Équations hyperboliques
(équations de transport et de Burgers)

1 Pour commencer

Nous allons utiliser le langage python pour résoudre numériquement les EDP vues en cours. Il y a plusieurs façons d'utiliser python, en voici quelques-unes :

- En utilisant la plateforme Jupyter de l'université, accessible avec vos identifiants à l'adresse `https://jupyter.mecanique.univ-lyon1.fr/`. Ce sont des notebooks, le code python est entré dans des cellules qui sont exécutées une par une ensuite. Pratique pour le côté interactif.
- En l'installant sur votre ordinateur. Le plus simple est d'installer Anaconda (`https://anaconda.org/`) ou encore Miniconda (plus léger, `https://docs.conda.io/en/latest/miniconda.html`). Si vous optez pour Miniconda, il faudra ensuite installer les paquets python `numpy`, `scipy`, et `matplotlib`. L'idée ici est d'écrire le code python dans un fichier texte et de l'exécuter ensuite avec la commande `python`. Pour pouvoir faire des notebooks dans le navigateur, il faut installer le paquet `jupyter` avec Miniconda. Avec Anaconda, ce paquet est déjà installé.

2 Rappels sur le paquet `numpy`

Le paquet `numpy` est indispensable en python pour faire du calcul scientifique. Importez-le avant de faire les prochains exercices en tapant la commande suivante au début du fichier ou dans la première cellule de votre notebook suivant la manière dont vous avez choisi d'utiliser python.

```
import numpy as np
```

2.1 Création de tableaux

Entrer les commandes suivantes et comprendre ce qu'elles font. On pourra utiliser la commande `print` pour afficher les variables.

Tableaux 1D :

```
u = np.array([1, 2, 3, 4, 5])
print(u.shape)
print(u.size)
print(u)

Nx = 10

# allocation et initialisation à 0 d'un tableau 1D de taille Nx
u = np.zeros(Nx)

# allocation et initialisation à 1 d'un tableau 1D de taille Nx
v = np.ones(Nx)

# allocation sans initialisation d'un tableau 1D de taille Nx.
w = np.empty(Nx)
```

```
# initialisation à la main
for i in range(w.size):
  w[i] = i

u = np.linspace(0, 1, Nx)
u, dx = np.linspace(0, 1, Nx, retstep=True)

dx = 1 / Nx
v = np.arange(0, 1, dx)
v = np.arange(0, 1+0.5*dx, dx)
```

Tableaux nD :

```
u = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(u.shape)
print(u.size)
print(u)

Nx, Nt = 10, 5

# allocation et initialisation à 0 d'un tableau 2D de taille (Nx, Nt)
A = np.zeros((Nx, Nt))

u = np.ones(Nx)
v = np.ones(Nx-1)
A = np.diag(2 * u) - np.diag(v, -1) - np.diag(v, 1)
```

2.2 Opérations sur les tableaux

Concaténation de tableaux.

```
u = np.arange(6)
v = np.random.random(6)

w = np.hstack((u, v))
print(w)

w = np.vstack((u, v, 10*v, -u))
print(w)
```

Les fonctions mathématiques et les opérations arithmétiques sur les tableaux `numpy` s'effectuent **élément par élément**.

```
u = np.random.random(6)
v = 10 * np.ones(6)

print(u + v)
print(u * v)
print(u / v)

x = np.linspace(0, 1, 10)
p = np.arange(10, 0, -1)
print(x**p)
print(np.exp(-(x-0.5)**2 / 2))
```

```
print(np.sqrt(x))
print(np.sin(2 * np.pi * x))

v = np.random.random(6)
print(np.amax(u))
print(np.amin(u))
print(np.maximum(u, v))
```

2.3 Sous-tableaux

Extraction d'un sous-tableau.

```
# tableaux 1D
x = np.arange(0, 51)
x[1:11]
x[:11]
x[40:-1]
x[40:]
x[10:41:3]
x[::2]
x[::-1]

# tableaux nD
A = np.array([[1, 2, 3, 4], [11, 12, 13, 14],
              [21, 22, 23, 24], [31, 32, 33, 34]])

# Accès à un élément
print(A[2, 1])

# sous-tableaux
A[0:2, 1:3]
A[:, 1]
A[0, :]
A[-1, :]
A[:, -3]

# extraction en spécifiant les indices
A[[0, 2], :]
A[[0, 2], [0, -1]]

# extraction à l'aide d'un tableau de booléens
print(A > 11)
print(A[A > 11])
A[A > 11] = -1
print(A)
```

2.4 Algèbre linéaire

```
A = np.array([[4, 0, 1], [1, 4, 1], [1, 0, 4]])
B = np.arange(1, 10).reshape((3, 3))
print(A)
```

```
print(B)

# produit
A * B
A @ B
np.matmul(A, B)

# transposée
A.T
A.transpose()
np.transpose(A)

# fonctions usuelles
np.trace(A)
np.linalg.det(A)

# produit matrice / vecteur
x = np.ones(3)
np.matmul(A, x)
np.dot(A, x)
A @ x

# résolution d'un système linéaire (LU avec pivot partiel)
np.linalg.solve(A, x)

# normes
x, dx = np.linspace(0, np.pi / 2, 20, retstep=True)
np.linalg.norm(dx * np.sum(np.sin(x[:-1])) - 1)
```

