Informatique en PCSI et MPSI
Champollion 2013-2014
Méthodes d'Analyse Numérique
Implémentation et Application en Python
Équations différentielles ordinaires

A. HASSAN

19 février 2014

Résolution des équation différentielles ordinaires (EDO)

Soit I un intervalle de $\mathbb{R}$, $\mathcal{D}$ un ouvert de $\mathbb{R}^n$ et f une fonction **connue** de $\mathbb{R}^n \times I \longrightarrow \mathbb{R}^n$. On cherche $y : I \to \mathbb{R}^n$ telle que

$$y'(t) = f(y(t);t) \text{ où } f(y(t);t) = \begin{bmatrix} f_1(y_1(t), \cdots, y_n(t);t) \\ \vdots \\ f_n(y_1(t), \cdots, y_n(t);t) \end{bmatrix} \text{ et } y(t) = \begin{bmatrix} y_1(t) \\ \vdots \\ y_n(t) \end{bmatrix}$$

Ce type d'équations s'appelle **Équation résolue en** y'.

Par contre $\sin(t.y' + \cos(y' + y)) = t.y + y'$ n'est pas résolue en y'.

I.e. Impossible d'écrire $y' = f(t,y)$ (avoir y' explicitement en fonction de t et y)
Ordinaire : la dérivation est effectuée par rapport à t uniquement.

Équation aux dérivées partielles(EDP) : $\dfrac{\partial T}{\partial t} = \dfrac{\partial^2 T}{\partial x^2} + \dfrac{\partial^2 T}{\partial y^2}$, fonction inconnue $T(t,x,y)$ (Équation de la chaleur ou de la diffusion)

Problème de Cauchy (ou de la condition initiale) :

$$\text{Trouver } t \mapsto y(t) = \begin{bmatrix} y_1(t) \\ \vdots \\ y_n(t) \end{bmatrix} \text{ de } I \longrightarrow \mathbb{R}^n \text{ telle que}$$

$$\begin{cases} y(t_0) = y_0 \\ y'(t) = f(y(t);t) \end{cases}$$

Problème des conditions aux limites :

Cas où l'on dispose d'informations sur y à des valeurs différentes de t.

Conditions initiales

$$\begin{cases} y_1'(t) = -y_1(t) + y_2(t) + \sin(t) \\ y_2'(t) = 2y_1(t) - 3y_2(t) \\ (y_1(0); y_2(0)) = (0, 3) \end{cases}$$

Conditions aux limites

$$\begin{cases} y_1'(t) = -y_1(t) + y_2(t) + \sin(t) \\ y_2'(t) = 2y_1(t) - 3y_2(t) \\ y_1(0) = 2 \\ y_2(\frac{\pi}{4}) = -1 \end{cases}$$

Problème de Cauchy

Sauf quelques cas particuliers, il est presque impossible de trouver des solutions analytiques(i.e. se ramener à un calcul de primitives).
On cherche des solutions approchées, donc Résolutions numériques.
Principe : Si l'on connaît y à l'instant (abscisse) t, comment obtenir y à l'instant $t+h$, $y(t+h)$, puis recommencer avec $y(t+h)$?


Choisir h assez petit et utiliser les Développements limités

$$y(t+h) = y(t) + hy'(t) + \frac{1}{2}h^2y''(t) + \cdots + o(h^n) \xrightarrow{y'(t)=f(t,y(t))} = y(t) + \Phi(t,h,f(y,t)) + o(h^n)$$

On cherche $\Phi(t,h,f(y,t))$ de telle sorte l'ordre de $o(h^n)$ soit le plus élevé possible (en minimisant le coût et la complexité des calculs).

Méthodes de résolution : Ordres 1 et 2

- Le **plus souvent** $h = (t_f - t_0)/N$ où $[t_0; t_f]$ désigne l'intervalle de résolution et N le nombre de sous-intervalles.
- **Sinon** : On considère $T = \{t_0; t_1; \cdots; t_{n-1}; t_n\}$ et $h_i = (t_{i+1} - t_i)$

Ainsi $y(t)$ est l'estimation courante en t et $y(t+h)$ l'estimation au pas suivant

Pour $n = 1 \Rightarrow$ Méthode d'Euler (ou Runge Kutta d'ordre 1).

$$y(t+h) \approx y(t) + hf(t, y(t))$$

Pour $n = 2 \Rightarrow$ Méthode de Runge Kutta d'ordre 2.

$$K_1 = h.f(t, y(t))$$
$$K_2 = h.f(t + \frac{1}{2}h; \frac{1}{2}K_1 + y(t))$$
$$y(t+h) \approx y(t) + K_2$$

Méthode d'Euler ou RK1 (Runge-Kutta d'ordre 1)

Principe de la méthode :

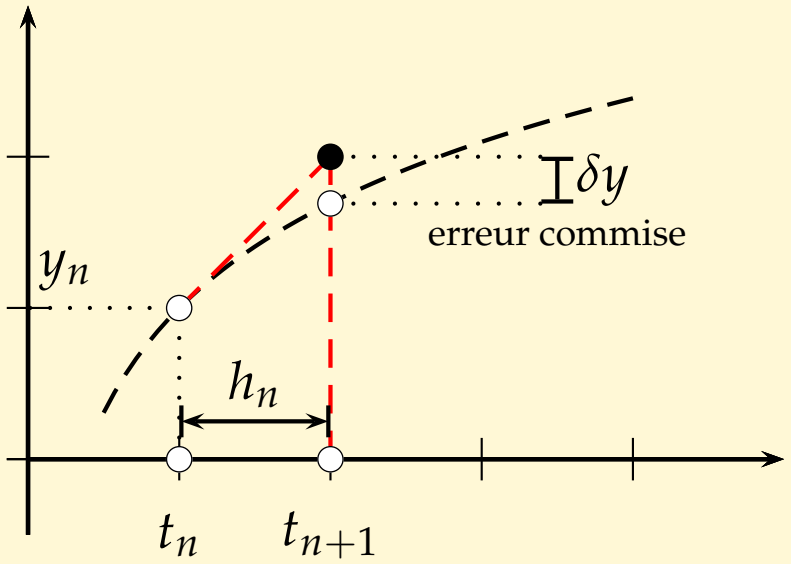
Chercher une solution approchée de $\begin{cases} y' = f(t;y) \\ y_0 = y(t_0) \end{cases}$ sur l'intervalle $I = [t_0; t_f]$

pas d'intégration : $h = (\frac{t_f - t_0}{N})$ où

$h = (t_{i+1} - t_i) \Rightarrow y(t_i + h) \approx y(t_i) + h_i.f(y(t_i), t_i)$

Géométriquement : Remplacer la courbe sur $[t_i; t_i + h]$ par la tangente.

Si $M_n(t_n; y_n)$ est le point courant de la courbe "solution", le nouveau point : $M_{n+1}(t_n + h; y_n + hf(t_n, y_n))$.



Algorithme d'Euler (Runge Kutta d'ordre un)

Euler (f, y_0, t_0, t_f, N)

Entrées : f fonction données
$(t_0; y_0)$ point initial
t_f abscisse final
N nombre de points de $[t_0; t_f]$

Sorties : Ly liste des ordonnées y_k, $k = 0; 1; \cdots; N$

$h \leftarrow \dfrac{(t_f - t_0)}{N}$

$L_y \leftarrow y_0$, $L_t \leftarrow t_0$

pour k *de* 1 *à* N **faire**

- $y_0 \leftarrow y_0 + h.f(t_0, y_0)$
- $t_0 \leftarrow t_0 + h$
- $L_y \leftarrow L_y, y_0$; # stocker les solutions
- $L_t \leftarrow L_t, t_0$ # stocker les abscisses

retourner L_y *et* L_t

Implémentation de l'algorithme d'Euler

1. On charge les modules `numpy` (ou `scipy`) pour les calculs numériques et `matplotlib.pyplot` (ou `pylab`) pour les graphiques.
2. La fonction $(y;t) \mapsto f(y;t)$ est définie au préalable.
3. On rappelle que $y_0 = y(t_0) \in \mathbb{R}^n$ éventuellement

```
1  import numpy as np
2  import matplotlib.pyplot as mp
3  ##
4  def Euler(f,t0,y0,tf,N):
5      h=.........              # definiton du pas fixe
6      # sinon h=t[i+1]-t[i]
7      Ly,Lt=[...],[...]        # stockage du point initial
8      for k in ..........:     # A la limite Lt peut etre donne ailleurs
9  # prevoir que y0 peut etre vectoriel
10         y0=.........................
11         t0+=........................
12         ............................
13         ............................
14     return array(Lt),array(Ly)
15 ##
```

Implémentation de la méthode d'Euler en Python

1. On charge les modules `numpy` (ou `scipy`)pour les calculs numériques et `matplotlib.pyplot` (ou `pylab`)pour les graphiques
2. La fonction $(y;t) \mapsto f(y;t)$ est définie au préalable.
3. On rappelle que $y_0 = y(t_0) \in \mathbb{R}^n$ éventuellement

