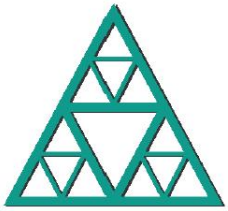
ECOLE NATIONALE DES PONTS ET CHAUSSEES



PROJET D'INITIATION A LA RECHERCHE 2018

Réduction de modèles appliquée à la modélisation du transport de polluants

Léo Baty, Clément Lasuen, Chiheb Eddine Najjar,
Nathan Godey, Régis Santet, Song Phuc Duong
sous la direction de
Damiano Lombardi, Sébastien Boyaval
Laboratoire INRIA

Table des matières

ABSTRACT : The ability of predicting the behaviour of a pollutant in a fluid is crucial in several realistic applications, such as when a drilling rig explodes undersea (*Deepwater Horizon* explosion in 2010 for example). Despite the increasing available computational power, the number of the parameters in the system and the number of degrees of freedom needed to approximate its solution make the prediction task prohibitively expensive from a computational point of view.. We have to use reduction models such as POD (Proper Orthogonal Decomposition) in order to compute the solution in a reasonable time. It involves keeping the major dynamic of the solution with a few degrees of freedom and accepting an error between the approximation and the real solution. We use the advection equation to simulate a pollution peak represented by a Gaussian function in a fluid where the fluid speed varies in a closed region. After computing a couple of solutions using our initial integration schemes for some parameters like the finite volumes scheme and explicit Euler scheme to simulate the fundamental dynamic, we use the POD algorithm to compute approximate solutions for a new set of parameters, this time in a much shorter period. We decided to approach the problem with two different methods. The Eulerian method, in which we consider the velocity as a function of space and time, and try to compute the concentration of the pollutant as a function of the same variables. Whereas in the Lagrangian method, we follow some particles arraigned initially in a random way according to a probability law adapted to the initial concentration and compute their trajectory as a function of time and the initial location. Thus, we will try to compare both methods to see which one suits the problem better.

KEYWORDS : Mathematical Models, Numerical Simulations, Computational Complexity, Model Order Reduction, Proper Orthogonal Decomposition, Singular Value Decomposition

1 Introduction

Le but de ce projet d'initiation à la recherche est d'observer numériquement l'évolution d'une concentration de polluants (fluide que l'on considère incompressible) dans un liquide (rejets dans une rivière) ou dans un gaz (nuage radioactif). Ce projet est associé au cours intitulé "Méthodes numériques pour les problèmes en grandes dimensions" [1] tenu en première année à l'École nationale des ponts et chaussées par Damiano Lombardi (INRIA) et Sébastien Boyaval (INRIA). Nous allons donc chercher à utiliser des outils de réductions de modèles afin de pouvoir prédire, étant donné une concentration initiale de polluant en un lieu donné, l'évolution de la concentration de façon fiable. La dynamique de la solution réelle dépend d'un nombre de paramètres beaucoup trop grand si l'on souhaite la simuler tout le temps avec des paramètres différents à chaque fois (position initiale, champs de vitesse,...). La discrétisation du problème pose aussi problème, avec un phénomène de diffusion souvent inévitable. Le but de la méthode de réduction de modèles est de pouvoir se ramener à une base de solutions qui contiennent en elle-même la majeure partie de la dynamique de la solution réelle pour un certain nombre de paramètres où on aura la solution fine, afin de pouvoir simuler de façon approchée et surtout plus rapide des nouveaux schémas avec des paramètres différents.
Il faut donc d'abord penser à l'approche que l'on va avoir sur la discrétisation du problème, puis savoir quel outil de la réduction de modèle utiliser afin d'avoir une erreur relativement faible pour un temps relativement court : un compris est donc fait entre temps de calcul et précision.

2 Matériels et méthodes

Tout notre projet repose sur l'équation dite d'*advection*, ou de *transport de matière* :

$$\partial_t c + u\nabla c = 0. \tag{1}$$

où c est la concentration de polluant et u est le champs de vitesse présent dans le liquide ou le gaz. Pour une simulation numérique, nous devons nous munir d'un code donnant un maillage de la zone que nous voulons étudier, d'un schéma de discrétisation en temps et en espace de l'équation (1), et d'une condition initiale.

Une fois ce code mis en place, nous utilisons l'algorithme POD (Proper Orthogonal Decomposition) pour réduire notre solution. Nous nous posons alors trois questions :

— L'algorithme POD est-il adapté à l'équation d'advection ?
— Quelle approche, lagrangienne ou eulérienne, convient-il de prendre ?
— Peut-on trouver un moyen d'augmenter la compression de données ?

Nous allons explorer ces questions à travers deux exemples : un écoulement à vitesse constant (où juste l'angle d'incidence change) et un écoulement cellulaire (des tourbillons).

2.1 Méthode POD : *Proper Orthogonal Decomposition*

L'inconnue du problème est notée $C(t) \in \mathbb{R}^N$ pour $(t, x) \in [0, T]$. La première étape de cette méthode consiste à calculer des valeurs de la solution $X(t_0), X(t_1), ..., X(t_n)$ à des instants $t_0, t_1, ...t_n$, appelées snapshots. Ces vecteurs sont ensuite classés dans une matrice M :

$$M = \Big(C(t_0) \quad | \quad C(t_1) \quad | \quad \cdots \quad | \quad C(t_n)\Big)$$

où : $C(t_i)_j = c(x_j, t_i)$. $(x_j)_{1 \leqslant j \leqslant N}$ sont les points où la solution c du problème de transport est calculée précisement. La décomposition SVD (Singular Value Decomposition) de la matrice M s'écrit : $M = USV^T$ où U et V sont des matrices orthogonales respectivement de taille N et n. La matrice S est diagonale, ses coefficients sont les valeurs singulières de M (les valeurs propres de M^TM). Cette décomposition est unique lorsque les coefficients sur la diagonale de S, notés $\sigma_1, ..., \sigma_p$ avec $p = rang(S) = rang(M)$, sont ordonnés par ordre décroissant.

Les colonnes de U, notées $U^{(1)}, ..., U^{(N)}$ sont les modes propres de la matrice (ce sont les vecteurs propres de M^TM). Soit r un entier compris entre 1 et le rang de S. Les vecteurs $U^{(1)}, ..., U^{(r)}$ forment alors une base orthonormée d'un sous-espace vectoriel de $\mathbb{R}^N$ dans lequel on projette la dynamique du système.

Cette méthode s'applique lorsque la dynamique du système reste proche d'un sous-espace vectoriel de dimension faible. Cela correspond à une décroissance rapide des valeurs singulières de M. L'erreur de représentation est donnée par : $\sum_{i=r+1}^{p} \sigma_i^2$. On obtient ainsi une representation simple de la solution. Cela nous permet ensuite de faire varier les paramètres du modèle ainsi que le champs de vitesse.

2.2 Paramétrage

Dans tout ce qui suit, nous avons opté comme maillage un quadrillage de la zone $(0, 1)^2$, et le nombre de cases sur le maillage est égal à $(2 \times 10^8 + 1)^2$: c'est un cas où le maillage est plutôt fin et le temps de calcul, bien qu'un peu long, reste raisonnable.

On effectue un nombre n_s de simulations différentes dans chaque cas afin de pouvoir remplir la matrice qui va nous servir pour faire l'algorithme POD (matrice contenant quelques solutions fines du problème).
On prend un temps maximal de simulation égal à $T_{max} = 1$.
Lors de l'algorithme POD, on se fixe un nombre nb_{modes} que l'on choisit tel que l'erreur sur la solution soit satisfaisante. Cette erreur est choisie a posteriori, étant un compromis entre le temps de calcul (nombre de modes) et l'erreur de la précision de l'algorithme.

Cas d'une vitesse constante Dans le cas où la vitesse est constante, seule sa direction varie et est donnée par un angle $\theta \in [0, \frac{\pi}{2}]$ (on prend ces angles pour éviter que l'on n'atteigne le bord). On a donc

$$\begin{cases} u_x = ||u|| \cos(\theta) \\ u_y = ||u|| \sin(\theta) \end{cases}$$

Les particules sont distribuées selon une gaussienne centrée en $(0.25\,;\, 0.25)$ de variance $1/50$. Leurs vitesses sont identiques, constante et orientées avec un angle θ compris entre 0 et $\frac{\pi}{2}$.

Cas d'un écoulement cellulaire Dans ce cas, le champ de vitesse dérive du potentiel suivant :

$$\psi(x, y) = \sin(2\pi x) \sin(2\pi y) + \theta_0 \cos(2\pi\theta_1 x) \cos(2\pi\theta_2 y)$$

où $\theta_0 \in [0, 2.5]$ et $(\theta_1, \theta_2) \in [0.5, 4]^2$. Nous avons pris $\theta_1 = \theta_2 = \frac{1}{2}$. La vitesse s'exprime alors :

$$\begin{cases} u_x(x,y) = \frac{\partial \psi}{\partial y} = 2\pi \sin(2\pi x)\cos(2\pi y) - \theta_0 2\pi\theta_2 \cos(2\pi\theta_1 x)\sin(2\pi\theta_2 y) \\ u_y(x,y) = -\frac{\partial \psi}{\partial x} = -2\pi \sin(2\pi y)\cos(2\pi x) + \theta_0 2\pi\theta_1 \cos(2\pi\theta_2 y)\sin(2\pi\theta_1 x) \end{cases}$$

Champ de vitesse de Lamb-Oseen Nous prenons dans ce cas une vitesse analytique donnée par :

$$\mathbf{V}(r,\theta,t) = \frac{\Gamma}{2\pi r}\left(1 - \exp\left(\frac{-r^2}{4\nu t + r_c^2}\right)\right)\mathbf{u}_\theta$$

où ν désigne la viscosité cinématique, Γ est la circulation de l'écoulement et r_c le rayon moyen.

2.3 Approche Eulérienne

2.3.1 Description

L'approche Eulérienne consiste à déterminer en chaque point de notre maillage les valeurs de la concentration en polluant. On va donc, à chaque itération de l'algorithme, mettre à jour l'ensemble du maillage.

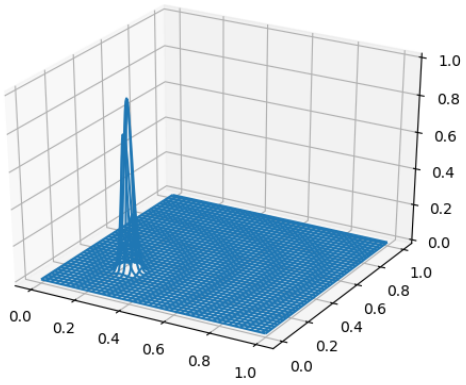
Il se trouve qu'une simple approche par la méthode des différences finies n'est pas un schéma stable pour notre étude : le schéma est un peu trop instable (concentration pouvant devenir négative). On va lui préférer la méthode des volumes finis avec un schéma de Lax-Friedrichs, qui est un schéma qui assure, par exemple, la positivité (et qui donc respecte une des propriétés fondamentales de la solution du problème continu). Ce schéma numérique classique introduit de la diffusion numérique. Les paramètres de discretisation seront donc choisis afin de limiter ce phénomène. Pour plus de détails, voir la section 2.3.2.

Nous avions à notre disposition un code d'éléments finis 2D sous Scilab faisant tourner le schéma de Lax-Friedrichs sur des cas très génériques et un maillage uniforme sur $(0,1)^2$. En déchiffrant une partie de ce code, et en s'y inspirant, nous avons codé ce schéma en Python, de façon simplifiée et en retenant seulement ce qui allait nous servir pour notre étude (maillage, fonction permettant l'actualisation, affichage de la solution). La zone étudiée est toujours $(0,1)^2$, et notre condition initiale est représentée par une gaussienne étroite (équivalente à un pic de polluant). De plus, les conditions aux bords sont périodiques : nous nous plaçons sur une sphère afin de rendre le code plus simple, même si nous ferons en sorte de ne jamais atteindre les bords. Ce code Python est donné en annexe.

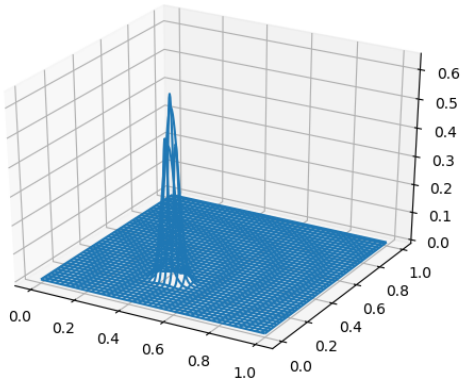
Notre condition initiale est donnée par une gaussienne :

$$c_0(x,y) = \exp\left(-\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}\right)$$

avec $(x_0, y_0) = (0.25, 0.25)$, $\sigma = \frac{1}{50}$. L'approche eulérienne consiste donc à transporter cette gaussienne le long du maillage.



Condition initiale...



...et après un certain temps

On stocke les valeurs de la solution au cours du temps sur tout le maillage dans une matrice à laquelle on va appliquer l'algorithme POD. Après un choix adapté du nombre de modes, on va pouvoir simuler la dynamique de notre système avec un nombre de degré de libertés bien plus faible.

2.3.2 Méthode des volumes finis

Le but de cette méthode est d'implémenter un schéma qui permet d'approcher la solution faible de l'équation :

$$\frac{\partial c}{\partial t} + \text{div}(c\ u) = 0 \tag{2}$$

de manière à assurer la convergence et la stabilité (voir le paragraphe Critères de convergence et de stabilité).

Nous regardons cette équation dans un compact en temps et en espace $[0, T] \times [0, L]^2$. Pour cela nous considèrerons un maillage en espace rectangulaire uniforme $K_{i,j}$ de pas Δx, Δy, et un maillage en temps de pas Δt_n qui dépend *a priori* de l'instant t_n. La méthode des volumes finis cherche à exprimer la valeur moyenne de la concentration c sur chaque cellule $K_{i,j}$ en fonction de la valeur moyenne cellulaire de la vitesse et des différents paramètres du maillage à savoir le pas et le volume des cellules. En d'autres termes, on exprime :

$$c_{i,j}(t_n) = \frac{1}{|K_{i,j}|} \int_{K_{i,j}} c(t_n, X) dX$$

en fonction de $u_{i,j}(t_n)$, Δx, Δy et Δt_n. Pour cela, nous intégrons l'équation (2) sur la cellule $K_{i,j}$ à l'instant t_n. Nous obtenons :

$$\frac{\partial c_{i,j}}{\partial t}(t_n) + \frac{1}{|K_{i,j}|} \int_{\partial K_{i,j}} c\ u \cdot n\ dL = 0$$

Nous approchons la dérivée temporelle par un schéma d'Euler explicite et le flux par un schéma de Lax-Friedrichs :
Pour toute cellule $K_{i,j}$

$$c_{i,j}(t_n) = c_{i,j}(t_{n-1}) - \frac{t_n - t_{n-1}}{|K_{i,j}|} \sum_{k,l} |\Gamma_{i,j}^{k,l}| g(c_{i,j}(t_{n-1}), c_{k,l}(t_{n-1}))\ \forall n \tag{3}$$

où $\Gamma_{i,j}^{k,l} = K_{i,j} \cap K_{k,l}$ et :

$$\begin{cases} \lambda_{i,j\to k,l} = \max\{u_{i,j} \cdot n_{i,j\to k,l}, u_{k,l} \cdot n_{i,j\to k,l}\} \\ g(c_{i,j}, c_{k,l}) = \frac{1}{2}(c_{i,j}u_{i,j} + c_{k,l}u_{k,l}) \cdot n_{i,j\to k,l} + \frac{1}{2\lambda_{i,j\to k,l}}(c_{i,j} - c_{k,l}) \end{cases} \tag{4}$$

à l'instant t_n. Pour plus de détails voir la section (6.1).

Critères de convergence et de stabilité Lorsque l'on part d'une condition initiale bornée, l'équation dans le sens faible (2) admet une unique solution entropique, c'est à dire la solution vérifie : $\forall S \in \mathcal{C}^2$ convexe, on a

$$\frac{\partial S(c)}{\partial t} + \mathrm{div}(G(c)) \leq 0$$

où G est un flux d'entropie qui vérifie $G' = S'u$.

Le schéma numérique (3) converge vers cette solution (lorsque $\Delta x, \Delta y, \Delta t_n \longrightarrow 0$) dès qu'il vérifie l'inégalité entropique discrétisée : $\forall S \in \mathcal{C}^2$ convexe, $\exists \Phi(c_1, c_2)$ telle que

$$\begin{cases} \Phi(c, c) = G(c) \\ \frac{S(c_{i,j}(t_n)) - S(c_{i,j}(t_{n-1}))}{t_n - t_{n-1}} + \sum_{k,l} \frac{|\Gamma_{i,j}^{k,l}|}{|K_{i,j}|} \Phi(c_{i,j}(t_{n-1}), c_{k,l}(t_{n-1})) \leq 0 \quad \forall i, j, n \end{cases} \tag{5}$$

Ainsi, l'entropie doit être dissipée ou conservée, son augmentation est une source d'instabilité du schéma.

