Université Paul Sabatier Novembre 2015

F.S.I **TP N°3** CB

Le but de ce TP est d'apprendre a utiliser les librairies NumPy, SciPy et Matplotlib.

1 NumPy

On trouve dans le module **NumPy** les outils de manipulation des tableaux pour le calcul numérique

— Nombreuses fonctions de manipulation de tableaux

— Bibliothèque mathématique importante

Il s'agit d'un module stable, bien testé et relativement bien documenté : `http://docs.scipy.org/doc/`, `http://docs.scipy.org/doc/numpy/reference/`

Pour l'importer, on recommande d'utiliser

```
>>> import numpy as np
```

Toutes les fonctions NumPy seront alors préfixées par *np*.

1.1 Introduction rapide a NumPy

Le module NumPy permet la manipulation simple et efficace des tableaux

```
>>> x = np.arange(0,2.0,0.1)   # De 0 (inclus) a 2 (exclus) par pas de 0.1
>>> x
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])
>>> np.size(x) # Sa taille
20
>>> x[0] # Le premier element
0.0
>>> x[1] # Le deuxieme element
0.10000000000000001
>>> x[19] # Le dernier element
1.9000000000000001
>>> x[20] # Pas un element !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 20 is out of bounds for axis 0 with size 20
>>> a = np.array ([[1,2,3], [4,5,6], [7,8,9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b = 2 * a  # Multiplication de chaque terme
>>> c = a + b  # Sommation terme a terme
>>> np.dot(a, b) # Produit de matrices
array([[ 60,  72,  84],
       [132, 162, 192],
       [204, 252, 300]])
>>> a * b      # Produit terme a terme
array([[  2,   8,  18],
       [ 32,  50,  72],
       [ 98, 128, 162]])
>>> a=np.array([1,2])
>>> np.outer(a,a)    # Calcule le produit a*transpose(a), soit une matrice.
array([[1, 2],
       [2, 4]])
```

On peut facilement effectuer des coupes dans un tableau numpy. Cette fonctionnalité est particulièrement importante en calcul scientifique pour éviter l'utilisation de boucles.

```
>>> t = np.array([1,2,3,4,5,6])
>>> t[1:4]    # de l'indice 1 a l'indice 4 exclu !!!ATTENTION!!!
array([2, 3, 4])
>>> t[:4]     # du debut a l'indice 4 exclu
array([1, 2, 3, 4])
```

```
>>> t[4:]    # de l'indice 4 inclus a la fin
array([5, 6])
>>> t[:-1]   # excluant le dernier element
array([1, 2, 3, 4, 5])
>>> t[1:-1]  # excluant le premier et le dernier
array([2, 3, 4, 5])
```

Pour extraire des sous-parties d'un tableau numpy, on a vu qu'on peut faire de l'indexation simple `t[0]` et des coupes `t[1:3]`. Une autre possibilité très pratique est de sélectionner certaines valeurs d'un tableau grâce a un autre tableau de booléens (un "masque"), de taille compatible avec le tableau d'intérêt. Cette opération s'appelle de l'indexation par masque

```
>>> a = np.arange(6)**2
>>> a
array([ 0,  1,  4,  9, 16, 25])
>>> a > 10 # le masque, tableau de booleens
array([False, False, False, False,  True,  True], dtype=bool)
>>> a[a > 10] # une maniere compacte d'extraire les valeurs > 10
array([16, 25])
```

Attention a la copie de tableau !

Pour un scalaire on a le comportement "intuitif" :

```
>>> a = 1.0
>>> b = a
>>> b
1.0
>>> a = 0.0
>>> b
1.0
```

Pour un tableau NumPy, par defaut on ne copie que l'adresse du tableau (pointeur) pas son contenu (les deux noms correspondent alors aux mêmes adresses en mémoire).

```
>>> a = zeros((2, 2))
>>> b = a
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> a[1, 1] = 10
>>> b
array([[  0.,   0.],
       [  0.,  10.]])
```

Pour effectuer une copie des valeurs, il faut utiliser .copy()

```
>>> c = b.copy()
>>> c
array([[  0.,   0.],
       [  0.,  10.]])
>>> b[1, 1] = 0
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> c
array([[  0.,   0.],
       [  0.,  10.]])
```