Version pédagogique

```
1  import numpy as np
2  import matplotlib.pyplot as mp
3  ##
4  def Euler(f,t0,y0,tf,N):
5      h=(tf-t0)/N
   # definition du pas fixe
6      # sinon h=t[i+1]-t[i]
7      Ly,Lt=[y0],[t0]
   # stockage du point initial
8      for k in range(N):
   # A la limite Lt peut etre donne ailleurs
9  # prevoir que y0 peut etre vectoriel
10         y0=y0+h*np.array(f(t0,y0))
11         t0+=h
12         Ly.append(y0)
13         Lt.append(t0)
14     return np.array(Lt),np.array(Ly)
15 ##
```

Version optimisée

```
1  import numpy as np
2  def euler(dXdt,X0,t):
3      n=len(t)
4      X=np.zeros((len(t),)+np.shape(X0))
5      X[0]=X0
6      for i in range(n-1):
7          h=(t[i+1]-t[i])
8          X[i+1]=X[i]+dXdt(X[i],t[i])*h
9      return X
```

Runge Kutta d'ordre 4 (RK4)

Soit $(t; y(t))$ un point courant, $y(t+h)$ est une estimation de la solution en $t+h$

Pour $n = 4 \Rightarrow$ Méthode ou Runge Kutta d'ordre 4.

$$K_1 = h.f(y(t); t);$$
$$K_2 = h.f(\frac{1}{2}K_1 + y(t); t + \frac{1}{2}h)$$
$$K_3 = h.f(y(t) + \frac{1}{2}K_2; t + \frac{1}{2}h);$$
$$K_4 = hf(y(t) + K_3 : t + h)$$
$$y(t+h) \approx y(t) + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$

Il existe des méthodes à pas adaptatif (h n'est pas fixe).

Méthode de Runge-Kutta d'ordre 4 en Python

On charge au préalable le module `numpy` (ou `scipy`). La fonction f est pointée par `dXdt`.

```
1  import numpy as np
```

Voici un code de RK4 :

```
1  def rk4(dXdt,X0,t):# Runge Kutta d'ordre 4
2      """
3      Entrees:
4          t        : contient les points de la subdivision
5          X0       : les conditions initiales
6          dXdt     : la fonction connue
7      Sorties:
8          X        : Solutions approchees aux points de t """
9      #import numpy as np
10     n=len(t)
11     X=np.zeros((len(t),)+np.shape(X0))  # on construit une matrice de 0
12     X[0]=X0                             # stocker les valeurs initiales
13     for i in range(n-1):
14         k1=...............              # k1=f(y,t)
15         h=...............
16         k2=...............
17         k3=...............
18         k4=...............
19         X[i+1]=X[i]+...............     # on a deja divise h par 2
20     return(X)
```

Méthode de Runge-Kutta d'ordre 4 en Python

On charge au préalable le module `numpy` (ou `scipy`)

```
1  import numpy as np
```

Voici un code de RK4 :

```
1  def rk4(dXdt,X0,t):# Runge Kutta d'ordre 4
2      """
3      Entrees:
4          t        : contient les points de la subdivision
5          X0       : les conditions initiales
6          dXdt     : la fonction connue
7      Sorties:
8          X        : Solutions approchees aux points de t """
9      #import numpy as np
10     n=len(t)
11     X=np.zeros((len(t),)+np.shape(X0))  # on construit une matrice de 0
12     X[0]=X0                             # stocker les valeurs initiales
13     for i in range(n-1):
14         k1=dXdt(X[i],t[i])             # k1=f(y,t)
15         h=(t[i+1]-t[i])/2
16         k2=dX_dt(X[i]+h*k1,t[i]+h)
17         k3=dX_dt(X[i]+h*k2,t[i]+h)
18         k4=dX_dt(X[i]+2*h*k3,t[i]+2*h)
19         X[i+1]=X[i]+h/3*(k1+2*k2+2*k3+k4)# on a deja divise h par 2
20     return(X)
```

Utilisation de la commande `odeint` du module `scipy.integrate`

Le module `scipy.integrate` fournit la commande `odeint` qui est un **"mix"** de Runge Kutta d'ordre 4 avec des méthodes à pas adaptatif.

Syntaxe et utilisation :

1. chargement du module adéquat :
   ```
   from scipy.integrate import odeint
   ```
2. forme simplifiée :
   ```
   odeint(func,CondInit,t)
   ```
3. forme complète :
   ```
   odeint(func, y0, t, args=(), Dfun=None,
   col_deriv=0, full_output=0, ml=None, mu=None,
   rtol=None, atol=None, tcrit=None, h0=0.0,
   hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0,
   mxordn=12, mxords=5, printmessg=0)
   ```

Commande `odeint` (suite)

Forme simplifiée :

```
Ys=odeint(f, CondInit, t)
```

1. `f` pointe sur la fonction CONNUE f dans $y'(t) = f(y(t), t)$. **Attention** : le premier argument de f est la fonction vectorielle inconnue y.
2. `CondInit` pointe sur le vecteur des conditions initiales

$$y(t_0) = \begin{bmatrix} y_1(t_0) \\ \vdots \\ y_n(t_0) \end{bmatrix}$$

3. `t` pointe sur les points définissant le découpage de l'intervalle de résolution $I = [t_0; t_0 + T]$

La solution `Ys` contient les valeurs des solutions approchées sur les différents points indiqués de I

Exercice : L'équation de Van Der Pol (1924)

Il s'agit de l'équation différentielle

$$y'' + f(y^2 - 1)y' + y = 0 \qquad (f \geq 0)$$

Où $f.(y^2 - 1)$ représente un facteur d'amortissement non linéaire ($f \in \mathbb{R}_+$).

1. Préciser à quoi correspond le cas $f = 0$? $f \neq 0$ et $a \neq 0$?.
2. Écrire l'équation de Van der Pol sous forme d'une équation vectorielle de premier ordre.
3. Montrer que 0 est le seul point d'équilibre et qu'il n'est pas stable.
4. Expliquer qualitativement le comportement lorsque $y \approx 0$ ou bien lorsque y croît.

Pour les simulations on prend
$(y_0; y'_0) \in \{r(\cos(\frac{k\pi}{5}); \sin(\frac{k\pi}{5})) : k \in [\![0, 9]\!] \text{ et } r \in \{1; 3\}\}$

Préambule

```
## modules a charger
from scipy import *
from matplotlib.pyplot import *
from scipy.integrate import odeint
```

Programmation du système correspondant à l'équation de Van Der Pol :

```
def VdPol(.....):   #y'' + f(y^2 - 1)y' + y = 0
    ..........
    ..........
    ..........
    ..........
    ..........
#
```

Simulation et utilisation de `odeint` :

```
t=linspace(..........)
ci=[....,....]
Ys=odeint(....., ....., ..........)
```

Réponses Temporelles Si $Y(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix}$ solution de $Y' = f(Y,t)$. Les courbes temporelles correspondent aux courbes $(t; y_1(t))$ et $(t; y_2(t))$

```
clf()
# Tracer la reponse temporelle
plot(    ,    ,'r') # premiere composante
plot(    ,    ,'g') # seconde composante
grid();             # grille
show()
```

Portraits de phase : Le portrait de phase correspond la trajectoire $M(t) = (y_1(t); y_2(t))$

```
CIs=[[  ,  ],[  ,  ],[  ,  ],[  ,  ]]
# ensemble de conditions initiales
for n in ..............:

    Ysol=................

    plot(........................)
show()
```

