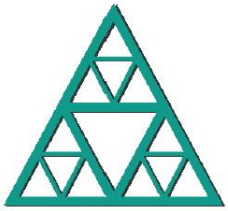
ECOLE NATIONALE DES PONTS ET CHAUSSEES



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 : Predicting the behavior of a pollutant in a fluid is one of the major problem today, but also very complex to do as there are too much variables to describe the phenomenon and thus have a good precision overtime. We have to use reducing models such as POD (*Proper Orthogonal Direct*) in order to compute the solution in a reasonable time. It involves keeping the major dynamic of the solution with a few degrees of liberties 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 real solutions for some parameters to simulate the fundamental dynamic, we use the POD algorithm to compute approximations of solutions for a new set of parameters, this time in a much shorter period. We study two cases : a constant velocity in the fluid and a cell structure within it. We use two approach : an Eulerian one in which we compute the velocity everywhere at any time, and a Lagrangian one in which we follow particles instead and compute their trajectory. We will see which one is better for our cases and discuss why.

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

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.

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*

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 $n_s = 16$ 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 $T_{max} = 1$.
Lors de l'algorithme POD, on se fixe un nombre nb_{modes} que l'on choisit tel que l'erreur sur la solution soit environ de 10^{-3}. Cette erreur a été choisie a posteriori, étant un compromis entre le temps de calcul (nombre de modes) et l'erreur que 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 la gaussienne n'atteigne le bord). On a donc

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

Cas d'un écoulement cellulaire

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 amène un peu plus de phénomène de diffusion, mais qui assure la positivité de la concentration. Pour plus de détails, voir la section 2.4.

Nous avions à notre disposition un code d'éléments finis 2D sous Scilab faisant tourner le schéma de Lax-Friedrichs sur des cas très compliqués 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 :

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

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.

images condition initiale 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.

2.4 Méthode des volumes finis

2.5 Approche Lagrangienne

2.5.1 Description

L'approche lagrangienne consiste à suivre les particules le long de leur trajectoires. Cela revient à résoudre :

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

Où $X(\xi,t) \in \mathbb{R}^{n\times 2}$, n étant le nombre de particules, $t \in [0,T]$ et ξ étant les positions initiales des particules.

2.5.2 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.

2.5.3 Interpolation

La résolution numérique du problème nécessite de connaître le champ de vitesse des particules, noté v. Nous supposons que l'ensemble $\Omega = [-1,1]^2$ 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 que nous avons utilisé pour approximer le champ de vitesse dans l'ensemble du domaine Ω.

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 :

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

2.5.4 Champ de vitesse de LambOseen

Nous prenons dans ce cas une vitesse analytique donnée par :

$$\mathbf{V}(r,\theta,t) = \frac{\Gamma}{2\pi r}(1 - exp(\frac{-r^2}{4\nu t}))$$

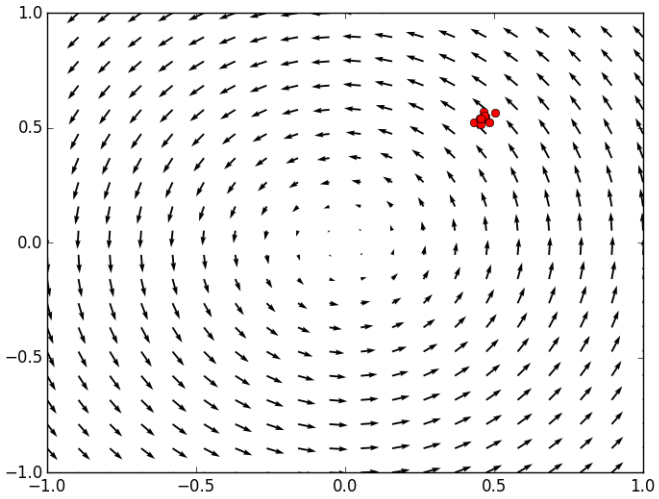


FIGURE 1 –

2.5.5 Champ de vitesse constant

Cette fois, 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 $\frac{\pi}{2}$. Les positions des différentes particules sont ensuite stockées dans une matrice M de taille $2nb_particules \times nb_steps$. Nous avons ensuite effectué la décomposition SVD de la matrice M : $M = USV^T$

Dans ce cas particulier, la projection suivant le premier mode propre (première colonne de U) permet d'obtenir une approximation satisfaisante. Comme on peut le voir sur la figure suivante, la

décroissance des valeurs singulières est très rapide.