2.5 Vues et copies

Quand c'est possible, `numpy` retourne des « vues » sur les tableaux ou sous-tableaux. Ce ne sont pas des copies. Si ces tableaux ou sous-tableaux sont modifiés, l'original l'est aussi.

```
A = np.arange(1, 10).reshape((3, 3))
print(A)

B = A[1:, 1:]
B[0, 0] = 20
print(A)
B[:, :] = -5
print(A)

B = A.transpose()
B[0, :] = -1
print(A)
```

Tout comme pour les listes python, une égalité entre deux tableaux ne déclenche pas une copie des éléments mais une copie des références vers les tableaux :

```
A = np.arange(1, 10).reshape((3, 3))
B = A
print(B)
print(A)
```

```
B[1, :] = -1
print(B)
print(A)
```

Pour faire une copie, il faut l'écrire explicitement :

```
A = np.arange(1, 10).reshape((3, 3))
B = A.copy()
B[1, :] = -1
print(B)
print(A)

# ou encore
A = np.arange(1, 10).reshape((3, 3))
B = np.empty_like(A)
B[:] = A

# ou même
A = np.arange(1, 10).reshape((3, 3))
B = np.empty_like(A)
np.copyto(B, A)
```

3 Animation avec `matplotlib`

Pour faire une animation, l'idée est de faire un premier graphique dont on mettra à jour les données dans une boucle. C'est beaucoup plus rapide, et donc agréable, de mettre à jour les données d'un graphique plutôt que de refaire un graphique complet. Voici une première façon de faire, dans le style matlab :

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

x = np.linspace(-1, 1, 200)
dt = 1e-2
T = 10
t = 0

pp, = plt.plot(x, np.sinc(t * x))
plt.xlim(-1, 1)
plt.ylim(-0.5, 1.5)
plt.pause(dt)

while t < T:
  t += dt
  pp.set_ydata(np.sinc(t * x))
  plt.pause(dt)
```

Et voici une méthode plus pythonesque :

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

```
x = np.linspace(-1, 1, 200)
dt = 1e-2
T = 10

fig = plt.figure()
pp, = plt.plot(x, np.sinc(t * x))
plt.xlim(-1, 1)
plt.ylim(-0.5, 1.5)

def update(t):
  global pp, x
  pp.set_ydata(np.sinc(t * x))

anim = animation.FuncAnimation(fig, update, frames=np.int(T / dt) +
   1, interval=1000*dt, repeat=True)
plt.show()
```

Dans un notebbok, ajouter la ligne suivante dans une cellule avant la première animation :

