solution of

$$\max_{\pm}\left(B^{\pm}x - b^{\pm}\right) = 0, \quad \text{in} \quad \mathbb{R}^I$$

or, more precisely:

$$\max\left(B^{+}x - b^{+}, B^{-}x - b^{-}\right) = 0, \quad \text{in} \quad \mathbb{R}^I$$

In the python code, we program this scheme by implementing a **("semi-smooth")** Newton's method for finding $x \in \mathbb{R}^I$ solution of

$$F(x) = \max(Bx - b, Cx - c) \equiv 0 \quad \text{for} \quad x \in \mathbb{R}^I$$

for given vectors $b, c \in \mathbb{R}^I$ and matrices $B, C \in \mathbb{R}^{I \times I}$ (the same way as the previous EI scheme).

Note: In order to compute the first step U_1 , an implicit EI step can be used.

We program this scheme using python and the implementation is given in the section Annex 10. We plot some graphs as shown in Fig. 24 below using this second order BDF2 scheme for butterfly option pricing with uncertain volatility model with the value of $(I, N) = (400, 400)$ and $(I, N) = (80, 8)$. Indeed, we notice that this scheme is unconditionally stable with respect to the mesh parameters.

In order to estimate the error, we simply estimate the difference between to successive computations, let's say $U^{(k-1)}$ and $U^{(k)}$ (obtained with mesh (I_{k-1}, N_{k-1}) and (I_k, N_k), resp.), so that $error_k = U^{(k-1)} - U^{(k)}$ (and e_1 is not defined). The table 25 and 26 bellow illustrate the error of this scheme when $N = I$ and $N = I/10$ respectively with the value of $S_{val} = 100$.

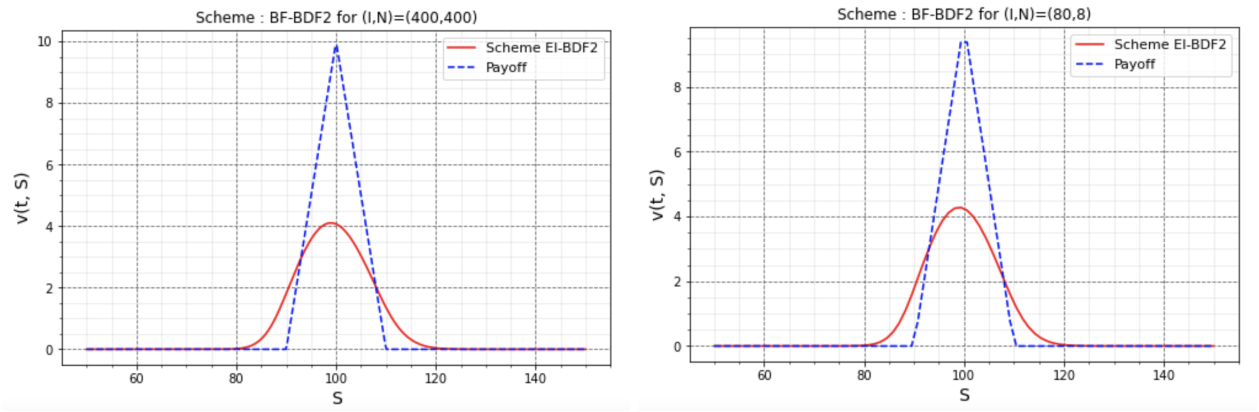

Figure 24: Graph of the Butterfly option pricing with uncertain volatility model using Implicit second order (BDF2) scheme. (a) For $N = I = 400$. (b) For $I = 80, N = 8$.

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	10	3.858052	NaN	NaN	NaN	0.013716
1	20	20	4.093780	0.235728	NaN	NaN	0.009090
2	40	40	4.083195	-0.010585	4.477002	4.638253	0.029513
3	80	80	4.078000	-0.005194	1.027029	1.045537	0.087404
4	160	160	4.073026	-0.004975	0.062343	0.062905	0.490489
5	320	320	4.070137	-0.002888	0.784403	0.787939	1.957636
6	640	640	4.068580	-0.001558	0.890905	0.892913	17.650732

Figure 25: Error Table of second order BDF2 scheme for $N = I$

	I	N	U(s)	error	order alpha time	order alpha space	tcpu
0	10	1	4.035820	NaN	NaN	NaN	0.000810
1	20	2	4.759163	0.723343	NaN	NaN	0.001749
2	40	4	4.475658	-0.283505	1.351305	1.399976	0.004208
3	80	8	4.233923	-0.241735	0.229949	0.234093	0.013328
4	160	16	4.141165	-0.092759	1.381872	1.394328	0.059418
5	320	32	4.102172	-0.038992	1.250299	1.255934	0.289880
6	640	64	4.084127	-0.018046	1.111529	1.114035	1.573058

Figure 26: Error Table of second order BDF2 scheme for $N = I/10$

We observe that the second order implicit (BDF2) scheme gives a better results (in terms of errors that we defined earlier) compared to the previous first order EI scheme.