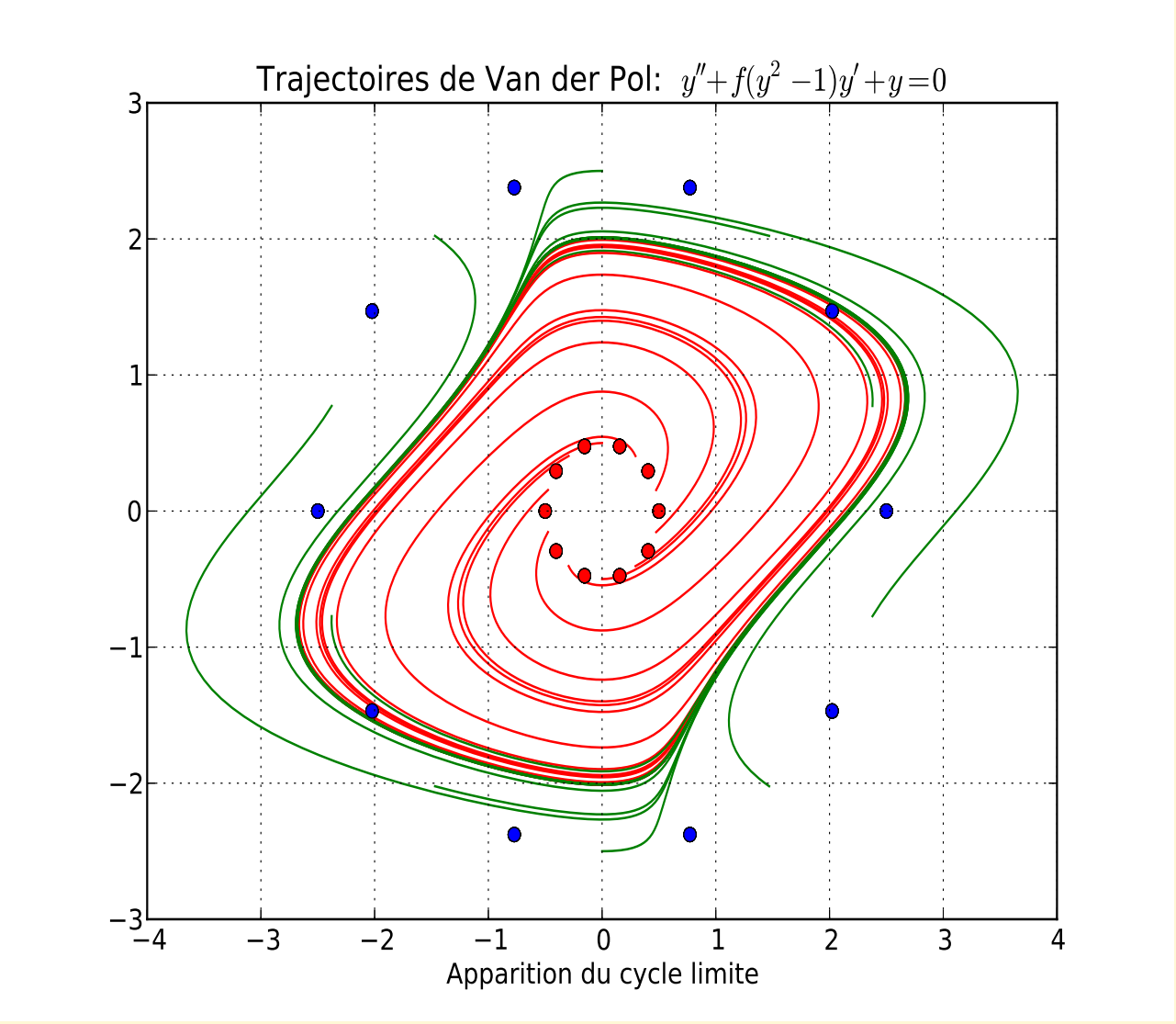
Exploitation graphique : Champs de vecteurs

On voudrait en chaque point M du plan faire apparaître un vecteur $\vec{v}$ tangent à la courbe intégrale passant par ce point (i.e. $\vec{v}$ colinéaire à $(1; f(y,t))$ dans notre cas)

```
## Creation des champs de vecteurs
x=linspace(-4,4,20)      # subdiviser l'intervalle de x [-4;4]
y=linspace(-4,4,20)      # subdiviser l'intervalle de y [-4;4]
X1,Y1=meshgrid(x,y)      # grille de noeuds en x, y
dX,dY=VdPol([X1,Y1],0)   # generer les vecteurs tangents
M=hypot(dX,dY)           # normalisation
M[M==0]=1.               # Normes d'eventuels vecteurs nuls
dX /= M                  # remplaces par 1 avant division
dY /= M
quiver(Y1,X1,dY,dX,M)    # generation du champs de vecteurs
show()
##
```

```
## Resolution de l'equation de Van der Pol y''+f*(y^2-1)y'+y=0
def VdPol(y,t):#
    """ Attention d'abord y(t) puis t """
    f=1
    v=y[0];              # y
    vp=y[1];             # y'
    vpp=f*(1-v**2)*vp-v # y''=-y+f(1-y^2)y'
    return array([vp,vpp])
##
t=linspace(0,20,1000)   # on simule sur 1000 points de T=[0;20]
# genere des conditions initiales
ci=array([[0.5*cos(k*pi/5),0.5*sin(k*pi/5)] for k in range(10)])
ci2=array([[2.5*cos(k*pi/5),2.5*sin(k*pi/5)] for k in range(10)])
#
for i in range(10):
    Ys=odeint(VdPol,ci[i],t)
    Yys=odeint(VdPol,ci2[i],t)
    Vp=Ys[:,1]; V=Ys[:,0]       # recuperer les reponses temporelles
    plot(Vp,V,'r')              # le portrait de phase (rouge)
    plot(Yys[:,1],Yys[:,0],'g') # encore des portraits de phases
    plot(ci[:,0],ci[:,1],'o')
    plot(ci2[:,0],ci2[:,1],'ob')
title("Trajectoires de Van der Pol:  y''+f(y^2-1)y'+y=0")
xlabel("Apparition du cycle limite"); grid()
savefig("G:\\Azzam\\MyPython\\TpChampollion\\VanDerPol.eps")
show()
```

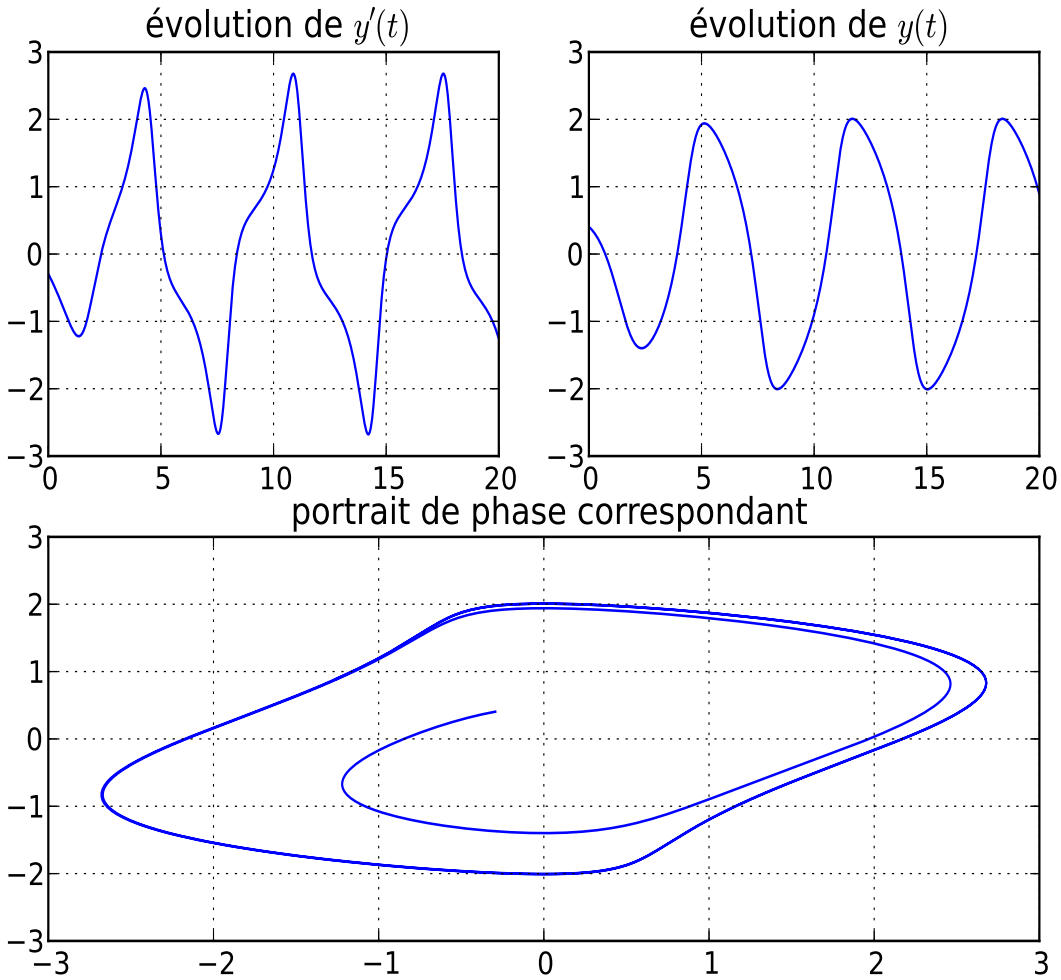
Portrait de phase



Réponses temporelles :