Pour que le schéma des volumes finis (3) soit stable, il faut satisfaire la condition CFL (Courant, Friedrichs, Lewy) qui s'écrit :

$$t_n - t_{n-1} < \frac{|K_{i,j}|}{2|\Gamma_{i,j}^{k,l}|\lambda_{i,j\to k,l}(t_{n-1})} \quad \forall i, j, k, l$$

Cette condition nous permet de choisir le pas de temps adapté à la vitesse et à la géométrie du maillage à chaque instant pour assurer la stabilité.

2.3.3 Modèle Réduit

La POD fournit une base orthonormée de r fonctions scalaires $(P^k)_{k=1}^r$ discrétisées en espace 2D, on note $P_{i,j}^k$ la valeur moyenne de P^k dans la cellule $K_{i,j}$.

Nous allons approcher la concentration c en utilisant cette base. Ainsi on note :

$$c(X, t) \simeq c^r(X, t) = \sum_{k=1}^{r} a_k(t) P^k(X) \tag{6}$$

On injecte (6) dans (1), on obtient :

$$
\begin{aligned}
&\partial_t \left(\sum_{k=1}^{r} a_k P^k \right) + u \cdot \nabla \left(\sum_{k=1}^{r} a_k P^k \right) = 0 \\
\Rightarrow \quad &\sum_{k=1}^{r} \partial_t a_k P^k + \sum_{k=1}^{r} a_k u \cdot \nabla P^k = 0 \\
\Rightarrow \quad &\forall\ 1 \leq k_0 \leq r \quad \sum_{k=1}^{r} \partial_t a_k \langle P^k, P^{k_0} \rangle + \sum_{k=1}^{r} a_k \langle u \cdot \nabla P^k, P^{k_0} \rangle = 0
\end{aligned}
$$

avec $\langle f, g \rangle = \int_{[0,1]^2} fg$, qui est le produit scalaire de L^2. Comme la base $(P^k)_{k=1}^r$ est orthonormée, on obtient :

$$
\forall\ 1 \leq k_0 \leq r \quad \partial_t a_{k_0} + \sum_{k=1}^{r} a_k \langle u \cdot \nabla P^k, P^{k_0} \rangle = 0
$$

Ainsi on construit une matrice D de dimension $r \times r$ telle que $[D]_{l,k} = \langle u \cdot \nabla P^k, P^l \rangle$ et un vecteur $A(t)$ de dimension r tel que $[A(t)]_k = a_k(t)$, et par suite l'équation devient :

$$
\partial_t A(t) + D \cdot A(t) = 0 \tag{7}
$$

Discrétisation spatiale et calcul de D : Toute fonction P^k est discrétisée en espace de la façon suivante :

$$
P^k_{i,j} = \frac{1}{|K_{i,j}|} \int_{K_{i,j}} P^k(X) \mathrm{d}X
$$

D'abord, on essaye d'approcher ∇P^k sur $K_{i,j}$ en utilisant un schéma de différences finis :

$$
\nabla P^k_{i,j} = \begin{pmatrix} \frac{P^k_{i,j+1} - P^k_{i,j-1}}{2\Delta x} \\ \frac{P^k_{i+1,j} - P^k_{i-1,j}}{2\Delta y} \end{pmatrix}
$$

ainsi :

$$
(u \cdot \nabla P^k)_{i,j} = u^x_{i,j} \frac{P^k_{i,j+1} - P^k_{i,j-1}}{2\Delta x} + u^y_{i,j} \frac{P^k_{i+1,j} - P^k_{i-1,j}}{2\Delta y}
$$

et finalement on peut calculer les coefficients de D

$$
\begin{aligned}
[D]_{l,k} &= \int_{[0,1]^2} (u \cdot \nabla P^k)\ P^l \\
&\simeq \sum_{i,j} |K_{i,j}| (u \cdot \nabla P^k)_{i,j}\ P^l_{i,j} \\
&= \sum_{i,j} |K_{i,j}| \left(u^x_{i,j} \frac{P^k_{i,j+1} - P^k_{i,j-1}}{2\Delta x} + u^y_{i,j} \frac{P^k_{i+1,j} - P^k_{i-1,j}}{2\Delta y} \right)\ P^l_{i,j} \quad \forall 1 \leq k, l \leq r
\end{aligned}
$$

Discrétisation temporelle : Nous utilisons le schéma d'Euler implicite pour discrétiser en temps l'équation (7) qui s'écrit :

$$
\frac{1}{t_n - t_{n-1}} (A(t_n) - A(t_{n-1})) + D \cdot A(t_n) = 0
$$

ou en réarrangeant les termes :

$$
A(t_n) = A(t_{n-1}) - \Delta t_n D \cdot A(t_n) \tag{8}
$$

Condition initiale Posons $C(t_n)$ le vecteur qui contient les $c_{i,j}(t_n)$ et C_r le vecteur contient les $c^r_{i,j}(t_n)$. On remarque alors que $C_r(t_n)$ (de dimension $n_x \times n_y$) est complètement déterminé par le vecteur $A(t_n)$ (de dimension r) via l'équation :

$$C(t_n) \simeq C_r(t_n) = U_r \cdot A(t_n)$$

où

$$U_r = (P^1|P^2|\dots P^r)$$

Pour déterminer la condition initiale $A(t_0)$, il suffit de projeter la condition initiale $C(t_0)$ sur la base (P^k), cela revient à :

$$A(t_0) = U_r^T \cdot C(t_0)$$

Algorithme : On part de la condition initiale $A(t_0) = U_r^T \cdot C(t_0)$ puis pour $n > 0, t_n < T$ on résout l'équation $(\Delta t_n D + I_r) \cdot A(t_n) = A(t_{n-1})$. Les vecteurs $A(t_n)$ ainsi obtenus sont utilisés pour reconstruire les vecteurs $C(t_n)$ avec la relation $C(t_n) = U_r \cdot A(t_n)$.

On voit ici l'intérêt de la méthode POD, on a pu réduire le problème de dimension $(n_x \times n_y \times n_t)$ en un problème de dimension $(r \times n_t)$. Dans le cas où $r << n_x \times n_y$, le temps de calcul est donc conséquemment réduit.

Conditions de stabilité condition CFL ...

2.4 Approche Lagrangienne

2.4.1 Description

L'approche lagrangienne consiste à suivre les particules le long de leur trajectoires. Si l'on note $X(\xi, t) \in \mathbb{R}^{n\times 2}$ les positions, à l'instant $t \in [0, T]$, des particules qui était initialement aux positions $\xi \in \mathbb{R}^{n\times 2}$, cela revient à résoudre :

$$\begin{cases} \partial_t X = v(X(\xi, t), t) \\ X(\xi, 0) = \xi \end{cases}$$

Supposons $u \in \mathcal{C}^0(\mathbb{R}^{n\times 2} \times \mathbb{R}) \bigcap W^{1,\infty}(\mathbb{R}^{n\times 2} \times \mathbb{R})$. C'est toujours le cas dans notre étude. Alors le théorème de Cauchy-Lipschitz assure l'existence et l'unicité d'une solution locale. De plus, l'application est globalement lipschitzienne. Le transport est donc à vitesse finie.

2.4.2 Résolution numérique

Nous avons utilisé le schéma de Crank-Nicholson :

$$X^{(k+1)} = X^{(k)} + \frac{\Delta t}{2}(v(X^{(k)}, t^k) + v(X^{(k+1)}, t^{k+1}))$$

Nous avons utilisé une méthode du point fixe pour évaluer $X^{(k+1)}$ en connaissant $X^{(k)}$. Initialisation (r=0) :

$$X_0^{(k+1)} = X^{(k)}$$

$$X_1^{(k+1)} = X^{(k)} + \Delta t v(X^{(k)}, t^k)$$
$$X_{r+1}^{(k+1)} = X^{(k)} + \frac{\Delta t}{2}(v(X^{(k)}, t^k) + v(X_r^{(k+1)}, t^{k+1}))$$

Et on arrête lorsque :

$$||X_{r+1}^{(k+1)} - X_r^{(k+1)}|| < \epsilon$$

où ϵ est une erreur donnant un critère d'arrêt : on suppose qu'on est "suffisamment proche de la solution" quand l'algorithme s'arrête.

2.4.3 Interpolation

La résolution numérique du problème nécessite de connaître le champ de vitesse des particules, noté v. Cependant, le champ de vitesse n'est pas toujours connu analytiquement Par exemple, si la vitesse est mesurée expérimentalement, alors elle n'est connue qu'aux points de mesure. Nous présentons ici une méthode d'interpolation, c'est à dire qui nous permet d'approximer le champs de vitesse sur l'ensemble du domaine lorsqu'il n'est connu qu'en certain points. Nous supposons que l'ensemble $\Omega = (0,1)^2$ soit muni d'un maillage carré régulier de pas noté Δx en abscisse et Δy en ordonnée. Nous supposons également que le champ de vitesse est connu en tout point du maillage. Nous présentons ici la méthode d'interpolation qu'il conviendrait d'utiliser dans ce cas.

Considérons une maille dont les coordonnées du coin inférieur gauche sont notées (x^{LL}, y^{LL}). Soit un point (x, y) appartenant à cette maille. On pose :

$$\begin{cases} \tilde{x} = \frac{x - x^{LL}}{\Delta x} \\ \tilde{y} = \frac{y - y^{LL}}{\Delta y} \end{cases}$$

On définit :

$$\begin{cases} \varphi_1(\tilde{x}, \tilde{y}) = (1 - \tilde{x})(1 - \tilde{y}) \\ \varphi_2(\tilde{x}, \tilde{y}) = \tilde{x}(1 - \tilde{y}) \\ \varphi_3(\tilde{x}, \tilde{y}) = \tilde{x}\tilde{y} \\ \varphi_4(\tilde{x}, \tilde{y}) = (1 - \tilde{x})\tilde{y} \end{cases}$$

La vitesse au point (x, y) à l'instant t est alors donnée par :

$$v(x, y, t) = v(x^{LL}, y^{LL}, t)\varphi_1(\tilde{x}, \tilde{y}) + v(x^{LL} + \Delta x, y^{LL}, t)\varphi_2(\tilde{x}, \tilde{y})$$
$$+ v(x^{LL} + \Delta x, y^{LL} + \Delta y, t)\varphi_3(\tilde{x}, \tilde{y})$$
$$+ v(x^{LL}, y^{LL} + \Delta y, t)\varphi_4(\tilde{x}, \tilde{y})$$

3 Résultats

3.1 Approche Eulerienne

3.1.1 Cas vitesse constante

On effectue $n_s = 16$ simulations, avec des valeurs d'angles uniformément réparties dans $[0, \frac{\pi}{2}]$. Voici d'abord le tracé du log des valeurs singulières en fonction de leur rang dans la décomposition SVD.

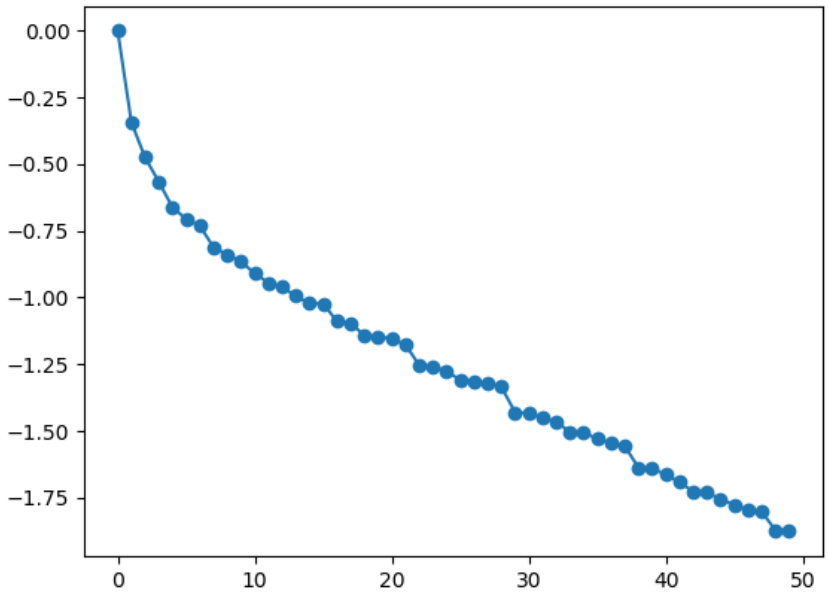
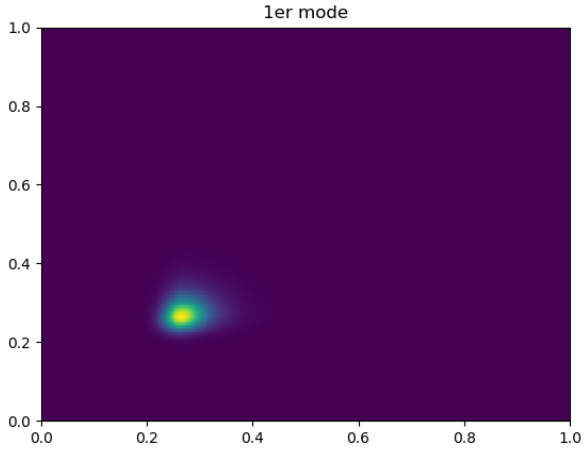
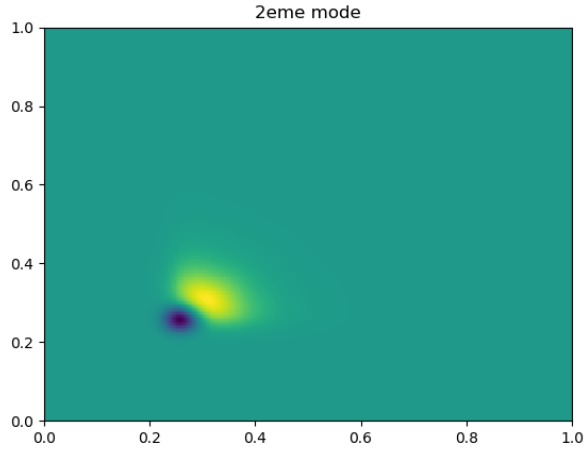


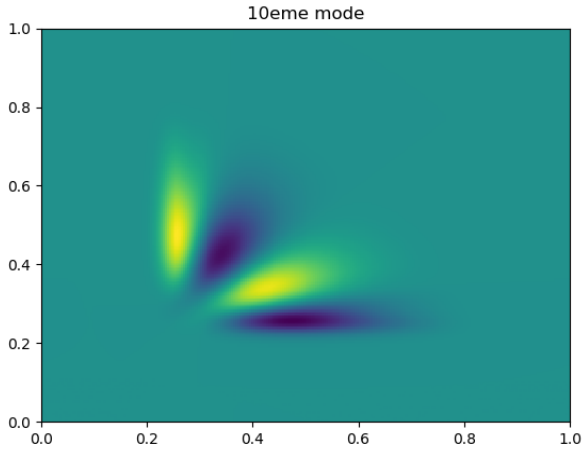
FIGURE 1 – Tracé du log des valeurs singulières en fonction du rang

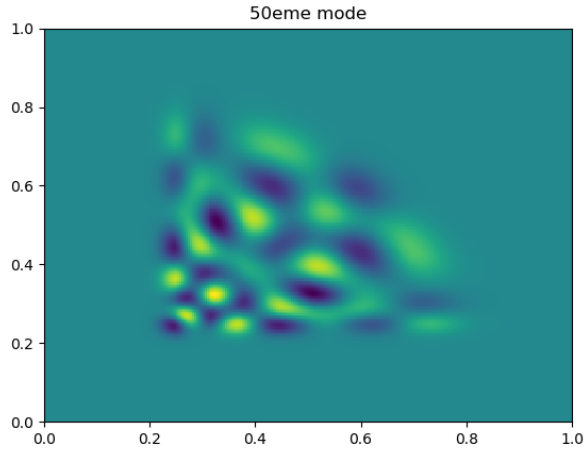
Ceci nous permet de choisir le nombre de modes que l'on va utiliser. Avec un nombre maximal de 50 modes (que nous nous sommes fixés), on voit que l'on n'arrive pas à atteindre l'erreur relative de 10^{-3}. En fait, il faudrait environ 100 modes propres pour obtenir cette erreur là. On voit donc que l'approche eulerienne dans le cas d'une vitesse constante n'est pas optimale : en effet, pour chaque valeur de la vitesse, la dynamique de la solution est différente (son parcours n'est jamais le même et on ne se recoupe jamais), on ne peut donc pas facilement exhiber un sous-espace linéaire qui contiendrait la majorité de la dynamique. Il est donc inutile d'essayer de réduire ce modèle sous l'approche eulérienne.
On note cependant qu'avec 16 simulations, on obtient une matrice de *snapshots* d'environ $3,7Go$ ($257 \times 257 \times 8 \times 7346$, 7346 étant le nombre de colonnes), que l'on arrive à compresser en $517ko$ en prenant 50 modes : même si l'erreur reste trop importante, la compression de données est très bonne grâce à l'algorithme POD.

