PROJET DE GROUPE EN UPICI

(Utilisation de plateforme industrielle pour le calcul intensif)

M1 EDP-MODELISATION-APPROXIMATION

RÉSOLUTION DU SYSTÈME D'ÉQUATIONS DE SAINT-VENANT

Présenté par

RAPHAEL GARNAUD

YOUSSEF FELLOUS

FLORIAN ROBERT

Encadré par

PR. YVES COUDIERE

– DÉCEMBRE 2018 –

Table des matières

Remerciements

Nous voudrons exprimer nos vifs remerciements à notre encadrant Pr. YVES COUDIERE, qui a encadré ce projet avec beaucoup de patience et de gentillesse. Il a su motiver chaque étape de notre travail par des remarques pertinentes .Nous le remercions très sincèrement pour sa disponibilité et son accueil chaleureux chaque fois que nous avons des problèmes ou lorsque nous avons besoin des conseils.

Chapitre 1

Introduction

Le système de Saint-Venant est un système d'équations aux dérivées partielles non linéaires et hyperboliques intervenant dans la résolution de nombreux problèmes de modélisation physique. Ainsi est-il, par exemple, de la modélisation d'un tsunami, d'une avalanche ou d'une rupture de barrage.

La forme générale de l'équation physique complète est la suivante :

$$\begin{cases} \partial_t w + \partial_x Q = 0 \\ \partial_t Q + \partial_x \left(\dfrac{Q^2}{h} + g\dfrac{h^2}{2} \right) = 0 \end{cases}$$

où h est la hauteur d'eau et Q le flux. D'autres paramètres peuvent être pris en compte dans la littérature, comme le facteur de forme (dépend du fluide et du cas), le frottement au fond, voire la topographie ou forme du fond. Dans le cadre de c

On ne peut pas résoudre ce système analytiquement dans le cas général. Par conséquent, une résolution numérique de ce système s'impose. La simulation de l'écoulement d'eau peu profonde à surface libre revient à résoudre le système de Saint-Venant à l'aide d'un schéma numérique robuste c'est-à-dire capable de donner une solution numérique proche de la réalité quelles que soient les particularités de l'écoulement.

On peut citer de nombreuses applications soit dans le cadre des écoulements d'eaux peu profondes en présence d'un traceur, soit l'aménagement des ressources en eau, soit la protection de l'environnement et de l'écosystème : la simulation des écoulements dus à la rupture d'un barrage, la simulation du processus de changement du lit d'une rivière, la simulation des écoulements et du transport sédimentaire ou des polluants en milieux estuariens et côtiers, etc... Ces problèmes régis par les équations ont été beaucoup traités dans le cadre de la mécanique des fluides, en particulier on cite le travail de Stoker [**?**] et Ouazar [**?**]. En 1957, Stoker [**?**] a résolu les problèmes de propagation des ondes en eau peu profonde, avec termes de pente et de frottement et plus récemment Ouazar en 1999 [**?**] a résolu les problèmes d'écoulements d'eau dans les nappes et notamment de l'intrusion d'eau salée dans les nappes côtières.

Notre travail s'inspire fortement du livre de Randall J. Leveque, *Numerical methods for conservation laws* [**?**] mais nous avons compulsé des informations d'autres PDF. En l'occurence, le manuel *Mathématiques appliquées L3* d'Alain Yger [**?**] et la thèse de Vivien Desveaux sur les méthodes numériques [**?**] nous ont rendu de grands services, notamment pour le traitement du cas non-linéaire.

Chapitre 2

Méthode de travail

Notre dépôt Github, nommé `SaintVenant-croustillant` est organisé en trois dossiers principaux :

— `refs` pour les références bibliographiques ayant servi à la partie théorique

— `code` pour le code des fonctions Python servant à la modélisation du problème de Saint-Venant. Nous n'avons gardé que six fichiers (cf. plus bas) par souci de simplicité et d'ordre

— `rapports` pour les rapports finaux. Nous y avons également stocké les images servant lors de la compilation LaTeXdu fichier `rapport1.tex` contenant notre rapport final.

Raphaël a organisé le code en 5 programmes, lesquels correspondent aux différents cas sur lesquels nous avons travaillé, à savoir :

— Le cas linéaire avec la solution exacte et Godunov dans le programme `programme1.py`

— Le cas non-linéaire avec la solution exacte et Godunov dans le programme `programme2.py`

— Le cas 2D de Godunov et Lax Friedrichs dans le programme `programme3.py`

— Le cas des équations de Saint-Venant proprement dites dans le programme `programme4.py`

L'ensemble des fonctions nécessaires à l'implémentation a été regroupé dans un seul fichier `initiales.py`

Le fichier `fct.py`, quant à lui, contient toutes les méthodes employées, notamment les schémas numériques (Godunov, Lax Friedrichs).J'ai essayé d'implémenter les fonctions de conditions initiales correspondant à plusieurs conditions initiales avec rupture de barrage (en distinguant le cas fond sec et fond mouillé, comme dans le document [**?**]) mais ces implémentations sont dysfonctionnelles. De la même façon, utiliser un schéma centré en non-linéaire ne fonctionne pas, ce que nous avons testé mais non gardé dans le code car inutile.

Chapitre 3

Équation de transport simple : cas linéaire 1D

Dans ce chapitre, nous étudierons le modèle le plus simple de modélisation d'une vague créneau, à savoir le cas linéaire qui revient à étudier l'équation de transport linéaire

3.1 Résolution par la méthode des volumes finis : flux de Godunov

On considère l'équation de transport linéaire définie par :

$$\frac{\partial w}{\partial t} + c\frac{\partial w}{\partial x} = 0, \quad x \in \mathbb{R}, \quad t \in \mathbb{R}^+, \tag{3.1}$$

où $w : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$, ce système est complété par une condition initiale :

$$w(x, t = 0) = w_0(x), \quad x \in \mathbb{R} \tag{3.2}$$

On s'intéresse ici à l'approximation numérique des solutions du système(3.1).Pour cela, on discrétise l'espace R. Pour simplifier les calculs, on suppose que cette discrétisation est uniforme, c'est-à-dire $w_{i+1/2} - w_{i-1/2} = \Delta(x)$ où $\Delta(x)$ est le pas d'espace supposé constant. On définit alors les volumes de controle $K_i = \left[x_{i-1/2}, x_{i+1/2}\right]$ .On dis-crétise également le temps de la façon suivante $t^n = n\Delta(t)$, où $\Delta(t)$ est le pas de temps. On note alors w_i^n une approximation de la solution exacte sur la cellule K_i et au temps t^n . Dans les méthodes de volumes finis, on cherche à approcher la moyenne de la solution w de (3.1) sur chaque cellule K_i :

$$w_i^n \approx \frac{1}{\Delta(x)} \int_{K_i} w(x, t^n)\mathrm{d}x$$

.