```
%matplotlib tk
```

4 Équation d'advection

On considère l'équation d'advection (convection, transport) à vitesse constante $a > 0$:

$$\partial_t u + a\partial_x u = 0,$$

avec une donnée initiale $u(0, x) = u^0(x)$. Le domaine spatial est $[0, 1]$. Le schéma de volumes finis général qui nous intéresse est écrit

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\frac{u_{j+1/2}^n - u_{j-1/2}^n}{\Delta x} = 0$$

où Δt et $\Delta x = 1/J$ ($J \in \mathbb{N}^*$) sont des réels strictement positifs, et où u_j^n (pour $n \in \mathbb{N}$ et $j = 1, \ldots, J$) est censé être une approximation de la moyenne de la solution u au temps $t^n = n\Delta t$ sur la maille j,

$$\frac{1}{\Delta x}\int_{x_{j-1/2}}^{x_{j+1/2}} u(n\Delta t, x)\, dx,$$

avec $x_{j-1/2} = (j-1)\Delta x$ pour $j = 1, \ldots, J+1$. En dimension 1 d'espace, ces schémas ressemblent beaucoup aux schémas de différences finies, pour lesquels u_j^n est censé être une approximation de $u(t^n, x_j)$, avec $t^n = n\Delta t$ et $x_j = (j - 1/2)\Delta x$.

1. Programmer les schémas
   - **upwind (décentré amont)** :
     $$u_{j+1/2}^n = u_j^n$$
     pour tout j et tout n (ce serait $u_{j+1/2}^n = u_{j+1}^n$ si a était négatif) ;
   - **centré** :
     $$u_{j+1/2}^n = \frac{u_j^n + u_{j+1}^n}{2}$$
     pour tout j et tout n ;

— **downwind** :

$$u^n_{j+1/2} = u^n_{j+1}$$

pour tout j et tout n (ce serait $u^n_{j+1/2} = u^n_j$ si a était négatif) ;

— de **Lax-Friedrichs** :

$$u^n_{j+1/2} = \frac{u^n_j + u^n_{j+1}}{2} + \frac{\Delta x}{2a\Delta t}(u^n_j - u^n_{j+1})$$

pour tout j et tout n ;

— de **Rusanov** :

$$u^n_{j+1/2} = \frac{u^n_j + u^n_{j+1}}{2} + \frac{c}{2a}(u^n_j - u^n_{j+1})$$

pour tout j et tout n, pour un certain coefficient c qui *doit* être supérieur ou égal à $|a|$;

— de **Lax-Wendroff** :

$$u^n_{j+1/2} = \frac{u^n_j + u^n_{j+1}}{2} - \frac{a\Delta t}{2\Delta x}(u^n_{j+1} - u^n_j)$$

pour tout j et tout n.

On remarque que l'on a besoin de spécifier des valeurs aux bords ! On pourra pour cela :

— soit considérer le problème sur $\mathbb{R}$ entier et supposer que la donnée initiale est périodique de période 1 (alors la solution est périodique de période 1 pour tout temps) ;

— soit implémenter des conditions de type Dirichlet et Neumann homogènes en ajoutant (virtuellement) à gauche et à droite du domaine spatial des mailles, indexées par 0 et $J+1$ et dites « fantômes », dans lesquelles on fixe arbitrairement la solution à chaque pas de temps de la manière suivante : $u^n_0 = 0$ et $u^n_{J+1} = u^n_J$, et à l'aide desquelles on calcule les *flux numériques* [1].

2. Remarquer par l'expérience que seuls le premier et les trois derniers semblent converger (lorsque Δt et Δx tendent vers 0), sous une condition liant le pas de temps et le pas d'espace : $a\Delta t/\Delta x \leq 1$ (ceci se montre facilement [2] [3]).

3. Montrer que cette condition est celle sous laquelle u^{n+1}_j est une combinaison convexe des u^n_i, $i = 1, \ldots, J$, sauf pour le schéma de Lax-Wendroff. Pour ce dernier schéma, la stabilité a lieu dans l^2, et est donc un peu plus difficile à montrer (mais ce n'est pas insurmontable !). On pourra montrer que tous ces schémas sont consistants à l'ordre 1 en Δt et Δx (sous la condition évoquée dans la note en bas de page 3 en ce qui concerne le schéma de Lax-Friedrichs), sauf le schéma centré (ordre 1 en Δt et 2 en Δx) et le schéma de Lax-Wendroff (ordre 2 en Δt et Δx).

4. Écrire le schéma *upwind implicite*, qui consiste à prendre les mêmes flux que ceux du schéma upwind classique mais au temps t^{n+1} dans le schéma volumes finis. Vérifier par l'expérience que ce schéma semble converger sans qu'aucune condition de type Courant-Friedrichs-Lewy ne soit vérifiée.

5. Tracer numériquement les ordres de convergence des différents schémas pour les normes L^1 et L^∞ et pour des conditions initiales régulières, lipschitziennes mais pas de classe $\mathcal{C}^1$, puis discontinues (mais à variation bornée).

6. On s'intéresse maintenant à l'équation de transport avec une vitesse a dépendant de manière régulière de la variable spatiale x, positive pour tout x pour simplifier. Programmer le schéma upwind et le schéma de Lax-Wendroff, vérifier leur convergence (par exemple pour $a(x) = 1 + \sin(2\pi x)$, mais ce n'est qu'un exemple !).

1. Le traitement des conditions aux limites en EDP hyperboliques est un problème trop ardu pour être abordé de manière précise dans le cadre de ce cours-TP.

2. En fait le schéma centré converge dans l^2 sous la condition, pas standard du tout pour une équation hyperbolique, qu'il existe une constante C telle que $\Delta t \leq C\Delta x^2$.

3. En réalité il y a une autre condition pour le schéma de Lax-Friedrichs : il faut que $\Delta x^2/\Delta t$ tende vers 0, pour des raisons de consistance. Cette condition est réalisée lorsqu'on fait le choix *naturel* de fixer le rapport $\Delta t/\Delta x$.

5 Équation de Burgers

On s'intéresse maintenant à l'équation non linéaire

$$\partial_t u + \partial_x u^2/2 = 0,$$

dans le même cadre que ci-dessus concernant conditions au bord et donnée initiale.

1. Écrire un schéma numérique de type upwind pour discrétiser (de manière consistante !) l'équation de Burgers pour une donnée initiale u^0 positive régulière (en supposant que la solution reste régulière et positive), et le tester. En réalité on pourra écrire *deux* schémas de type upwind : l'un conservatif, et l'autre discrétisant la forme quasi-linéaire non conservative $\partial_t u + u\partial_x u = 0$. On constatera que ce dernier schéma, non conservatif, n'est pas convergent en général, en particulier lorsque la solution comporte un choc.
2. Programmer aussi le schéma de Rusanov. Faire différents tests, en changeant la donnée initiale, le flux de l'équation...