Stage LIESSE Python

Calcul Numérique

Matthieu Moy

Ensimag

octobre 2015

Sommaire

1. Résolution de $AX = Y$ par l'algorithme du pivot de Gauss
2. Formalisation
3. Mise en œuvre
4. Le pivot de Gauss en NumPy
5. Analyse, comparaison avec NumPy
6. Tracer de courbes avec matplotlib

Pivot de Gauss

- Méthode pour résoundre un système linéaire.
- Exemple : résolution dans $\mathbb{R}^4$ du système

$$(S_0)\begin{cases} x - y + 2z + t = 1 \\ 2x - 4y + z + t = 6 \\ x - y + z - t = 2 \\ 3x + y + z + 2t = 1 \end{cases}$$

1. Effectuer les opérations :
   - $L_2 \leftarrow L_2 - 2L_1$
   - $L_3 \leftarrow L_3 - L_1$
   - $L_4 \leftarrow L_4 - 3L_1$
   - $\leadsto$ Résultat = (S_1).
2. Effectuer la transformation $L_4 \leftarrow L_4 + 2L_2$ pour obtenir le système (S_2).
3. Effectuer la transformation $L_4 \leftarrow L_4 - 11L_3$ pour obtenir le système triangulaire (S_3).
4. Résoudre dans $\mathbb{R}^4$ le système (S_3).

Exemple : solution

- Système final (S_3) : $\left(\begin{array}{cccc|c} 1 & -1 & 2 & 1 & 1 \\ 0 & -2 & -3 & -1 & 4 \\ 0 & 0 & -1 & -2 & 1 \\ 0 & 0 & 0 & 19 & -5 \end{array}\right)$
- solution : $X = \frac{1}{19}\begin{pmatrix} 20 \\ -22 \\ -9 \\ -5 \end{pmatrix}$
- Remarques :
  - Dans la ligne L_1 de (S_0), le coefficient de x est le *premier pivot*.
  - Dans la ligne L_2 de (S_1), le coefficient de y est le *deuxième pivot*.
  - Dans la ligne L_3 de (S_2), le coefficient de z est le *troisième pivot*.
  - Existence et unicité de la solution si tous les pivots sont non nuls (système de Cramer).

En résumé

Deux phases :

- mise sous forme triangulaire
  - $L_j \leftarrow L_j - \lambda L_i$ = « *transvection* ».
  - En cas de pivot nul, échange avec une autre ligne (il y en a toujours au moins une dans un système de Cramer).
- résolution du système triangulaire (phase de remontée)
  - Si on recontre l'équation $0 = \beta$ lors de la phase de remontée, le système n'a aucune solution.
  - Si on rencontre l'équation $0 = 0$, le système a une infinité de solutions.

Stabilité numérique

- Attention aux arrondis !

```
>>> 12*(1/3-1/4)-1
-2.220446049250313e-16
```

- Notion de "pivot non nul" assez fragile !
- Si le pivot est très petit (en valeur absolue), on va obtenir des nombres très grands lors des transvections.
- Solution : « pivot partiel ».
  - On sélectionne la ligne contenant le pivot le plus grand en valeur absolue.
  - Heuristique : marche bien en pratique, mais pas toujours !

Notations

- *Entrée :* $A = (a_{ij})$ est une matrice $n \times n$ composée de lignes L_i
- pour i de 0 à $n - 2$:
  - trouver j, $i \leq j \leq n - 1$, tel que $|a_{ji}|$ soit maximal
  - échanger L_i et L_j
  - pour k de $i + 1$ à $n - 1$:
    - $L_k \leftarrow L_k - \frac{a_{ik}}{a_{ii}} L_i$
- *Sortie :* le résultat est dans A

Structure de données « matrice » : tableaux NumPy

- Bibliothèque très performante (code C optimisé appelable depuis Python)
- Structure de base = tableau (homogène) multidimensionnel : `numpy.array`
- `http://www.numpy.org/`