4 Conclusion

The aim of our project was to test different schemes for HJB equations. The first part concerned first order HJB equations, related to deterministic control, where it is more easy to study the numerical schemes, the stability and the monotonicity of the schemes. The second part concerns the approximation of a second order HJB equation for the uncertain volatility model.

- Concerning the first order HJB equations : as usual and expected the Euler Explicit (EE) scheme for first order HJB equation is stable and monotone for an appropriate (sufficient) CFL condition. The consistency error can be improved by using the left-sided three points schemes for approximating the derivative with respect to the space variable. However this makes us lose the monotony of the scheme.

  It is a scheme of order $(1, 1)$ in time and in space, but by introducing an RK2 scheme, we obtain an numerical order of $(2, 2)$ for the Euler Explicit scheme. However it is still subjected to some CFL condition for stability.

  To avoid that, we can use the Implicit Euler (EI) scheme which is still of order 1 both in time and space, but does not suffer from some CFL conditions. In order to improve the order of the scheme we have again used left-sided three points schemes for approximating the derivative (with respect to the time and space variables). So we are finally able to get a scheme of order $(2, 2)$, which is not subjected to some CFL conditions for stability.

- Concerning the approximation of a second order HJB equation for the uncertain volatility model : the same conclusion can be done, indeed the Euler Explicit scheme is not stable for time mesh step smaller than space mesh step, contrary to implicit scheme (again with some CFL conditions).

  Like the first order HJB equation, we can improve the order and the consistency error by using the left-sided three points schemes for approximating the derivative with respect to time.

References

[Bokanowski et al., 2018] Bokanowski, O., Picarelli, A., and Reisinger, C. (2018). High-order filtered schemes for time-dependent second order hjb equations. *ESAIM: Mathematical Modelling and Numerical Analysis*, 52(1):69–97.

[Mariani et al., 2016] Mariani, A., Nugrahani, E. H., and Lesmana, D. C. (2016). Numerical solution for option pricing with stochastic volatility model. In *IOP Conference Series: Earth and Environmental Science*, volume 31, page 012024. IOP Publishing.

A Annex : Python code used in this numerical project

```
def Euler_EE(c, T, Smin, Smax, I, N):

    h = (Smax - Smin)/(I+1)
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 0.0

    def vright(t):
        return 0.0

    # D_
    def get_D1():
        D = np.zeros((I,I))
        for i in range(0, I):
            D[i, i] = c/h
        for i in range(1, I):
            D[i, i-1] = -c/h

        return sparse(D)

    # D+
    def get_D2():
        D = np.zeros((I,I))
        for i in range(0, I):
            D[i, i] = -c/h
        for i in range(0, I-1):
            D[i, i+1] = c/h

        return sparse(D)


    # q_
    def get_Q1(t):
        Q = np.zeros((I,1))
        Q[0] = -(c/h)*vleft(t)

        return Q

    # q+
    def get_Q2(t):
        Q = np.zeros((I,1))
        Q[I-1] = (c/h)*vright(t)

        return Q

    def v0(x):
        res = -(np.maximum(1 - abs(x)**2, 0))**2
        return res.reshape(I,1)

    #init
```

```
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
        t = n*dt
        U = U - dt*np.maximum(get_D1()@U + get_Q1(t), -(get_D2()@U +
    get_Q2(t)))

    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 1: Euler Explicit (EE) code

```
def Euler_EE_not_working(c, T, Smin, Smax, I, N):

    h = (Smax - Smin)/(I+1)
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 0.0

    def vright(t):
        return 0.0

    # A'
    def get_A():
        A = np.zeros((I,I))
        for i in range(0, I-1):
            A[i, i+1] = 1/(2*h)
        for i in range(1, I):
            A[i, i-1] = -1/(2*h)

        return A


    # q'
    def get_Q(t):
        Q = np.zeros((I,1))
        Q[0] = (1/(2*h)) * vleft(t)
        Q[I-1] = (1/(2*h)) * vright(t)
        return Q

    def v0(x):
        res = -(np.maximum(1 - abs(x)**2, 0))**2
        return res.reshape(I,1)

