Simulation Numérique 1 : Numpy et Matplotlib

Nous supposons que les modules suivants sont importés pour Python :

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

Ainsi, les commandes de numpy sont appelé avec

`np.nom_commande`

celles de scipy avec

`sp.nom_commande`

et celles de matplotlib avec

`plt.nom_commande`

A noter : une commande `np.who()` permet de visualiser les informations relatives à l'ensemble des variables utilisées.

1 Les tableaux

a array

Les tableaux avec Numpy sont de type `array`, correspondant au type habituel `list` mais dont les éléments sont tous de même type et dont la taille est fixé (un vrai tableau informatique, quoi !) [0]

On peut définir un tableau à partir d'une liste et préciser de manière optionnelle le type des éléments.

```
>>> t=np.array([1,2,3])
>>> t
array([1, 2, 3])

>>> t2=np.array([1,2,3.])
>>> t2
array([ 1.,  2.,  3.])

>>> t3=np.array([1,2,3],float)
>>> t3
array([ 1.,  2.,  3.])

>>> t4=np.array([1,2,3],complex)
>>> t4
array([ 1.+0.j,  2.+0.j,  3.+0.j])

>>> np.who()
Name    Shape    Bytes  Type
===================================

t3      3        24     float64
t2      3        24     float64
t       3        24     int64
t       3        48     complex128

Upper bound on total bytes  =  120
```

La taille d'un tableau s'obtient avec `len` comme pour les listes.

```
>>> len(t)
3
```

On accède aux éléments d'un tableau, et on les modifie comme pour les listes.

0. Sauf que l'on peut créer des tableau hétérogène avec comme type `object`... Python...

```
>>> t=np.array([1,2,3,4])

>>> t[1]
2

>>> t[1]=0

>>> t
array([1, 0, 3, 4])
```

On peut faire du slicing sur les tableaux comme pour les listes. On peut aussi le remplacer par une liste ou un tableau contenant les indices à extraire.

```
>>> t[1:3] # [début:fin]
array([0, 3])

>>> t[1:]
array([0, 3, 4])

>>> t[::2] # [début:fin:pas]
array([1, 3])

>>> t[1::2]
array([0, 4])

>>> t[[1,3]]
array([2, 4])
```

b arange

On peut créer des tableaux contenant des suites arithmétiques avec `np.arange` qui fonctionne comme `range`, mais accepte aussi des flottants et renvoie un `array`.

```
>>> np.arange(9)
array([0, 1, 2, 3, 4, 5, 6, 7, 8])

>>> np.arange(2,7)
array([2, 3, 4, 5, 6])

>>> np.arange(2,7,2)
array([2, 4, 6])

>>> np.arange(1.1, 2.1, .2)
array([ 1.1, 1.3, 1.5, 1.7, 1.9])
```

c linspace

La commande

`np.linspace(a,b,n)`

renvoie le tableau contenant la subdivision régulière de $[a,b]$ en n termes, c'est-à-dire n nombres régulièrement espacés dont le premier vaut a et le dernier b.

```
>>> np.linspace(1.5, 3.7, 5)
array([1.5, 2.05, 2.6, 3.15, 3.7])
```

d Tableaux particuliers

- `np.zeros(n)` tableau de zéros (taille n)
- `np.ones(n)` tableau (taille n) rempli de 1
- `np.random.rand(n)` tableau (taille n) à coefficients aléatoires uniformes sur $[0,1[$

2 Calculs mathématiques

a Constantes

Les constantes mathématiques usuelles sont accessibles avec NumPy : `pi` et `e`.

```
>>> np.e , np.pi
(2.7182818284590, 3.1415926535897)
```

D'autres constantes scientifiques sont disponibles dans SciPy (voir doc).

b Fonctions

Les principales fonctions mathématiques sont disponibles dans le module NumPy : `sqrt` ($\sqrt{\cdot}$), `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `exp`, `log`, `log10`, `floor` ($\lfloor\cdot\rfloor$), `ceil` ($\lceil\cdot\rceil$), etc.

Notons également qu'appliquée à un tableau, la fonction est appliquée à tous les cœfficients.