Construire un tableau NumPy

On peut créer un tableau (unidimensionnel : un vecteur) NumPy à partir d'une liste Python ou même d'une liste de listes (il sera alors bi-dimensionel : une matrice) :

```
>>> import numpy
>>> numpy.array([1, 2, 3])
array([1, 2, 3])
>>> numpy.array([[1, 2], [3, 4], [5,6]])
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> a.shape[0]
3
>>> a.shape[1]
2
```

Opérations sur tableaux numpy

NumPy permet de faire des opérations directement sur ces tableaux, sans avoir à réécrire chaque fonction :

```
>>> 2 * numpy.array([1, 2, 3])
           + numpy.array([8, 10, 12])
array([10, 14, 18])
```

⇝ ce à quoi on s'attend en algèbre linéaire.

Accéder aux éléments d'un tableau NumPy

- Accès aux éléments comme pour une liste : `a[i]`
- Numérotation à partir de 0
- Matrices : `M[ligne][colonne]`

```
>>> b = numpy.array([1.414, 0.5, 3.14, 2.718])
>>> b[2]
3.1400000000000001
>>> a = numpy.array([[ 0.19,  0.14,  0.21],
...                  [ 0.79,  0.43,  0.74],
...                  [ 0.12,  0.40,  0.30]])
>>> a[2, 1]
0.40000000000000002
```

Accès aux lignes et aux colonnes

```
>>> a = numpy.array([[ 0.19,  0.14,  0.21],
...                  [ 0.79,  0.43,  0.74],
...                  [ 0.12,  0.40,  0.30]])

>>> a[:, 1]
array([ 0.14,  0.43,  0.4 ])
>>> a[0, :]
array([ 0.19,  0.14,  0.21])
```

Accès aux lignes et aux colonnes

⚠ NumPy ne retourne pas une copie des données dans ce cas mais seulement un *vue* sur celles-ci :

```
>>> a = numpy.array([[ 0.19,  0.14,  0.21],
...                  [ 0.79,  0.43,  0.74],
...                  [ 0.12,  0.40,  0.30]])

>>> v = a[:, 2]
>>> v
array([ 0.21,  0.74,  0.3 ])
>>> v[1] = 1.0
>>> v
array([ 0.21,  1.  ,  0.3 ])
>>> a
array([[ 0.19,  0.14,  0.21],
       [ 0.79,  0.43,  1.  ],
       [ 0.12,  0.4 ,  0.3 ]])
```

Tranches de tableaux

```
>>> v = numpy.array([-1, -2, -3, -4, -5])
# elements entre les indices 2 (inclus) et 4 (exclu)
>>> v[2:4]
array([-3, -4])
# elements entre les indices 0 (inclus) et 3 (exclu)
>>> v[:3]
array([-1, -2, -3])
# elements entre les indices 2 (inclus)
#                           et fin (inclus)
>>> v[2:]
array([-3, -4, -5])
```

⇝ Là aussi, on ne possède que des *vues* sur les données et pas des copies de celles-ci.

Exercice 1 : échange de lignes

Solution naive

Étant donnés une matrice NumPy `a` et deux entiers `i` et `j`, échanger les lignes `i` et `j` de `a`.

- 1ere tentative :

```
>>> i = 0
>>> j = 2
>>> tmp = a[i, :]
>>> a[i, :] = a[j, :]
>>> a[j, :] = tmp
>>> a
array([[ 0.12,  0.4 ,  0.3 ],
       [ 0.79,  0.43,  1.  ],
       [ 0.12,  0.4 ,  0.3 ]])
```

- Oups, ne fonctionne pas ! (rien changé)
  - `tmp = a[i, :]` ne fait pas de copie, juste une vue
  - `a[i, :] = a[j, :]` écrase la ligne
- ⇒ Il faut copier explicitement

Exercice 1 : échange de lignes

Solution avec recopie [1]

```
>>> a = numpy.array([[ 0.19,  0.14,  0.21],
...                  [ 0.79,  0.43,  1.  ],
...                  [ 0.12,  0.4 ,  0.3 ]])
>>> tmp = a[i, :].copy()  # <-- là
>>> a[i, :] = a[j, :]
>>> a[j, :] = tmp
>>> a
array([[ 0.12,  0.4 ,  0.3 ],
       [ 0.79,  0.43,  1.  ],
       [ 0.19,  0.14,  0.21]])
```

1. En fait, la "bonne" manière de faire en NumPy n'est pas de faire une copie mais plutôt d'utiliser l'indexation avancée : a[[i, j], :] = a[[j, i], :].