De plus, cet exemple illustre parfaitement le rôles des modes propres dans la reconstitution de la solution approchée. Voici en effet le tracé des modes propres sur $(0,1)^2$ (en utilisant le module *pcolor* de *matplotlib.pyplot*) :





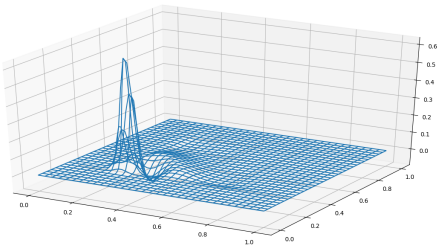




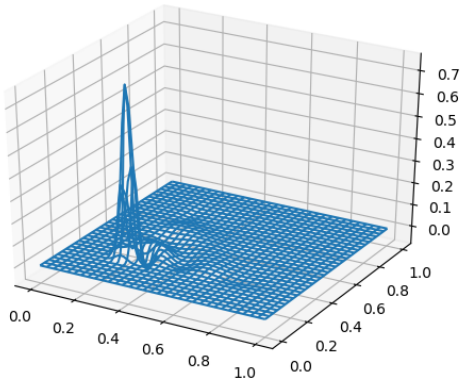
Les différentes couleurs montrent les oscillations de la fonction. On voit que le premier mode rassemble la majeure dynamique de ce qu'on a simulé : la condition initiale, qui est restée la même durant toutes nos simulations, est représentée de façon très claire.
On voit ensuite, dès le deuxième mode, la première oscillation qui va permettre de reconstituer les solutions. L'oscillation se dirige dans le cadrant en haut à droite, lieu de nos simulations car $\theta \in [0, \frac{\pi}{2}]$. Le 10ème mode représente bien les différentes trajectoires de nos simulations (encore via des oscillations), et enfin, quand on va chercher des modes plus grands, on va essayer de reproduire de plus en plus les détails de la dynamique sur l'ensemble de nos valeurs de paramètres $\theta \in [0, \frac{\pi}{2}]$.

Ensuite, nous avons testé la reconstruction d'une solution avec notre base de modes propres pour un nouveau paramètre qui ne faisait pas partie des simulations précédentes : on est donc sûr qu'aucun vecteur solution ayant servi à la POD ne peut directement représenter la solution. En l'occurrence, nous avons pris $\theta = \frac{\pi}{9}$. Ceci est un test dit *out-of-data in-bound* : on utilise une valeur de paramètre non présente dans la base qui va servir à réduire la modèle, mais qui est quand même dans l'intervalle que nous nous sommes fixés au départ.

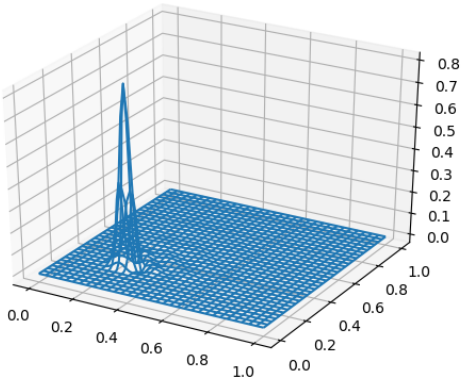
Voici les différentes reconstructions pour un nombre de modes propres valant 10, 20, 50, 100, 200 et 500, d'abord pour la condition initiale :



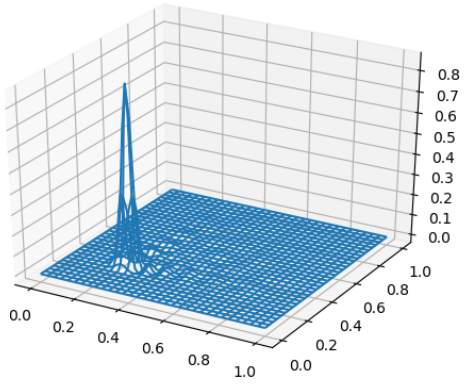
Condition initiale, 10 modes



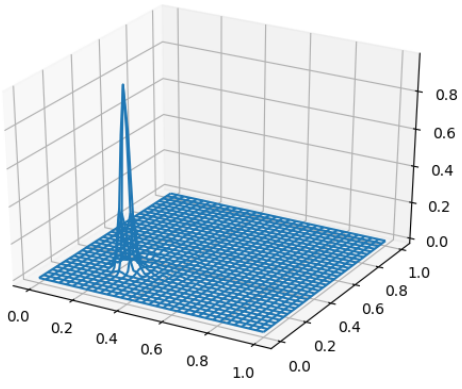
Condition initiale, 20 modes



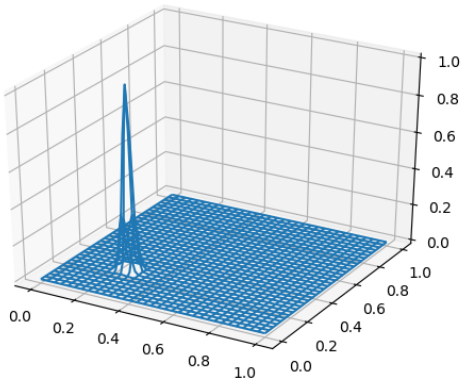
Condition initiale, 50 modes



Condition initiale, 100 modes



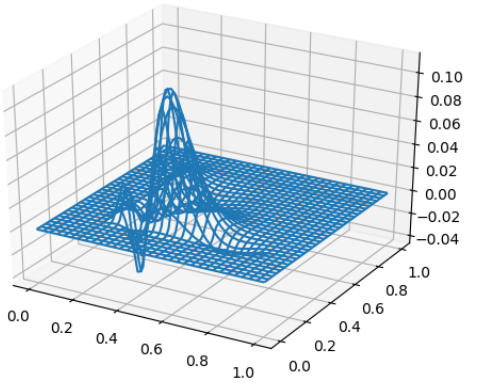
Condition initiale, 200 modes



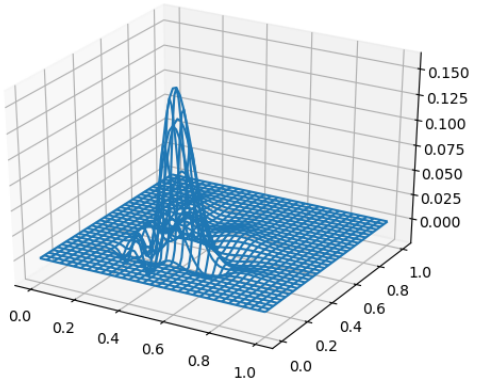
Condition initiale, 500 modes

On voit premièrement que la condition initiale est plutôt bien restituée. Avec un nombre faibles

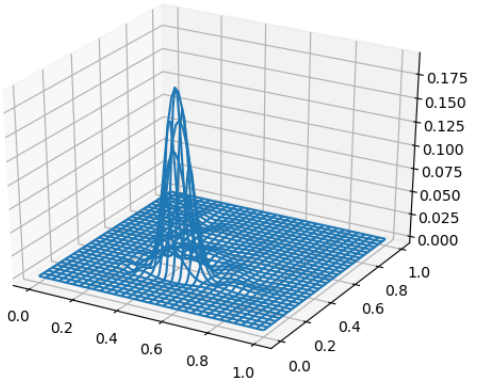
de modes, on observe des oscillations mais elles disparaissent vite avec un nombre de modes de plus en plus grand. Cela est notamment dû au fait que la condition initiale a été la même dans toutes les simulations. On remarque aussi et surtout que la positivité de la solution n'est plus conservée après reconstruction via la base propre. C'est à contraster avec ce qu'on voulait via l'utilisation du schéma de Lax-Friedrichs. Regardons maintenant ce qu'il se passe à mi-parcours :



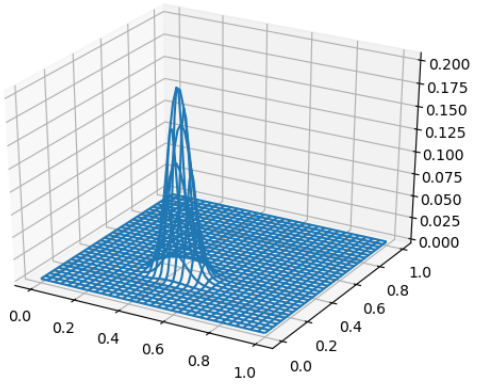
A mi-parcours, 10 modes



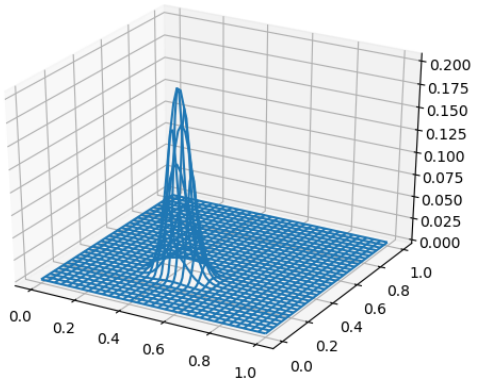
A mi-parcours, 20 modes



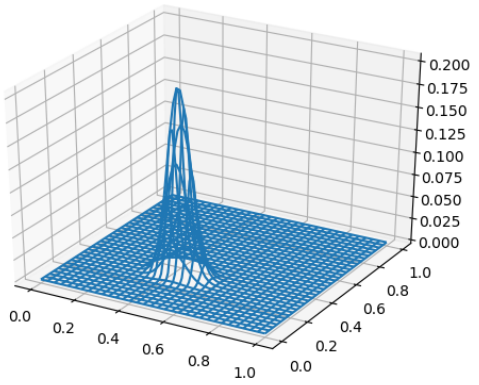
A mi-parcours, 50 modes



A mi-parcours, 100 modes



A mi-parcours, 200 modes



A mi-parcours, 500 modes

Cette fois-ci, on voit très clairement qu'un faible nombre de modes ne permet plus de restituer une dynamique cohérente. Des oscillations de grande amplitudes sont observées pour 10 et 20 modes, et il reste encore des petites "vagues" pour 50 modes. Cependant, passé une centaine de modes, on n'observe plus de grands changements dans la gaussienne. Il faudrait donc une centaine de modes environ pour s'en sortir dans la restitution d'une solution fine pour un nouveau paramètre dans l'intervalle de départ.
Si l'on fait le calcul (comme ce sera le cas pour l'écoulement cellulaire), on observe que l'erreur pour un même nombre de modes augmentent au cours du temps : cela est dû au fait que le paramètre n'est pas représenté dans la base propre dès le départ.

3.1.2 Cas des écoulements cellulaires

Ensuite, nous avons considéré le cas des écoulements cellulaires (cf. paramétrages) dans le carré $[-0.5, 1.5] \times [-0.5, 1.5]$. On considère une condition initiale sous forme de gaussienne d'écart-type $\sigma = \frac{1}{20}$ Le problème étant paramétré par la position initiale (x_0, y_0) ainsi que par les paramètres $(\theta_0, \theta_1, \theta_2)$, on a fixé $\theta_1 = \theta_2 = 0.5$ et tiré aléatoirement de manière uniforme 64 triplés (x_0, y_0, θ_0) dans $[0.25, 0.75] \times [0.25, 0.75] \times [0, 2.5]$.

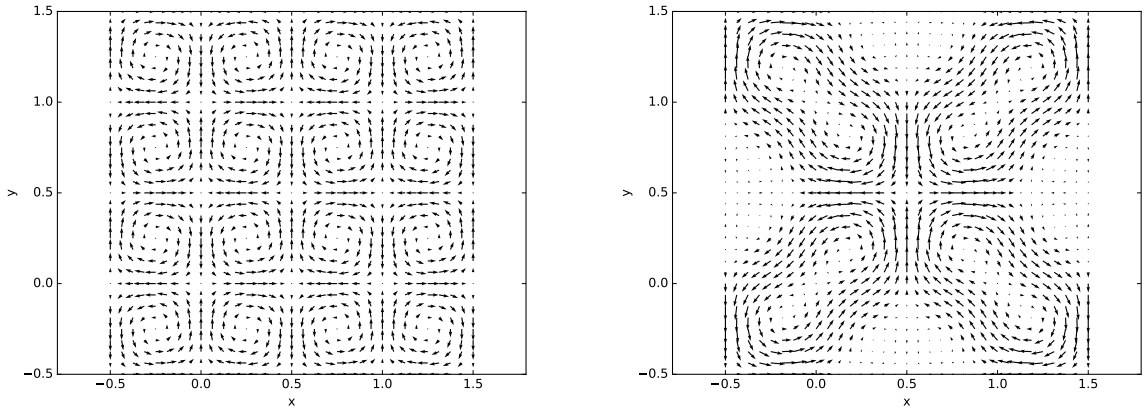


FIGURE 2 – Exemples de champs de vitesse d'écoulements cellulaires pour $\theta_0 = 0$ et $\theta_0 = 2$

On a donc résolu l'équation de transport pour ces 64 jeux de paramètres pendant une durée $T = 0.5$. On a pris un pas de maillage de $\frac{1}{2^7}$, i.e. deux fois moins fin que pour un champ de vitesse constant, sans quoi le temps de calcul aurait été trop important. Après trois heures de simulation, on obtient une matrice M des snapshots de taille 16641×14980, et qui pèse 2 Go.

On a effectué la SVD (décomposition en valeurs singulières) de M, puis on a sauvegardé les 500 premiers modes propres, pour un total de 66.6 Mo, ce qui met bien en évidence la compression de donnée qu'apporte la réduction de modèles. On a tracé les 500 premières valeurs singulières de M :

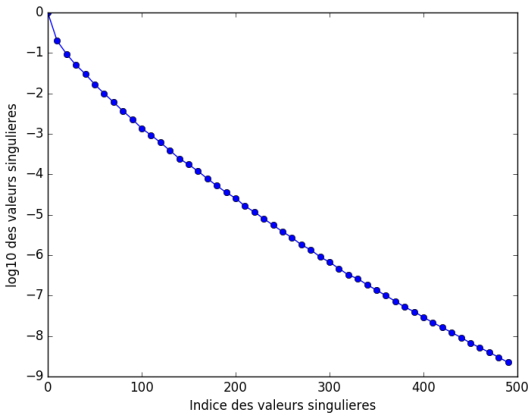
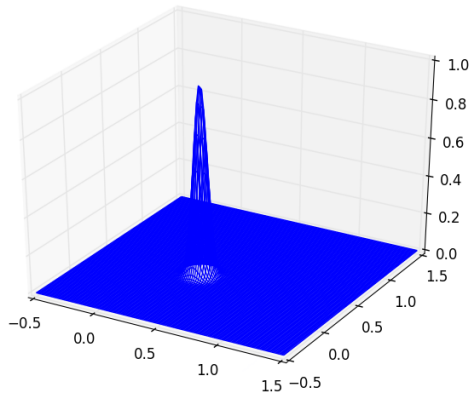


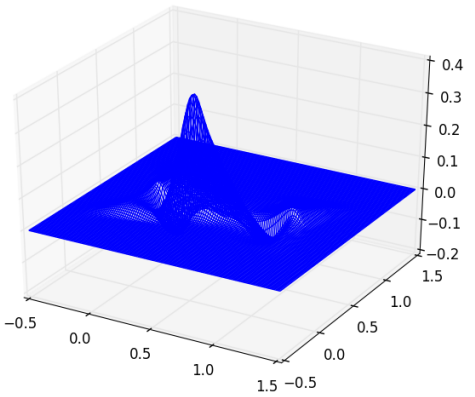
FIGURE 3 – Tracé du log des 500 premières valeurs singulières

On observe une décroissance exponentielle des valeurs singulières, ce qui indique que la POD est une solution envisageable dans le cas des écoulements cellulaires. On peut donc sélectionner un nouveau jeu de paramètres "out of data in range", i.e (x_0, y_0, θ_0) dans $[0.25, 0.75] \times [0.25, 0.75] \times [0, 2.5]$ calculer la solution fine à l'aide des volumes finis, puis projeter chaque snapshot sur les r premiers vecteurs de la base POD afin de calculer l'erreur.

