TD4

Équations différentielles ordinaires et visualisation

On cherche dans ce TD à étudier et manipuler des équations différentielles ordinaires en python et à les visualiser.

1 Visualisation des fonctions

Comme premier contacte avec la visualisation, nous considérons l'exemple suivant simple

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

x=np.linspace(-np.pi,np.pi,101)
y=np.sin(x)+np.sin(3*x)/3.0

plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Simple plot')

plt.show()
plt.savefig("simple_plot.pdf")
```

La fonction `np.linspace(a,b,n)` permet de construire array avec n points entre entre a et b (par défaut, avec la même séparation entre eux). Ces instructions permettent de visualiser rapidement uns fonction.

Quand la figure devient plus complexe, c'est conseillé d'utiliser plutôt une syntaxe *orientée aux objets*, notamment

```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(x,y)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Simple plot")
plt.show()
fig.savefig("simple_plot.pdf")
```

Pour illustrer ceci, on considère la fonction $f :]-\pi,\pi] \to \mathbb{R}$ avec

$$f(x) = \begin{cases} -1, & -\pi < x < 0 \\ 1, & 0 \leq x \leq \pi \end{cases} \tag{1}$$

(on peut l'étendre à et $f : \mathbb{R} \to \mathbb{R}$ en imposant 2π-périodicité). La "série de Fourier" associée à l'expression

$$\hat{f}(x) = \frac{4}{\pi} \sum_{n=0}^{\infty} \frac{\sin(2n+1)x}{2n+1} \ . \tag{2}$$

Pour avoir une intuition graphique de la convergence, on peut écrire :

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

x = np.linspace(-np.pi,np.pi,101)
f = np.ones_like(x)
f[x<0] = -1
y1=(4/np.pi)*(np.sin(x)+np.sin(3*x)/3.0)
y2=y1+(4/np.pi)*(np.sin(5*x)/5.0+np.sin(7*x)/7.0)
y3=y2+(4/np.pi)*(np.sin(9*x)/9.0+np.sin(11*x)/11.0)

fig = plt.figure()
ax= fig.add_subplot(111)

ax.plot(x,f,'b-',lw=3,label='f(x)')
ax.plot(x,y1,'c--',lw=2,label='two terms')
ax.plot(x,y2,'r-.',lw=2,label='four terms')
ax.plot(x,y3,'b:',lw=2,label='six terms')
ax.legend(loc='best')
ax.set_xlabel('x',style='italic')
ax.set_ylabel('partial sums',style='italic')
fig.suptitle('Partial sums for Fourier series of f(x)', size=16,weight=

fig.savefig("not_so_simple_plot.pdf")
```

Remarque : Noter que la fonction `np.sin(x)` agît coefficient par coefficient sur l'array `x`.

Exercice 1. Construire, avec la fonction $arctan(x)$ un suite des fonctions $f_n(x)$ qui "convergent" vers la fonction $f : \mathbb{R} \to \mathbb{R}$, avec $f(x) = -\pi/2$, pour $x < 0$, et $f(x) = \pi/2$, pour $x \geq 0$. Visualiser simultanément quelques éléments de la suite et sa limite.

2 Résolution des équations différentielles ordinaires

Nous considérons maintenant des problèmes suivant : trouver une fonction $y : [t_o, T] \to \mathbb{R}$ solution de

$$\dot{y} = f(y(t),t), \qquad t \geq t_o, \qquad y(t_o) = y_o \ , \tag{3}$$

avec $\dot{y} = \dfrac{dy}{dt}$. Par exemple

$$y' + ay = 0, \qquad y(0) = 1 \tag{4}$$

Exercice 2. Résoudre l'équation (4) analytiquement et visualiser la solution.