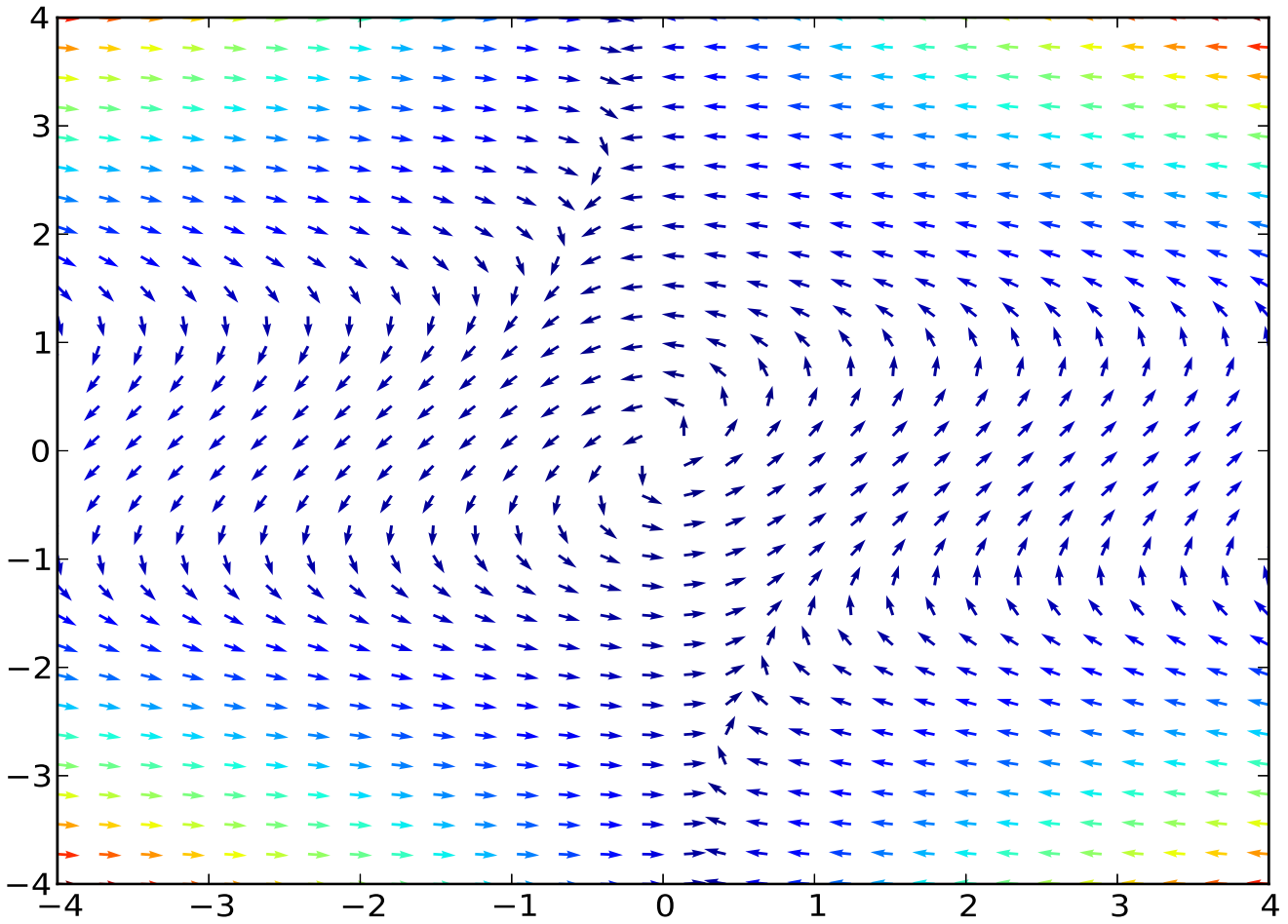
```
1  subplot(221)
2  plot(t,Vp); grid();
3  title("␣évolution␣de␣y'(t)")
4  subplot(222)
5  plot(t,V); grid();
6  title("␣évolution␣de␣y(t)")
7  subplot(212)
8  plot(Vp,V); grid();
9  title("␣portrait␣de␣phase␣correspondant")
10 show()
11 savefig("C://Users//David//Desktop//Info//Python//Cours//analysenum//RepTempo.eps")
```

Solution : Graphiques(suite)

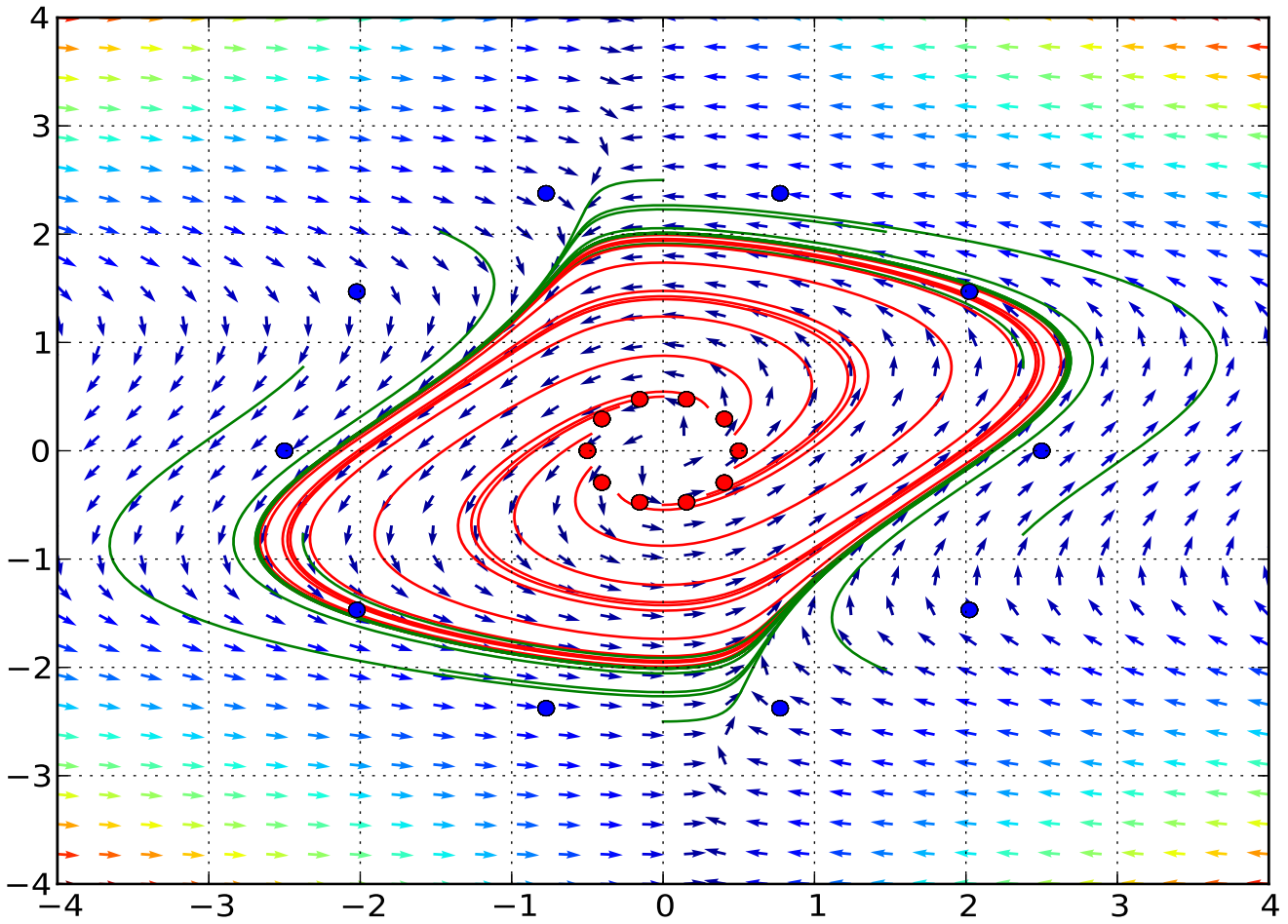


Solution : Champs de vecteurs

```
clf()
x=linspace(-4,4,30)
y=linspace(-4,4,30)
X1,Y1=meshgrid(x,y)
dX,dY=VdPol([X1,Y1],0)
M=hypot(dX,dY)
M[M==0]=1.0
dX /= M
dY /= M
#
quiver(Y1,X1,dY,dX,M)
show()
savefig("C://Users//David//Desktop//Info//Python//Cours//analysenum//ChampVdPol.eps")
```


4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4

Solution : Superposition Portrait de phase/ champs de vecteurs

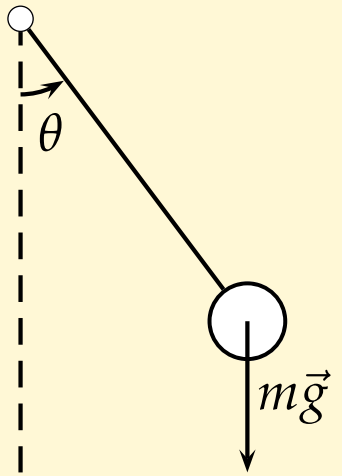


Exemple : Le pendule simple (non amorti)

Soit une masse m suspendu à un point fixe par un fil non extensible de longueur ℓ.
Les lois de la mécanique impliquent :

$$m\ell^2\theta'' = -mg\ell\sin(\theta) \iff \boxed{\theta'' = -\frac{g}{l}\sin(\theta)}$$

1. θ angle de l'écart avec la verticale en radians, $\theta'(t)$ (resp. $\theta''(t)$) vitesse (resp. accélération)angulaire
2. ℓ longueur en mètre.
3. g accélération de la pesanteur en $\mathrm{m.s}^{-2}$
4. À l'instant $t = 0$, $\theta = \theta_0$ écart par rapport à la verticale et $\theta'_0 = 0$ (vitesse initiale nulle)



Question : Trouver la loi du mouvement $t \mapsto \theta(t)$?