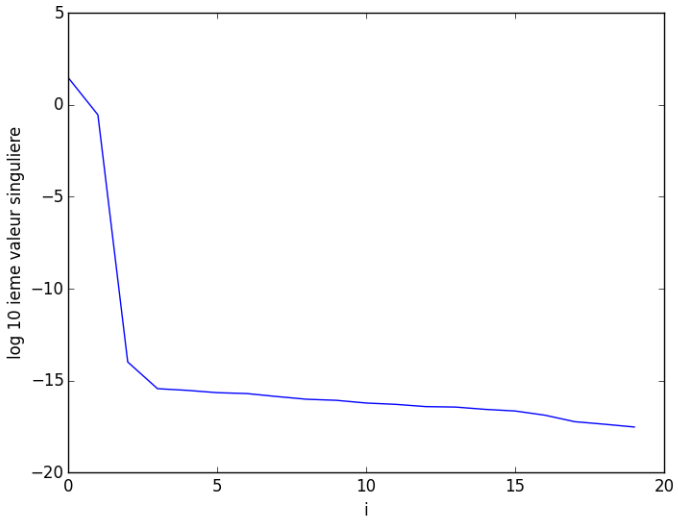


FIGURE 2 – Tracé du log des valeurs singulières en fonction de leur rang

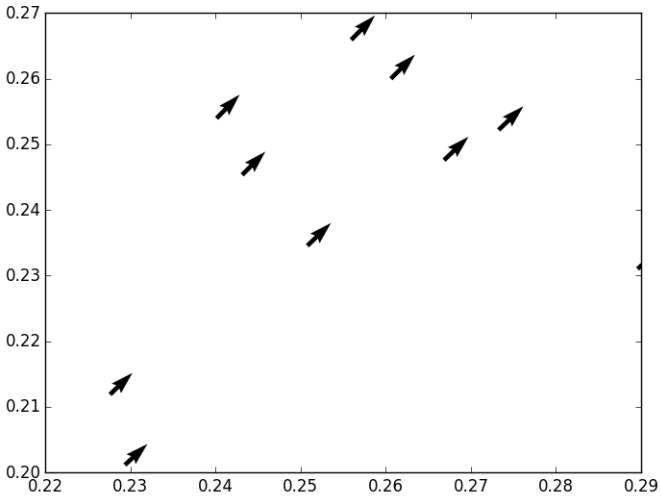


FIGURE 3 – Projection suivant le premier mode

3 Résultats

3.1 Approche Eulerienne

3.1.1 Cas vitesse constante

Voici d'abord le tracé du log des valeurs singulières en fonction de leur rang dans la décomposition SVD.

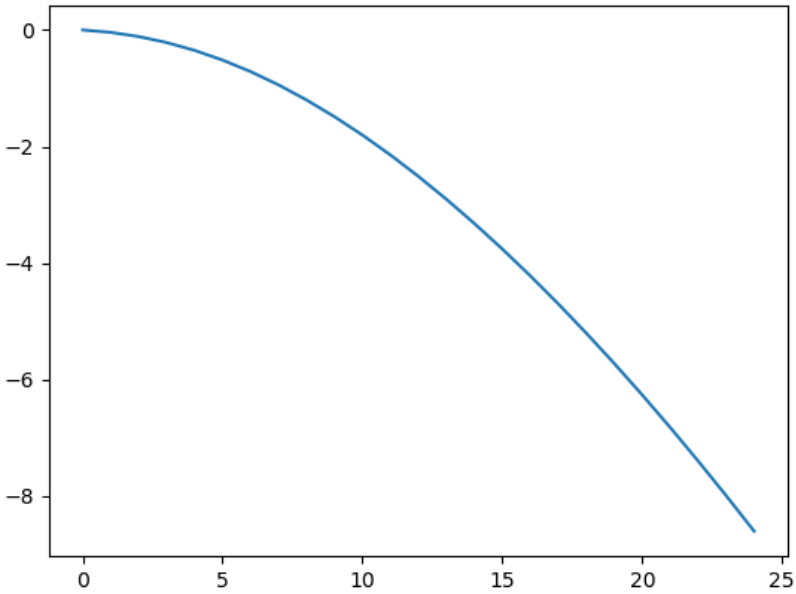


FIGURE 4 – Tracé du log des valeurs singulières en fonction du rang

Ceci nous permet de choisir le nombre de modes qu'on va utiliser, l'erreur étant fixée à 10^{-3} : on choisit $nb_{modes} = 13$.