En intégrant l'équation (3.1) sur le rectangle $K_i \times [t^n, t^{n+1}]$ et en divisant par $\Delta(x)$, il vient que :

$$\begin{aligned} \frac{1}{\Delta(x)} \int_{K_i} w(x, t^{n+1})\mathrm{d}x &= \frac{1}{\Delta(x)} \int_{K_i} w(x, t^n)\mathrm{d}x \\ -\frac{c}{\Delta x} &\left(\int_{t^n}^{t^{n+1}} \left[w(x_{i+1/2}, t) - w(x_{i-1/2}, t) \right] dt \right) \end{aligned} \tag{3.3}$$

L'équation (3.3) nous suggère de considérer des méthodes numériques de la forme

$$w_i^{n+1} = w_i^n - \frac{\Delta(t)}{\Delta(x)} \left(F_{i+1/2}^n - F_{i-1/2}^n \right)$$

.

où $F_{i+1/2}$ est une approximation de la moyenne du flux en temps sur une cellule :

$$F_{i+1/2} \approx \frac{c}{\Delta(t)} \int_{t^n}^{t^{n+1}} w(x_{i+1/2}, t)\mathrm{d}t$$

Il est raisonnable alors de considérer que $F_{i+1/2}$ peut être obtenu à partir des deux valeurs w_i^n et w_{i+1}^n par interpolation linéaire (c'est la plus simple), c'est-à-dire :

$$F_{i+1/2} = F(w_i^n, w_{i+1}^n)$$

où la fonction F est appelée flux numérique et définit les valeurs à chaque nouvelle itérations.

Cela nous amène à la forme générale des schémas volumes finis :

$$w_i^{n+1} = w_i^n - \frac{\Delta(t)}{\Delta(x)} \left(F(w_i^n, w_{i+1}^n) - F(w_{i-1}^n, w_i^n) \right)$$

.

Une méthode de volumes finis est donc entièrement déterminée par le choix d'un flux numérique. Ceci nous permettra d'implémenter plus tard la méthode non-linéaire, notamment pour Lax Friedrichs, et de généraliser ces schémas en deux dimensions pour Godunov (par ajout d'une coordonnée)

3.2 Équation de transport : cas linéaire 1D

Dans cette section, on étudie le schéma de Godunov linéaire [?] pour l'équation de transport linéaire (3.1) .

D'après le choix de flux de godunov [?] pour la forme générale obtenue des schémas volumes finis (**??**), on a :

$$F^n_{i+1/2} = F(w^n_i, w^n_{i+1}) = \begin{cases} cw^n_{i+1} \text{ si } c < 0, \\ \\ cw^n_i \text{ si } c > 0. \end{cases} \tag{3.4}$$

Par conséquent, on obtient :

$$F(w^n_i, w^n_{i+1}) = \frac{c}{2}(w^n_i + w^n_{i+1}) - \frac{|c|}{2}(w^n_{i+1} - w^n_i)$$

.

On peut réécrire la forme générale des schémas volumes finis (**??**) comme suit :

- **Cas 1 : c>0** , on a :

$$w^{n+1}_i = w^n_i - \frac{\Delta(t)}{\Delta(x)}\left(cw^n_i - cw^n_{i-1}\right). \tag{3.5}$$

- **Cas 1 : c<0** , on a :

$$w^{n+1}_i = w^n_i - \frac{\Delta(t)}{\Delta(x)}\left(cw^n_{i+1} - cw^n_i\right). \tag{3.6}$$

On obtient donc cette algorithme

Et pour la suite, on utilisera cette algorithme général. On aura donc juste à changer le calcul de la fonction F.

Algorithm 1 Godunov linéaire

procedure GODUNOV LINÉAIRE(w_G, w_D,c)
    $c_{moins} \leftarrow min(0, c)$ ▷ C'est pour le calcul de la valeur absolue
    $c_{plus} \leftarrow max(0, c)$
    return $\frac{c}{2} \cdot (w_D - w_G) - \frac{c_{plus} - c_{moins}}{2} \cdot (w_D - w_G)$

Algorithm 2 Algorithme général pour la solution 1D

procedure CALCUL DE LA SOLUTION 1D(w,n_{temps},n_{espace},F,Δt, Δx)
    for n entre 0 et n_{temps} **do** ▷ En réalité c'est $n_{temps} - 1$
        $w^{n+1} \leftarrow w^n$
        $w_0^{n+1} \leftarrow 0$ ▷ On impose une condition initiale
        for i entre 0 et n_{espace} **do**
            $F \leftarrow F(w_i^n, w_{i+1}^n)$
            $w_i^{n+1} \leftarrow w_i^{n+1} - \frac{\Delta t}{\Delta x} \cdot F$
            $w_{i+1}^{n+1} \leftarrow w_{i+1}^{n+1} - \frac{\Delta t}{\Delta x} \cdot F$

3.3 Équation de transport linéaire 2D

3.3.1 Théorie

On on ne peut plus utiliser l'algorithme général parce que l'on prend maintenant en compte deux dimensions d'espaces au lieu d'une. On choisit d'utiliser la méthode des volumes finies centrées en espace et décentrées en temps :

$$\int_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} \int_{y_{k-\frac{1}{2}}}^{y_{k+\frac{1}{2}}} \int_{t^n}^{t^{n+1}} \partial_t u + \int_{y_{k-\frac{1}{2}}}^{y_{k+\frac{1}{2}}} \int_{t^n}^{t^{n+1}} \int_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} \partial_x u + \int_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} \int_{t^n}^{t^{n+1}} \int_{y_{k-\frac{1}{2}}}^{y_{k+\frac{1}{2}}} \partial_y u = 0 \quad (3.7)$$

$$\delta x \delta y [u]_{t^n}^{t^{n+1}} + c \cdot \delta y \delta t [u]_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} + c \cdot \delta x \delta t [u]_{y_{k-\frac{1}{2}}}^{y_{k+\frac{1}{2}}} = 0 \quad (3.8)$$

$$\delta x \delta y (u_{k,j}^{n+1} - u_{k,j}^n) + c \cdot \delta y \delta t (u_{k+\frac{1}{2},j}^n - u_{k-\frac{1}{2},j}^n) + c \cdot \delta x \delta t u_{k,j+\frac{1}{2}}^n - u_{k,j-\frac{1}{2}}^n) = 0 \quad (3.9)$$

On obtient pour des vitesse positive, en prenant l'information à gauche $c > 0$:

$$u_{k,j}^{n+1} = u_{k,j}^n - \frac{c\delta t}{\delta x \delta y}(\delta y(u_{k,j}^n - u_{k-1,j}^n) + \delta x(u_{k,j}^n - u_{k,j-1}^n) \tag{3.10}$$

Et pour des vitesse négative, en prenant l'information à droite $c < 0$:

$$u_{k,j}^{n+1} = u_{k,j}^n - \frac{c\delta t}{\delta x \delta y}(\delta y(u_{k+1,j}^n - u_{k,j}^n) + \delta x(u_{k,j+1}^n - u_{k,j}^n) \tag{3.11}$$

3.3.2 Algorithme

On utilise donc un autre algorithme, que l'on a pas réussi à améliorer (réduire le nombre de fois où l'on calcule la fonction F).