Pas de solution exprimable à l'aide des fonction usuelles (composées de $\exp()$ et de $\ln()\ldots$), (on démontre même que cela est impossible).

Remarque : Résolution pour des "*petits*" θ :

$$\xrightarrow{\sin(\theta)\approx\theta} \theta(t) = \theta_0\cos(\omega t) \text{ où } \omega = \sqrt{\frac{g}{\ell}}$$

On cherche une fonction $y : \mathbb{R}_+ \longrightarrow I \times \mathbb{R}$ où $I = [-\pi; \pi]$ où

$$y = \begin{bmatrix} \theta \\ \theta' \end{bmatrix} \text{ où } \begin{cases} \theta & \text{position angulaire} \\ \theta' & \text{vitesse angulaire} \end{cases}$$

$$y' = \begin{bmatrix} \theta' \\ \theta'' \end{bmatrix} = \begin{bmatrix} \theta' \\ -\frac{g}{\ell}\sin(\theta) \end{bmatrix} = \begin{bmatrix} y_2 \\ -\frac{g}{\ell}\sin(y_1) \end{bmatrix}$$

Donc

$$\begin{aligned} f : \quad & I \times \mathbb{R} \longrightarrow \mathbb{R} \times \mathbb{R} \\ & (x_1; x_2) \mapsto (x_2; -\frac{g}{\ell}\sin(x_1)) \end{aligned}$$

L'équation devient (Problème de Cauchy) :

$$y' = f(y) \qquad \text{avec } y(0) = \begin{bmatrix} \theta_0 \\ 0 \end{bmatrix}$$

Équation différentielle **autonome**(stationnaire) : f ne dépend pas de t **explicitement**.

Exemples : Résolution de $y'' + y = 0$ (petite oscillation du pendule)

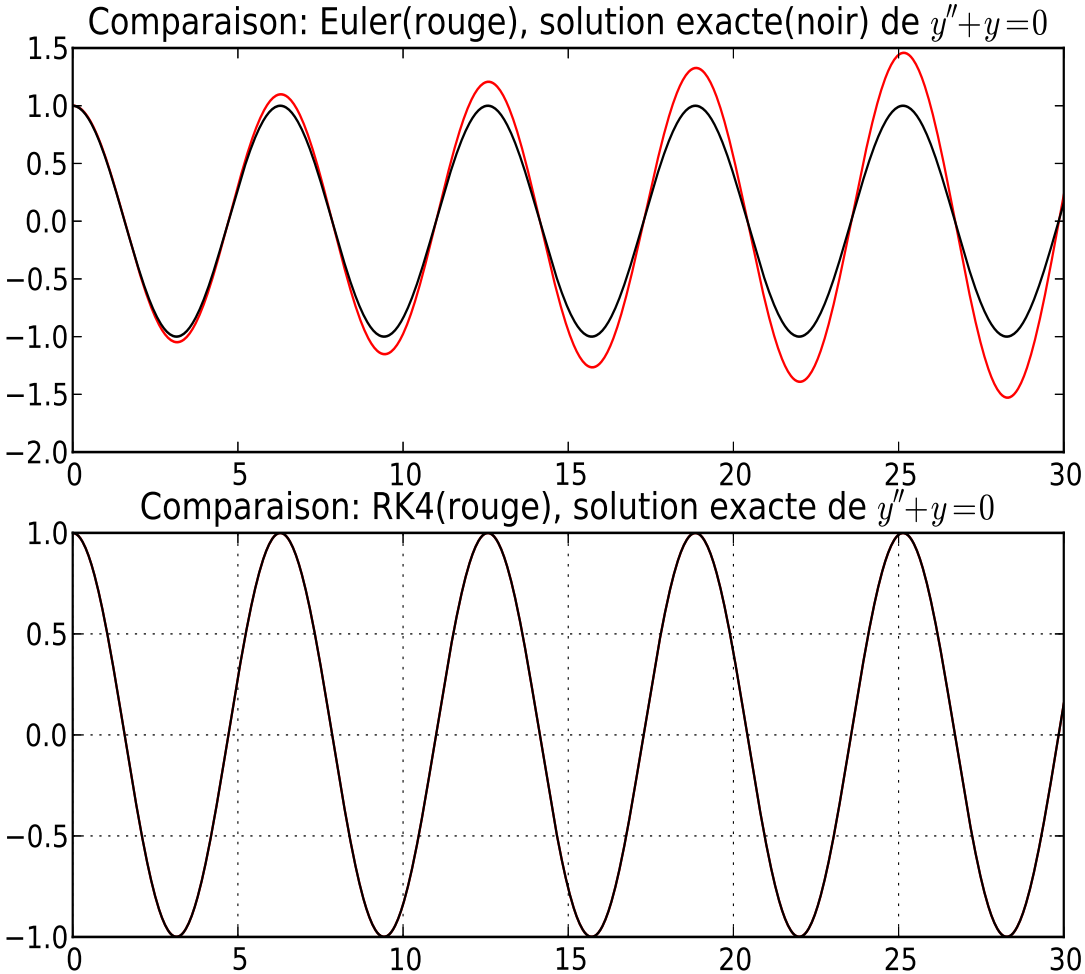
On dispose d'une solution exacte de $y'' + y = 0$ (n'est-ce pas ?) où $(y(0); y'(0)) = (a, b)$:

$$y(t) = a\cos(t) + b\sin(t)$$

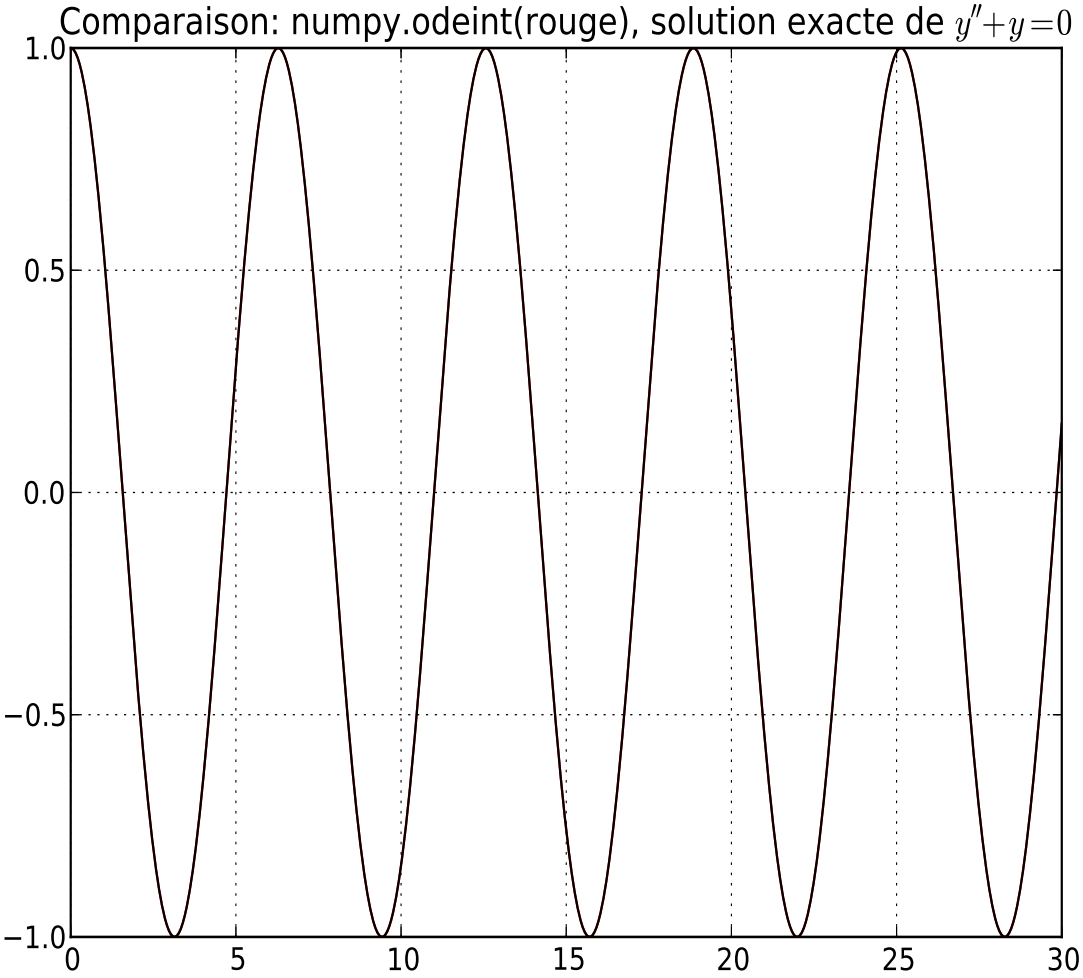
```
## Un exemple: y''+y=0
def f(y,t):# definition de la fonction vectorielle
    """ y(1)= ypp,
        y(2)= yp """
    return(np.array([-y[1] , y[0] ]))
#
t=np.linspace(0,30,1000) # découpage de l'intervalle [0,10] en 1000

# utilisation des méthodes d'Euler et RK4 programmées
# cond. init.(y'(0),y(0)) = (0;1)
ci=np.array([0,1])
# solution par Euler
sol1=euler(f,ci,t)
# solution par RK4
sol2=rk4(f,ci,t)
# solution exacte de y''+y=0
solexact=np.cos(t)# y(t) = cos(t)
# utilisation de la methode odeint de numpy.integrate
sol3=sint.odeint(f,ci,t)
#
```

```
# Exploitation graphique
py.subplot(211)
py.title("Comparaison: Euler(rouge), solution exacte(noir) de y'' + y = 0")
py.plot(t,sol1[:,1],color='red')
py.plot(t,solexact,color='black')
#
py.subplot(212)
py.title("Comparaison: RK4(rouge), solution exacte de y'' + y = 0")
py.plot(t,sol2[:,1],color='red')
py.plot(t,solexact,color='black')
py.grid(True)
py.savefig("edo1.eps")# sauvegarder l'image dans un fichier, SI JE VEUX
py.subplot(111)
py.title("Comparaison: numpy.odeint(rouge), solution exacte de y'' + y = 0")
py.plot(t,sol3[:,1],color='red')
py.plot(t,solexact,color='black')
py.grid(True)
py.savefig("edo2.eps")
```