    #init
    U = v0(s)
    Id = np.eye(I)
```

```
38
39      # main loop
40      for n in range(0, N):
41          t = n*dt
42          U = U - dt*c*np.absolute(get_A()@U + get_Q(t))
43
44      U = np.insert(U, 0, vleft(T))
45      U = np.insert(U, U.size, vright(T))
46
47      return U
```

Listing 2: Euler Explicit (EE) (not working) code

```
1  def Euler_EE_modified(c, T, Smin, Smax, I, N):
2
3      h = (Smax - Smin)/(I+1)
4      s = Smin + h*np.arange(1, (I+1))
5      dt = T/N
6
7      def vleft(t):
8          return 0.0
9
10     def vright(t):
11         return 0.0
12
13     # D_ tilde
14     def get_D1():
15         D = np.zeros((I,I))
16         for i in range(0, I):
17             D[i, i] = 3/(2*h)
18         for i in range(1, I):
19             D[i, i-1] = -4/(2*h)
20         for i in range(2, I):
21             D[i, i-2] = 1/(2*h)
22
23         return D
24
25     # D+ tilde
26     def get_D2():
27         D = np.zeros((I,I))
28         for i in range(0, I):
29             D[i, i] = -3/(2*h)
30         for i in range(0, I-1):
31             D[i, i+1] = 4/(2*h)
32         for i in range(0, I-2):
33             D[i, i+2] = -1/(2*h)
34
35         return D
36
37
38     # q_ tilde
39     def get_Q1(t):
40         Q = np.zeros((I,1))
```

```
        Q[0] = (-4/(2*h))*vleft(t)
        Q[1] = (1/(2*h))*vleft(t)

        return Q

    # q+ tilde
    def get_Q2(t):
        Q = np.zeros((I,1))
        Q[I-1] = (4/(2*h))*vright(t)
        Q[I-2] = (-1/(2*h))*vright(t)

        return Q

    def v0(x):
        res = -(np.maximum(1 - abs(x)**2, 0))**2
        return res.reshape(I,1)

    #init
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
        t = n*dt
        U = U - dt*np.maximum(c*get_D1()@U + get_Q1(t), -c*(get_D2()@U +
   get_Q2(t)))

    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 3: Euler Explicit (EE) modified code

```
def RK2(c, T, Smin, Smax, I, N):

    h = (Smax - Smin)/(I+1)
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 0.0

    def vright(t):
        return 0.0

    # D_ tilde
    def get_D1():
        D = np.zeros((I,I))
        for i in range(0, I):
            D[i, i] = 3/(2*h)
        for i in range(1, I):
            D[i, i-1] = -4/(2*h)
```

```
        for i in range(2, I):
            D[i, i-2] = 1/(2*h)

        return D

    # D+ tilde
    def get_D2():
        D = np.zeros((I,I))
        for i in range(0, I):
            D[i, i] = -3/(2*h)
        for i in range(0, I-1):
            D[i, i+1] = 4/(2*h)
        for i in range(0, I-2):
            D[i, i+2] = -1/(2*h)

        return D


    # q_ tilde
    def get_Q1(t):
        Q = np.zeros((I,1))
        Q[0] = (-4/(2*h))*vleft(t)
        Q[1] = (1/(2*h))*vleft(t)

        return Q

    # q+ tilde
    def get_Q2(t):
        Q = np.zeros((I,1))
        Q[I-1] = (4/(2*h))*vright(t)
        Q[I-2] = (-1/(2*h))*vright(t)

        return Q

    def v0(x):
        res = -(np.maximum(1 - abs(x)**2, 0))**2
        return res.reshape(I,1)

    def S_one(t, U):
        """
            U : matrix
        """
        return U - dt*np.maximum(c*get_D1()@U + get_Q1(t), -c*(get_D2()@U
    + get_Q2(t)))

    #init
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
        t = n*dt
        U = 0.5*(U + S_one(t, S_one(t, U)))

```

```
    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 4: Euler Explicit (EE) RK2 variant code