Exercice : transvection

Étant donnés une matrice NumPy `a`, deux entiers `i` et `j` et un flottant `x`, appliquer la transvection $L_j \leftarrow L_j + xL_i$ sur la matrice `a`.

```
>>> a
array([[ 0.12,  0.4 ,  0.3 ],
       [ 0.79,  0.43,  1.  ],
       [ 0.19,  0.14,  0.21]])
>>> i = 0
>>> j = 1
>>> x = -a[j, i] / a[i, i] # pivot : a[i, i]
>>> a[j, :] = a[j, :] + x * a[i, :]  # Lj <- Lj + xLi
>>> a
array([[ 0.12      ,  0.4        ,  0.3       ],
       [ 0.        , -2.20333333, -0.975     ],
       [ 0.19      ,  0.14       ,  0.21      ]])
```

Opérations sur les tableaux NumPy (1/2)

- Déjà vu : `M1 + M2`, `scalaire * M`
- Opérations classiques, composante par composante : `-`, `*`, `/`, `abs`

```
>>> u = numpy.array([1, 2, 3, 4])
>>> v = numpy.array([4, 3, 2, 1])
>>> u * v
array([4, 6, 6, 4])
>>> u / v
array([ 0.25,  0.66666667,  1.5,  4.])
>>> abs(u - v)
array([3, 1, 1, 3])
```

Opérations sur les tableaux NumPy (2/2)

- produits matriciels, produits scalaires : `numpy.dot`

```
>>> u = numpy.array([1, 2, 3, 4])
>>> v = numpy.array([4, 3, 2, 1])
>>> numpy.dot(u, v)   # sum_{i=0}^{n-1} u_i . v_i
20
>>> w = u * v
>>> w.sum()
20
>>> a = numpy.array([[1, 2], [3, 4]])
>>> x = numpy.array([1, -1])
>>> b = numpy.dot(a, x)
>>> b
array([-1, -1])
```

Exercice : Valeur maximale dans une colonne

Étant donnée une matrice `A` et un indice i, écrire la fonction `pivot_index(A,i)` qui renvoie l'indice $j \geq i$ t.q. $|a_{ji}|$ est maximal.

```
def pivot_index(A, i):
    n = A.shape[0]   # nombre de lignes
    j = i
    for k in range(i + 1, n):
        if abs(A[k, i]) > abs(A[j, i]):
            j = k
    return j
```

Résolution de système triangulaire supérieur

Étant donnés une matrice `A` triangulaire supérieure, et un vecteur `b` de même taille, implémenter la résolution de $Ax = b$.

```
def solve_triangular(A, b):
    n = len(b)
    x = numpy.zeros([n])   # vecteur de n zeros
    for i in range(n - 1, -1, -1):
        # x_i = 1/a_ii (b_i - sum_{j=i+1}^{n-1} a_ij x_j)
        sum = numpy.dot(A[i, :], x)
        x[i] = (b[i] - sum) / A[i, i]
    return x
>>> M = numpy.array([[1, 2], [0, 4]])
>>> x = numpy.array([1, -1])
>>> v = numpy.dot(M, x)   # array([-1, -4])
>>> solve_triangular(M, v)
array([ 1., -1.])   # on retrouve x ...
```

Algorithme du pivot de Gauss (1/3)