Algorithm 3 Calcul de la solution 2D

1: **procedure** CALCUL DE LA SOLUTION 2D(w,n_{temps},$n1_{espace}$,$n = 2_{espace}$,F,Δt, Δx,Δy)
2:     **for** n entre 0 et n_{temps} **do** ▷ En réalité c'est $n_{temps} - 1$
3:         $w^{n+1} \leftarrow w^n$
4:         $w_0^{n+1} \leftarrow 0$ ▷ On impose une condition initiale
5:         **for** i entre 0 et $n1_{espace}$ **do**
6:             **for** i entre 0 et $n2_{espace}$ **do**
7:                 $F_x \leftarrow F(w_{i,j}^n, w_{i,j+1}^n, c) - F(w_{i,j-1}^n, w_{i,j}^n, c)$
8:                 $F_y \leftarrow F(w_{i,j}^n, w_{i+1,j}^n, c) - F(w_{i-1,j}^n, w_{i,j}^n, c)$
9:                 $w_{i,j}^{n+1} \leftarrow w_i^n - \frac{c\Delta t}{\Delta x \cdot \Delta y} \cdot (\Delta x \cdot F_x + \Delta y \cdot F_y)$

3.3.3 Résultats

Bien qu'il y est de la diffusion, pour une vitesse positive, le résultat est intéressant. En revanche, pour une vitesse négative, bien qu'elle soit en accord avec la condition de CFL (en admettant qu'elle soit la même pour le cas 1D et 2D), la solution diverge.

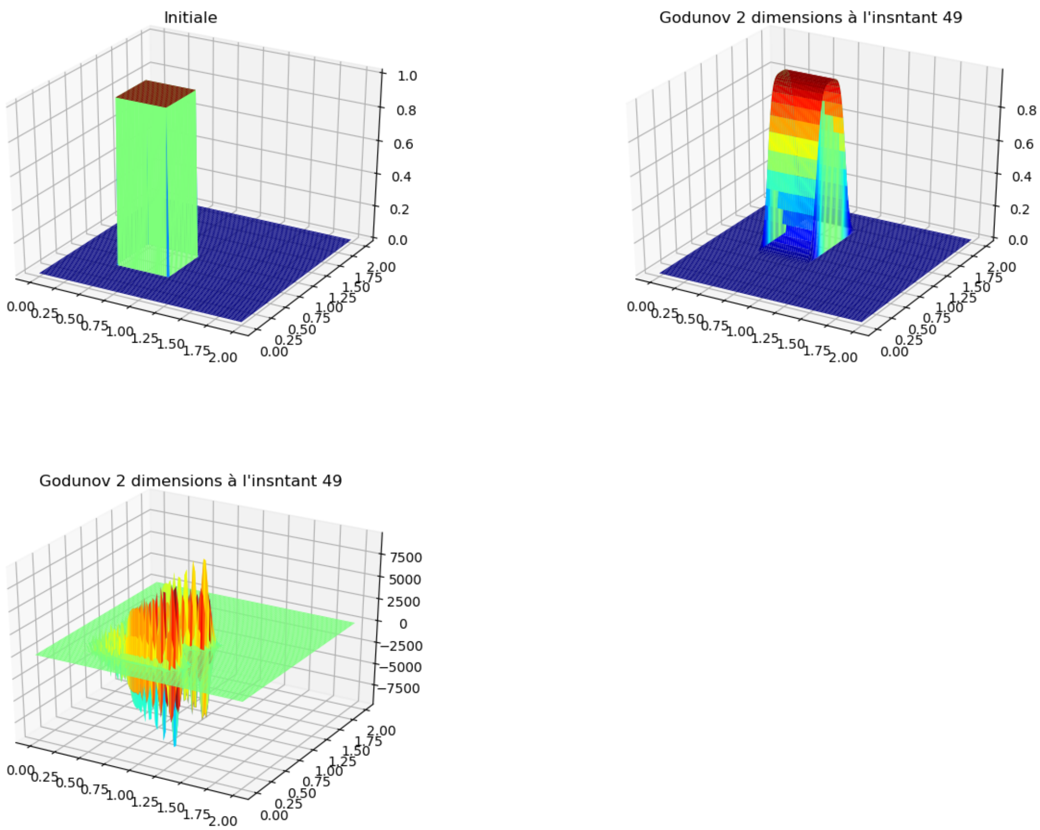


FIGURE 3.1 – "Godunov en 2D avec une vitesse de 1 et -0.5"

Chapitre 4

Équation de Bürgers : cas non-linéaire simple 1D

Dans le chapitre précédent, nous avons traité de l'équation du transport linéaire. Nous avons vu notamment que la solution était constante le long des droites caractéristiques, ce qui nous a permis d'encoder facilement le calcul de la solution analytique sous Python. Nous avons aussi vu que l'aspect linéaire ou non-linéaire de l'équation était entièrement déterminé par son flux numérique, lequel est lui-même issu du flux analytique.

Or, les équations linéaires reflètent rarement la réalité : elles permettent de reconstituer le cas d'une vague se déplaçant à vitesse constante le long des caractéristiques $x - ct$ dans des conditions idéales (fond parfaitement plat, eaux superficielles, fluide parfaitement incompressible). Or la réalité physique obéit rarement à un modèle aussi simple.

Par exemple, dans le cas d'une rupture de digue ou de barrage, l'on observe une forte discontinuité en un point avec propagation d'un choc à la vitesse c, pour lequel les caractéristiques se croisent.

Considérons à présent l'équation de Bürgers (du physicien hollandais Martin Bürgers) défini pour le flux $f(w) = \dfrac{1}{2}w^2$. Alors l'équation de départ n'est plus linéaire et se réécrit :

$$\partial_t w + u\partial_x(cu) = 0 \tag{4.1}$$

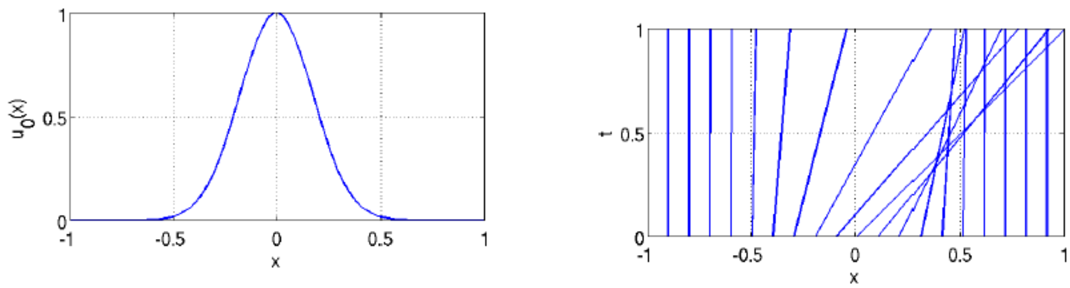
En certains points, deux flux passent de vitesses inégales et la forme de la vague est altérée. Il se produit la même chose lors d'un phénomène de tsunami qui gonfle la vague avant que celle-ci ne s'effondre sur la côte : la solution n'est plus translatée dans l'espace à l'identique mais selon le chemin emprunté.

Le cas le plus simple approchant Saint-Venant est le modèle des équations de Bürgers (d'après le mathématicien hollandais Martin Bürgers) pour lequel on considère l'équation sous sa forme conservative :

$$\partial_t u + \partial_x(F(u)) = 0$$

où u, la hauteur d'eau, est une fonction $\mathcal{C}^1$. F est la **fonction de flux** dépendant de u. Dans le cas Bürgers, même en 1D, la solution de l'équation n'est plus calculable explicitement et il faut recourir à des méthodes numériques.