```
def Euler_EI(c, T, Smin, Smax, I, N, v0_init="v0_1", eta=1e-5, kmax=100,
    show_iter=False):

    h = (Smax - Smin)/(I+1)
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 0.0

    def vright(t):
        return 0.0

    # the matrix A_
    def get_A1():
        A = np.zeros((I,I))
        for i in range(0, I):
            A[i, i] = c/h
        for i in range(1, I):
            A[i, i-1] = -c/h

        return A

    # the matrix A+
    def get_A2():
        A = np.zeros((I,I))
        for i in range(0, I):
            A[i, i] = c/h
        for i in range(0, I-1):
            A[i, i+1] = -c/h

        return A


    # the vector function q_
    def get_Q1(t):
        Q = np.zeros((I,1))
        Q[0] = -(c/h)*vleft(t)

        return Q

    # the vector function q+
    def get_Q2(t):
        Q = np.zeros((I,1))
        Q[I-1] = (c/h)*vright(t)

```

```
        return Q

    def v0(x):
        if (v0_init == "v0_1"):
            return (-(np.maximum(1 - abs(x)**2, 0))**2).reshape(I,1)
        elif (v0_init == "v0_2"):
            return ((np.maximum(1 - abs(x)**2, 0))**2).reshape(I,1)
        else:
            print("Function v0 is not well defined!")

    # initialization
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
        t = n*dt
        t1 = (n+1)*dt
        B1 = Id + dt*get_A1()
        B2 = Id + dt*get_A2()
        b1 = U - dt*get_Q1(t1)
        b2 = U - dt*get_Q2(t1)

        x = U.copy()
        xold = np.zeros((I,1))
        k = 0
        err = 1 # we start with an error >> 1e-5, so we can choose error=1

        while (err > eta and k < kmax):
            xold = x.copy()
            F = np.maximum(B1 @ x - b1, B2 @ x - b2)
            Fp = B2.copy()
            i, j = np.where((B1 @ x - b1) >= (B2 @ x - b2))
            Fp[i,:] = B1[i,:]
            x = x - lng.solve(Fp, F)

            err = lng.norm(x - xold, np.inf)
            if (show_iter):
                print("k=%s, err=%10.8f" %(k, err))
            k = k+1
        U = x