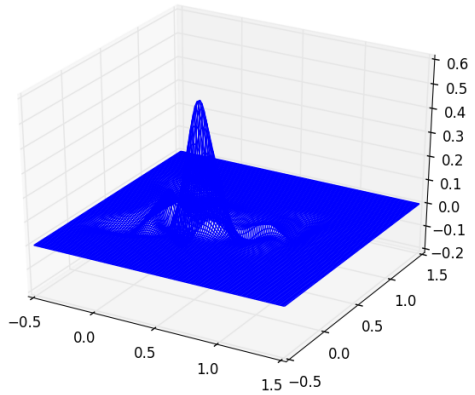
Voici les différentes reconstructions pour un nombre de modes propres valant 10, 20, 50, 100 et 500, pour la condition initiale $(x_0, y_0, \theta_0) = (0.35, 0.35, 1)$:



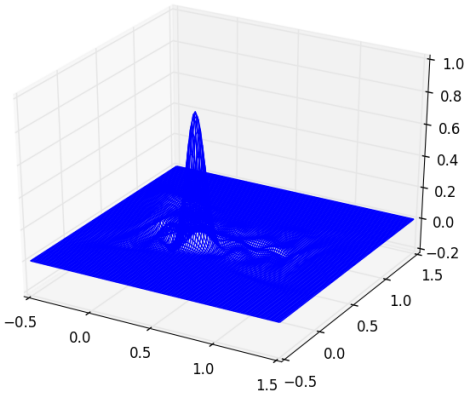
Condition initiale de référence



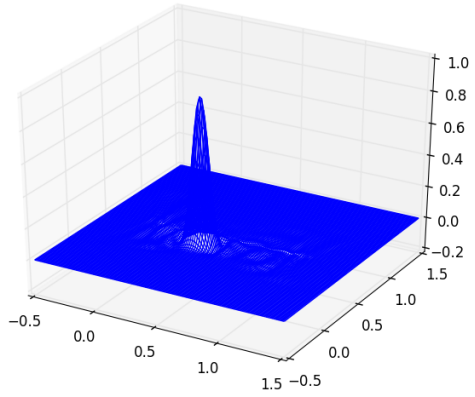
Condition initiale, 10 modes



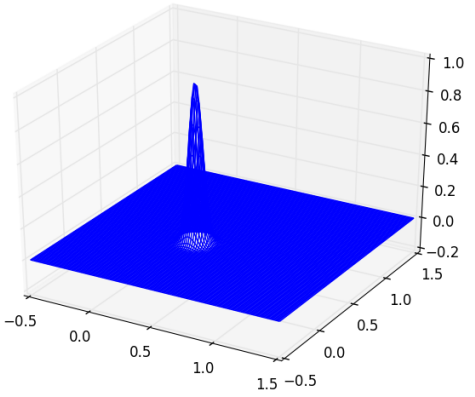
Condition initiale, 20 modes



Condition initiale, 500 modes



Condition initiale, 100 modes

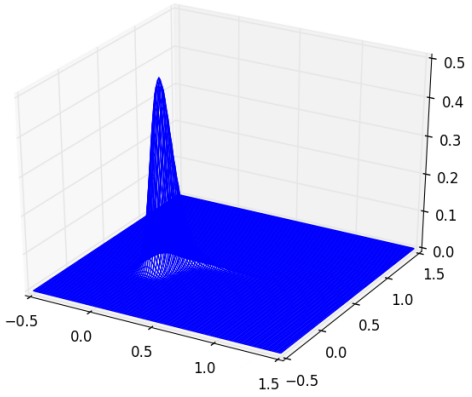


Condition initiale, 500 modes

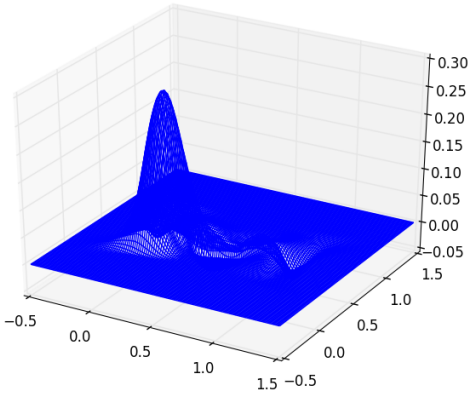
On remarque que la condition initiale "in range" est plutôt bien restituée au bout d'un certain

nombre de modes. On remarque de même, que la projection donne des concentrations négatives et des oscillations à certains endroits, ce qui est un inconvénient par rapport à la méthode des volumes finis qui conserve la positivité de la solution.

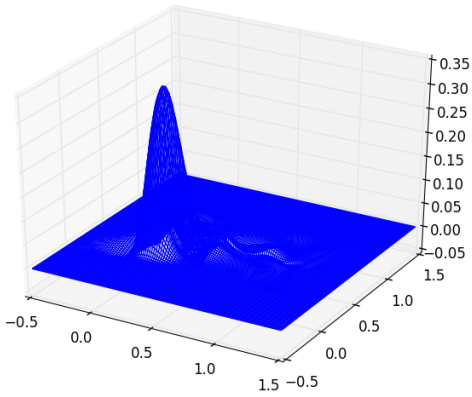
On obtient le même genre de résultats si on fait de même pour un snapshot au milieu de la simulation :



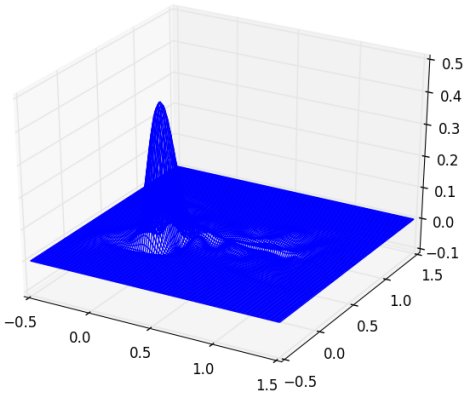
A mi-parcours, référence



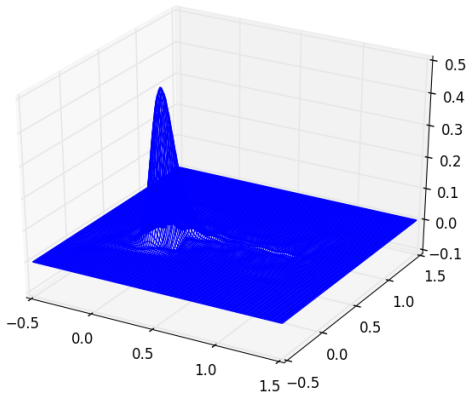
A mi-parcours, 10 modes



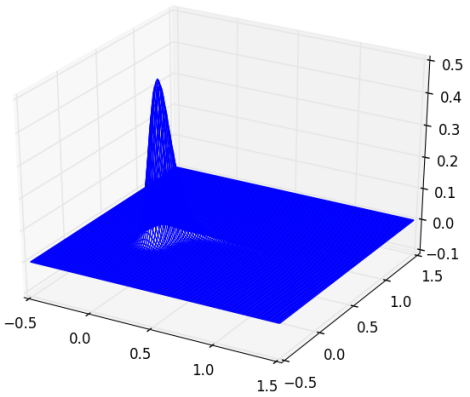
A mi-parcours, 20 modes



A mi-parcours, 500 modes



A mi-parcours, 100 modes



A mi-parcours, 500 modes

Lorsqu'on calcule l'erreur relative en norme de Frœbenius, on obtient :

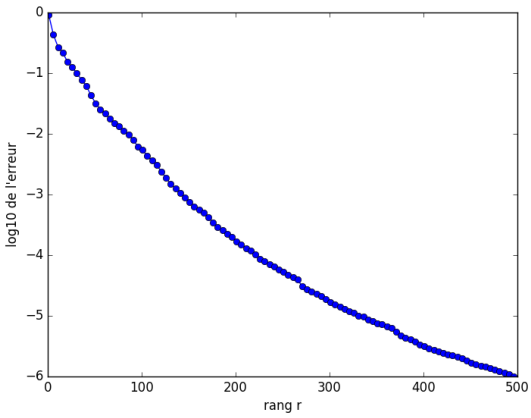


FIGURE 4 – Erreur relative en norme de Frœbenius

Cela comfirme bien les observations précédentes.

Ensuite, on s'est interessé à des jeux de paramètres "out of data-out of range ", i.e. (x_0, y_0, θ_0) en dehors de $[0.25, 0.75] \times [0.25, 0.75] \times [0, 2.5]$. Ainsi, pour $(x_0, y_0, \theta_0) = (0, 0, 1)$, la gaussienne initiale devient :

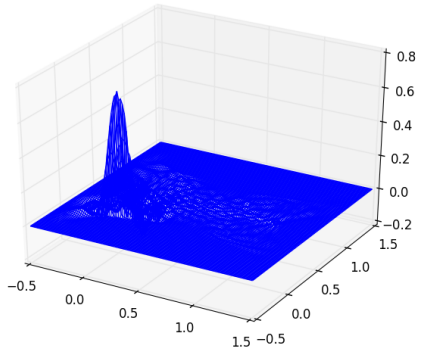


FIGURE 5 – Projection d'une gaussienne "out of range" pour 500 modes

On observe déja que la gaussienne n'est pas très bien restituée.

Puis, au milieu de la simulation, la projection sur les 500 modes propres n'est pas du tout restituée :

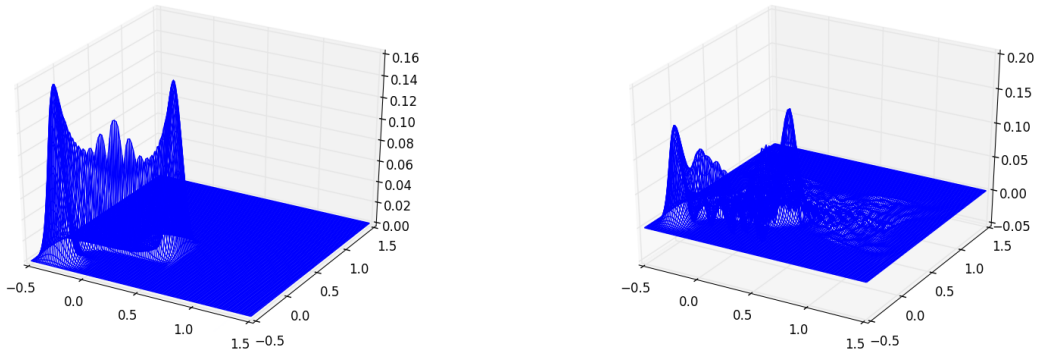


FIGURE 6 – Solution de référence et sa projection sur 500 modes propres

Cela est du au fait que les modes propres sont quasiment nuls en dehors de l'intervalle $[0.25, 0.75]\times [0.25, 0.75] \times [0, 2.5]$. On constate donc les limites de la méthode POD.

3.1.3 Prise en compte de la positivité de la concentration

Afin de prendre en compte la positivité de la solution, on peut ajuster les coefficients de la POD de sorte à ce que :

$$(a_j)_j = \text{argmin} \|c_* - \sum_j a_j \varphi_j\|_{L^2} \text{ s.c. } \sum_j a_j \varphi_j \geq 0$$

Pour cela, nous avons utilisé les algorithme d'optimisation de la bibliotèque **scipy.optimize**.

Voici un exemple de correction d'une gaussienne projetée sur la base POD :

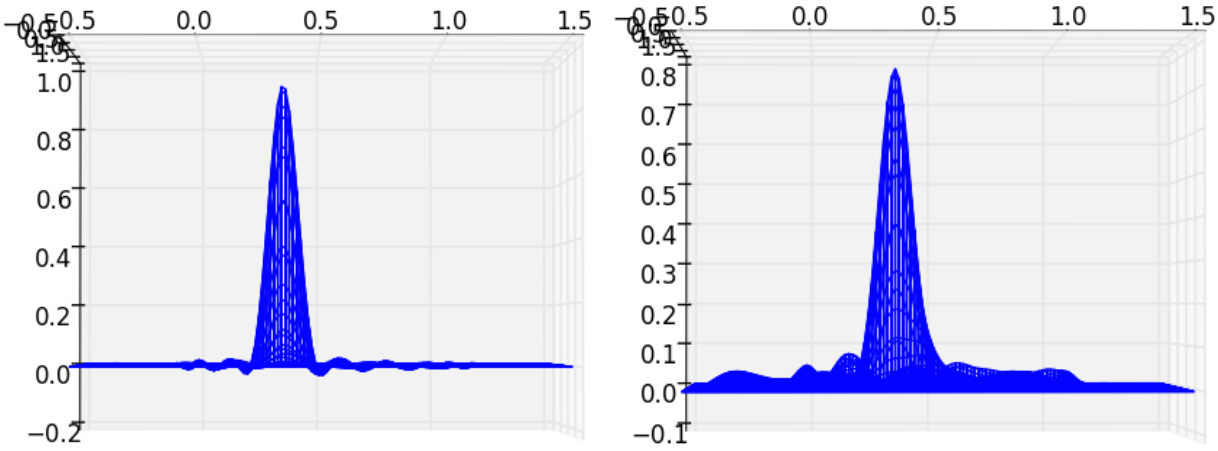


FIGURE 7 – Projection d'une gaussienne sur la base propre, et son optimisation

Voici un autre exemple en cours de simulation :

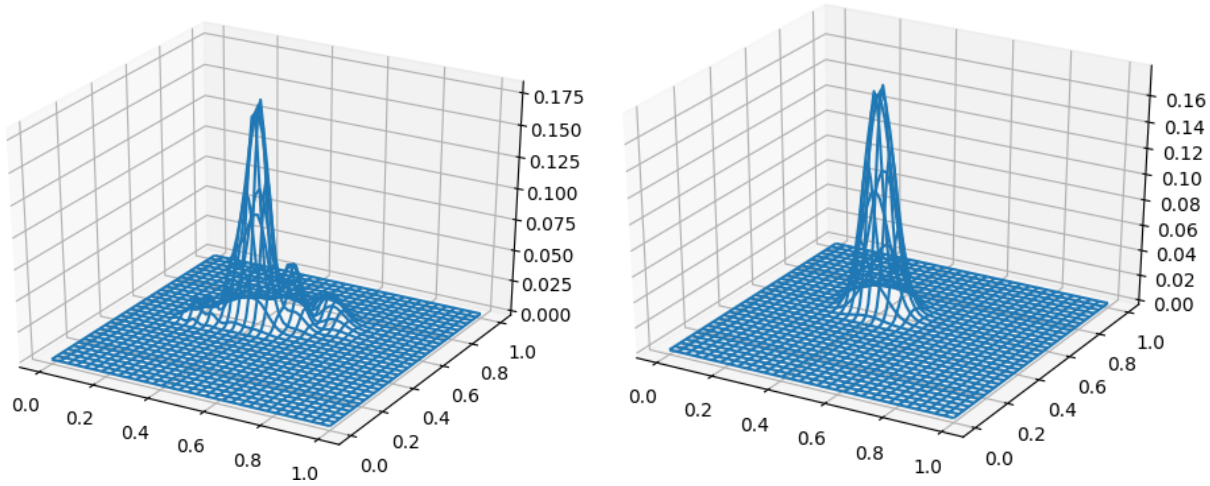


FIGURE 8 – Correction d'une gaussienne au cours d'une simulation, pour 20 et 50 modes

Ainsi, cette méthode d'optimisation est pluttôt efficace afin de respecter la condition de positivité de la concentration. Cependant, il faut lancer l'algorithme pour chaque snapshot, ce qui rallonge considérablement le temps de calcul, et donc supprime l'avantage principal de la réduction de modèle.

3.2 Approche Lagrangienne

3.2.1 Vitesse constante

Voici les valeurs singulières de la matrice des snapshots pour 10 particules. Nous avons également effectué le calcul pour 100 particules, le resultat obtenu est le même.