L'on considère dans le cas Bürgers que $F(u) = u^2/2$, la fonction de flux est donc une fonction convexe. Sous la condition de CFL, la solution est donc stable et donc convergente d'après le théorème de Lax, donc il y a bien unicité... sous réserve de respecter quelques conditions supplémentaires.



(a) (b)

Figure 2: (a) The initial data $u_0(x) = \exp(-16x^2)$. (b) The corresponding characteristics of the Burgers equation.

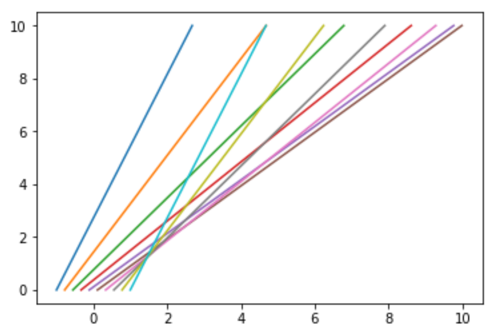


FIGURE 4.1 – "Exemple de solutions initiales avec discontinuités : les caractéristiques se croisent. Le problème de Riemann à résoudre nécessite alors le choix d'une valeur en chaque croisement à chaque itération. La première image a été empruntée à [**?**]"

4.1 Algorithme de Godunov non-linéaire

Dans le domaine triangulaire $\mathcal{T}$ où les caractéristiques se croisent, il y a apparition d'un choc-détente, plus difficile à rendre numériquement. Le formalisme mathématique autour des choc-détente étant très riche, nous l'avons synthétisé. L'on trouvera plus d'informations dans les ouvrages suivants : [**?**] (sur les ruptures de barrages sèches et mouillés) et [**?**].

Le choix de la valeur à substituer dans l'algorithme dépend alors d'un paramètre supplémentaire qui est la **condition de Rankine-Hugoniot**, définie par :

$$\sigma = \frac{f(w_2) - f(w_1)}{(w_2 - w_1)}$$

Autrement dit σ est le coefficient directeur au saut. Le choix de $w1$ ou $w2$ dépend du signe de $\Delta = \frac{x}{t} - \sigma$. Plus précisément, si on considère une solution dite **autosimilaire**, c'est-à-dire ne dépendant que de $\frac{x}{t}$ (on écrit alors $u(x,t) = \tilde{u}(\frac{x}{t})$, l'écriture de la solution se trouve grandement simplifiée.

En fait, le but est de ne considérer que les cas où les pentes forment un faisceau divergent. Il reste alors un triangle central où le choix de w_1 ou de w_2 doit s'opérer selon un autre critère, lié à un solveur de Riemann.

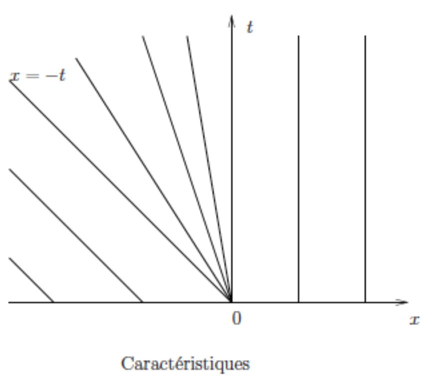


FIGURE 4.2 – Le triangle de droites centrales $\mathcal{T}$ est une zone d'incertitude : il est nécessaire d'y ajouter une fonction dépendant du flux pour le choix d'une valeur de flux. Image empruntée à [**?**]

On résout un problème de Riemann sur chacune des interfaces K_i. Bien sûr, cela suppose que la condition de CFL soit validée, donc que le cône de dépendance théorique soit inclus dans le cône de dépendance numérique afin que l'information soit prise aux bons points.

Il faut néanmoins distinguer plusieurs cas :

4.1.1 Cas détente

Le cas de la détente correspond à la condition initiale suivante pour le solveur de Riemann sur la cellule K_i [?] [?]

$$\forall i, \quad w_0^R(x) = \begin{cases} w_1 & x < x_i \\ w_2 & x > x_i \end{cases}$$

avec $w_1 > w_2$ et w_0^R qui correspond à la condition de Riemann sur l'interface K_i. En fonction du signe de Δ, on choisit une des valeurs selon le critère suivant :

— Si $\frac{x}{t} < w_1$, on choisira $w_i = w_1$
— Si $\frac{x}{t} > w_2$, on choisira $w_i = w_2$
— Si $w_1 < \frac{x}{t} < w_2$, il faut alors compléter les pentes manquantes. On choisit alors $w_i = (f^{-1})'(x/t)$ avec f^{-1} la fonction réciproque de la fonction de flux, déterminable en résolvant une équation inverse du type $y = f(x)$.

En fait, dans le triangle $\mathcal{T}$, tout est fait pour "compléter" les pentes de la façon la plus naturelle possible ; or, dans ce domaine, pour une valeur donnée de $\frac{x}{t}$, il est difficile de savoir à la limite si le point du plan choisi est plus proche des droites du domaine de w_1 ou du domaine de w_2.

4.1.2 Cas choc

Ce cas correspond à une solution de Riemann au point x_i de la forme :

$$\forall i, \quad w_0^R(x) = \begin{cases} w_1 & x < x_i \\ w_2 & x > x_i \end{cases}$$

avec $w_1 > w_2$ [1]

C'est le cas qui correspond au tsunami et à la rupture de barrage car il y a variation brutale de la hauteur d'eau (même si, en réalité, il faudrait considérer une rupture de barrage sèche ou mouillée, voir à ce propos

Mathématiquement, il mène (voir les calculs détaillés dans à une solution plus simple que la détente selon le signe de $\Delta = \frac{x}{t} - \sigma$:

$$\forall i, \quad F^G(x,t,\sigma) = \begin{cases} w_1 \text{ si } \Delta < 0 \\ w_2 \text{ si } \Delta > 0 \end{cases}$$

On obtient donc l'algorithme suivant :

4.2 Flux de Lax-Friedrichs

Le flux de Lax-Friedrichs n'est que d'ordre 1 en temps et espace : il est donc moins précis que celui de Godunov mais est plus stable. En outre, il coûte moins cher en nombre d'opérations et donc de complexité pour le cas linéaire car il n'y a pas à effectuer le choix en cas de choc ou de détente

Le flux de Lax-Friedrichs pour l'équation $\partial_t u + \partial_x f(u) = 0$ a pour expression générale :

$$w_i^{n+1} = u_i^n - \frac{\Delta x}{\Delta y}(f_{i+1/2}^n - f_{i-1/2}^n)$$

avec $f_{i-1/2}^n = \frac{1}{2}(f_{i-1}^n + f_i^n) - \frac{\Delta x}{2\Delta t}(u_i^n - u_{i-1}^n)$

On obtient cela en remplaçant w_i^n par une moyenne simple en espace, à savoir $\frac{1}{2}(w_{i+1}^n + w_{i-1}^n)$.