Pour la résolution numérique des équations différentielles, python compte avec la fonction `odeint` dans le module

```
from scipy.integrate import odeint
```

En particulier, l'équation doit être écrite dans la forme (3). Comme exemple, nous pouvons résoudre l'équation (4) de la manière suivante

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def rhs(Y,t,a):
    y = Y
    return -a*y

t_arr=np.linspace(0,10,101)
y_init=[1]
a=1.0
y_arr=odeint(rhs, y_init, t_arr, args=(a,))
y=y_arr[:,0]
```

Exercice 3. Visualiser la résolution numérique de (4) et comparer graphiquement avec la solution analytique.

L'application de `odeint` a besoin de la construction pour le term à droite dans (3), qui doit aussi incorporer les possibles paramètres. Dans notre exemple, la fonction `rhs(Y,t,a)` proportionne cette terme à droite.

De façon plus générale, la fonction à résoudre peut être un `array`. C'est le cas des systèmes d'équations différentielles et des équations d'ordre supérieure. Comme exemple de système d'équations différentielles ordinaires, nous considérons

$$\begin{aligned} \dot{x} &= -ay \\ \dot{y} &= -bx \end{aligned} \tag{5}$$

Pour construire la solution numérique à ce problème on peut procéder de la manière suivante

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def rhs(u,t,a,b):
    x,y = u
    return -a*y, -b*x

t_arr=np.linspace(0,10.,101)
u_init=[1,0]
a, b = 1., -1.
u_arr=odeint(rhs, u_init, t_arr, args=(a,b,))
x, y = u_arr[:,0], u_arr[:,1]


fig=plt.figure()
ax1=fig.add_subplot(121)
ax1.plot(t_arr, x, t_arr, y)
ax1.set_xlabel('t')
ax1.set_ylabel('x et y')
ax2=fig.add_subplot(122)
ax2.plot(x,y)
ax2.set_xlabel('x')
ax2.set_ylabel('y')
fig.tight_layout()
#fig.subplots_adjust(top=0.90)

plt.savefig("system_EDO.pdf")
```

Pour des systèmes d'équations différentielles d'ordre supérieur on doit procéder d'abord pour réduire l'équation à une système de première ordre. C'est le but de l'exercice suivant.

Exercice 4. Considérer l'équation

$$\begin{aligned} y'' - 3y' + 2y &= 2\cos x \\ y(0) = 1, y'(0) &= 0 \ . \end{aligned} \tag{6}$$

i) Résoudre analytiquement cette équation.
ii) Re-écrire l'équation (6) comme un système de première ordre.
iii) Résoudre numériquement l'équation avec `odeint`.
iv) Visualiser analytiquement la solution numérique, la solution numérique et la différence.

3 Exemples des systèmes dynamiques

3.1 Dimension 2

3.1.1 Oscillateur harmonique

On on considère l'équation

$$\ddot{y} + \omega^2 y = 0 \tag{7}$$

qui représente une oscillation sinusoïdale de fréquence ω autour un point de repos $y = 0$. En particulier, dans le cas du pendule simple $y(t) = \theta(t)$, il s'agît de l'approximation linéaire de l'équation du pendule

$$\ddot{\theta} + \omega^2 \sin(\theta) = 0 \tag{8}$$

Exercice 5. Résoudre numériquement l'équation (7) pour différents valeurs initiales. Comparer avec les solutions de (8) et estimer l'amplitude initiale à partir de laquelle l'approximation linéaire n'est plus valable. Étudier et comparer les diagrammes de phases (c'est à dire, les trajectoires dans l'espace de solutions $(theta, \dot{\theta})$) de ces deux systèmes.

Comme ouverture, écrire le code suivant et le sauver dans un fichier python, exemple "trajectoires.py".

```
#%% Resolution of harmonic oscillator 2
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def rhs(Y,t, omega):
    y,ydot =Y
    return ydot, -omega**2*y

t_arr=np.linspace(0,2*np.pi,101)
y_init=[1,0]
omega=2.0

fig=plt.figure()
y,ydot=np.mgrid[-3:3:21j, -6:6:21j]
u,v=rhs(np.array([y,ydot]),0.0,omega)
mag=np.hypot(u,v)
mag[mag==0]=1.0
plt.quiver(y,ydot,u/mag,v/mag,color='red')

#Permet dessiner un nombre arbitraire de trajectoires
print "\n\n\nUtilise la souris pour choisir le point de depart"
print "Timout, apres 30 seconds"
print "\n\n"
choice=[(0,0)]
while len(choice)>0:
    y01=np.array([choice[0][0],choice[0][1]])
    y=odeint(rhs,y01,t_arr,args=(omega,))
    plt.plot(y[:,0],y[:,1],lw=2)
    choice=plt.ginput()
print "Le temps est passe !"
```

Exécuter le programme depuis une terminale en faisant

```
python trajectoires.py
```

Répéter le code pour l'équation non-linéaire du pendule simple.