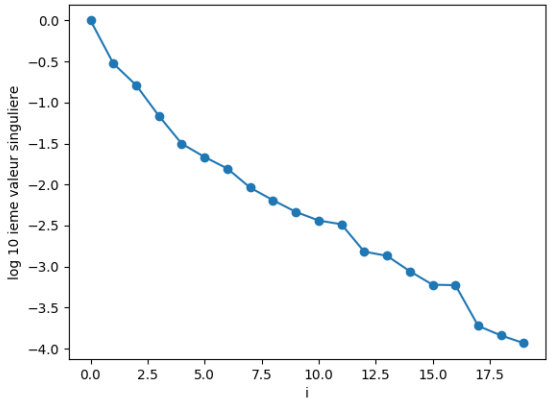


FIGURE 9 – Valeurs singulières pour 10 particules

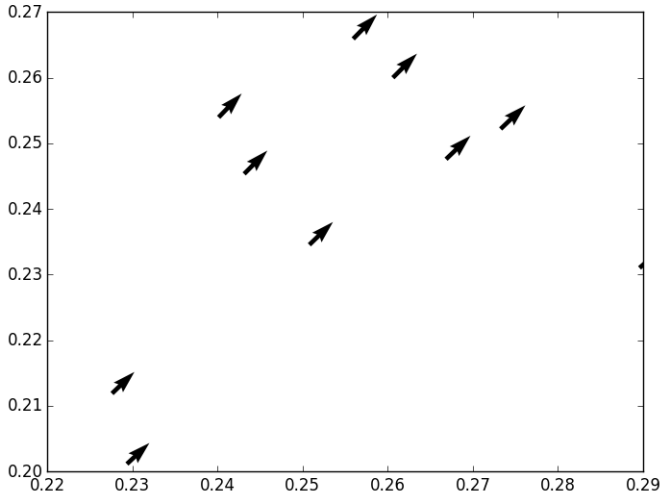


FIGURE 10 – Projection suivant le premier mode

La décroissance des valeurs singulières est très rapide dans les deux cas. Les deux premiers modes propres (colonnes de U suffisent à représenter très correctement la solution. Ce résultat était prévisible puisque les particules possèdent des trajectoires rectilignes uniformes, ce qui est relativement simple à representer.

3.2.2 Écoulement cellulaire

La dynamique a été simulée en utilisant les paramètres suivants : $\theta_0 = 0.2; \theta_1 = 3.12; \theta_2 = 2.69$.

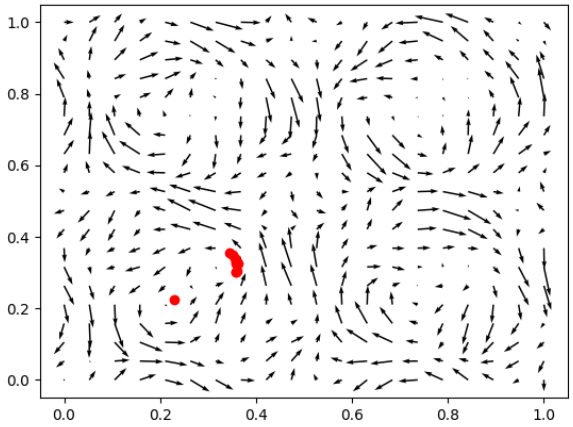


FIGURE 11 – 10 particules dans un écoulement cellulaire

La figure suivante montre les valeurs singulières de la matrice des snapshots pour 100 particules, pour les paramètres donnés ci-dessus.

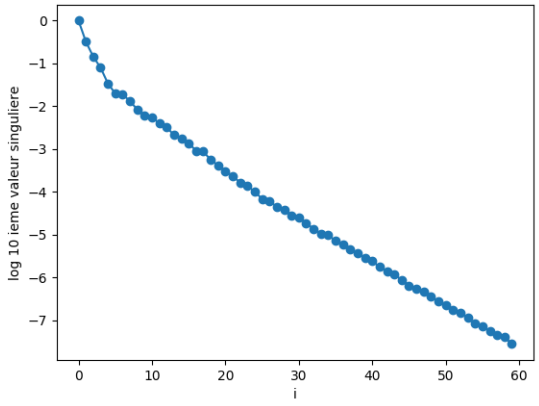


FIGURE 12 – Valeurs singulières pour 100 particules, 1 simulation

On remarque que la décroissance est assez lente.

La simulation suivante a nécessite plus de temps de calcul. Les positions initiales ainsi que le premier paramètre du modèle sont choisis aléatoirement. 64 trajectoires ont été simulées, pour 100 particules. Les positions initiales suivent une loi gaussienne centrée en (0.25 , 0.5) et de variance 0.05. Le premier paramètre (θ_0) a été choisi uniformément sur $[0, 2.5]$.

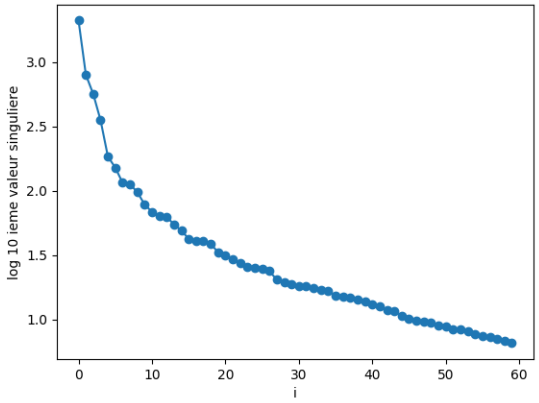


FIGURE 13 – Valeurs singulières pour 500 particules, 64 simulations

D'une part, les valeurs singulières sont très élevées et d'autre part, leur décroissance est plutôt lente.

3.2.3 Champs de Lamb-Oseen

Dans cette simulation : $\Gamma = 10$, $\nu = 0.5$ et $r_c = 0.7$. Le tourbillon est centré en 0 mais nous n'étudions ici que le domaine $(0, 1)^2$.

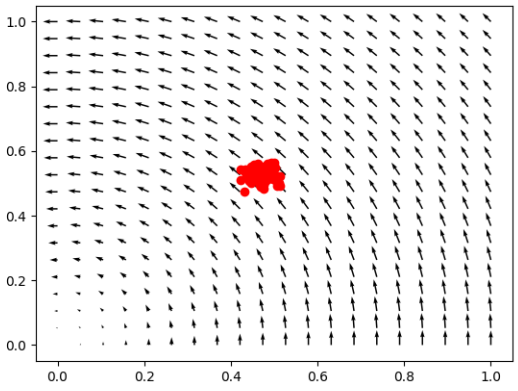


FIGURE 14 – 100 particules dans un champs de Lamb-Oseen

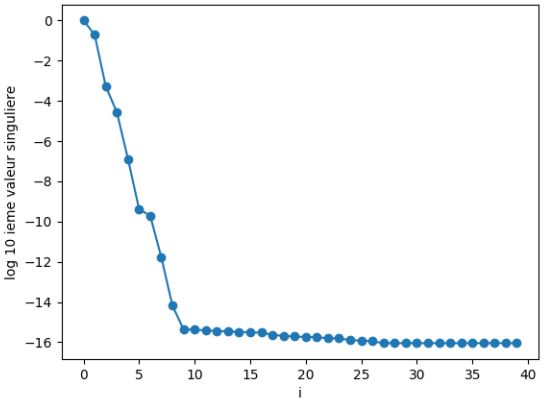


FIGURE 15 – Valeurs singulières pour 100 particules

Comme dans le cas où le champs de vitesse est uniforme, les valeurs singulières de la matrice des snapshots décroissent assez rapidement.

4 Discussion

4.1 Approche Eulerienne

De manière générale, le schéma de Lax-Frierichs diffuse beaucoup et il faudrait un maillage très fin pour que cela ne soit pas visible sur une grande période de temps. Pour diminuer la diffusion, on peut aussi penser à faire une méthode d'ordre 2 (non-linéaire) qui est liée au schéma de Lax-Friedrichs, mais elle serait plus difficile à implémenter et plus coûteuse en temps.

4.1.1 Vitesse constante

La dynamique de la solution change pour chaque valeur de θ. Il faudrait donc faire beaucoup plus de simulations et donc avoir plus de solutions fines pour permettre une bonne réduction (couvrir tout l'espace possible de propagation finalement).
Ceci impliquerait un stockage mémoire encore plus conséquent, et un temps de calcul très grand. Cependant, un nouveau paramètre dans l'intervalle de simulations est plutôt bien reconstruit avec une centaine de modes.

Le coût en temps en amont est donc assez conséquent, et on ne peut pas se permettre autant de simulations fines *a priori*. Enfin, on voit que les modes n'oscillent que sur le cadran supérieur droit. Un angle n'étant pas compris dans l'intervalle sera très mal représenté, c'est aussi une limite de l'algorithme POD, exposée notamment dans le cas de l'écoulement cellulaire.

Dans le cas d'une vitesse constante, l'approche lagrangienne semble donc plus adaptée.

4.1.2 Écoulement cellulaire

Tout comme pour le cas vitesse constante, la dynamique dépend très fortement de la position initiale de la gaussienne ainsi que des paramètres régissant l'écoulement. Cependant, tant que l'on reste "in range", la base POD arrive à bien reconstruire la solution avec un nombre raisonnable de modes propres. On observe les mêmes limites que dans le cas constant, i.e. que les solutions "out of range" trop éloignées du domaine initial ne sont pas très bien rendues.

4.2 Approche Lagrangienne

4.2.1 Vitesse constante

Cependant, Les modes propres dépendent de l'angle θ qui caractérise la vitesse des particules. Si ce paramètre est modifié alors le mode propre est également modifié. L'approche lagrangienne est préferable à l'approche eulerienne dans ce cas.

4.2.2 Écoulement cellulaire

La décroissance des valeurs singulières est assez lente. Un grand nombre de modes propres est donc nécessaire pour donner une bonne approximation de la solution. D'après la figure 13, 60 modes propres sont nécessaires pour obtenir une erreur de l'ordre de 10^{-3}. De plus, dans la seconde étude, l'approximation n'est donc pas satisfaisante puisque les valeurs singulières sont trop élevées. L'approche eulerienne est donc largement préferable dans ce cas de figure.

4.2.3 Champ de Lamb-Oseen

Un faible nombre de modes propres est par conséquent nécessaire pour obtenir une approximation convenable. Ce résultat était prévisible. En effet, la vitesse est uniquement tangentielle, les particules ont donc toutes des trajectoires similaires. Les premiers modes propres servent donc à representer ces trajectoires circulaires. Les autres modes propres ne contiennent que peu d'informations. Là encore, l'approche lagrangienne est plus efficace et plus précise que l'approche eulerienne.

5 Conclusions

Tout au long de ce projet, nous avons donc pu observer que la POD, utilisée dans de bonnes conditions, était non seulement un outil fonctionnel mais surtout indispensable pour modéliser des équations telles que l'équation d'advection en des temps de calcul raisonnables. De plus, nous avons montré qu'il était possible de maîtriser les incertitudes en jeu tout en réduisant la taille des données utilisées, en stockant pour cela moins de modes de la POD.

En revanche, nous avons déterminé une séparation entre les écoulements pour lesquels l'approche lagrangienne était bien plus adaptée (vitesse constante, Lamb-Oseen) et ceux pour lesquels l'approche eulérienne donnait de meilleurs résultats (écoulement cellulaire). Il est interessant de remarquer que l'approche lagrangienne est plus efficace lorsque les particules posèdent des trajectoires similaires, ce qui n'est pas le cas pour des écoulements cellulaires. Nous avons également déterminé ainsi qu'une limitation de l'algorithme dans le cas d'une reconstruction d'une solution fine pour un

paramètre n'étant pas dans l'intervalle de départ choisi pour construire la base de modes propres. Ainsi, il semble que la POD demande de bien choisir l'approche choisie en fonction de l'écoulement observé afin d'obtenir une réduction optimale du modèle, et de se poser la question du nombre de simulations (et donc du temps de calcul en amont) et de quels paramètres pertinents choisir pour créer la base de la POD.

Enfin, grâce à une interpolation d'ordre 1 d'un champ de vitesse discrétisé, nous avons pu entrevoir comment résoudre et compresser un tel modèle à partir de relevés expérimentaux comme on y serait confronté dans une situation réelle.

Ainsi, nous avons remarqué que la réduction de modèles en grande dimension via la POD était une méthode permettant, sous réserve d'un bon choix d'approche et de paramètres, de résoudre rapidement et avec une certaine fiabilité les équations décrivant le transport de polluants.

6 Appendice

6.1 Méthode des volumes finis [2]

Dans le cas de l'approche eulérienne, on utilise un schéma qui est différent de ceux utilisés jusqu'à présent dans nos cours. Détaillons un peu comment cette méthode fonctionne.

Considérons l'équation de transport dans le cas général en dimension $d > 0$ sur un compact en temps et en espace $[0, T] \times [a, b]^d$ pour $T \in \mathbb{R}_+^*$ et $a < b$ dans $\mathbb{R}$

$$\frac{\partial c}{\partial t} + u \cdot \nabla c = f \tag{9}$$

où $c : [0, T] \times [a, b]^d \longrightarrow [0, 1]$ la concentration du polluant, $u : [0, T] \times [a, b]^d \longrightarrow \mathbb{R}^d$ le champ des vitesses imposé dans le milieu et $f : [0, T] \times [a, b]^d \longrightarrow \mathbb{R}$ la source de pollution.

Dans notre étude nous prenons $f = 0$, nous considérons qu'il n'y a pas de source.

6.1.1 Maillage

Nous posons $[a, b] = [0, L]$ afin de simplifier. Maintenant, pour discrétiser l'équation (9) en espace, nous utilisons deux suites arithmétiques $(x_i)_{i=1}^{n_x}$ sur l'axe des abscisses et $(y_i)_{i=1}^{n_y}$ sur l'axe des ordonnées, vérifiant :

$$\begin{cases} x_i = i\Delta x = i\frac{L}{n_x} \\ y_i = i\Delta y = i\frac{L}{n_y} \end{cases}$$

On obtient une grille de $(n_x - 1)(n_y - 1)$ cellules. On appelle alors $K_{i,j} = [x_i, x_{i+1}] \times [y_j, y_{j+1}]$, et $\Gamma_{i,j}^{k,l} = K_{i,j} \cap K_{k,l}$.

On discrétise $[0,T]$ en utilisant la suite $(t_i)_{i=1}^{n_t}$ (qui n'est pas nécessairement arithmétique) vérifiant la condition CFL et :

$$\begin{cases} t_1 = 0 \\ t_{n_t} = T \\ t_i < t_{i+1} \ \forall i < n_t \end{cases}$$

Ainsi, nous posons pour $1 \leq i \leq n_x$, $1 \leq j \leq n_y$ et $1 \leq n \leq n_t$: :

$$\begin{aligned} X = (x,y),\ dX = dxdy \\ |K_{i,j}| = (x_{i+1} - x_i)(y_{j+1} - y_j) = \Delta x \Delta y = \frac{L^2}{n_x n_y} \\ c_{i,j}(t_n) = \frac{1}{|K_{i,j}|} \int_{K_{i,j}} c(t_n, X) dX, \\ u_{i,j}(t_n) = \frac{1}{|K_{i,j}|} \int_{K_{i,j}} u(t_n, X) dX. \end{aligned}$$

6.1.2 Schéma numérique

Nous considérons l'équation (9) dans sa forme faible, c'est à dire au sens des distributions, en utilisons le fait que $\mathrm{div}(c\ u) = u\nabla c + c\ \mathrm{div}(u)$. Nous pouvons la réécrire sous la forme suivante :

$$\frac{\partial c}{\partial t} + \mathrm{div}(c\ u) - c\ \mathrm{div}(u) = 0$$

Dans notre projet, on considère des vitesse de fluides incompressibles : $\mathrm{div} u = 0$. L'équation devient :

$$\frac{\partial c}{\partial t} + \mathrm{div}(c\ u) = 0 \tag{10}$$

Maintenant, en fixant $t_n \in [0,T]$, nous intégrons sur chaque cellule $K_{i,j}$,

$$\frac{1}{|K_{i,j}|} \int_{K_{i,j}} \left(\frac{\partial c}{\partial t} + \mathrm{div}(c\ u) \right) dX = 0$$

On peut permuter l'intégrale et la dérivation par rapport au temps puisque les cellules $K_{i,j}$ sont indépendantes de temps, et on utilise le théorème de Green pour des fonctions suffisamment régulières : $\int_\Omega \mathrm{div}(a)\ d\Omega = \int_{\partial\Omega} a \cdot n\ d$Ł , où n est le vecteur normal sortant. Ainsi :

$$\frac{\partial c_{i,j}}{\partial t}(t_n) + \frac{1}{|K_{i,j}|} \int_{\partial K_{i,j}} c\ u \cdot n\ dL = 0$$

Chaque cellule est un rectangle, ces bords sont donc des segments et nous posons, pour chaque cellule, Γ la quantité $\frac{1}{|\Gamma|} \int_\Gamma c\ u\ dL = F_\Gamma(c)$:

$$\int_{\partial K_{i,j}} c\ u \cdot n\ dL = \sum_{k,l} |\Gamma_{i,j}^{k,l}| F_{\Gamma_{i,j}^{k,l}}(c) \cdot n_{i,j \to k,l}$$

Cette somme contient bien sûr quatre termes puisque les cellules sont des rectangles.

$F_\Gamma \cdot n$ est le flux sortant de la cellule $K_{i,j}$ à partir de la face Γ, on cherche maintenant à l'approcher numériquement de manière stable, on choisit pour cela un schéma de flux en 2-points, c'est à dire,

$$F_{\Gamma_{i,j}^{k,l}}(c) \cdot n_{i,j\to k,l} \simeq g(c_{i,j}, c_{k,l})$$

Le flux sortant de Γ dépend de la valeur de c dans les deux cellules voisines qui contiennent Γ.