Résultats graphiques



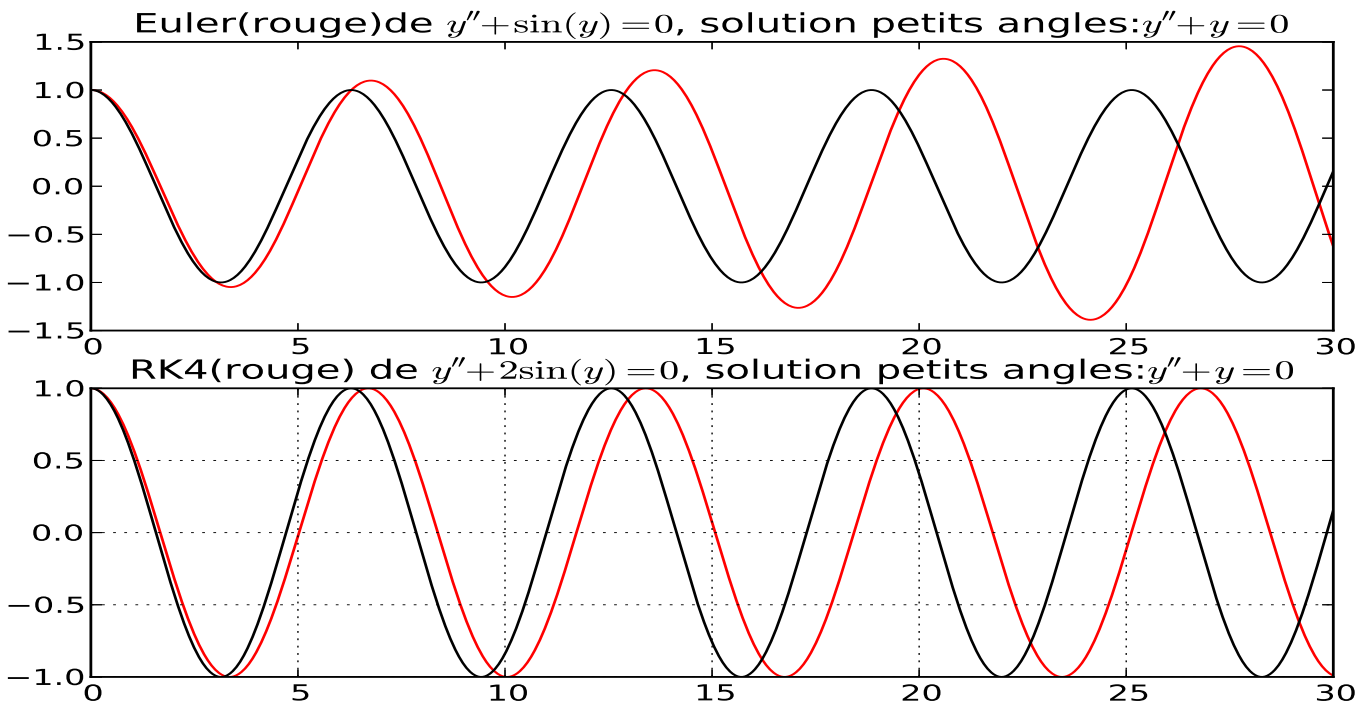
Résultats graphiques

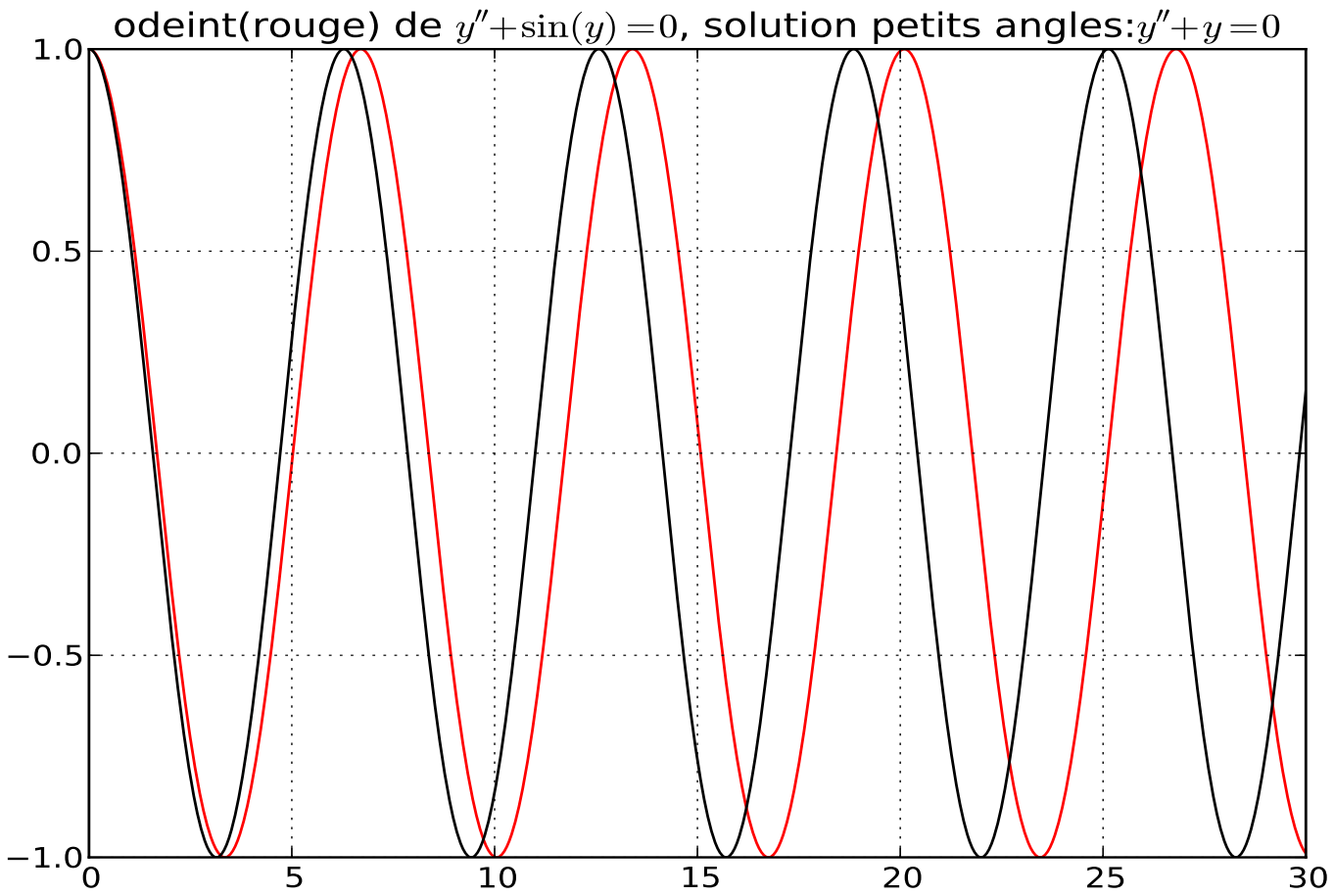


Pendule simple : $y'' + \omega^2 \sin(y) = 0$

Cette fois on ne fait pas l'approximation $\sin(y) \approx y$.
Pas de solution exacte simple, la comparaison avec la solution "petits angles" donne :

```
1 ## Resolution de y''+sin(y)=0
2 def f(y,t):
3     return(np.array([-np.sin(y[1]) , y[0] ]))
```