Remarque : la même chose s'applique aux coupes :

```
>>> a = np.arange(10)
>>> b = a[:5]
>>> a[0] = 10
>>> b
array([10,  1,  2,  3,  4])
```

Le module NumPy comporte beaucoup de fonctions qui permettent de créer des tableaux spéciaux, manipuler des tableaux, de faire des opérations sur ces tableaux, etc.

```
>>> a = np.arange(10)
>>> np.sum(a)
45
>>> np.mean(a)
4.5
>>> M=np.array([[0,1],[2,3]])
>>> M
array([[0, 1],
       [2, 3]])
>>> M.transpose()  # Calcule la transposée de la matrice M.
array([[0, 2],
       [1, 3]])
```

Il est également possible de résoudre des systèmes linéaires. Par exemple, pour résoudre le système d'équations $3x_0 + x_1 = 9$ et $x_0 + 2x_1 = 8$, on peut executer

```
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> x
array([ 2.,  3.])
```

On peut également calculer des coordonnées et évaluer les fonctions facilement sur ces coordonnées.

```
>>> def f(x):
...     return x**3
>>> n = 4
>>> x = np.linspace(0, 1, n)  # Calcule les coordonnées de n points répartis de manière
                              # équidistante entre 0 et 1
>>> y = np.zeros(n)           # Génère un vecteur remplis de 0
>>> for i in xrange(n):       # xrange est plus efficace que range pour de grandes valeurs
... y2[i] = f(x2[i])          # de n
...
>>> y2
array([ 0. , 0.015625, 0.125 , 0.421875, 1. ])
```

Les boucles sur de grands tableaux peuvent être très couteuse et engendrer de long temps de calcul. Il est possible d'être beaucoup plus efficace en utilisant la vectorisation.

```
>>> y2 = f(x2)
>>> y2
array([ 0. , 0.015625, 0.125 , 0.421875, 1. ])
```

NumPy met en oeuvre les fonctions standard de manière vectorielle. Ainsi, au lieu de taper

```
from math import sin, cos, exp
import numpy as np
r = np.zeros(len(x))
for i in xrange(len(x)):
r[i] = sin(x[i])*cos(x[i])*exp(-x[i]**2) + 2 + x[i]**2
```

on préférera

```
r = np.sin(x)*np.cos(x)*np.exp(-x**2) + 2 + x**2
```

ou encore

```
from numpy import sin, cos, exp
r = sin(x)*cos(x)*exp(-x**2) + 2 + x**2
```

Certaine fonction ne se vectorise par toujours facilement. Par exemple, on peut remarquer qu'on a des problèmes pour la fonction de Heaviside

```
>>> def H(x):
...     return (0 if x < 0 else 1)
>>> import numpy as np
>>> x = np.linspace(-10, 10, 5)
>>> x
array([-10., -5., 0., 5., 10.])
>>> H(x)
...
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

Le problème est relié au test `x < 0`, qui résulte en un tableau de valeurs booléenes, tandis que le test `if` a besoin

d'une simple valeur booléene. On peut alors réutiliser la version “scalaire” de la fonction `H`

```
>>> def H_loop(x):
>>>     r = np.zeros(len(x))
>>>     for i in xrange(len(x)):
>>>         r[i] = H(x[i])
>>>     return r
>>> # Exemple:
>>> x = np.linspace(-5, 5, 6)
>>> y = H_loop(x)
```

1.2 Input/output : comment sauver des tableaux, et charger des fichiers

On peut facilement sauver un tableau NumPy sous deux formats :

— en texte ascii avec `np.savetxt` (si on a besoin de voir le tableau dans un éditeur de texte) pour les tableaux de dimension <= 2.

— en format binaire de NumPy avec `np.save` (pour des fichiers moins gros en mémoire)

```
>>> a = np.arange(25).reshape((5, 5))
>>> a
array([[ 0,  1,  2,  3,  4],
    [ 5,  6,  7,  8,  9],
    [10, 11, 12, 13, 14],
    [15, 16, 17, 18, 19],
    [20, 21, 22, 23, 24]])
>>> np.savetxt('tableau.txt', a)
>>> np.save('tableau.npy', a)
>>> import os
>>> os.path.getsize('tableau.txt')
625
>>> os.path.getsize('tableau.npy')
280
```

Pour des tableaux de dimension trois ou plus, il faut utiliser `np.save`