La méthode de Godunov permet de donner une fonction g qui assure la stabilité du schéma. En effet, il propose pour des scalaires a et b,

$$g(a,b) = F_{\Gamma_{i,j}^{k,l}}(c_R(t_n, 0)) \cdot n_{i,j\to k,l}$$

où c_R est la solution du problème du Riemann définit par :

$$\begin{cases} \text{trouver c(t, x) telle que} \\ \frac{\partial c}{\partial t} + \text{div}(c\ u) = 0 \\ c(0,x) = a\mathbb{1}_{x<0} + b\mathbb{1}_{x>0} \end{cases} \tag{11}$$

mais vu la difficulté de la résolution du problème exact (11), nous utilisons le schéma de Lax-Friedrichs qui peut être considéré comme une méthode approchée de la méthode de Godunov, plus simple à implémenter et rapide :

$$\begin{cases} \lambda_{i,j\to k,l} = \max\{u_{i,j} \cdot n_{i,j\to k,l}, u_{k,l} \cdot n_{i,j\to k,l}\} \\ g(c_{i,j}, c_{k,l}) = \frac{1}{2}(c_{i,j}u_{i,j} + c_{k,l}u_{k,l}) \cdot n_{i,j\to k,l} + \frac{1}{2\lambda_{i,j\to k,l}}(c_{i,j} - c_{k,l}) \end{cases} \tag{12}$$

Il faut noter que le flux que nous avons utilisé g est un flux conservatif c'est à dire

$$g(c_{i,j}, c_{k,l}) + g(c_{k,l}, c_{i,j}) = 0,$$

ce qui est rassurant vu la nature conservative de l'équation (10).

On utilise pour discrétiser le terme en ∂_t un schéma d'Euler explicite :

$$\frac{\partial c}{\partial t}(t_n) \simeq \frac{c(t_n) - c(t_{n-1})}{t_n - t_{n-1}}$$

Enfin le schéma numérique à implémenter est :
Pour toute cellule $K_{i,j}$

$$c_{i,j}(t_n) = c_{i,j}(t_{n-1}) - \frac{t_n - t_{n-1}}{|K_{i,j}|} \sum_{k,l} |\Gamma_{i,j}^{k,l}| g(c_{i,j}(t_{n-1}), c_{k,l}(t_{n-1}))\ \forall n \tag{13}$$

7 Annexes

7.1 Création du maillage

```
#------------------ Creation du maillage carre en 2 dimensions
    ---------------------

import math
import numpy as np
import matplotlib.pyplot as plt

x0 = 0; y0 = 0 # On definit l'origine du repere 2D (coordonnees de la
    premiere face)
# Domaine de resolution
Lx = 1 # intervalle [x0,Lx] selon x
Ly = 1 # intervalle [y0,Ly] selon y

LL = min(Lx,Ly)
mylevel = 8 # Parametre entier a modifier pour modifier le pas du
    maillage
NN = 2**mylevel

Nx = NN * int(Lx/LL) # nombre de faces selon la direction x
Ny = NN * int(Ly/LL) # nombre de faces selon la direction y

hx = Lx/Nx # pas d'espace selon x
hy = Ly/Ny # pas d'espace selon y
volume = hx*hy # volume d'une cellule

neighbours = 4 # nombre de voisins par cellule

## Initialisation
Ncell = 0   # nombre de cellules
codim0 = np.array([[0,0]]) # chaque ligne stocke les coordonnees (x,y)
    du centre de chaque cellule
Nface = 0   # nombre de frontieres
codim1 = np.array([[0,0,0,0]]) # chaque ligne stocke les coordonnees (x,
    y)
        # de deux points delimitant une face
        # (de la gauche vers la droitepour les faces horizontales
        # du bas vers le haut pour les faces verticales)

codim0to1E = []  # liste des longueurs des frontieres (correspond avec
    codim1)
codim0to1NX = [] # composantes x des normales des faces (correspond avec
     codim1)
codim0to1NY = [] # composantes y des normales des faces (correspond avec
     codim1)
Nghost = 0 # nombre de cellules fantome^88 (au bord)
```

```
codim0to1A = [] # liste des numeros des cellules en amont des frontieres
    correspondant a codim1
codim0to1B = [] # liste des numeros des cellules en aval des frontieres
   correspondant a codim1


## Creation du maillage
# On parcours le maillage ligne par ligne, de la gauche vers la droite,
   et de bas en haut
ny = 0
while ny<=Ny:
    nx = 0
    while nx<=Nx:
        #(numcell = nx + ny*Nx # numero de la cellule (on commence la
            numerotation a 0))

        # ---------------- On ajoute une cellule
            ------------------------------------
        xx = x0 + nx*hx # abscisse du centre de la cellule
        yy = y0 + ny*hy # ordonnee du centre de la cellule

        codim0=np.concatenate((codim0,np.array([[xx,yy]])))

        # --- On cree la face verticale a gauche de la cellule ---
        # Cas du bord x=0
        if nx==0:
            Nface+=1
            codim1=np.concatenate((codim1,np.array([[xx-hx/2,yy-hy/2,xx-
                hx/2,yy+hy/2]])))

            ex = codim1[Nface,2]-codim1[Nface,0] # >0 (face horizontale)
                , ou 0
            ey = codim1[Nface,3]-codim1[Nface,1] # >0 (face verticale),
               ou 0
            assert(ex>=0 and ey>=0)
            E = math.sqrt(ex*ex + ey*ey) # norme de la face
            NX = ey/E; # composante x de la normale a la face (ne pas
               confondre avec Nx!!)
            NY = ex/E; # composante y de la normale a la face (ne pas
               confondre avec Ny!!)
            codim0to1E.append(E)
            codim0to1NX.append(NX)
            codim0to1NY.append(NY)

            # On ajoute les deux cotes de la face
```

```
        codim0to1A.append(Ncell+Nx) # car il n'y a pas de cellule en
            amont
        codim0to1B.append(Ncell) # la cellule que l'on vient de
           creer est en aval
        Nghost+=1


# --- On cree la face verticale a droite de la cellule ---
Nface+=1
codim1=np.concatenate((codim1,np.array([[xx+hx/2,yy-hy/2,xx+hx
    /2,yy+hy/2]])))

ex = codim1[Nface,2]-codim1[Nface,0] # >0 (face horizontale), or
     0
ey = codim1[Nface,3]-codim1[Nface,1] # >0 (face verticale), or 0
assert(ex>=0 and ey>=0)
E = math.sqrt(ex*ex + ey*ey) # norme de la face
NX = ey/E; # composante x de la normale a la face (ne pas
   confondre avec Nx!!)
NY = ex/E; # composante y de la normale a la face (ne pas
   confondre avec Ny!!)
codim0to1E.append(E)
codim0to1NX.append(NX)
codim0to1NY.append(NY)

# Cas du bord x=Nx
if nx==Nx:
        codim0to1A.append(Ncell) # la cellule que l'on vient de
           creer est en amont
        codim0to1B.append(-(Nghost+1)) # car il n'y a pas de cellule
            en aval
        Nghost+=1

else:
        codim0to1A.append(Ncell) # la cellule que l'on vient de
           creer est en amont
        codim0to1B.append(Ncell+1)


# --- On cree la face horizontale en-dessous de la cellule ---
# Cas ou y=0
if ny==0:
        Nface+=1
        codim1=np.concatenate((codim1,np.array([[xx-hx/2,yy-hy/2,xx+
            hx/2,yy-hy/2]])))
```

```
        ex = codim1[Nface,2]-codim1[Nface,0] # >0 (face horizontale)
            , ou 0
        ey = codim1[Nface,3]-codim1[Nface,1] # >0 (face verticale),
            ou 0
        assert(ex>=0 and ey>=0)
        E = math.sqrt(ex*ex + ey*ey) # norme de la face
        NX = ey/E; # composante x de la normale a la face (ne pas
            confondre avec Nx!!)
        NY = ex/E; # composante y de la normale a la face (ne pas
            confondre avec Ny!!)
        codim0to1E.append(E)
        codim0to1NX.append(NX)
        codim0to1NY.append(NY)
        codim0to1A.append(-(Nghost+1)) # il n'y a rien en amont
        codim0to1B.append(Ncell) # la cellule que l'on vient de
            creer est en aval

        Nghost+=1

    # --- On cree la face horizontale au-dessus de la cellule ---
    Nface+=1
    codim1=np.concatenate((codim1,np.array([[xx-hx/2,yy+hy/2,xx+hx
        /2,yy+hy/2]])))

    ex = codim1[Nface,2]-codim1[Nface,0] # >0 (face horizontale), ou
         0
    ey = codim1[Nface,3]-codim1[Nface,1] # >0 (face verticale), ou 0
    assert(ex>=0 and ey>=0)
    E = math.sqrt(ex*ex + ey*ey) # norme de la face
    NX = ey/E; # composante x de la normale a la face (ne pas
        confondre avec Nx!!)
    NY = ex/E; # composante y de la normale a la face (ne pas
        confondre avec Ny!!)
    codim0to1E.append(E)
    codim0to1NX.append(NX)
    codim0to1NY.append(NY)

    # Cas ou y=Ny-1
    if ny==Ny:
        codim0to1A.append(Ncell) # la cellule que l'on vient de
            creer est en amont
        codim0to1B.append(Ncell-Ny*(Nx+1)) # il n'y a rien en aval

        Nghost+=1
```

```
            else:
                codim0to1A.append(Ncell)
                codim0to1B.append(Ncell+Nx+1)

            # On passe a la cellule suivante
            Ncell+=1
            nx+=1
        ny+=1

codim0=codim0[1:,:]
codim1=codim1[1:,:]

'''
## Visualisation et verification du maillage:
    #En noir le maillage, en bleu le passage des face
for i in range(len(codim1[:,1])):
    a=[codim1[i,0],codim1[i,2]]
    b=[codim1[i,1],codim1[i,3]]
    plt.plot(a,b,"black")
    if codim0to1A[i]>=0 and codim0to1B[i]>=0:
        X=[codim0[codim0to1A[i],0],codim0[codim0to1B[i],0]]
        Y=[codim0[codim0to1A[i],1],codim0[codim0to1B[i],1]]
        plt.plot(X,Y,'b')
plt.axis('equal')
plt.show()
'''

## Sauvegarde du maillage
np.save('codim0', codim0)
np.save('codim1', codim1)
np.save('codim0to1A', codim0to1A)
np.save('codim0to1B', codim0to1B)
np.save('codim0to1E', codim0to1E)
np.save('codim0to1NX', codim0to1NX)
np.save('codim0to1NY', codim0to1NY)
np.save('mylevel', mylevel)
```

7.2 Approche Eulerienne

7.2.1 Calcul du flux : solveur de Lax-Friedrichs Riemann

```
def RIEMANN(lambda1,state1,lambda2,state2):
    lambda0=max(abs(lambda1),abs(lambda2)) ## maximal wave speed
    rightf=(lambda1*state1+lambda2*state2+lambda0*(state1-state2))/2.
    leftf=-rightf ## conservative, entropy flux=0
    return [leftf,rightf,lambda0] ## end of Lax-Friedrichs Riemann
```

solver

7.2.2 Volumes finis - cas constant

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from riemann_scal import *

## Recuperation des parametres du maillage
codim0=np.load('codim0.npy')
codim1=np.load('codim1.npy')
codim0to1A=np.load('codim0to1A.npy')
codim0to1B=np.load('codim0to1B.npy')
codim0to1E=np.load('codim0to1E.npy')
codim0to1NX=np.load('codim0to1NX.npy')
codim0to1NY=np.load('codim0to1NY.npy')
x0 = 0.0; y0 = 0.0 # origine du repere 2D
Lx = 1.0; Ly = 1.0 # domaine de resolution [x0,x0+Lx]*[y0,y0+Ly]

LL = min(Lx,Ly)
mylevel = np.load('mylevel.npy')
NN = 2**mylevel

Nx = NN * int(Lx/LL) # nombre de faces selon la direction x
Ny = NN * int(Ly/LL) # nombre de faces selon la direction y

hx = Lx/Nx # pas d'espace selon x
hy = Ly/Ny # pas d'espace selon y
volume = hx*hy # volume d'une cellule

neighbours = 4 # nombre de voisins par cellule

## Parametres

t = 0
Tfinal = 1 # Temps final d'une simulation

ns = 16 # nombre de simulations
THETA=np.linspace(0,np.pi/2,ns) # differents angles de vitesse entre 0
    et pi/2

normev = 0.5 # cas constant : norme de la vitesse

x0,y0=0.25,0.25 # position initiale de la gaussienne
```

```
sigma = 1/50 # largeur de la gaussienne

# Definition de la vitesse
def ux(theta):
    return normev*np.cos(theta)

def uy(theta):
    return normev*np.sin(theta)

## Initialisation

def initialisation(sigma,x0,y0):
    qini=np.zeros(codim0.shape[0])
    for i in range(len(qini)):
        x,y=codim0[i]
        qini[i]=np.exp(-((x-x0)**2+(y-y0)**2)/(2*sigma**2))
    return qini

qini=initialisation(sigma,x0,y0) # Condition initiale




compteur=0
qhist=[]

## Boucle sur les differents angles de la vitesse
for theta in THETA : # Cas vitesse constante, pas uniforme sur les
   angles
    print(theta)
    qhist.append(qini.copy()) # Recuperation des vecteurs solutions a
       chaque instant de la simulation

    q0 = qini.copy()

    q1 = qini.copy()

    compteur+=1
    print (compteur)
    t=0

    uxtheta = ux(theta)
    uytheta = uy(theta)
```

```
    ## Boucle temporelle
    while t<Tfinal:
        k = 1000
        flux=np.zeros(codim0.shape[0]) # Matrice des flux
        # On parcours toutes les faces
        for iface in range(codim1.shape[0]):
            i=codim0to1A[iface] # Cellule en amont
            j=codim0to1B[iface] # Cellule en aval
            if i<0 or j<0: # Conditions de bord periodiques
                continue
            # Vitesse du fluide en i et j projete selon la normale a la
                face
            lambdai = uxtheta*codim0to1NX[iface]+uytheta*codim0to1NY[
                iface]
            lambdaj = uxtheta*codim0to1NX[iface]+uytheta*codim0to1NY[
                iface]
            statei = q0[i] # Concentration en i
            statej = q0[j] # Concentration en j
            # Schema de Lax-Friedriech: calcul du flux en i et en j a
                travers la face
            [leftf,rightf,Lambda] = RIEMANN(lambdai,statei,lambdaj,
                statej)
            # Mise a jour des flux
            flux[i]+=leftf*codim0to1E[iface]
            flux[j]+=rightf*codim0to1E[iface]
            # Calcul du pas de temps associz
            k=min(k,volume/(2*(hx+hy)*Lambda))
        # Mise a jour de la concentration
        q1+=flux*(k/volume)
        qhist.append(q1.copy())
        q0 = q1.copy()
        t+=k
    # Remplissage d'une matrice compose des vecteurs solutions que l'on
        va rajouter a la matrice de snapshots

SNAPSHOTMATRIX=np.zeros((codim0.shape[0],len(qhist)))
for ligne in range(codim0.shape[0]):
    for colonne in range(len(qhist)):
        SNAPSHOTMATRIX[ligne,colonne]=qhist[colonne][ligne]

print (SNAPSHOTMATRIX.shape[0]) # Compte le nombre de snapshots
np.save('SNAPSHOTMATRIX',SNAPSHOTMATRIX) # Sauvegarde. Attention a
    changer le nom pour differentes simulations
```

```
## Animation 3D

for q in qhist:
    plt.clf()
    fig = plt.figure(1)
    ax = fig.gca(projection='3d')
    X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
    Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
    x=X.reshape(Nx+1,Ny+1)
    y=Y.reshape(Nx+1,Ny+1)
    ax.plot_wireframe(x,y,q.reshape(Nx+1,Ny+1))#,False)
    plt.pause(0.1)

from math import *
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.optimize import minimize

## RÃ©cupÃ©ration des paramÃ¨tres du maillage

# Domaine de resolution
Lx = 1 # intervalle [x0,Lx] selon x
Ly = 1 # intervalle [y0,Ly] selon y

LL = min(Lx,Ly)
mylevel = np.load('mylevel.npy') # Parametre entier a modifier pour
   modifier le pas du maillage
          # Vaut 5 dans le code scilab
NN = 2**mylevel

Nx = NN * int(Lx/LL) # nombre de faces selon la direction x
Ny = NN * int(Ly/LL) # nombre de faces selon la direction y

hx = Lx/Nx # pas d'espace selon x
hy = Ly/Ny # pas d'espace selon y
volume = hx*hy # volume d'une cellule

neighbours = 4 # nombre de voisins par cellule

codim0=np.load('codim0.npy')
codim1=np.load('codim1.npy')
codim0to1A=np.load('codim0to1A.npy')
codim0to1B=np.load('codim0to1B.npy')
```