    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 5: Euler Implicit (EI) code

```
def Euler_EI_variant_second_order(c, T, Smin, Smax, I, N, v0_init="v0_1",
    eta=1e-5, kmax=100, show_iter=False):

    h = (Smax - Smin)/(I+1)
```

```
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 0.0

    def vright(t):
        return 0.0

    def v0(x):
        if (v0_init == "v0_1"):
            return (-(np.maximum(1 - abs(x)**2, 0))**2).reshape(I,1)
        elif (v0_init == "v0_2"):
            return ((np.maximum(1 - abs(x)**2, 0))**2).reshape(I,1)
        else:
            print("Function v0 is not well defined!")

    # the matrix A_
    def get_A1():
        A = np.zeros((I,I))
        for i in range(0, I):
            A[i, i] = c/h
        for i in range(1, I):
            A[i, i-1] = -c/h

        return A

    # the matrix A+
    def get_A2():
        A = np.zeros((I,I))
        for i in range(0, I):
            A[i, i] = c/h
        for i in range(0, I-1):
            A[i, i+1] = -c/h

        return A


    # the vector function q_
    def get_Q1(t):
        Q = np.zeros((I,1))
        Q[0] = -(c/h)*vleft(t)

        return Q

    # the vector function q+
    def get_Q2(t):
        Q = np.zeros((I,1))
        Q[I-1] = (c/h)*vright(t)

        return Q


    # the matrix A_ tilde
```

```
    def get_A1_():
        A = np.zeros((I,I))
        for i in range(0, I):
            A[i, i] = (c*3)/(2*h)
        for i in range(1, I):
            A[i, i-1] = -(c*4)/(2*h)
        for i in range(2, I):
            A[i, i-2] = c/(2*h)

        return A

    # the matrix A+ tilde
    def get_A2_():
        A = np.zeros((I,I))
        for i in range(0, I):
            A[i, i] = (c*3)/(2*h)
        for i in range(0, I-1):
            A[i, i+1] = -(c*4)/(2*h)
        for i in range(0, I-2):
            A[i, i+2] = c/(2*h)

        return A

    # the vector function q_ chapeau
    def get_Q1_(t):
        Q = np.zeros((I,1))
        Q[0] = (-3*c/(2*h))*vleft(t)
        Q[1] = (c/(2*h))*vleft(t)

        return Q

    # the vector function q+ chapeau
    def get_Q2_(t):
        Q = np.zeros((I,1))
        Q[I-1] = (-3*c/(2*h))*vright(t)
        Q[I-2] = (c/(2*h))*vright(t)

        return Q


    #init
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
        t = n*dt
        t1 = (n+1)*dt

        if(n==0) :
            B1 = Id + dt*get_A1()
            B2 = Id + dt*get_A2()
            b1 = U - dt*get_Q1(t1)
            b2 = U - dt*get_Q2(t1)
```

```
112
113             x = U.copy()
114             xold = np.zeros((I,1))
115             k = 0
116             err = 1 # we start with an error >> 1e-5, so we can choose
    error=1
117
118             while (err > eta and k < kmax):
119                 xold = x.copy()
120                 F = np.maximum(B1 @ x - b1, B2 @ x - b2)
121                 Fp = B2.copy()
122                 i, j = np.where((B1 @ x - b1) >= (B2 @ x - b2))
123                 Fp[i,:] = B1[i,:]
124                 x = x - lng.solve(Fp, F)
125
126                 err = lng.norm(x - xold, np.inf)
127                 if (show_iter):
128                     print("k=%s, err=%10.8f" %(k, err))
129                 k = k+1
130             U = x.copy()
131             U_moins = U.copy()
132
133         else:
134             # Utilisation of the left-sided three points finite difference
     scheme :
135
136             B1_ = 3*Id + 2*dt*get_A1_()
137             B2_ = 3*Id + 2*dt*get_A2_()
138             b1_ = 4*U - U_moins - 2*dt*get_Q1_(t1)
139             b2_ = 4*U - U_moins - 2*dt*get_Q2_(t1)
140
141             x_ = U.copy()
142             U_inter = U.copy()
143             xold_ = np.zeros((I,1))
144
145             k_ = 0
146             err_ = 1 # we start with an error >> 1e-5, so we can choose
    error=1
147
148             while (err_ > eta and k_ < kmax):
149                 xold_ = x_.copy()
150                 F = np.maximum(B1_ @ x_ - b1_, B2_ @ x_ - b2_)
151                 Fp = B2_.copy()
152                 i, j = np.where((B1_ @ x_ - b1_) >= (B2_ @ x_ - b2_))
153                 Fp[i,:] = B1_[i,:]
154                 x_ = x_ - lng.solve(Fp, F)
155
156                 err_ = lng.norm(x_ - xold_, np.inf)
157                 if (show_iter):
158                     print("k=%s, err=%10.8f" %(k_, err_))
159                 k_ = k_+1
160             U = x_.copy()
161             U_moins = U_inter.copy()
162
```

```
    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 6: Euler Implicit second order (EI2) code

```
def Euler_EE_volatility_model(sigma, T, Smin, Smax, I, N):

    h = (Smax - Smin)/(I+1)
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 1.0

    def vright(t):
        return -1.0

    # the matrix D
    def get_D():
        D = np.zeros((I,I))
        for i in range(1, I):
            D[i, i-1] = 1/(h**2)
        for i in range(0, I):
            D[i, i] = -2/(h**2)
        for i in range(0, I-1):
            D[i, i+1] = 1/(h**2)

        return sparse(D)


    # the vector function q(t)
    def get_Q(t):
        Q = np.zeros((I,1))
        Q[0] = (1/(h**2))*vleft(t)
        Q[I-1] = (1/(h**2))*vright(t)

        return Q

    def v0(x):
        res = np.sign(x) * ((np.maximum(1 - abs(x), 0))**4 - 1)
        return res.reshape(I,1)

    # initialization
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
        t = n*dt
        U = U - dt*np.minimum(0, -0.5*(sigma**2)*(get_D()@U + get_Q(t)))

```