```
>>> b = np.ones((2, 1, 3, 4))
>>> b.shape
(2, 1, 3, 4)
>>> np.savetxt('b.txt', b)
Traceback (most recent call last):
File "<ipython-input-13-1d7a297e3d85>", line 1, in <module>
    np.savetxt('b.txt', b)
File "/usr/lib/python2.7/dist-packages/numpy/lib/npyio.py", line 979,
in savetxt
    fh.write(asbytes(format % tuple(row) + newline))
TypeError: float argument required, not numpy.ndarray

>>> np.save('b.npy', b)
```

De la même manière, il existe deux fonctions pour charger un tableau numpy a partir d'un fichier

```
>>> c = np.loadtxt('tableau.txt')
>>> d = np.load('tableau.npy')
```

Remarque La librairie standard de Python permet de sauver n'importe quelle chaîne de caractères dans un fichier. Voir `http://scipy-lectures.github.io/intro/language/io.html`.

1.3 Exercices

Exercice 1.

Soit la fonction $h(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}x^2}$. Créer des tableaux xt et ht qui contiennent respectivement 41 valeurs reparties uniformement pour $x \in [-4, 4]$ et l'évaluation de la fonction h sur ces valeurs.

Exercice 2. Vectoriser le code suivant

```
import numpy as np
import math
```

```python
x = np.zeros(N);  y = np.zeros(N)
dx = 2.0/(N-1) # spacing of x coordinates
for i in range(N):
    x[i] = -1 + dx*i
    y[i] = math.exp(-x[i])*x[i]
```

Exercice 3. *Factorisation d'une matrice sous la forme QR par la méthode de Householder*

On souhaite pouvoir factoriser une matrice A, ayant m lignes et n colonnes, sous la forme du produit d'une matrice unitaire Q de taille $m \times m$ ($QQ* = I$, où $Q*$ désigne l'adjointe de Q, ou encore la conjuguée de la matrice transposé et I la matrice identité de taille $m \times m$), et d'une matrice R possédant m lignes et n (voir Fig. 1). Pour fabriquer cette factorisation, on peut suivre l'algorithme suivant pour

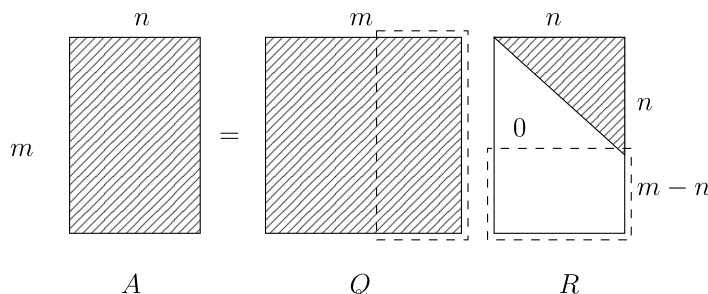

FIGURE 1 – Factorisation QR