Presque tout est fait !

```
import numpy

def pivot_index(A, i):
    n = A.shape[0] # nombre de lignes
    j = i
    for k in range(i + 1, n):
        if abs(A[k, i]) > abs(A[j, i]):
            j = k
    return j

def swap_lines(A, i, j):
    tmp = A[i, :].copy()
    A[i, :] = A[j, :]
    A[j, :] = tmp

def transvection_lines(A, i, k, factor):
    A[k, :] = A[k, :] + A[i, :] * factor
```

Algorithme du pivot de Gauss (2/3)

```
def gauss(A0, b0):
    A = A0.copy() # pour ne pas détruire A
    b = b0.copy()
    n = len(b)   # on pourrait aussi la taille de a
    for i in range(n - 1):
        ipiv = pivot_index(A, i)
        if ipiv != i:   # echanges
            swap_lines(A, i, ipiv)
            tmp = b[ipiv]
            b[ipiv] = b[i]
            b[i] = tmp
        for k in range(i + 1, n):   # pivotage
            factor = -A[k, i] / A[i, i]
            transvection_lines(A, i, k, factor)
            b[k] = b[k] + b[i] * factor
    return [A, b]
```

Algorithme du pivot de Gauss (3/3 : programme principal)

```
def solve_triangular(A, b):
    n = len(b)
    x = numpy.zeros([n])
    for i in range(n - 1, -1, -1):
        sum = numpy.dot(A[i, :], x)
        x[i] = (b[i] - sum) / A[i, i]
    return x

# Fonction principale
def solve(A0, b0):
    Ab = gauss(A0, b0)
    return solve_triangular(Ab[0], Ab[1])
```

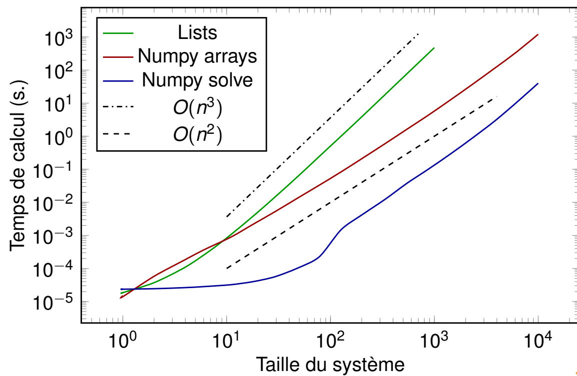
Algorithme du pivot de Gauss : test

```
>>> n = 5
# les matrices inversibles sont denses
>>> M = numpy.random.random([n, n])
>>> x = numpy.random.random([n])
>>> v = numpy.dot(M, x)
>>> x_solved = solve(M, v)
>>> abs(x - x_solved).max()
7.2164496600635175e-16
```

Performances

- Coût en $O(n^3)$ (démonstration dans Wack *et al.* pour les curieux).
- En général, il vaut mieux faire confiance aux algorithmes proposés par les bibliothèques de Python : plus rapides, plus robustes aux cas limites

Performances : comparaison avec numpy



Et la stabilité numérique ?

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 1 & 5/4 & 12 & 0 \\ 1 & 1/3 & 1 & 1 \\ 1 & 5/4 & 13 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 0 & 5/4-1/4 & 12 & 0 \\ 0 & 1/3-1/4 & 1 & 1 \\ 0 & 5/4-1/4 & 13 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

Et la stabilité numérique ?

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 1 & 5/4 & 12 & 0 \\ 1 & 1/3 & 1 & 1 \\ 1 & 5/4 & 13 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 0 & 1 & 12 & 0 \\ 0 & 1/3-1/4 & 1 & 1 \\ 0 & 1 & 13 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

Et la stabilité numérique ?

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 1 & 5/4 & 12 & 0 \\ 1 & 1/3 & 1 & 1 \\ 1 & 5/4 & 13 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 0 & 1 & 12 & 0 \\ 0 & 0 & 1-12*(1/3-1/4) = \text{2.2e-16} & 1 \\ 0 & 1 & 13 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

Et la stabilité numérique ?