```
    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 7: Euler Explicit (EE) for volatility model code

```
def Euler_EI_volatility_model(sigma, T, Smin, Smax, I, N):

    eta=1e-5
    kmax=100
    show_iter=False
    h = (Smax - Smin)/(I+1)
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 1.0

    def vright(t):
        return -1.0

    # the matrix A
    def get_A():
        A = np.zeros((I,I))
        for i in range(1, I):
            A[i, i-1] = -(sigma**2)/(2 * h**2)
        for i in range(0, I):
            A[i, i] = (sigma**2)/(h**2)
        for i in range(0, I-1):
            A[i, i+1] = -(sigma**2)/(2 * h**2)

        return A


    # the vector function q(t)
    def get_Q(t):
        Q = np.zeros((I,1))
        Q[0] = -(sigma**2)/(2 * h**2) * vleft(t)
        Q[I-1] = -(sigma**2)/(2 * h**2) * vright(t)

        return Q

    def v0(x):
        res = np.sign(x) * ((np.maximum(1 - abs(x), 0))**4 - 1)
        return res.reshape(I,1)

    # initialization
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
```

```
        t = n*dt
        t1 = (n+1)*dt
        B = Id + dt*get_A()
        b = U - dt*get_Q(t1)
        c = U.copy()

        x = U.copy()
        xold = np.zeros((I,1))
        k = 0
        err = 1 # we start with an error >> 1e-5, so we can choose error=1

        while (err > eta and k < kmax):
            xold = x.copy()
            F = np.minimum(B @ x - b, x - c)
            Fp = np.eye(I)
            i, j = np.where(B @ x - b <= x - c)
            Fp[i,:] = B[i,:]
            x = x - lng.solve(Fp, F)

            err = lng.norm(x - xold, np.inf)
            if (show_iter):
                print("k=%s, err=%10.8f" %(k, err))
            k = k+1
        U = x

    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 8: Euler Implicit (EI) for volatility model code

```
def Euler_EI_butterfly(r, sigma1, sigma2, K1, K2, T, Smin, Smax, I, N):

    eta=1e-5
    kmax=100
    show_iter=False
    h = (Smax - Smin)/(I+1)
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 0.0

    def vright(t):
        return 0.0

    # the matrix A
    def get_A(sigma):
        A = np.zeros((I,I))
        for i in range(0, I):
            A[i, i] = (sigma**2)*(s[i]**2)/(h**2) + r
        for i in range(0, I-1):
```

```
            A[i, i+1] = -(sigma**2)*(s[i]**2)/(2*h**2) - r*s[i]/(2*h)
        for i in range(1, I):
            A[i, i-1] = -(sigma**2)*(s[i]**2)/(2*h**2) + r*s[i]/(2*h)

        return A


    # the vector function q^{+}(t)
    def get_Q1(t):
        Q = np.zeros((I,1))
        Q[0] = (-(sigma1**2)*(s[0]**2)/(2*h**2) + r*s[0]/(2*h)) * vleft(t)
        Q[I-1] = (-(sigma1**2)*(s[I-1]**2)/(2*h**2) - r*s[I-1]/(2*h)) *
    vright(t)

        return Q

     # the vector function q^{-}(t)
    def get_Q2(t):
        Q = np.zeros((I,1))
        Q[0] = (-(sigma2**2)*(s[0]**2)/(2*h**2) + r*s[0]/(2*h)) * vleft(t)
        Q[I-1] = (-(sigma2**2)*(s[I-1]**2)/(2*h**2) - r*s[I-1]/(2*h)) *
    vright(t)

        return Q

    def v0(x):
        res = np.maximum(x - K1, 0) - 2*np.maximum(x - 0.5*(K1+K2), 0) +
    np.maximum(x - K2, 0)
        return res.reshape(I,1)

    # initialization
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
        t = n*dt
        t1 = (n+1)*dt
        B1 = Id + dt*get_A(sigma1)
        B2 = Id + dt*get_A(sigma2)

        b1 = U - dt*get_Q1(t1)
        b2 = U - dt*get_Q2(t1)

        x = U.copy()
        xold = np.zeros((I,1))
        k = 0
        err = 1 # we start with an error >> 1e-5, so we can choose error=1

        while (err > eta and k < kmax):
            xold = x.copy()
            F = np.maximum(B1 @ x - b1, B2 @ x - b2)
            Fp = B2.copy()
            i, j = np.where((B1 @ x - b1) >= (B2 @ x - b2))
```

```
            Fp[i,:] = B1[i,:]
            x = x - lng.solve(Fp, F)

            err = lng.norm(x - xold, np.inf)
            if (show_iter):
                print("k=%s, err=%10.8f" %(k, err))
            k = k+1
        U = x