```
1      Q=eye(m)          # eye est une fonction NumPy qui fabrique des matrices
2                        # identite de taille m x m
3      R=A
4
5      Pour k=0 a n,
6        Q_k=eye(m)
7        x=R[k:m,k]
8        v_k=signe(x[0])*norme(x)*e1+x  # e1, vecteur de même taille que x,
9                                       # tel que e1[0]=1 et 0 pour toutes les autres
10                                      # composantes
11       v_k=v_k/norme(v_k)
12       H=eye(m-k)-2*(v_k*transpose(v_k))  # Attention, v_k*transpose(v_k) est une matrice
13       Qk[k:m,k:m]=H
14       R=Qk*R
15       Q=Qk*Q
16     Fin pour
17     Q=transpose(Q)
```

1. Fabriquer une fonction Python qui prend une matrice A en argument et renvoie les deux matrices Q et R.
2. Tester votre méthode en vérifiant que le produit de Q par R redonne bien A.
3. Utiliser la méthode QR pour résoudre un système linéaire $Ax = b$, où A est une matrice possédant n lignes et n colonnes, et x et b sont des vecteurs a n composantes.

Exercice 4. *Approximation par la méthode des moindres carrés*

Supposons que l'on dispose de m couples de valeurs ($x_i \neq x_j \ \forall(i,j)$) $(x_1, y_1), \cdots, (x_m, y_m)$ résultats d'une expérience physique ou chimique. On cherche a construire une fonction (ici un polynôme) qui approche de la meilleure façon possible (ici au sens des moindres carrés) l'ensemble des points (x_i, y_i). Pour cela, on considère le polynôme $P(X) = c_0 + c_1 X + \cdots + c_{m-1} X^{n-1}$, où $n \leq m$. Si $n = m$, on parle d'interpolation polynomiale. On considère ici que $n < m$. Les coefficients de P vérifient alors le système linéaire surdeterminé de m équations a

n inconnues

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & & & & \vdots \\ \vdots & & & & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^{n-1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_m \end{pmatrix}$$

$$A \qquad x \quad = \quad b$$

Résoudre ce système surdeterminé revient a rendre le plus petit possible le résidu $Ax - b \in \mathbb{C}^m$ au sens de la norme euclidienne $\| \cdot \|_2$. On peut montrer que le problème "Trouver x qui minimise $\|Ax - b\|_2^2$" (c'est à dire $(Ax - b)^*(Ax - b)$) possède au moins une solution. L'ensemble des solutions coïncide avec celui du système

$$A^*Ax = A^*b.$$

De plus, si x_1 et x_2 sont solutions, alors $Ax_1 = Ax_2$. Enfin, si la matrice A^*A est régulière, c'est à dire si $rg(A) = n$, alors, la solution est unique.

Utiliser la fonction qui calcule la factorisation QR de l'exercice précédent pour résoudre un problème de moindre carré.

Exercice 5. *Calcul des valeurs propres d'une matrice diagonalisable*

On peut utiliser la méthode QR pour calculer les valeurs propres d'une matrice carrée diagonalisable.

L'algorithme est le suivant

```
1    T_k=A;
2    repeter
3      T_k=Q_k*R_k       # Calcul la factorisation QR de T_k
4      T_k=R_k*Q_k       # Multiplication
5    jusqu'a convergence.
```

On peut montrer que la méthode QR converge et $T_k = Q_k^* A Q_k$ converge vers la forme de Schur de A, c'est à dire une matrice triangulaire supérieure avec les valeurs propres sur la diagonale. Les éléments triangulaires inférieurs de T_k tendent progressivement vers 0. Si A est une matrice réelle mais à valeurs propres complexes, alors QR ne converge pas (car la décomposation QR ne fait intervenir que des opérations réelles). On doit donc appliquer QR à $A + iI$ et enlever i aux valeurs propres trouvées.

Programmer la méthode et la tester.

2 Matplotlib

Le module Matplotlib est chargé de tracer les courbes :

```
>>> import matplotlib.pyplot as plt
```

L'objet n'est pas ici de donner une vue complète de toutes les possibilités de Matplotlib. Pour plus de détails, on se réfèrera a la documentation en ligne. On pourra également consulter le site `http://scipy-lectures.github.io/intro/matplotlib/matplotlib.html` qui donne de nombreux exemples.

D'une manière générale les fonctions `plt.plot` prennent en arguments des vecteur/matrice, bref des tableaux de points du plan. Selon les options, ces points du plan sont reliés entre eux de façon ordonnée par des segments : le résultat est une courbe.

Commençons par la fonction sinus.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
plt.plot(x,np.sin(x))  # on utilise la fonction sinus de Numpy
plt.ylabel('fonction sinus')
plt.xlabel("l'axe des abcisses")
plt.show()
```

Après la commande `plt.show`, une fenêtre s'ouvre comme la figure 2. Il est possible d'utiliser des menus au bas de cette fenêtre : zoomer, déplacer la figure, etc et surtout sauvegarder dans un format PNG, PDF, EPS, etc.