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 1 & 5/4 & 12 & 0 \\ 1 & 1/3 & 1 & 1 \\ 1 & 5/4 & 13 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 0 & 1 & 12 & 0 \\ 0 & 0 & \text{2.2e-16} & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

Et la stabilité numérique ?

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 1 & 5/4 & 12 & 0 \\ 1 & 1/3 & 1 & 1 \\ 1 & 5/4 & 13 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1/4 & 0 & 0 \\ 0 & 1 & 12 & 0 \\ 0 & 0 & \text{2.2e-16} & 1 \\ 0 & 0 & 0 & 1-1/\text{2.2e-16} = \text{-4.5e15} \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ 0 \\ 1e16 \\ 4.7e31 \end{pmatrix}$$

Erreur = 10^{16} (33 seulement avec un pivot partiel)

Vous voulez des détails ?

- TP6 du cours d'informatique CPP Semestre 3.

Tracer une courbe avec un ensemble de points

matplotlib/courbe_simple.py

```
# plt devient l'abrege de matplotlib.pyplot:
import matplotlib.pyplot as plt

x = [0, 1, 3]   # liste des abscisses
y = [1, -1, 0]   # liste des ordonnees

plt.plot(x, y)   #  trace y en fonction de x
plt.show()   # affiche la fenetre du trace
```

Tracer de Fonction

matplotlib/oscilateur.py

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

# Valeurs sur lesquelles on va calculer
x = np.linspace(0, 10 * np.pi, 1000)
# application d'une fonction à chaque point
y1 = np.cos(x) # np.cos s'applique sur des tableaux
y2 = np.exp(-x/10) * y1

# y1 en fonction de x, y2 en fonction de y
plt.plot(x, y1)
plt.plot(x, y2)
```

Tracer de Fonction (suite)

matplotlib/oscilateur.py

```
# Titres sur les axes
plt.title('Oscillateurs harmoniques libre et amorti')
plt.xlabel('x')
plt.ylabel('y = f(x)')

# Tracer de l'axe y=0 et grille
plt.axhline(y=0, color='k')
plt.grid()

plt.show()
```

Tracer de courbes paramétrées

- Jusqu'ici :

```
x = np.linspace(...)
y1 = f1(x)
y2 = f2(x)
plt.plot(x, y1)
plt.plot(x, y2)
```

- Courbes paramétrées :

```
x = np.linspace(...) # ou t = ...
y1 = f1(x)  # ou x = ...
y2 = f2(x)  # ou y = ...
plt.plot(y1, y2)
```

Tracer de courbes paramétrées

matplotlib/spirale.py

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

t = np.linspace(0, 10 * np.pi, 1000)
x = np.cos(t) * t
y = np.sin(t) * t

plt.plot(x, y)
```

Joli, mais on peut faire bouger tout ça ?

$\rightsquigarrow$ Oui ! ! !

Animations

matplotlib/oscilateur_anim.py

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

# figure = tout le contenu de la fenêtre
fig = plt.figure() # A appeler avant de tracer autre chose

# Une courbe fixe
x = np.linspace(0, 10 * np.pi, 500)
y1 = np.cos(x)

parfait = plt.plot(x, y1)
y2 = y1 # Pas d'amortissement pour l'instant
amorti = plt.plot(x, y2)

plt.title('Oscillateurs libre et amorti')
```

Animations

matplotlib/oscilateur_anim.py

```
plt.xlabel('x')
plt.ylabel('y = f(x)')

# Fonction appelée à chaque pas de l'animation
def animate(i):
    C = i + 1 # i vaut successivement 0, 1, 2, ...
    # On peut changer le titre
    plt.title('Oscillateurs libre et amorti (C = '
              + str(C) + ')')
    # Et changer les données
    y2 = np.exp(-x/C) * y1
    # Attention, amorti est une liste a 1 élément.
    amorti[0].set_data(x, y2)

ani = animation.FuncAnimation(fig, animate, interval=100)

plt.show()
```

Animations : à essayer

matplotlib/oscilateur_anim.py

- Ajouter une instruction `print(i)` dans la fonction `animate`
- Jouer avec les paramètres de `FuncAnimation` :
  - `interval` (en miliseconde)
  - `frames` (arguments passés à `animate`, par exemple `np.linspace(...)`)
  - `repeat=False` (s'arrêter à la fin au lieu de répéter)

http://matplotlib.org/api/animation_api.html#matplotlib.animation.FuncAnimation

Animations : dessinons un peu

matplotlib/pendule_graphique.py