1. Dans le cas d'une fonction de flux quelconque, le choc correspond à $f'(w_1) > f'(w_2)$ - ici la dérivée du flux est l'identité

Algorithm 4 Godunov non linéaire

procedure GODUNOV NON LINÉAIRE(w_G,w_D,x,t)
  if $w_G > 0$ et $w_D > 0$ **then**
    return $f(w_G)$
  if $w_D < 0$ et $w_G < 0$ **then**
    return $f(w_D)$
  if $w_D < 0$ et $w_G > 0$ **then**
    $sigma \leftarrow \frac{f(w_D)-f(w_G)}{w_D-w_G}$
    if $\frac{x}{t} < sigma$ **then**
      return $f(w_G)$
    if $\frac{x}{t} > sigma$ **then**
      return $f(w_G)$
  if $w_G < 0$ et $w_D > 0$ **then**
    if $\frac{x}{t} > f'(w_G)$ **then**
      return $f(w_G)$
    if $\frac{x}{t} < f'(w_G)$ **then**
      return $f(w_D)$
    else
      return $(f')^{-1}(\frac{x}{t})$
  else
    return $f(w_G)$

Le dernier terme correspond ici à une diffusion numérique d'ordre 2 – ou terme "visqueux" – qui stabilise le schéma. Ce schéma est stable sous la condition CFL classique, à savoir $c \leq \frac{\delta x}{\delta t}$. Nous l'avons testé dans le cas linéaire comme non-linéaire et il donne des résultats corrects, bien que moins probants que Godunov dans le cas non-linéaire.

4.3 Résultats du code

Nous présentons ici les graphes commentés de nos différentes implémentations pour tous les cas : linéaire, non-linéaire et Saint-Venant.

4.3.1 Cas linéaire

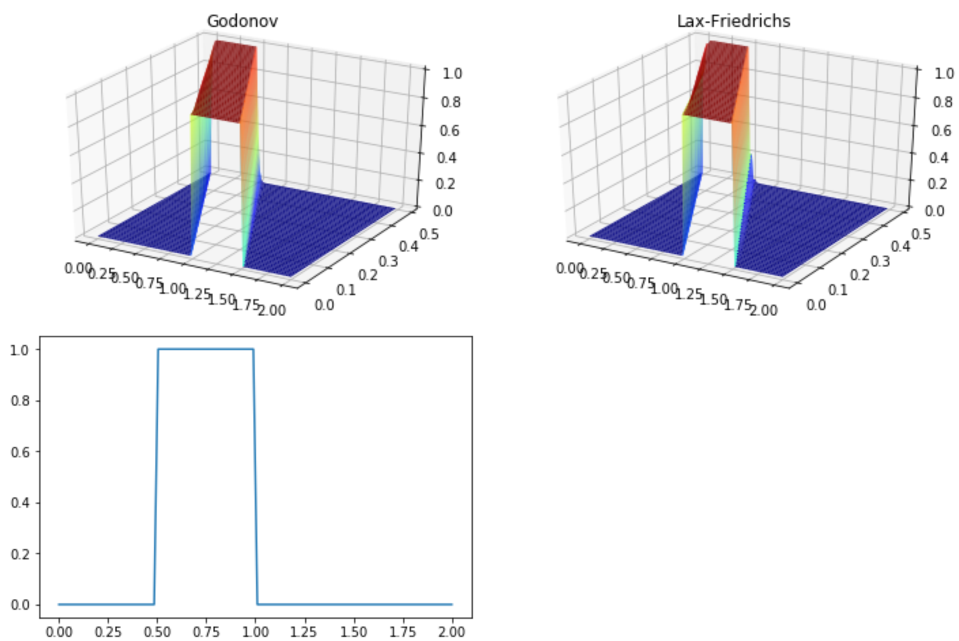


FIGURE 4.3 – Comparaison de la reconstitution de la solution initiale en créneau - ici en bas - par Godunov et Lax Friedrichs linéaires

On constate que dans le cas linéaire, le calcul de la solution exacte à chaque pas de temps permet une reconstitution parfaite, sans diffusion numérique, et ce qu'importe l'ordre ou la complexité de la méthode.

4.3.2 Cas non-linéaire

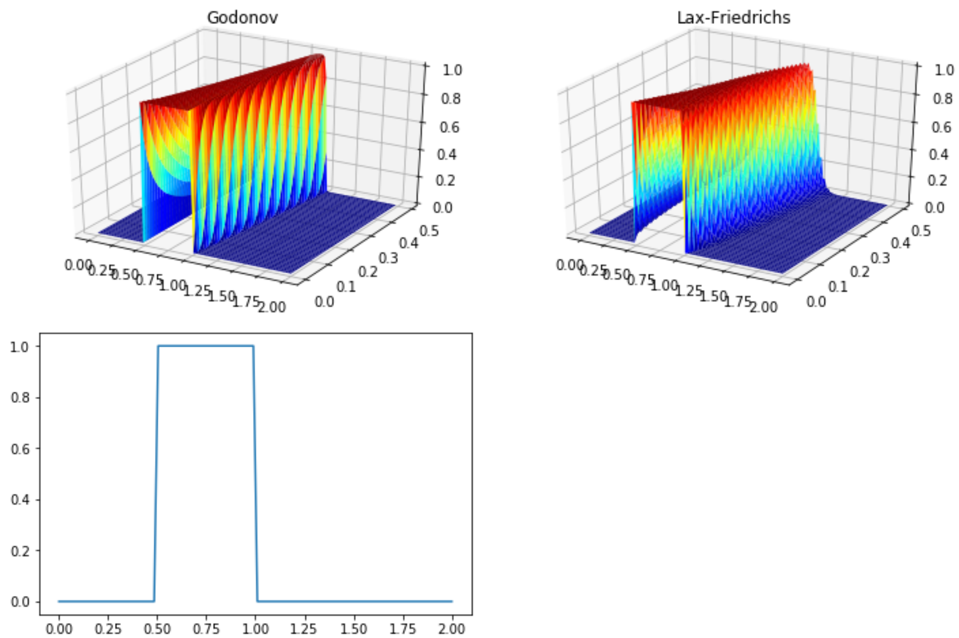


FIGURE 4.4 – Comparaison de la reconstitution de la solution initiale en créneau - ici en bas - par Godunov et Lax Friedrichs non-linéaires pour l'équation de Bürgers (cf. partie 2)

On constate que la reconstitution de la vague est altérée dans les deux cas : cela est dû à la diffusion numérique. Néanmoins, le schéma de Godunov semble le plus proche de la réalité car la forme de la vague demeure plus "carrée" que dans le cas de Lax Friedrichs, qui est moins précis. Notamment, sur le bas du versant de la vague, Lax Friedrichs présente un plus fort amortissement, là où la vague créneau reconstituée par Godunov est plus droite.