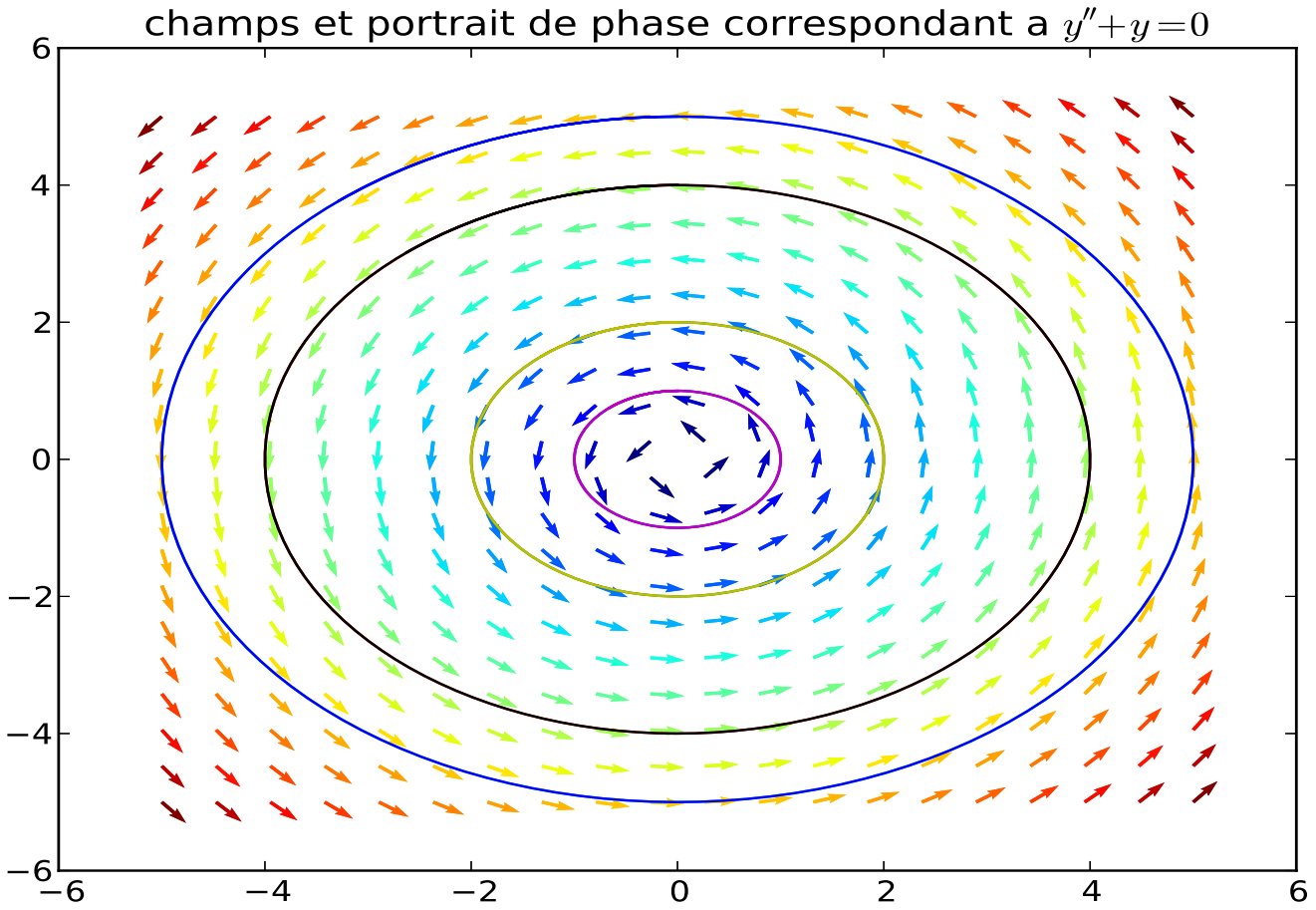



odeint(rouge) de $y''+\sin(y)=0$, solution petits angles:$y''+y=0$
1.0
0.5
0.0
−0.5
−1.0
0
5
10
15
20
25
30

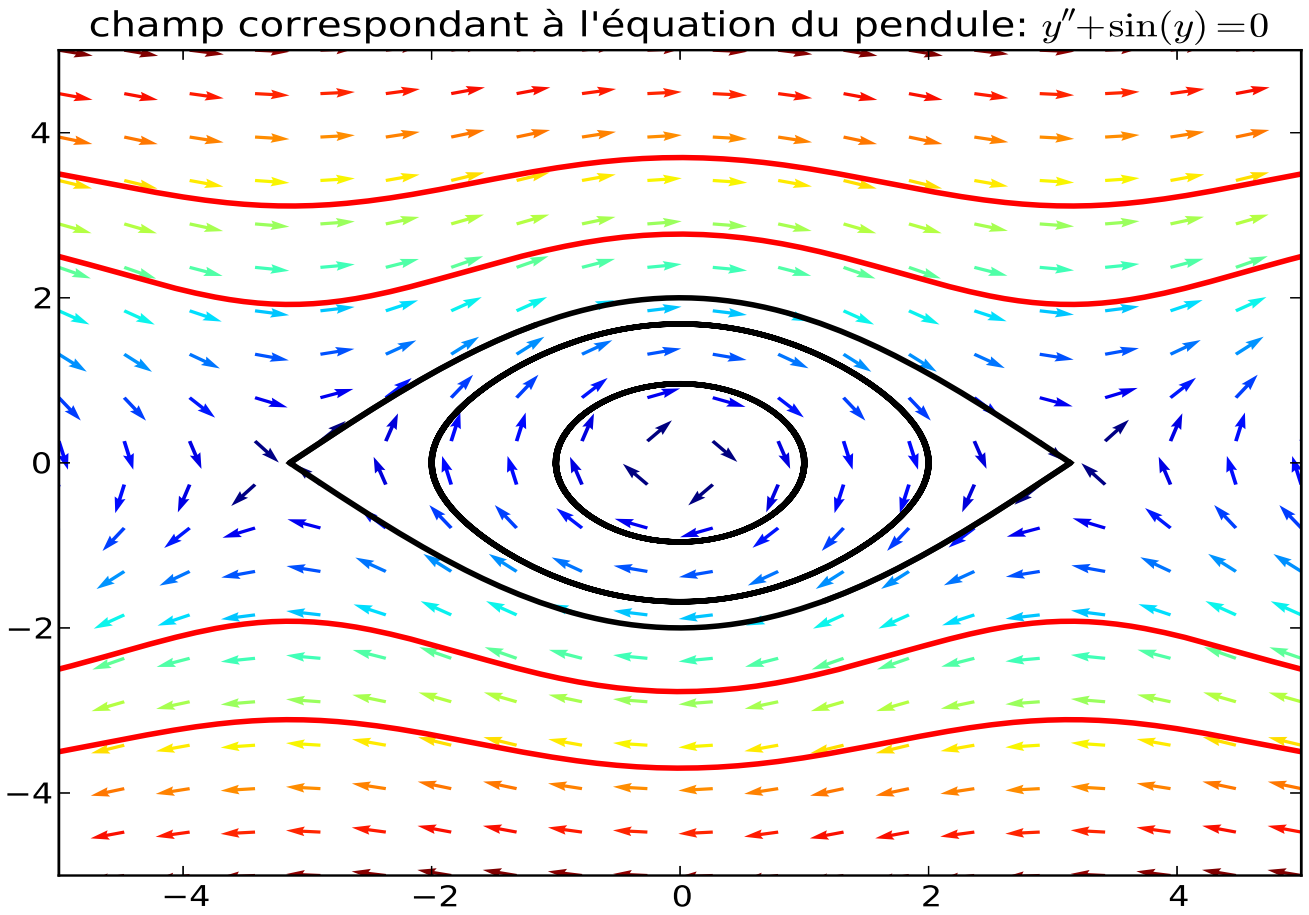
Construction du portrait de phases

```
t=np.linspace(0,7,300)
#On stocke pour plusieurs cond init.
courbes=[\
    ([0,1],t),\
    ([0,2],t),\
    ([0,4],t),\
    ([0,5],t)]
#
for i in range(len(courbes)):
    sol=sint.odeint(f,courbes[i][0],courbes[i][1])
    py.plot(sol[:,0],sol[:,1])# portrait de phase
#dessin du champs de vecteurs
#creation d'une grille
x=np.linspace(-5,5,20)
y=np.linspace(-5,5,20)
X,Y=py.meshgrid(x,y)
#calcul des vecteurs du champ
DX,DY=f([X,Y],0)
#normalisation
N=py.hypot(DX,DY)
DX/=N; DY/=N
#dessin du champ le dernier paramètre permet\
#de coloriser en fonction de la norme
py.quiver(X,Y,DX,DY,N)
py.title("champs et portrait de phase correspondant a $y''+y=0$")
py.grid()
py.savefig("phase1.eps")
```