    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 9: Euler Implicit (EI) for Butterfly option code

```
def Euler_BDF2_butterfly(r, sigma1, sigma2, K1, K2, T, Smin, Smax, I, N):

    eta=1e-5
    kmax=100
    show_iter=False
    h = (Smax - Smin)/(I+1)
    s = Smin + h*np.arange(1, (I+1))
    dt = T/N

    def vleft(t):
        return 0.0

    def vright(t):
        return 0.0

    # the matrix A
    def get_A(sigma):
        A = np.zeros((I,I))
        for i in range(0, I):
            A[i, i] = (sigma**2)*(s[i]**2)/(h**2) + r
        for i in range(0, I-1):
            A[i, i+1] = -(sigma**2)*(s[i]**2)/(2*h**2) - r*s[i]/(2*h)
        for i in range(1, I):
            A[i, i-1] = -(sigma**2)*(s[i]**2)/(2*h**2) + r*s[i]/(2*h)

        return A


    # the vector function q^{+}(t)
    def get_Q1(t):
        Q = np.zeros((I,1))
        Q[0] = (-(sigma1**2)*(s[0]**2)/(2*h**2) + r*s[0]/(2*h)) * vleft(t)
        Q[I-1] = (-(sigma1**2)*(s[I-1]**2)/(2*h**2) - r*s[I-1]/(2*h)) *
   vright(t)

        return Q

```

```
    # the vector function q^{-}(t)
    def get_Q2(t):
        Q = np.zeros((I,1))
        Q[0] = (-(sigma2**2)*(s[0]**2)/(2*h**2) + r*s[0]/(2*h)) * vleft(t)
        Q[I-1] = (-(sigma2**2)*(s[I-1]**2)/(2*h**2) - r*s[I-1]/(2*h)) *
    vright(t)

        return Q

    def v0(x):
        res = np.maximum(x - K1, 0) - 2*np.maximum(x - 0.5*(K1+K2), 0) +
    np.maximum(x - K2, 0)
        return res.reshape(I,1)

    # initialization
    U = v0(s)
    Id = np.eye(I)

    # main loop
    for n in range(0, N):
        t = n*dt
        t1 = (n+1)*dt

        if (n==0):
            B1 = Id + dt*get_A(sigma1)
            B2 = Id + dt*get_A(sigma2)
            b1 = U - dt*get_Q1(t1)
            b2 = U - dt*get_Q2(t1)

            x = U.copy()
            xold = np.zeros((I,1))
            k = 0
            err = 1 # we start with an error >> 1e-5, so we can choose
    error=1

            while (err > eta and k < kmax):
                xold = x.copy()
                F = np.maximum(B1 @ x - b1, B2 @ x - b2)
                Fp = B2.copy()
                i, j = np.where((B1 @ x - b1) >= (B2 @ x - b2))
                Fp[i,:] = B1[i,:]
                x = x - lng.solve(Fp, F)

                err = lng.norm(x - xold, np.inf)
                if (show_iter):
                    print("k=%s, err=%10.8f" %(k, err))
                k = k+1
            U = x
            U_moins = U.copy()

        else:

            B1_ = 3*Id + 2*dt*get_A(sigma1)
            B2_ = 3*Id + 2*dt*get_A(sigma2)
```

```
            b1_ = 4*U - U_moins - 2*dt*get_Q1(t1)
            b2_ = 4*U - U_moins - 2*dt*get_Q2(t1)

            x_ = U.copy()
            U_inter = U.copy()
            xold_ = np.zeros((I,1))

            k_ = 0
            err_ = 1 # we start with an error >> 1e-5, so we can choose
    error=1

            while (err_ > eta and k_ < kmax):
                xold_ = x_.copy()
                F = np.maximum(B1_ @ x_ - b1_, B2_ @ x_ - b2_)
                Fp = B2_.copy()
                i, j = np.where((B1_ @ x_ - b1_) >= (B2_ @ x_ - b2_))
                Fp[i,:] = B1_[i,:]
                x_ = x_ - lng.solve(Fp, F)

                err_ = lng.norm(x_ - xold_, np.inf)
                if (show_iter):
                    print("k=%s, err=%10.8f" %(k_, err_))
                k_ = k_+1
            U = x_.copy()
            U_moins = U_inter.copy()

    U = np.insert(U, 0, vleft(T))
    U = np.insert(U, U.size, vright(T))

    return U
```

Listing 10: Implicit second order scheme (BDF2) for Butterfly option code