Chapitre 5

Saint-Venant en une dimension

5.1 Théorie

En plus de la hauteur notée h, on considère aussi la vitesse notée u.

$$\begin{cases} \partial_t h + \partial_x(h \cdot u) = & 0 \\ \partial_t h \cdot u + \partial_x(h \cdot u) = & 0 \end{cases} \tag{5.1}$$

On choisi de considérer ces équation sous formes de vecteur :

$$\partial_t V + \partial_x f(V) = 0 \tag{5.2}$$

avec $V = \begin{pmatrix} h \\ h \cdot u \end{pmatrix}$ et $f(V) = \begin{pmatrix} h \cdot u \\ h \cdot u^2 + \frac{1}{2} g \cdot h^2 \end{pmatrix}$

5.2 Algorithme

Pour la modélisation, il sera plus facile de considérer $h \cdot u$ comme une variable à elle seule, plutôt que comme un produit. On pose donc :

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} h \\ h \cdot u \end{pmatrix}$$

Finalement, on obtient, pour l'équation de Saint-Venant appliqué à w :

$$f \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} w_2 \\ \frac{(w_2)^2}{w_1} + \frac{1}{2} g (w_1)^2 \end{pmatrix} \tag{5.3}$$

5.3 Résultat

On observe un résultat intéressant quand on impose une condition non physique (vitesse en créneau). En revanche quand on impose une condition plausible (vitesse nulle comme condition initiale), alors le résultat donne une hauteur nulle en tout temps. Ce dernier résultat étant absurde, nous ne pouvons pas valider ce code.

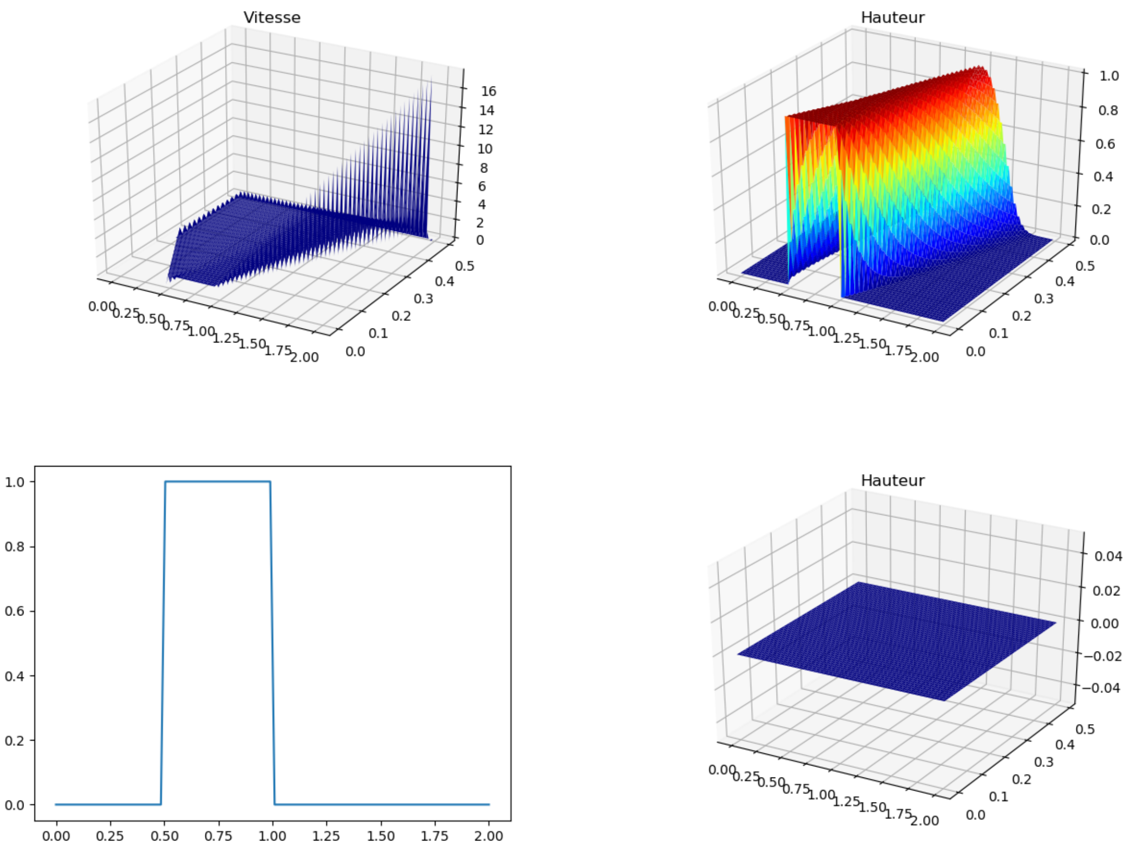


FIGURE 5.1 – Saint-Venant avec vitesse en créneaux et nulle

Chapitre 6

Conclusion et perspectives

Les équations de Saint-Venant sont un modèle important pour la prévision de nombreux phénomènes naturels potentiellement dangereux comme les tsunamis . Notre étude nous a mené à une étude complète d'une modélisation en Python, en commençant par un modèle linéaire simple - le transport d'une vague - puis en extrapolant (Bürgers puis Saint-Venant à proprement parler).

Il serait possible de modéliser plus finement certaines réalités physiques obéissant à Saint-Venant avec des paramètres plus réalistes comme la topographie - une forme de fond pas forcément plate - dans le cas de la vague. D'autres fluides comme la neige ou le sable obéissant à cette modélisation font appel à un facteur de forme ou à des lois plus complexes dérivées de ce modèle.

Nous avons beaucoup apprécié ce travail en équipe qui nous a permis de nous perfectionner en Python et de ne pas "rouiller" en LaTeX. Il a néanmoins été très ambitieux, autant au niveau de la bibliographie que du codage et nous n'avons pas pu justifier tous nos résultats, ni implémenter toutes nos conditions initiales (dont certaines sont dysfonctionnelles malgré une révision assidue).

Bibliographie

Annexe

6.1 Les différentes méthodes utilisées

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import csv, math, numpy
import matplotlib.pyplot as plt

#w1 = valeur Ã  gauche de la solution du pbm de Riemann
#w2 = valeur Ã  droite de la solution du pbm de Riemann

def F(w1,w2,c): #caractÃ©ristiques
    if c>0:
        y=(w2-w1)*c
    elif c<0:
        y=(w1-w2)*c
    return y

#Equation de transport
##LinÃ©aire 1D
###Godunov
```

```

def GL(w1,w2,c):#Godunov linÃ©aire
    cp=max(0,c)
    cm=min(0,c)
    return (c/2)*(w2+w1)-((cp-cm)/2)*(w2-w1)#cp-cm est la valeur absolue de c

###Lax Friedrichs

def Lf(u,c): #fonction f pour Lax-friedrichs linÃ©aire
    return c*u

def LLF(w1,w2,dx,dt,c):#Lax-Friedrichs Ã©quation de transport linÃ©aire
    return 0.5*(Lf(w1,c)+Lf(w2,c))-(0.5*dx/dt)*(w2-w1)

##Cas non linÃ©aire
###Lax-Friedrichs

def f(u): #fonction de flux pour BÃ¼rgers
    return 0.5*u**2


def LF(w1,w2,dx,dt):#Lax-Friedrichs Ã©quation de transport cas non linÃ©aire
    return 0.5*(f(w1)+f(w2))-(0.5*dx/dt)*(w2-w1)

###Godunov

def fDer(x):#dÃ©rivÃ© de la fonction f
        return x #dÃ©rivÃ©e de la fonction de flux

def fDerRec(x):#RÃ©ciproque de la dÃ©rivÃ© de la fonction f qui est Ã©gal Ã mÃªme
```