```
codim0to1E=np.load('codim0to1E.npy')
codim0to1NX=np.load('codim0to1NX.npy')
codim0to1NY=np.load('codim0to1NY.npy')

Ur = np.load('ur_eulerien_constant.npy')
Cr = np.load('SNAPSHOTMATRIX.npy')

r = Ur.shape[1]

sol = []

## Programme d'optimisation

for t in range(50,51):
    def objective(a):
        s = 0
        for k in range(Nx):
            sk = 0
            for l in range(Ny):
                sl = 0
                for j in range(r):
                    sl += a[j]*Ur[k+l*(Nx+1),j]
                sk += (Cr[k+l*(Nx+1),t] - sl)**2
            s += sk
        return s

    # initial guesses
    x0 = np.zeros(r)

    # show initial objective
    print('Initial Objective: ' + str(objective(x0)))

    # optimize
    def constraint(a):
        return Ur @ a

    con = {'type': 'ineq', 'fun': constraint}

    solution = minimize(objective,x0,method='SLSQP',constraints=con)
    x = solution.x

    sol.append(x)

    # show final objective
    print('Final Objective: ' + str(objective(x)))
```

```
V = [Ur @ q for q in sol]

for q in V:
    plt.clf()
    fig = plt.figure(1)
    ax = fig.gca(projection='3d')
    X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
    Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
    x=X.reshape(Nx+1,Ny+1)
    y=Y.reshape(Nx+1,Ny+1)
    ax.plot_wireframe(x,y,q.reshape(Nx+1,Ny+1))#,False)
    plt.show()

from math import *
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from random import *

from riemann_scal import *

codim0=np.load('codim0.npy')
codim1=np.load('codim1.npy')
codim0to1A=np.load('codim0to1A.npy')
codim0to1B=np.load('codim0to1B.npy')
codim0to1E=np.load('codim0to1E.npy')
codim0to1NX=np.load('codim0to1NX.npy')
codim0to1NY=np.load('codim0to1NY.npy')

x0 = -0.5; y0 = -0.5 # origine du repere 2D
Lx = 2.0; Ly = 2.0 # domaine de resolution [x0,x0+Lx]*[y0,y0+Ly]

LL = min(Lx,Ly)
mylevel = np.load('mylevel.npy') # Parametre entier a modifier pour
   modifier le pas du maillage
          # Vaut 5 dans le code scilab
NN = 2**mylevel

Nx = NN * int(Lx/LL) # nombre de faces selon la direction x
Ny = NN * int(Ly/LL) # nombre de faces selon la direction y

hx = Lx/Nx # pas d'espace selon x
hy = Ly/Ny # pas d'espace selon y
volume = hx*hy # volume d'une cellule
```

```
neighbours = 4 # nombre de voisins par cellule

v0=2#0.75
vx=1/2
vy=1/2
k=0.25
x_0,y_0=0.25,0.25


"""
# Liste des conditions initiales et parametres
X0=[[0.25,0.25],[0.5,0.5],[0.5,0.25]]
V0=[0,0.5]
VX=[1,3]
CI=[]
for a in X0:
    for b in V0:
        for c in VX:
            for d in VX:
                CI.append([a[0],a[1],b,c,d])

"""

# Liste des conditions initiales et parametres
CI=[]
for i in range(64):
    X0=random()*0.5+0.25
    Y0=random()*0.5+0.25
    Vx=0.5
    Vy=0.5
    V0=random()*2.5
    CI.append([X0,Y0,V0,Vx,Vy])

np.save('CI',CI)

#print(CI)
CI=[[0,0,1,0.5,0.5]]

sigma=1/20

def initialisation(sigma,x1,y1):
    qini=np.zeros(codim0.shape[0])
    for i in range(len(qini)):
        x,y=codim0[i]
```

```
        qini[i]=np.exp(-((x-x1)**2+(y-y1)**2)/(2*sigma**2))
    return qini

def ux(icell,v0,vx,vy):
    x,y=codim0[icell]
    return  2*pi*sin(2*pi*x)*cos(2*pi*y)-2*pi*vy*v0*cos(2*pi*vx*x)*sin
        (2*pi*vy*y)

def uy(icell,v0,vx,vy):
    x,y=codim0[icell]
    return -2*pi*sin(2*pi*y)*cos(2*pi*x)+2*pi*vx*v0*cos(2*pi*vy*y)*sin
        (2*pi*vx*x)

def uux(x,y,v0,vx,vy):
    u=  2*pi*sin(2*pi*x)*cos(2*pi*y)-2*pi*vy*v0*cos(2*pi*vx*x)*sin(2*pi*
        vy*y)
    return u

def uuy(x,y,v0,vx,vy):
    u= -2*pi*sin(2*pi*y)*cos(2*pi*x)+2*pi*vx*v0*cos(2*pi*vy*y)*sin(2*pi*
        vx*x)
    return u



#qhist = [qini.copy()]

#q0 = qini.copy()

#q1 = qini.copy()

"""
fig = plt.figure(5)
ax = fig.gca(projection='3d')
X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
x=X.reshape(Nx+1,Ny+1)
y=Y.reshape(Nx+1,Ny+1)
ax.plot_wireframe(x,y,qhist[0].reshape(Nx+1,Ny+1))#,False)
plt.pause(0.1)
"""

t = 0
Tfinal = 0.5
```

```
#print(codim0.shape[0],(Nx+1)*(Ny+1))



X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
x=X.reshape(Nx+1,Ny+1)
y=Y.reshape(Nx+1,Ny+1)
plt.figure(1)
U=np.array([ux(i,v0,vx,vy) for i in range(codim0.shape[0])])
V=np.array([uy(i,v0,vx,vy) for i in range(codim0.shape[0])])
Q = plt.quiver(x,y,U.reshape(Nx+1,Ny+1),V.reshape(Nx+1,Ny+1))#, units='
    width')
#qk = plt.quiverkey(Q, 0.9, 0.9, 2, r'$2 \frac{m}{s}$', labelpos='E',
    coordinates='figure')
#plt.title('Ecoulement cellulaire')
plt.xlabel('x')
plt.ylabel('y')
plt.axis('equal')
plt.show()
plt.figure(2)
U=np.array([sqrt(ux(i,v0,vx,vy)**2+uy(i,v0,vx,vy)**2) for i in range(
    codim0.shape[0])])
CS = plt.contour(x, y, U.reshape(Nx+1,Ny+1))
plt.clabel(CS, inline=1, fontsize=10)
plt.plot([0.25,0.25,0.75,0.75,0.25],[0.25,0.75,0.75,0.25,0.25],'r')
plt.xlabel('x')
plt.ylabel('y')
plt.axis('equal')
plt.title('Norme de la vitesse')
plt.show()
plt.figure(3)
U=np.array([ux(i,v0,vx,vy) for i in range(codim0.shape[0])])
CS = plt.contour(x, y, U.reshape(Nx+1,Ny+1))
plt.clabel(CS, inline=1, fontsize=10)
plt.plot([0.25,0.25,0.75,0.75,0.25],[0.25,0.75,0.75,0.25,0.25],'r')
plt.xlabel('x')
plt.ylabel('y')
plt.axis('equal')
plt.title('ux')
plt.show()
plt.figure(4)
V=np.array([uy(i,v0,vx,vy) for i in range(codim0.shape[0])])
```

```
CS = plt.contour(x, y, V.reshape(Nx+1,Ny+1))
plt.clabel(CS, inline=1, fontsize=10)
plt.plot([0.25,0.25,0.75,0.75,0.25],[0.25,0.75,0.75,0.25,0.25],'r')
plt.xlabel('x')
plt.ylabel('y')
plt.axis('equal')
plt.title('uy')
plt.show()
plt.pause(0.1)



t = 0
Tfinal = 0.1
compteur=0
qhist=[]
#SNAPSHOTMATRIX=np.zeros((codim0.shape[0],1))

# boucle sur les conditions initiales
for ci in CI:
    compteur+=1
    print(compteur,ci)
    t=0
    [x_0,y_0,v0,vx,vy]=ci
    ## Initialisation
    #qini = np.zeros(codim0.shape[0])
    qini=initialisation(sigma,x_0,y_0)
    qhist.append(qini.copy())
    q0 = qini.copy()
    q1 = qini.copy()

    ## Boucle temporelle
    while t<Tfinal:
        k = 1000
        flux=np.zeros(codim0.shape[0]) # Matrice des flux
        # On parcours toutes les faces
        for iface in range(codim1.shape[0]):
            i=codim0to1A[iface] # Cellule en amont
            j=codim0to1B[iface] # Cellule en aval
            if i<0 or j<0: # Conditions de bord periodiques
                continue
            # Vitesse du fluide en i et j projete selon la normale a la
                face
            lambdai = ux(i,v0,vx,vy)*codim0to1NX[iface]+uy(i,v0,vx,vy)*
                codim0to1NY[iface]
```

```
            lambdaj = ux(j,v0,vx,vy)*codim0to1NX[iface]+uy(j,v0,vx,vy)*
                codim0to1NY[iface]
            statei = q0[i] # Concentration en i
            statej = q0[j] # Concentration en j
            # Schema de Lax-Friedriech: calcul du flux en i et en j a
                travers la face
            [leftf,rightf,Lambda] = RIEMANN(lambdai,statei,lambdaj,
                statej)
            # Mise a jour des flux
            flux[i]+=leftf*codim0to1E[iface]
            flux[j]+=rightf*codim0to1E[iface]
            # Calcul du pas de temps associe
            k=min(k,abs(volume/(2*(hx+hy)*Lambda)))
        # Mise a jour de la concentration
        q1+=flux*(k/volume)
        qhist.append(q1.copy())
        q0 = q1.copy()
        t+=k

print('fini')


SNAPSHOTMATRIX=np.zeros((codim0.shape[0],len(qhist)))
for ligne in range(codim0.shape[0]):
    for colonne in range(len(qhist)):
        SNAPSHOTMATRIX[ligne,colonne]=qhist[colonne][ligne]

print(SNAPSHOTMATRIX.shape[0]) # Compte le nombre de snapshots
np.save('snapshot'+str(CI[0]),SNAPSHOTMATRIX) # Sauvegarde. Attention a
    changer le nom pour differentes simulations


"""
M=np.load('SNAPSHOT_CELLULARFLOW.npy')

u,s,v=np.linalg.svd(M,False) # Thin SVD

np.save('u_eulerien_cellulaire',u)

np.save('s_eulerien_cellulaire',s)
"""

## Animation 3D
for q in qhist:
    fig = plt.figure(6)
```

```
        plt.clf()
        ax = fig.gca(projection='3d')
        X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
        Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
        x=X.reshape(Nx+1,Ny+1)
        y=Y.reshape(Nx+1,Ny+1)
        ax.plot_wireframe(x,y,q.reshape(Nx+1,Ny+1))#,False)
        plt.pause(0.1)
        plt.show()

from math import *
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


r = 500 # nombre de modes pour la POD
x0 = -0.5; y0 = -0.5 # origine du repere 2D
Lx = 2.0; Ly = 2.0 # domaine de resolution [x0,x0+Lx]*[y0,y0+Ly]

LL = min(Lx,Ly)
LL = min(Lx,Ly)
mylevel = np.load('mylevel.npy') # Parametre entier a modifier pour
    modifier le pas du maillage
              # Vaut 5 dans le code scilab
NN = 2**mylevel

Nx = NN * int(Lx/LL) # nombre de faces selon la direction x
Ny = NN * int(Ly/LL) # nombre de faces selon la direction y

hx = Lx/Nx # pas d'espace selon x
hy = Ly/Ny # pas d'espace selon y
volume = hx*hy # volume d'une cellule

codim0=np.load('codim0.npy')
codim1=np.load('codim1.npy')
codim0to1A=np.load('codim0to1A.npy')
codim0to1B=np.load('codim0to1B.npy')
codim0to1E=np.load('codim0to1E.npy')
codim0to1NX=np.load('codim0to1NX.npy')
codim0to1NY=np.load('codim0to1NY.npy')
CI=np.load('CI.npy')

print(CI)
```

```
v0=0#0.75
vx=0.5
vy=0.5
k=0.25
x_0,y_0=0.5,0.25

sigma=1/20

def initialisation(sigma,x1,y1):
    qini=np.zeros(codim0.shape[0])
    for i in range(len(qini)):
        x,y=codim0[i]
        qini[i]=np.exp(-((x-x1)**2+(y-y1)**2)/(2*sigma**2))
    return qini



def ux(icell):
    x,y=codim0[icell]
    return  2*pi*sin(2*pi*x)*cos(2*pi*y)-2*pi*vy*v0*cos(2*pi*vx*x)*sin
       (2*pi*vy*y)

def uy(icell):
    x,y=codim0[icell]
    return -2*pi*sin(2*pi*y)*cos(2*pi*x)+2*pi*vx*v0*cos(2*pi*vy*y)*sin
       (2*pi*vx*x)


u=np.load('u_eulerien_cellulaire2.npy')

print(u.shape)

ur=u[:,:r]
np.save('u_eulerien_cellulaire_reduit2',ur)


ur=np.load('u_eulerien_cellulaire_reduit2.npy')
s=np.load('s_eulerien_cellulaire2.npy')




# Plot des valeurs singulieres
plt.plot([i for i in range(0,r,10)],[np.log(s[i]/s[0]) for i in range(0,
```

```
    r,10)],'-o')
plt.xlabel('Indice des valeurs singulieres')
plt.ylabel('log10 des valeurs singulieres')
plt.show()
"""
# Modes propres
fig=plt.figure(2)
ax=fig.gca(projection='3d')
X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
x=X.reshape(Nx+1,Ny+1)
y=Y.reshape(Nx+1,Ny+1)
ax.plot_wireframe(x,y,u[:,0].reshape(129,129))
plt.show()
"""

q0=initialisation(sigma,0.5,0.25)

A0 = ur.T @ q0
q = ur @ A0

fig = plt.figure(5)
ax = fig.gca(projection='3d')
X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
x=X.reshape(Nx+1,Ny+1)
y=Y.reshape(Nx+1,Ny+1)
ax.plot_wireframe(x,y,q.reshape(Nx+1,Ny+1))#,False)
plt.pause(0.01)

"""
# matrice identite
Ir=np.zeros((r,r))
for i in range(r):
    Ir[i,i]=1

I=codim0.shape[0]

B=np.zeros((I,I))
for i in range(I):
    B[i,(i+Nx+1)%I]=1/hx/2*ux(i)
    B[i,(i+1)%I]=1/hx/2*uy(i)
    B[i,(i-1)%I]=-1/hx/2*uy(i)
    B[i,(i-Nx-1)%I]=-1/hx/2*ux(i)
```

```
Bt = ur.T @ B @ ur


P=1000
Tfinal = 0.1
dt=Tfinal/P

## Initialisation
qini = np.zeros(codim0.shape[0])

for icell in range(codim0.shape[0]):
    xi,yi=codim0[icell]
    norme=sqrt((xi-x_0)**2+(yi-y_0)**2)
    if norme<k:
        qini[icell]=exp(-1/(k**2-norme**2))
q0 = qini.copy()
q1 = qini.copy()


# Initialisation
A0 = ur.T @ q0
A = A0

Atot = np.zeros((r, P+1));
Atot[:,0]=A0

for p in range(1,P+1):
    A=np.linalg.solve(Ir-dt*Bt,A0)
    Atot[:,p]=A
    A0=A

Vr = ur @ Atot # matrice des snapshots

print(Vr.shape)

## Animation 3D
for s in range(1):
    q=Vr[:,s]
    fig = plt.figure(5)
    plt.clf()
    ax = fig.gca(projection='3d')
    X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
    Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
    x=X.reshape(Nx+1,Ny+1)
```