On récupère donc l'approximation de la matrice de *snapshots* avec 13 vecteurs propres, et on va pouvoir simuler la dynamique de notre système assez fidèlement.

4 Discussion

5 Conclusions

6 Appendice

6.1 Méthode des volumes finis

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 $\mathbb{R}$

$$\frac{\partial c}{\partial t} + u \cdot \nabla c = f. \tag{2}$$

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 champs des vitesses imposé dans le milieu et $f : [0, T] \times [a, b]^d \longrightarrow \mathbb{R}$ la source de pollution.

Pour la suite nous prenons $f = 0$, pas de source.

6.1.1 Maillage

Nous posons $[a, b] = [0, L]$ avec $L > 0$ afin de simplifier. Maintenant, pour discrétiser l'équation (2) 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 donc $K_{i,j} = [x_i, x_{i+1}] \times [y_j, y_{j+1}]$, et $\Gamma_{i,j}^{k,l} = K_{i,j} \cap K_{k,l}$.

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

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

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 (2) 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 des fluides incompressibles, donc $\text{div}(u) = 0$. L'équation devient :

$$\frac{\partial c}{\partial t} + \text{div}(c\ u) = 0 \tag{3}$$

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

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

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 $\int_\Omega \text{div}(a)\ d\Omega = \int_{\partial\Omega} a \cdot n\ dL$,où n est le vecteur normal sortant, ainsi :

$$\frac{\partial c_{i,j}}{\partial t}(t_n) + \frac{1}{|K_{i,j}|} \int_{\partial K_{i,j}} c\ u \cdot n\ dL = 0 \tag{4}$$

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

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

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 (voir la section 6.1.3) et on choisit pour cela un schéma de flux en 2-points, c'est à dire

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

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,

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

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

$$\begin{cases} \text{trouver c(t, x) telle que} \\ \frac{\partial c}{\partial t} + \text{div}(c\ u) = 0 \\ c(0, x) = a\mathbb{1}_{x<0} + b\mathbb{1}_{x>0} \end{cases} \tag{5}$$

mais vu la difficulté de la résolution du problème exacte (5) nous utilisons le schéma de Lax-Friedrichs qui peut être considéré comme une méthode approcher de la méthode de Godunov, plus simple à implémenter et rapide :

$$\begin{cases} \lambda = \max\{u_{i,j}, u_{k,l}\} \\ g(c_{i,j}, c_{k,l}) = \frac{1}{2}(c_{i,j}u_{i,j} + c_{k,l}u_{k,l}) \cdot n_{i,j \to k,l} + \frac{1}{2\lambda}(c_{i,j} - c_{k,l}) \end{cases} \tag{6}$$

6.1.3 Critères de stabilité

Entropie pour la vraie solution
Entropie discrétisé
consistance g(u,u) = F(u)
condition CFL

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(i,theta):
    return normev*np.cos(theta)

def uy(i,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)




SNAPSHOTMATRIX=np.zeros((codim0.shape[0],1))
compteur=0

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

```
q0 = qini.copy()

q1 = qini.copy()

compteur+=1
print (compteur)

## 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,theta)*codim0to1NX[iface]+uy(i,theta)*
            codim0to1NY[iface]
        lambdaj = ux(j,theta)*codim0to1NX[iface]+uy(j,theta)*
            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
M=np.zeros((codim0.shape[0],len(qhist)))
for ligne in range(codim0.shape[0]):
    for colonne in range(len(qhist)):
```

```
                M[ligne,colonne]=qhist[colonne][ligne]
        SNAPSHOTMATRIX=np.concatenate((SNAPSHOTMATRIX,M),axis=1)

SNAPSHOTMATRIX=SNAPSHOTMATRIX[:,1:] # On avait mis une colonne de zeros
    pour initialiser la matrice
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)
```

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)
```

8 Remerciements

9 Exemples de figures, tableau, équations...

Ce court texte illustre la façon de citer les références [1, **?**, **?**, **?**, **?**, **?**], de faire une note de bas de page [1], de faire référence à une partie du document (Sec. 4), à une figure (Figure **??**), à un tableau (Tableau **??**) ou à une équation (Equation 7).

$$H\psi = i\hbar\frac{\partial\psi}{\partial t}. \tag{7}$$

1. Exemple de note de bas de page.

Références

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