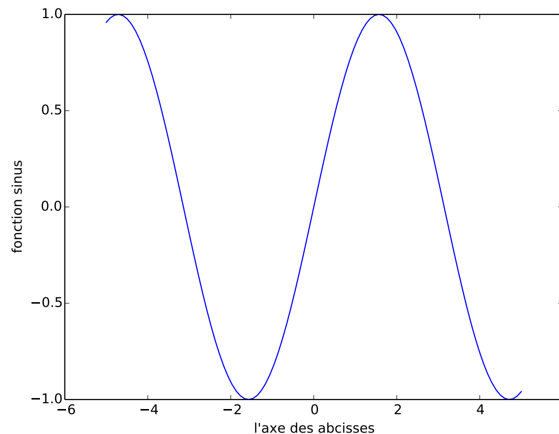

FIGURE 2 – Graphe de la fonction sinus

D'autres commandes sont utiles.

plt.clf() efface la fenêtre graphique

plt.savefig() sauvegarde le graphique. Par exemple `plt.savefig("mongraphe.png")` sauve sous le nom "mongraphe.png" le graphique. Par défaut le format est PNG. Il est possible d'augmenter la résolution, la couleur de fond, l'orientation, la taille (a0, a1, lettertype, etc) et aussi le format de l'image. Si aucun format n'est spécifié, le format est celui de l'extension dans "nomfigure.ext" (où "ext" est "eps", "png", "pdf", "ps" ou "svg"). Il est toujours conseillé de mettre une extension aux noms de fichier ; si vous y tenez `plt.savefig('toto',format='pdf')` sauvegardera l'image sous le nom "toto" (sans extension !) au format "pdf".

En mode interactif Python, une caractéristique est le mode interactif de cette fenêtre graphique. Si vous avez tapé l'exemple précédent, et si cette fenêtre n'a pas été fermée alors la commande `plt.xlabel("ce que vous voulez")` modifiera l'étiquette sous l'axe des abcisses. Si vous fermez la fenêtre alors la commande `plt.xlabel("ce que vous voulez")` se contentera de faire afficher une fenêtre graphique avec axe des abcisses, des ordonnées allant de 0 a 1 et une étiquette "ce que vous voulez" sous l'axe des abcisses. L'équivalent "non violent" de fermer la fenêtre est la commande `plt.close()`.

L'inconvénient, une fois un premier graphique fait, est le ré-affichage ou l'actualisation de cette fenêtre graphique au fur et a mesure des commandes graphiques : lenteur éventuelle si le graphique comporte beaucoup de données. Il est donc indispensable de pouvoir suspendre ce mode interactif. Heureusement tout est prévu !

plt.isinteractive() Retourne `True` ou `False` selon que la fenêtre graphique est interactive ou non.

plt.ioff() Coupe le mode interactif.

plt.ion() Passe en mode interactif.

plt.draw() Force l'affichage (le "retraçage") de la figure.

Ainsi une fois la première figure faite pour revenir a l'état initial, les deux commandes `plt.close()` et `plt.ioff()` suffisent.

Pour connaître toutes les options, le mieux est de se référer a la documentation de Matplotlib. Voyons ici quelques unes d'entre elles

— bornes : spécifier un rectangle de représentation, ce qui permet un zoom, d'éviter les grandes valeurs des fonctions par exemple, se fait via la commande `plt.axis([xmin,xmax,ymin,ymax])`

— couleur du trait : pour changer la couleur du tracé une lettre `g` vert (green), `r` rouge (red), `k` noir, `b` bleu, `c` cyan, `m` magenta, `y` jaune (yellow), `w` blanc (white). `plt.plot(np.sin(x),'r')` tracera notre courbe sinus en rouge. Les nuances de gris sont obtenues via `color='(un flottant entre 0 et 1)'`. Enfin pour avoir encore plus de couleurs, la séquence `color='#eeefff'` donnera la couleur attendu en hexadécimal. On peut également manipuler les couleurs RGB par `color=( R, G, B)` avec trois paramètres compris entre 0 et 1.

— symboles : mettre des symboles aux points tracés se fait via l'option `marker`. Les possibilités sont nombreuses parmi [`+` | `*` | `,` | `.` | `1` | `2` | `3` | `4` | `<` | `>` | `D` | `H` | `^` | `_` | `d` | `h` | `o` | `p` | `s` | `v` | `x` | `|` | TICKUP | TICKDOWN | TICKLEFT | TICKRIGHT | `None` |].