```
    y=Y.reshape(Nx+1,Ny+1)
    ax.plot_wireframe(x,y,q.reshape(Nx+1,Ny+1))#,False)
    plt.pause(0.01)
"""
```

7.2.3 Decomposition SVD

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

## Declaration des parametres et recuperation des donnees du maillage

nbmodes = 25 #nombre de modes pour la POD

# Domaine de resolution
Lx = 1 # intervalle [x0,Lx] selon x
Ly = 1 # intervalle [y0,Ly] selon y

LL = min(Lx,Ly)
mylevel = np.load('mylevel.npy')
NN = 2**mylevel

Nx = NN * int(Lx/LL) # nombre de faces selon la direction x
Ny = NN * int(Ly/LL) # nombre de faces selon la direction y

hx = Lx/Nx # pas d'espace selon x
hy = Ly/Ny # pas d'espace selon y
volume = hx*hy # volume d'une cellule

codim0=np.load('codim0.npy')
codim1=np.load('codim1.npy')
codim0to1A=np.load('codim0to1A.npy')
codim0to1B=np.load('codim0to1B.npy')
codim0to1E=np.load('codim0to1E.npy')
codim0to1NX=np.load('codim0to1NX.npy')
codim0to1NY=np.load('codim0to1NY.npy')


## Recuperation de la matrice de snapshots

M=np.load('SNAPSHOTMATRIX.npy')

## Decomposition SVD

u,s,v=np.linalg.svd(M,False) # Thin SVD
```

```
## Trace des valeurs singulieres
plt.plot([i for i in range(nbmodes)],[np.log(s[i]/s[0]) for i in range(
    nbmodes)])
plt.show()


## Trace du premier mode propre en 3D

fig=plt.figure(1)
ax=fig.gca(projection='3d')
X=np.array([codim0[i,0] for i in range(codim0.shape[0])])
Y=np.array([codim0[i,1] for i in range(codim0.shape[0])])
x=X.reshape(Nx+1,Ny+1)
y=Y.reshape(Nx+1,Ny+1)
ax.plot_wireframe(x,y,-u[:,0].reshape(Nx+1,Ny+1))
plt.show()

## Sauvegarde de la decomposition

np.save('u_eulerien_constant',u)
np.save('s_eulerien_constant',s)

## Recuperation et sauvegarde de la base de vecteurs propres

ur=u[:,[i for i in range(13)]]

np.save('ur_eulerien_constant',ur)
```

7.3 Approche Lagrangienne

7.3.1 Champs de vitesse uniforme

```
import scipy as sc
import numpy as np
from scipy import linalg as LS
from numpy import linalg as LA
import matplotlib.pyplot as plt
import time

# ---------- CODE DE CALCUL POUR LE CHAMP DE VITESSE UNIFORME
    ----------


dt=1/256
epsilon=0.000001
nb_steps=256
```

```
nbParticles=8192
center=[0.25,0.25]
cov=1/50
dx=dy=1/2**8

vitesse=0.5;




def norme(v):
    return np.sqrt(v[0]**2+v[1]**2)


def v(theta):
    return np.array([vitesse*np.cos(theta),vitesse*np.sin(theta)])

class particle:
    x=0
    x0=0
    y=0
    y0=0
    vx=0
    vy=0
    def __init__(self):
        self.x=self.y=0


Particles=[]
for i in range(nbParticles):
    Particles.append(particle())
    # Les particules sont initialement disposees selon une gaussienne
       cetree en (0.25 , 0.25) de variance 0.02
    Particles[i].x0=Particles[i].x=np.random.normal(center[0],cov)
    Particles[i].y=Particles[i].y0=np.random.normal(center[1],cov)

def point_fixe(A,k, theta):
    f_k = A + dt*v(theta)/2
    t = dt*(k+1)

    B = f_k + dt*v(theta)/2

    while norme( B-A) > epsilon :
        A,B = B, f_k + dt*v(theta)/2
    return B
```

```
def resoudre(X0,n,theta):
    X = np.array([ X0 for k in range(n)] )
    for k in range(n-1):
        X[k+1] = point_fixe(X[k],k,theta)

    return X
#
```

```
X,Y=np.meshgrid(np.linspace(0,1,20),np.linspace(0,1,20))

# Valeurs possibles de theta
THETA = np.array([ np.pi*k/32 for k in range(16)]);

#Le calcul se fait ici pour k = 8
k=8;
theta = THETA[k]

allXp=nbParticles*[1]
#Calcul des trajectoires des particules
for i_p in range(len(Particles)):
    p=Particles[i_p]
    allXp[i_p]=resoudre([p.x0,p.y0],nb_steps,theta)

for i_t in range(nb_steps):
    # Affichage du champs de vitesse
    Vx,Vy = [v(theta)],[v(theta)]

    plt.quiver(np.linspace(0,1,20),np.linspace(0,1,20),Vx,Vy)
    # Affichage de la trajectoire de chaque particule
    for i_p in range(len(Particles)):
        p=Particles[i_p]
        plt.plot([p.x],[p.y],marker='o',color='red')


        p.x,p.y=allXp[i_p][i_t]
    plt.show()

    plt.pause(0.02)
    plt.clf()

# Calcul de la matrice des snapshots pour une valeur de theta
```

```
def snap(theta):
    Snapshot = np.zeros([nbParticles*2, nb_steps])
    res=[]
    for i_p in range(nbParticles):
        p=Particles[i_p]
        res=resoudre([p.x0,p.y0],nb_steps,theta)
        for t in range(nb_steps):
            Snapshot[i_p,t] = res[t,0]
            Snapshot[i_p + nbParticles,t] = res[t,1]
    return Snapshot
M=snap(theta);
pos0 = np.copy(M[:,0])
for i in range(nb_steps):
    M[:,i] = M[:,i] -M[:,0]



#On calcule la decomposition SVD de la matrice M
print("Calcul de la decomposition SVD de la matrice M")
Ut,St,Vt = LS.svd(M,False)

#On sauve la matrice M obtenue et les vecteurs singuliers a gauche de M
np.save('M_mat', M)
np.save('Ut_mat', Ut)


#On trace les valeurs singulieres de M en fonction de leur indice
Sigma = np.diag(St)
sig = St[0:20]

#print(sig)

svec = np.zeros(20)
for i in range(0,20):
    svec[i] = i

plt.plot(svec, np.log(sig)/np.log(10),'--o')
plt.xlabel('i')
plt.ylabel('log 10 ieme valeur singuliere')
plt.show()

#Visualiser le premier mode

svec2 = np.zeros(2*nbParticles)
```

```
plt.plot( np.linspace(np.linspace(0,1,20),np.linspace(0,1,20)))
plt.plot( Ut[0,:nbParticles], Ut[0,nbParticles:2*nbParticles])

Vx,Vy=-Ut[:nbParticles,0], -Ut[nbParticles:2*nbParticles,0]


plt.quiver(pos0[:nbParticles],pos0[nbParticles:2*nbParticles],Vx,Vy)

#Calcul de la matrice des snapshots pour les 16 valeurs de theta
A=snap(0)
for k in range(1,8):
    print(k)
    B=snap(THETA[k])
    A = np.concatenate((A,B),axis=1);
np.save('Theta_mat',A)


#------------------------------ INTERPOLATION
    ------------------------------------------


def Lower_left(x,y):
    i = int(np.floor((x+1)/dx))
    j = int(np.floor((1+y)/dy))
    return np.array([i,j])

def phi(k,x,y):
    i,j= Lower_left(x,y)[0],Lower_left(x,y)[1]
    x_tilde = x/dx - i
    y_tilde = y/dy - j
    if k==1:
        return (1-x_tilde)*(1-y_tilde)
    if k==2:
        return x_tilde*(1-y_tilde)
    if k==3:
        return x_tilde*y_tilde
    else :
        return y_tilde*(1-x_tilde)
```

7.3.2 Champs de vitesse issu d'un écoulement cellulaire

```
import scipy as sc
import numpy as np
from scipy import linalg as LS
from numpy import linalg as LA
```

```
import matplotlib.pyplot as plt
import time


# ---------- CODE DE CALCUL POUR LE CHAMP DE VITESSE D'UN ECOULEMENT
    CELLULAIRE ----------

dt=0.002
epsilon=0.00001
nb_steps=256
nbParticles=100
center=[0.25,0.25]
cov=0.02
dx=dy=0.01

# theta est le vecteur contenant les paramÃ¨tres de cette simulation
theta = [0.2, 3.12, 2.69]

def norme(v):
    return np.sqrt(v[0]**2+v[1]**2)

#---------- Champ de vitesse dÃ©rivant d'un Ã©coulement
    cellulaire ----------

def Vx(x,y,theta):
    res = 2*np.pi*np.sin(2*np.pi*x)*np.cos(2*np.pi*y) - theta[0]*2*np.pi
       *theta[2]*np.cos(2*np.pi*theta[1]*x)*np.sin(2*np.pi*theta[2]*y)
    return res

def Vy(x,y,theta):
    res = 2*np.pi*np.sin(2*np.pi*y)*np.cos(2*np.pi*x) - theta[0]*2*np.pi
       *theta[1]*np.cos(2*np.pi*theta[1]*y)*np.sin(2*np.pi*theta[2]*x)
    return -res

def vitesse_cell(x,y,theta):
    return np.array([ Vx(x,y,theta), Vy(x,y,theta)])
#
    ------------------------------------------------------------------------------


class particle:
    x=0
    x0=0
    y=0
    y0=0
    vx=0
```

```
    vy=0
    def __init__(self):
        self.x=self.y=0


Particles=[]
# Les particules sont initialement disposÃ©es selon une gaussienne cetrÃ
  ©e en (0.25 , 0.25) de variance 0.02
for i in range(nbParticles):
    Particles.append(particle())
    Particles[i].x0=Particles[i].x=np.random.normal(center[0],cov)
    Particles[i].y=Particles[i].y0=np.random.normal(center[1],cov)

def point_fixe(A,theta):
    f_k = A + dt*vitesse_cell(A[0],A[1],theta)/2
    B = f_k + dt*vitesse_cell(A[0],A[1],theta)/2
    while norme( B-A) > epsilon :
        A,B = B, f_k + dt*vitesse_cell(A[0],A[1],theta)/2
    return B


def resoudre(X0,n,theta):
    X = np.array([ X0 for k in range(n)] )
    for k in range(n-1):
        X[k+1] = point_fixe(X[k],theta)
    return X
#
```

```
X,Y=np.meshgrid(np.linspace(0,1,100),np.linspace(0,1,100))


allXp=nbParticles*[1]


#Calcul des trajectoires des particules
for i_p in range(len(Particles)):
    p=Particles[i_p]
    allXp[i_p]=resoudre([p.x0,p.y0],nb_steps,theta)

for i_t in range(nb_steps):
    # Affichage du champs de vitesse
    Vx1,Vy1 = vitesse_cell(X,Y,theta)
```

```
        plt.quiver(np.linspace(0,1,100),np.linspace(0,1,100),Vx1,Vy1)

        # Affichage de la trajectoire de chaque particule
        for i_p in range(len(Particles)):
            p=Particles[i_p]
            plt.plot([p.x],[p.y],marker='o',color='red')


            p.x,p.y=allXp[i_p][i_t]
        plt.show()

        plt.pause(0.02)
        plt.clf()

# Calcul de la matrice des snapshots pour une valeur du vecteur theta
def snap(theta):
    Snapshot = np.zeros([nbParticles*2, nb_steps])
    res=[]
    for i_p in range(nbParticles):
        p=Particles[i_p]
        res=resoudre([p.x0,p.y0],nb_steps,theta)
        for t in range(nb_steps):
            Snapshot[i_p,t] = res[t,0]
            Snapshot[i_p + nbParticles,t] = res[t,1]
    return Snapshot


M=snap(theta);
#On calcule la dÃ©composition SVD de la matrice M
print("Calcul de la dÃ©composition SVD de la matrice M")
Ut,St,Vt = LS.svd(M,False)
Sigma = np.diag(St)
sig = St[0:40]

print(sig)

svec = np.zeros(40)
for i in range(0,40):
    svec[i] = i

# Affichage du log des valeurs singuliÃ¨res
plt.plot(svec, np.log(sig)/np.log(10))
plt.xlabel('i')
plt.ylabel('log 10 ieme valeur singuliere')
plt.show()
```

7.3.3 Champs de vitesse de Lamb Oseen

```
import scipy as sc
import numpy as np
from scipy import linalg as LS
from numpy import linalg as LA
import matplotlib.pyplot as plt
import time

# ---------- CODE DE CALCUL POUR LE CHAMP DE VITESSE DE LAMB OSEEN ----------


dt=0.002
epsilon=0.00001
nb_steps=256
nbParticles=100
center=[0.5,0.5]
cov=0.02
dx=dy=0.01

gamma=10
nu=0.5
rc_0=0.7




def norme(v):
    return np.sqrt(v[0]**2+v[1]**2)

def u_theta(x,y):

    r=norme([x,y])

    return [-y/r,x/r]


def v_LO(x,y,t):

    r=norme([x,y])

    rc=np.sqrt(4*nu*t+rc_0**2)
    V=gamma/(2*np.pi*r)*(1-np.exp(-(r/rc)**2))
    return np.array([V*u_theta(x,y)[0],V*u_theta(x,y)[1]])
```

```
class particle:
    x=0
    x0=0
    y=0
    y0=0
    vx=0
    vy=0
    def __init__(self):
        self.x=self.y=0


Particles=[]
for i in range(nbParticles):
    Particles.append(particle())
    # Les particules sont initialement disposÃ©es selon une gaussienne
       cetrÃ©e en (0.25 , 0.25) de variance 0.02
    Particles[i].x0=Particles[i].x=np.random.normal(center[0],cov)
    Particles[i].y=Particles[i].y0=np.random.normal(center[1],cov)

def point_fixe(A, k):
    f_k = A + dt*v_LO(A[0],A[1],k*dt)/2
    t = dt*(k+1)

    B = f_k + dt*v_LO(A[0],A[1],t)/2

    while norme( B-A) > epsilon :
        A,B = B, f_k + dt*v_LO(A[0],A[1],k*dt)/2
    return B


def resoudre(X0,n):
    X = np.array([ X0 for k in range(n)] )
    for k in range(n-1):
        X[k+1] = point_fixe(X[k],k)

    return X
#
```

```
X,Y=np.meshgrid(np.linspace(0,1,20),np.linspace(0,1,20))


allXp=nbParticles*[1]
```

```
#Calcul des trajectoires des particules
for i_p in range(len(Particles)):
    p=Particles[i_p]
    allXp[i_p]=resoudre([p.x0,p.y0],nb_steps)

for i_t in range(nb_steps):
    # Affichage du champs de vitesse
    Vx,Vy=v_LO(X,Y,i_t*dt)

    plt.quiver(np.linspace(0,1,20),np.linspace(0,1,20),Vx,Vy)

    # Affichage de la trajectoire de chaque particule
    for i_p in range(len(Particles)):
        p=Particles[i_p]
        plt.plot([p.x],[p.y],marker='o',color='red')


        p.x,p.y=allXp[i_p][i_t]
    plt.show()

    plt.pause(0.02)
    plt.clf()

# Calcul de la matrice des snapshots
def snap():
    Snapshot = np.zeros([nbParticles*2, nb_steps])
    res=[]
    for i_p in range(nbParticles):
        p=Particles[i_p]
        res=resoudre([p.x0,p.y0],nb_steps)
        for t in range(nb_steps):
            Snapshot[i_p,t] = res[t,0]
            Snapshot[i_p + nbParticles,t] = res[t,1]
    return Snapshot

M=snap();
#On calcule la dÃ©composition SVD de la matrice M
print("Calcul de la dÃ©composition SVD de la matrice M")
Ut,St,Vt = LS.svd(M,False)
Sigma = np.diag(St)
sig = St[0:40]
print(sig)

svec = np.zeros(40)
for i in range(0,40):
```

```
    svec[i] = i

# Affichage du log des valeurs singuliÃ¨res
plt.plot(svec, np.log(sig)/np.log(10))
plt.xlabel('i')
plt.ylabel('log 10 ieme valeur singuliere')
plt.show()


#------------------------------ INTERPOLATION
-------------------------------------------------


def Lower_left(x,y):
    i = int(np.floor((x+1)/dx))
    j = int(np.floor((1+y)/dy))
    return np.array([i,j])

def phi(k,x,y):
    i,j= Lower_left(x,y)[0],Lower_left(x,y)[1]
    x_tilde = x/dx - i
    y_tilde = y/dy - j
    if k==1:
        return (1-x_tilde)*(1-y_tilde)
    if k==2:
        return x_tilde*(1-y_tilde)
    if k==3:
        return x_tilde*y_tilde
    else :
        return y_tilde*(1-x_tilde)
#
-------------------------------------------------------------------------------
```

8 Remerciements

Nous tenons particulièrement à remercier M. Damiano Lombardi et M. Sébastien Boyaval pour leur patience et leur précieuse aide, ainsi que pour toute l'expérience qu'ils nous ont apporté au cours de ce projet. Nous remercions de même l'Ecole Nationale des Ponts et Chaussées pour la mise en œuvre d'un tel projet et les rencontres organisées avec les intervenants du cours d'ouverture de M. Boyaval et M. Lombardi.

Références

[1] V. Ehrlacher, S.Boyaval *Méthodes numériques pour les problèmes en grande dimension.* Cours de l'École nationale des ponts et chaussées, 2018.

[2] S.Boyaval *Numerical Hydrodynamics for the environment Illustrated by river flows* Cours de l'École nationale des ponts et chaussées, 2018.