```
        return x #ici la rÃ©ciproque de la dÃ©rivÃ© de f est elle-mÃªme car fDer est l'identitÃ©

def sigma(w1,w2):#Condition de Ranki-Hugoniot
    if(abs(w1-w2)<10**(-5)):
        return 1
    else:
        return (f(w2)-f(w1))/(w2-w1)

def GNL(w1,w2,x,t):#Godunov cas non linÃ©aire
    if w1>0 and w2>0:
        return f(w1)
    elif w2<0 and w1<0:
        return f(w2)
    elif (w1>0 and w2<0): #cas de choc ou w1>0>w2
        if (x/t)<=sigma(w1,w2):# on choisit ici arbitrairement la valeur Ã gauche quand sigma(w1,w2)=0
            return f(w1)
        elif (x/t)>sigma(w1,w2):
            return f(w2)
        else:
            return f(w1)
    elif w1<0 and w2>0: #cas de dÃ©tente pour w2>0>w1
        if (x/t)>fDer(w1):
            return f(w1)
        elif (x/t)<fDer(w1):
            return f(w2)
        else:
            fDerRec(x/t)

    else:
        return f(w1)

#Equation de Saint Venant
```

```
##Lax-Friedrichs

def f2(wa,wb,g):#fonction considÃ©rant la dexuiÃ¨me Ã©quation de Saint-Venant
    return numpy.array(wb,(wb**2)/wa+0.5*g*(wb**2))# on retourne un tableau parce que c'est une fonction Ã  deux variable

def LLF2(w1,w2,dx,dt,g):#Lax-Friedrichs gÃ©nÃ©ral 2
    return numpy.array(0.5*(f2(w1[0],w1[1],g)+f2(w2[0],w2[1],g))-(0.5*dx/dt)*(w2-w1))#on adapte la formule de Lax-Friedrichs avec une fonction Ã  2 variable en envoyant le vecteur w1


def initia(x): #fonction crÃ©neau pour l'exemple (fct Cinfini sur un compact)
    if (0.5<=x)and(x<=1):
        #u=1+x*x;
        u=1
    else:
        u=0
    return u

def initia2(x,y):#fonction crÃ©neau pour le cas 2D
    if ((0.5<=x)and(x<=1) and (0.5<=y) and (y<=1)):
        u=1
    else:
        u=0
    return u

def initia3(x): #fonction crÃ©neau pour les vitesses nÃ©gatives
    if (1<=x)and(x<=1.5):
        u=1
    else:
        u=0
```

```
    return u
```

6.2 Les fonctions initiales utilisées

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-$

import numpy as np

def initia(x): #fonction crÃ©neau pour l'exemple (fct Cinfini sur un compact)
    if (0.5<=x)and(x<=1):
        #u=1+x*x;
        u=1
    else:
        u=0
    return u

def initia2(x,y):#fonction crÃ©neau pour le cas 2D
    if ((0.5<=x)and(x<=1) and (0.5<=y) and (y<=1)):
        u=1
    else:
        u=0
    return u

def initia3(x): #fonction crÃ©neau pour les vitesses nÃ©gatives
    if (1<=x)and(x<=1.5):
        u=1
    else:
        u=0
    return u

```

```
def initiaGauss(x):
    if (abs(x)<=1):
        u= np.exp(-x**2)
    else:
        u=0
    return u
```

6.3 L'équation de transport linéaire 1D

```
# -*- coding: utf8 -*-
import csv, math
import numpy as np
import matplotlib.pyplot as plt
import pylab
from mpl_toolkits.mplot3d import Axes3D #Pour les graphes en 3D (cf. CM de
    mardi)

import fct
import initiales

nx=100;
nt=50;
dt=0.01;
dx=2.0/(nx-1);
x=np.linspace(0,2,nx)
u=np.zeros(nx);
un=np.zeros(nx);
dtt=0.1

for i in range(0,nx-1):
    u[i]=initiales.initia3(x[i])
w=np.zeros([nt,nx])
```

```
w1=np.zeros([nt,nx])
w[0,:]=u
w1[0,:]=u
c=2
plt.plot(x,u)

#Calcul de la solution exacte

sol=np.eye(nt,nx)*0
t=np.arange(0,0.5,dt)
for i in range(1,nt-1):
    for j in range(0,nx-1):
        sol[i,j]=initiales.initia3(x[j]-c*dt*i)
    sol[i,0]=0

#Calcul des solutions approchÃ©es

##Solution de Godunov

for i in range(0,nt-1):
    w[i+1,:]=w[i,:]# on initialise la solution pour tous les pas d'espace
    aux valeurs du temps prÃ©cÃ©dent
    w[i+1,0]=0#On impose une condition initiale (n'a de sens que pour une
    vitesse positive ou)
    for j in range(0,nx-1):
        F=fct.GL(w[i,j],w[i,j+1],c)
        w[i+1,j]=w[i+1,j]-(dt/dx)*F
        w[i+1,j+1]=w[i+1,j+1]+(dt/dx)*F
    w[i+1,nx-1]=w[i,nx-2]

##Solution de Lax-Friedrichs

for i in range(0,nt-1):
```

```
        w1[i+1,:]=w1[i,:]# on initialise la solution pour tous les pas d'espace aux valeurs du temps prÃ©cÃ©dent
        w1[i+1,0]=0 #On impose une condition initiale (n'a de sens que pour une vitesse positive ou)
        for j in range(1,nx-1):
            F=fct.LLF(w1[i,j],w1[i,j+1],dx,dt,c)
            w1[i+1,j]=w1[i+1,j]-(dt/dx)*F
            w1[i+1,j+1]=w1[i+1,j+1]+(dt/dx)*F
        w1[i+1,nx-1]=w[i,nx-2] #On donne comme approximation acceptable de la derniÃ¨re valeur, la valeur de l'avant derniÃ¨re

#Calculs des erreurs

maG=0
for i in range(0,nt-1):
    if maG<max(sol[i,:]-w[i,:]):
        miG=i
        maG=max(sol[i,:]-w[i,:])

print("L'erreur de la mÃ©thode de Godunov dans le cas linÃ©aire est : ",maG," et elle est Ã  l'instant ",miG)

maLF=0
for i in range(0,nt-1):
    if maLF<max(sol[i,:]-w1[i,:]):
        miLF=i
        maLF=max(sol[i,:]-w1[i,:])
print("L'erreur de la mÃ©thode de Lax-Friedrichs linÃ©aire est : ",maLF," et elle est Ã  l'instant ",miLF)

ma=0
for i in range(0,nt-1):
    if ma<max(w[i,:]-w1[i,:]):
        mi=i
```

```
        ma=max(w[i,:]-w1[i,:])

print("La diffÃ©rence maximale entre ces deux mÃ©thodes est : ",ma," et
    elle est Ã  l'instant ",mi)

if abs(c)*dt<dx:
    print("Le schÃ©ma est stable")
else:
    print("Le schÃ©ma est instable")

X,T = np.meshgrid(x,t)# On transforme les vecteurs X et Y en matrices pour
    pouvoir projeter la figure

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, T, w, cmap=plt.cm.jet, rstride=1, cstride=1, linewidth
    =0)
plt.title("Godunov")

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, T, w1, cmap=plt.cm.jet, rstride=1, cstride=1, linewidth
    =0)
plt.title("Lax-Friedrichs")

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, T, sol, cmap=plt.cm.jet, rstride=1, cstride=1, linewidth
    =0)
plt.title("Solution")

plt.show()
```