— style du trait : pointillés, absences de trait, etc se décident avec `linestyle`. Au choix `-` ligne continue, `--` tirets, `-.` points-tirets, `:` pointillés, sachant que 'None', '', ' ' donnent "rien-du-tout". Plutôt que `linestyle`, `ls` (plus court) fait le même travail.
— épaisseur du trait : `linewidth=flottant` (comme `linewidth=2`) donne un trait, pointillé (tout ce qui est défini par style du trait) d'épaiseur "flottant" en points. Il est possible d'utiliser `lw` en lieu et place de `linewidth`.
— taille des symboles (markers) : `markersize=flottant` comme pour l'épaisseur du trait. D'autres paramètres sont modifiables `markeredgecolor` la couleur du trait du pourtour du marker, `markerfacecolor` la couleur de l'intérieur (si le marker possède un intérieur comme 'o'), `markeredgsize=flottant` l'épaisseur du trait du pourtour du marker. Remarquez que si la couleur n'est pas spécifiée pour chaque nouvel appel la couleur des "markers" change de façon cyclique.
— étiquettes sur l'axe des abcisses/ordonnées : Matplotlib décide tout seul des graduations sur les axes. Tout ceci se modifie via `plt.xticks(tf)`, `plt.yticks(tl)` où `tf` et `tl` sont des vecteurs de flottants ordonnés de façon croissante.
— ajouter un titre : `plt.title("Mon titre")`
— légendes : il faut définir les labels des légendes au moment du tracé. Voir l'exemple ci-après.

```
>>> x=np.linspace(-np.pi,np.pi,100)
>>> plt.plot(x,np.sin(x),color="red",  linewidth=2.5, linestyle="-", label="sinus")
>>> plt.plot(x,np.cos(x),color="blue",  linewidth=2.5, linestyle="-", label="cosinus")
>>> plt.legend(loc='upper left')
>>> plt.axis([-np.pi,np.pi,-1,1])
>>> plt.yticks([-1, 0, +1],
  [r'$-1$', r'$0$', r'$+1$'])
>>> plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
  [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
```

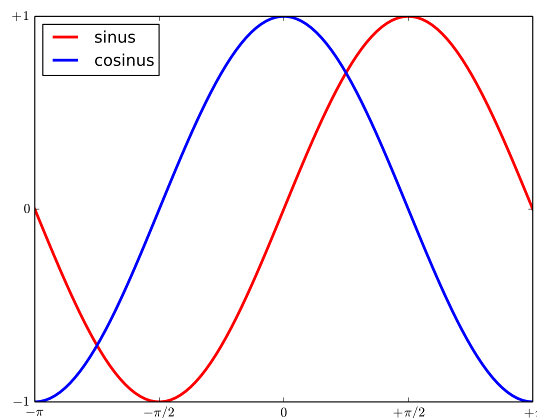

Figure 3 – Graphe des fonctions sinus et cosinus

Exercice 6.

Réaliser le graphe de la fonction $y(t) = v_0 t - \frac{1}{2} g t^2$ pour $v_0 = 10$, $g = 9.81$, et $t \in [0, 2v_0/g]$. Le label sur l'axe des x devra être "temps (s)" et le label sur l'axe des y "hauteur (m)".

Exercice 7.

Faire un programme qui lit un ensemble de valeurs v_0 en ligne de commandes et trace les courbes correspondantes $y(t) = v_0 t - \frac{1}{2} g t^2$ pour $g = 9.81$ dans la même fenêtre graphique. On prendra $t \in [0, 2v_0/g]$ pour chaque courbe, de telle sorte que on devra fabriquer un vecteur t différent pour chaque courbe.

Exercice 8.

La fonction

$$f(x,t) = e^{-(x-3t)^2} \sin\left(3\pi(x-t)\right)$$

décrit pour une valeur fixe de t une onde localisée en espace. Faire un programme qui visualise cette fonction comme une fonction de x dans l'intervalle $x \in [-4, 4]$ pour $t = 0$.

Exercice 9.

La fonction sinus peut être approchée par un polynôme grâce a la formule

$$\sin x \approx S(x;n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!}.$$

L'erreur dans l'approximation $S(x;n)$ décroit quand n augmente et a la limite on a $\lim_{n\to\infty} S(x;n) = \sin x$. Le but de cette exercice est de visualiser la qualité de diverses approximations $S(x;n)$ quand n croit.