3.1.2 Oscillateur de van der Pol

Une équation non-linéaire qui récupère comme cas particulière l'oscillateur harmonique est l'oscillateur de van der Pol

$$\begin{aligned} &\ddot{y} - \mu(1 - y^2)\dot{y} + y = 0 \ , \\ &y(0) = y_0 \ , \ \dot{y}(0) = y_1 \end{aligned} \tag{9}$$

La non-linéarité fait que l'équation devient "stiff", dur aux grands gradients dans la solution. Dans ce situation, op peut optimiser l'algorithme dedans `odeint` si on proportionne la matrice de Jacobi du terme à droite, quand on écrit l'équation comme un système de premier ordre. Si le terme à droite de

l'équation est écrit comme

$$\mathbf{f}(y,\dot{y}) = \begin{pmatrix} f_1(y,\dot{y}) \\ f_2(y,\dot{y}) \end{pmatrix} , \tag{10}$$

alors la matrice de Jacobi est

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial \dot{y}} \\ \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial \dot{y}} \end{pmatrix} \tag{11}$$

Exercice 6. Calculer la matrice de Jacobi correspondant au terme à droite dans l'équation de van de Pol et l'implémenter une fonction python `def jac(y,t,mu):`

Implementer le code suivant pour résoudre et visualiser les solutions aux équations de Van der Pol et explorer le comportement avec μ.

```
#%% Van der Pol
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def rhs(y,t,mu):
    return [y[1], mu*(1-y[0]**2)*y[1] - y[0]]

    def jac(y,t,mu): # remplir ici avec la fonction
                     #construite dans l'exercice 6

mu=1.0
t_final = 15.0 if mu<10 else 4.0*mu
n_points = 1001 if mu<10 else 1001*mu
t=np.linspace(0, t_final,n_points)
y0=np.array([2.0,0.0])
y,info=odeint(rhs,y0,t,args=(mu,),Dfun=jac,full_output=True)

print " mu = %g, le nombre d'appels au Jacobien est %d" % \
            (mu, info['nje'][-1])

plt.plot(y[:,0],y[:,1])
plt.savefig("Van_der_Pol.pdf")
```

d'une partie, que les périodes ne soient pas de μ (dans l'oscillateur ar

3.2 Dimension 3

3.2.1 Équations de Kepler (coordonnées Cartésiennes)

Les équations de Newton appliqués à la loi de la Gravitation universelle entre deux particules de masse M et m peuvent être écrites (en faisant $G = 1$)

$$\ddot{\vec{r}} = -\frac{Mm}{(x^2+y^2+z^2)^{\frac{3}{2}}}\vec{r} , \tag{12}$$

avec $\vec{r}(x,y,z)$. La visualisation en trois dimensions demande la construction des "axes à dimension 3. Pour construire les axes tri-dimensionnels il faut importer `from mpl_toolkits.mplot3d import Axes3D` et après la constrution d'un objet `figure` on l'applique la fonction `ax=Axes3D(fig)`

Exercice 7. Résoudre numériquement les trajectoires (kepleriennes) d'une particule de masse m déterminées par l'équation (refkepler). Explorer visuellement la limite entre trajectoires elliptiques et hyperboliques en fonction des données initiales.

3.2.2 Équations de Lorentz

Finalement, les équations de Lorentz comme modèle non-linéaire pour le climat planétaire son données par

$$\begin{cases} \dot{x} &= \sigma(y-x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= xy - \beta z \end{cases} \tag{13}$$

Ces équations présentent un comportement "chaotique", en particulier une grande sensitivité aux conditions initiales, pour de valeurs de $\rho > \rho_H \sim 24.7$.

Exercice 8. Résoudre numériquement les équations (13) pour $\sigma = 10$ et $\beta = 8/3$ et visualiser les trajectoires. Explorer les résultats en fonction de différents valeurs de ρ et des conditions initiales.