6.4 L'équation de transport linéaire 2D

```
# -*- coding: utf8 -*-
import csv, math #pour les tableaux et listes, je suppose
import numpy as np
import matplotlib.pyplot as plt
import pylab
from mpl_toolkits.mplot3d import Axes3D #Pour les graphes en 3D (cf. CM de
    mardi)

import fct
import initiales

nx=100;
nt=50;
dt=0.01;
dx=2.0/(nx-1);
x=np.linspace(0,2,nx)
t=np.arange(dt,0.5+dt,dt)
u=np.zeros(nx);
un=np.zeros(nx);

for i in range(0,nx-1):
    u[i]=initiales.initia(x[i])

w1=np.eye(nt,nx)*0#On multiplie par zÃ©ro pour qu'il n'y est pas des un qui
    vienne parasiter les rÃ©sultats
w1[0,:]=u
w=np.eye(nt,nx)*0
w[0,:]=u
plt.plot(x,u)
t=np.arange(0,0.5,dt)

```

```
#Calcul des solutions approchÃ©es

##Solution de Godunov

for i in range(0,nt-1):
    w[i+1,:]=w[i,:]# on initialise la solution pour tous les pas d'espace aux valeurs du temps prÃ©cÃ©dent
     for j in range(1,nx-1):
         F=fct.GNL(w[i][j],w[i][j+1],x[j],t[i])
         w[i+1,j]=w[i+1][j]-(dt/dx)*F
         w[i+1,j+1]=w[i+1][j+1]+(dt/dx)*F
    w[i+1,nx-1]=w[i,nx-2]

##Solution de Lax-Friedrichs
for i in range(0,nt-1):
    w1[i+1,:]=w1[i,:]# on initialise la solution pour tous les pas d'espace aux valeurs du temps prÃ©cÃ©dent
     for j in range(1,nx-1):
         F=fct.LF(w1[i][j],w1[i][j+1],dx,dt)
         w1[i+1,j]=w1[i+1][j]-(dt/dx)*F
         w1[i+1,j+1]=w1[i+1][j+1]+(dt/dx)*F
    w[i+1,nx-1]=w[i,nx-2]

#Calcul
ma=0
for i in range(0,nt-1):
     if ma<max(w[i,:]-w1[i,:]):
         mi=i
         ma=max(w[i,:]-w1[i,:])
print("La diffÃ©rence maximale entre ces deux mÃ©thodes est : ",ma," et elle est Ã  l'instant ",mi)

X,T= np.meshgrid(x,t)

```

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, T, w, cmap=plt.cm.jet, rstride=1, cstride=1, linewidth=0)
plt.title("Godonov")

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, T, w1, cmap=plt.cm.jet, rstride=1, cstride=1, linewidth=0)
plt.title("Lax-Friedrichs")

plt.show()
```

6.5 L'équation de transport non linéaire 1D

```
# -*- coding: utf8 -*-
import csv, math
import numpy as np
import matplotlib.pyplot as plt
import pylab
from mpl_toolkits.mplot3d import Axes3D #Pour les graphes en 3D (cf. CM de mardi)

import fct
import initiales

nx=100
ny=100
nt=50
f=0.5
dt=f/(nt-1)
```

```
dx=2.0/(nx-1)
dy=2.0/(ny-1)
x=np.linspace(0,2,nx)
y=np.linspace(0,2,ny)
u=np.zeros([nx,ny])
t=np.arange(dt,4+dt,dt)#On ne veut pas qu'il y est de temps 0 pour Ã©viter
    une Ã©ventuelle division par 0

for i in range(0,nx-1):
    for j in range(0,ny-1):
        u[i,j]=initiales.initia2(x[i],y[j])
w=np.zeros([nt,nx,ny])
w[0,:,:]=u
c=-0.5

#appliquer l'algorithme qu'on utilise pour le cas 1D est ici beaucoup plus
    difficile, on applique donc la discrÃ©tisation directement
for i in range(0,nt-1):
    for j in range(1,nx-1):
        for k in range(1,ny-1):
            w[i+1,j,k]=w[i,j,k]-(c*dt/(dx*dy))*(dx*(fct.GL(w[i,j,k],w[i,j
    +1,k],c)-fct.GL(w[i,j-1,k],w[i,j,k],c))+dy*(fct.GL(w[i,j,k],w[i,j,k+1],c
    )-fct.GL(w[i,j,k],w[i,j,k-1],c)))

X,T= np.meshgrid(x,t)
X,Y= np.meshgrid(x,y)
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, w[49,:,:], cmap=plt.cm.jet, rstride=1, cstride=1,
    linewidth=0)
plt.title("Godunov 2 dimensions Ã  l'insntant 49")

fig = plt.figure()
ax = plt.axes(projection='3d')
```

```
X,Y=np.meshgrid(x,y)
ax.plot_surface(X, Y, u, cmap=plt.cm.jet, rstride=1, cstride=1, linewidth
    =0)
plt.title("Initiale")


plt.show()
```

6.6 L'équation de Saint-Venant 1D

```
# -*- coding: utf8 -*-
import csv, math #pour les tableaux et listes, je suppose
import numpy as np
import matplotlib.pyplot as plt
import pylab
from mpl_toolkits.mplot3d import Axes3D #Pour les graphes en 3D (cf. CM de
    mardi)

import fct
import initiales

nx=100;
nt=50;
dt=0.01;
dx=2.0/(nx-1);
x=np.linspace(0,2,nx)
t=np.arange(dt,0.5+dt,dt)
ui=np.zeros(nx)

for i in range(0,nx-1):
    ui[i]=initiales.initia(x[i])

```

```
u=np.eye(nt,nx)*0
u[0,:]=ui
h=np.eye(nt,nx)*0
h[0,:]=ui
w=np.array([h,u*h])
g=10.0# valeur de la gravitation sur Terre
plt.plot(x,ui)

t=np.arange(0,0.5,dt)
for i in range(0,nt-1):
    w[:,i+1,:]=w[:,i,:]# on initialise la solution pour tous les pas d'
    espace aux valeurs du temps prÃ©cÃ©dent
    for j in range(1,nx-1):
        F=fct.LLF2(w[:,i,j],w[:,i,j+1],dx,dt,g)
        w[:,i+1,j]=w[:,i+1,j]-(dt/dx)*F
        w[:,i+1,j+1]=w[:,i+1,j+1]+(dt/dx)*F
    w[:,i+1,nx-1]=w[:,i,nx-2]

for i in range(0,nt-1):
    for j in range(1,nx-1):
        u[i,j]=w[1,i+1,j]/w[0,i+1,j+1]

X,T= np.meshgrid(x,t)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, T, w[1], cmap=plt.cm.jet, rstride=1, cstride=1,
    linewidth=0)
plt.title("Hauteur")

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, T, u, cmap=plt.cm.jet, rstride=1, cstride=1, linewidth
    =0)
```

```
plt.title("Vitesse")

plt.show()
```