La première partie de l'exercice est d'écrire une fonction Python `S(x,n)` qui calcule $S(x;n)$.

La partie suivante est de tracer simultanément la fonction $\sin x$ sur $[0, 4\pi]$ avec ses approximations $S(x;1)$, $S(x;2)$, $S(x;3)$, $S(x;6)$ et $S(x;12)$.

3 SciPy

Dans les modules scientifiques de Python, il existe une collection impressionnante de fonctions réalisant des opérations diverses sur les tableaux numériques de NumPy. Il faut surtout les utiliser, plutôt que de réinventer la roue en recodant (probablement moins bien) une fonctionnalité qui existe déja par ailleurs !

Dans NumPy, on a déja vu qu'il existe beaucoup d'opérations permettant par exemple de
- générer des tableaux particuliers : np.arange, np.ones, np.linspace, ...
- faire des opérations a partir des valeurs du tableau : np.sum, np.sin, np.histogram, etc.
- changer l'agencement des valeurs d'un tableau, sa forme, ou encore créer un nouveau tableau a partir de plusieurs autres : np.reshape, np.concatenate.
- etc. ...

Le module SciPy est la boîte a outils numérique pour les tableaux NumPy. On trouve dans SciPy les opérations de manipulation / traitement de données numériques classiques, mais spécifiques a un type d'application (algébre linéaire, statistiques, etc.). Ce sont donc des fonctions plus "haut niveau" que celles de NumPy.

SciPy est un module stable, bien testé et relativement bien documenté.

`http://docs.scipy.org/doc/`, `http://docs.scipy.org/doc/scipy/reference/`

```
>>> import scipy
```

Le module SciPy réalise les différentes opérations sur des tableaux numériques (`ndarray`) de NumPy. On peut donc directement utiliser ces tableaux comme arguments pour les différentes fonctions

```
>>> from scipy import linalg
>>> mat = np.array([[1, 2], [2, 4]])
>>> mat
array([[1, 2],
    [2, 4]])
>>> linalg.det(mat)
0.0
```

3.1 Pendule simple

Pour montrer l'utilisation de SciPy, nous allons nous intéresser a l'intégration d'équations différentielles, en considérant des systèmes dynamiques a base de pendules mécaniques.

En écrivant la relation fondamentale de la dynamique (conservation de la quantité de mouvement), on obtient l'équation du pendule simple donnée par

$$\ddot{\theta} + \omega^2 \sin\theta = 0$$

où θ est l'angle du pendule par rapport a la verticale, et on note avec un point la dérivée temporelle.

Pour les petites oscillations on peut faire l'approximation $\sin\theta \approx \theta$. Quand l'approximation n'est pas valide il faut intégrer numériquement cette équation différentielle pour obtenir l'évolution de la position et de la vitesse angulaire du pendule, au cours du temps.

Il nous faut donc disposer d'un intégrateur d'équations différentielles, que l'on peut s'attendre a trouver dans SciPy. Mais quelle est la fonction correspondante ? On peut consulter le sommaire de l'aide `http://docs.scipy.org/doc/scipy/reference/index.html`. Il existe un sous-module integrate, qui contient lui-même une fonction odeint

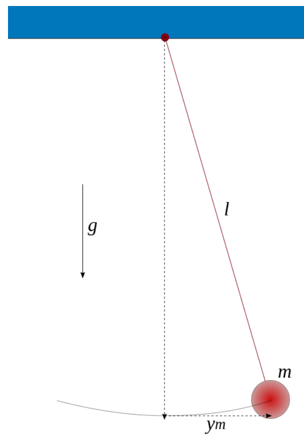

FIGURE 4 – Pendule simple

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

Regarder la doc de la fonction http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html#scipy.integrate.odeint et l'exemple http://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html#ordinary-differential-equations-odeint.