Portrait de phase de $y'' + y = 0$: Visualisation



Portrait de phase de $y'' + \sin(y) = 0$: Visualisation



Attracteur de Lorenz(vers 1963)

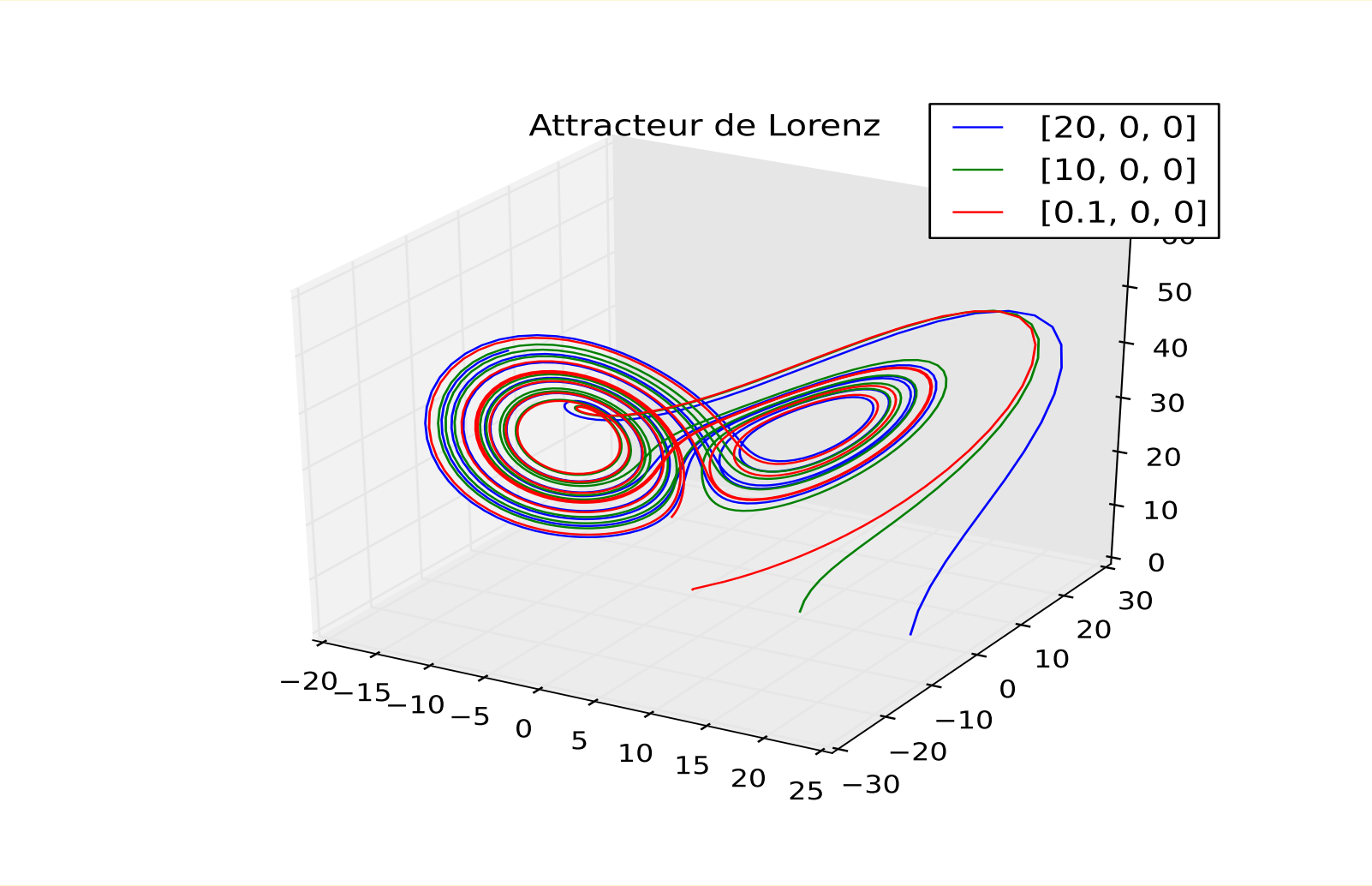
Les équations de Lorentz, basées sur la mécanique des fluides (équations de Navier-Stokes), modélisent entre autre, l'évolution atmosphérique et expliquent (en partie= les difficultés à rencontrées lors des prévisions météorologiques :

$$x'(t) = \sigma(y(t) - x(t))$$
$$y'(t) = \varrho x(t) - y(t) - x(t)z(t)$$
$$z'(t) = x(t)y(t) - \beta z(t)$$
$$\sigma \approx 10, \quad \beta \approx 2.66 \quad \varrho \approx 30$$

Où σ : (dépend de la viscosité et la conductivité thermique)
$x(t)$: est proportionnel au taux de convection.
$y(t)$, $z(t)$: des gradients de températures horizontal et vertical

Résolutions des équations de Lorenz avec Python

```
1  import matplotlib.pyplot as mp
2  import numpy as np
3  from scipy.integrate import odeint
4  from mpl_toolkits.mplot3d import Axes3D # Module pour graphique 3D
5  ## Attracteur de Lorentz
6  sigma , rho , b = 10 ,30 ,2 # valeurs des parametres
7  fig =mp.figure()
8  ax = fig.gca ( projection ='3d')
9  # v=[x,y,z] dans le modele de Lorenz
10 def fLorenz(v , t ):
11     return ([ sigma *(v[1]-v [0]),rho*v[0]-v [1]-v [0]*v [2] ,\
12              v [0]* v [1] - b *v [2]])
13 #
14 def OrbiteLorenz ( ci,n,T):
15     t = np.linspace (0 ,T , n +1)
16     for y0 in ci :
17         values = odeint (fLorenz , y0 , t )
18         ax.plot (values[:,0],values[:,1],values[:,2],label=str(y0))
19 OrbiteLorenz ([[20 ,0 ,0] ,[10 ,0 ,0] ,[0.1 ,0 ,0]] ,1000 ,10)
20 ax.legend ()
21 mp.savefig ("Lorenz.eps")
22 mp . show ()
```



Impossible de prévoir à long terme l'état du système.

Le module NUMPY : quelques commandes

Vecteurs

```
vl=array([a,b,..])          # vecteur ligne
vc=arry([[a],[b],..])       # vecteur colonne
vl[i]; vc[j]                # elemt : vl_i et vc_j
vl.size                     # longueur de vl
S=arange(d,f,i)             # S=[d,d+i,d+2i,...] (f ∉ S)
```

Matrices

```
mat=[[a1,a2,..],[b1,b2,..]] # remplissage par lignes
mat[i,j]                    # m_ij
mat.shape                   # renvoie le format de mat
mat[i,:]                    # extrait la ligne i de mat
mat[:,j]                    # extrait la colonne j de mat
zeros([n,p]); zeros(n,p)    # 0_{n×p}
m=eye(n,p)                  # m_ij = 1 si i = j, 0 sinon
ones([n,p])                 # matrice des uns (matrice d'Attila)
transpose(mat)              # no comment
reshape(mat,(m,q))          # re-dimensionner : mat ∈ M_{m×q}(K)
asarray(mat)                # transformer matrice en tableau
asmatrix(m)                 # transformer tableau -> matrice
det(mat)                    # det(mat) avec le module numpy.linalg
inv(mat)                    # mat^{-1} avec le module numpy.linalg
eigvals(mat)                # valeurs et vecteurs propres de mat avec le module nu
eig(mat)                    # vecteurs propres avec le module numpy.linalg
```

Polynômes

```
1  P=poly1d([a,b,..])          # P(x) = a x^n + b x^{n-1} + ...
2  Pf=poly1d([a,b,..],True)    # P_f(x) = (x-a)(x-b)...
3  P.order
4  P.roots; roots(P)           # renvoie les racines complexes (approchees)
5  P.coeffs                    # renvoie les coefficients de P
6  P(x); polyval(P,x)          # renvoie P(x) selon Horner
```

Opération sur les matrices (resp. vecteurs)

1. Combinaisons linéaires
2. Produit de matrices
3. inverse de matrices
4. Produit termes à termes
5. Produit tensoriel