```
amplitude = 1
dt = 0.05

fig = plt.figure()
# 111 : découpage 1x1, sous-figure numéro 1
# (i.e. pas de découpage, mais on doit utiliser add_subplot
# pour utiliser xlim et ylim)
ax = fig.add_subplot(111, autoscale_on=False,
                     xlim=(-1.5, 1.5), ylim=(-1.5, .5))
ax.grid()

x = [0, 0]
y = [0, -1]
# y = f(x). 'o-' pour afficher des ronds sur les points.
parfait = plt.plot(x, y, 'o-')

plt.title('Pendule')
plt.xlabel('x')
plt.ylabel('y')
```

Animations : dessinons un peu

matplotlib/pendule_graphique.py

```
def animate(i):
    t = i * dt   # t = 0, 0.05, 0.10, ...
    # On s'autorise quelques libertés par rapport
    # à la physique ...
    angle = sin(t) * amplitude
    x = (0, sin(angle))
    y = (0, -cos(angle))
    plt.title('Pendule parfait (t = ' +
              str(round(t * 10) / 10) + ')')
    # Attention, parfait est une liste a 1 élément.
    parfait[0].set_data(x, y)

ani = animation.FuncAnimation(fig, animate,
                              interval=10)
plt.show()
```

Résolution d'équation différentielle

matplotlib/odeint.py

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

# y' = f(y, x)
def f(y, x):
    return np.cos(10 * y) * np.cos(x)

x = np.linspace(0, 10, 100)
# Résolution numérique de l'équation
y = scipy.integrate.odeint(f, 0, x)

# Tracer du résultat
plt.plot(x, y)
plt.show()
```

Animations : dessinons un peu

matplotlib/pendule_anim_simple.py

```
w02 = 10
y0 = np.array([np.pi / 2, 0])

def f(y, t):
    # y[0] = angle, y[1] = vitesse angulaire
    # On renvoie : (vitesse, accélération)
    return np.array([y[1], -w02 * np.sin(y[0])])

t = np.linspace(0, 100, 10000)
y = scipy.integrate.odeint(f, y0, t)
theta, thetadot = y[:, 0], y[:, 1]

fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False,
            xlim=(-1.5, 1.5), ylim=(-1.5, .5))
ax.grid()

plt.title('Pendule')
plt.xlabel('x'); plt.ylabel('y')
pendules = plt.plot((0, y0[0]), (0, y0[1]), 'o-')

def animate(i):
    angle = theta[i]
    x = (0, sin(angle))
    y = (0, -cos(angle))
    pendules[0].set_data(x, y)

anim = animation.FuncAnimation(fig, animate, interval=10)
anim.save('pendule_anim_simple.mp4', writer='avconv', fps=30)
# En cas de problème avec writer, essayer writer='mencoder'

plt.show()
```

- $\theta'' + \omega_0^2 \sin\theta = 0$
- On pose $y = (y_1, y_2) = (\theta, \theta')$
- On remarque $y' = (y_2, -\omega_0^2 \sin y_1)$
- On résout cette équation (θ en fonction de t) : `scipy.integrate.odeint`
- On trace le segment $(0,0) \to (\sin(\theta), -\cos(\theta))$ à chaque instant : `animation.FuncAnimation`

Animations : amusons-nous un peu

matplotlib/pendule_anim.py

- 2 pendules
- Lois de la physique différentes pour les deux (frottement)
- Pour aller plus loin : `matplotlib/double_pendulum_animated.py`, `http://matplotlib.org/examples/`