Université Paul Sabatier — Décembre 2014

F.S.I — **TP N°3** — CB

Le but de ce TP est d'apprendre à utiliser la librairie 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 Introduction rapide à NumPy

Le module NumPy permet la manipulation simple et efficace des tableaux

```
>>> x = np.arange(0,2.0,0.1)   # De 0 (inclus) à 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 élément
0.0
>>> x[1] # Le deuxième élément
0.10000000000000001
>>> x[19] # Le dernier élément
1.9000000000000001
>>> x[20] # Pas un élément !
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 à terme
>>> np.dot(a, b) # Produit de matrices
array([[ 60,  72,  84],
       [132, 162, 192],
       [204, 252, 300]])
>>> a * b      # Produit terme à 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 à l'indice 4 exclu!!!ATTENTION!!!
array([2, 3, 4])
>>> t[:4]    # du debut à l'indice 4 exclu
array([1, 2, 3, 4])
```

```
>>> t[4 :]     # de l'indice 4 inclus à 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 à 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 à 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, comme pour des listes, 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, de 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 couteuses 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 œuvre 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)
```

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)
```

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 à 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`.

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
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 possèdant m lignes et n (voir Fig. 1). Pour fabriquer cette factorisation, on peut suivre l'algorithme suivant

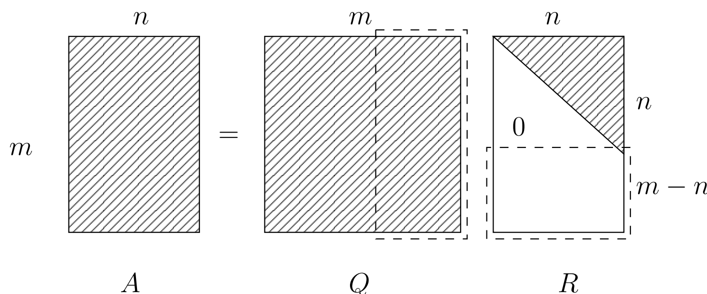

FIGURE 1 : Factorisation QR

```
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 à 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*transpose(v))  # Attention, v*transpose(v) 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 possèdant n lignes et n colonnes, et x et b sont des vecteurs à 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 à 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 à n inconnues

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

Résoudre ce système surdeterminé revient à 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.