Pour commencer, il faut mettre l'équation différentielle du 2nd ordre sous la forme d'un système d'équations du premier ordre

```
def simple_pendulum(theta_thetadot, t):
    theta, theta_dot = theta_thetadot
    return [theta_dot, - np.sin(theta)]
```

correspondant a

$$\frac{d\theta}{dt} = \dot{\theta},$$
$$\frac{d\theta}{dt} = -\sin\theta.$$

Nous pouvons maintenant intégrer une trajectoire a partir d'une condition initiale

```
>>> t = np.linspace(0, 5 * np.pi, 1000)
>>> sol = odeint(simple_pendulum, (np.pi/3, 0), t)
```

Nous pouvons par exemple vérifier la conservation de l'énergie mécanique au cours du temps

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

def simple_pendulum(theta_thetadot, t):
    theta, theta_dot = theta_thetadot
    return [theta_dot, - np.sin(theta)]

theta_thetadot = (np.pi / 3, 0)
t = np.linspace(0, 5 * np.pi, 1000)
sol = odeint(simple_pendulum, (np.pi/3, 0), t)
theta, theta_dot = sol.T

E_kin = 1./ 2 * theta_dot ** 2
E_pot = 1 - np.cos(theta)
E_mech = E_kin + E_pot

plt.figure()
ax = plt.gca()
plt.plot(t, E_kin, 'o-', lw=2, label=u'$E_\mathrm{kin}$')
plt.plot(t, E_pot, 'o-', lw=2, label=u'$E_\mathrm{pot}$')
plt.plot(t, E_mech, lw=2, label=u'$E_\mathrm{mech}$')
plt.xticks(np.pi * np.arange(5), [0, u'$\pi$', u'$2\pi$', u'$3\pi$', u'$4\pi$'])
plt.xlabel(u'$t$', fontsize=26)
plt.title(u'Conservation de l\'energie')
plt.gca().tick_params(axis='both', which='major', labelsize=20)
plt.legend(loc='best')
```

```
plt.grid()
plt.show()
```

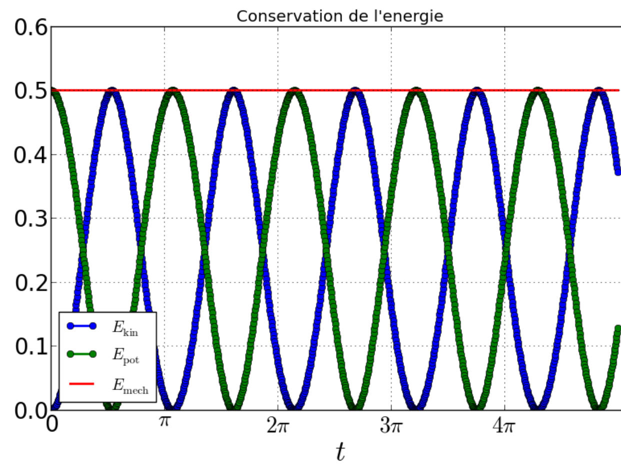

FIGURE 5 – Conservation de l'énergie

3.2 Exercice

Ecrire un script python pour construire le diagramme des phases du pendule simple, représenté Fig. 6 ci-dessous. Pour cela, il faut

— intégrer l'équation différentielle pour différentes conditions initiales entre 0 et π.

— représenter $\dot{\theta}$ en fonction de θ pour les différentes solutions.

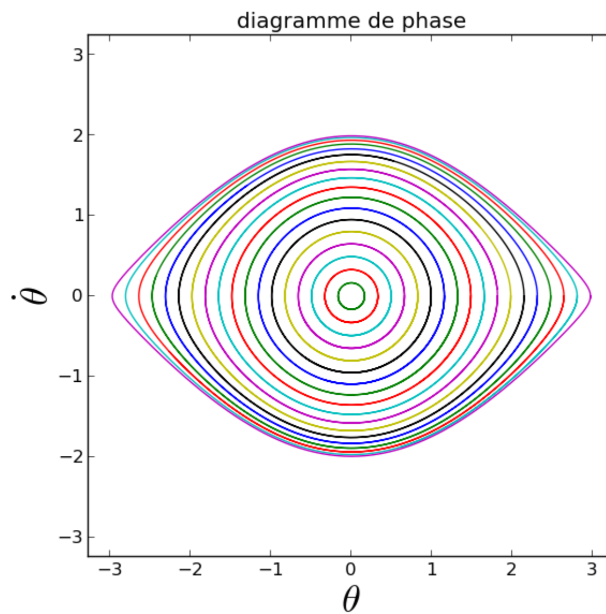

FIGURE 6 – Diagramme de phase