Les opérations `+, *, /, //, **, %` s'appliquent aussi directement aux/entre les cœfficients.

```
>>> t=np.array([1,2,3])

>>> t
array([1, 2, 3])

>>> t+t
array([2, 4, 6])

>>> t*t
array([1, 4, 9])

>>> t**3
array([ 1,  8, 27])

>>> 1/t
array([ 1. , 0.5 , 0.33333])

>>> np.sin(t)
array([ 0.8414, 0.9092, 0.1411])

>>> np.log(t)
array([ 0. , 0.69314,  1.09861])

>>> np.exp(t)
array([ 2.7182, 7.3890 , 20.0855])

>>> t % 2
array([1, 0, 1])
```

On peut vectorialiser une fonction pour qu'elle s'applique terme à terme avec `np.vectorize` :

```
>>> f=lambda n:1 if n==0 else n*f(n-1\
 )

>>> a=np.arange(6)

>>> f(a)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in
  <module>
    f(a)
  File "<pyshell#5>", line 1, in
  <lambda>
    f=lambda n:1 if n==0 else
    n*f(n-1)
ValueError: The truth value of an
array with more than one element
is ambiguous. Use a.any() or
a.all()

>>> np.vectorize(f)(a)
array([ 1, 1,  2,  6, 24, 120])
```

c Nombres complexes

On peut aussi manipuler des nombres complexes qui s'écrivent sous la forme suivante : `2.5 + 3.5j`, `1j` désignant le complexe i. Les opérations habituelles sur les nombres complexes (ou tableaux de nombres complexes.)

```
>>> z=2.5+3.5j
>>> z**2
(-6+17.5j)

>>> np.conj(z)
(2.5-3.5j)

>>> np.abs(z)
4.3011626335213133

>>> np.angle(z)
0.95054684081207519

>>> np.real(z)
array(2.5)

>>> np.imag(z)
array(3.5)
```

d Fonctions spécifiques aux tableaux

```
>>> t=np.linspace(2,1,5)
>>> t
array([2., 1.75, 1.5, 1.25, 1.])

>>> np.sum(t)
7.5
```

```
>>> np.product(t)
6.5625

>>> np.mean(t)
1.5
>>> np.min(t)
1.0

>>> np.max(t)
2.0

>>> np.sort(t)
array([1., 1.25, 1.5, 1.75, 2.])
```

3 Graphes

Le tracé de courbe se fait par le module MatPlotLib.PyPlot, donc bien le charger :

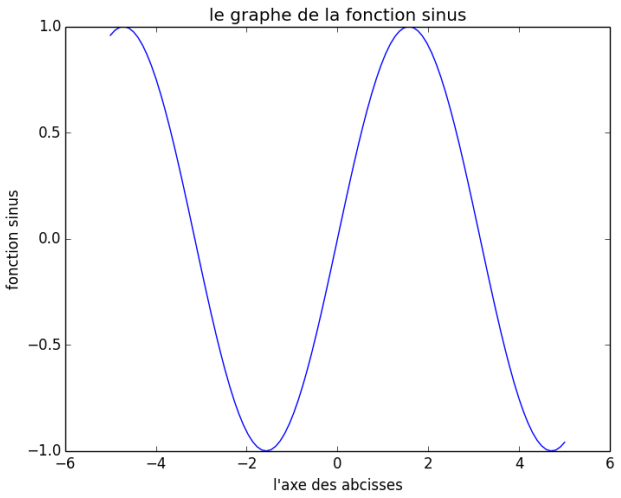
```
import matplotlib.pyplot as plt
```

On donne quelques possibilités, pour plus de précision, il faut consulter la doc de MatPlotLib.

On indique toutes les préférences du graphique avant de le tracer.

- `plt.plot(x,y)` où `x` et `y` sont des tableaux calcule la courbe correspondante.
- `plt.xlabel(chaine)` et `plt.ylabel(chaine)` permettent de préciser un titre pour chaque axe.
- `plt.title(chaine)` permet de préciser un titre.
- `plt.show()` permet d'afficher le graphe.
- `plt.savefig(nom_fichier)` permet d'enregistrer le graphe.

```
x=np.linspace(-5,5,100)
plt.plot(x,np.sin(x))
plt.ylabel('fonction sinus')
plt.xlabel("l'axe des abcisses")
plt.title('le graphe de la fonction \
 sinus')
plt.show()
```



- `plt.axis([xmin,xmax,ymin,ymax])` ou `plt.xlim(xmin,xmax))` ou `plt.ylim(ymin,ymax))` permet de préciser l'échelle des axes.
- `plt.grid()` permet de voir une grille.

  `plt.legend()` permet d'afficher la légende (précisée avec l'argument `label` de la fonction `plot`). L'argument `loc='best'` permet de laisser matplotlib placer la légende au meilleur endroit.

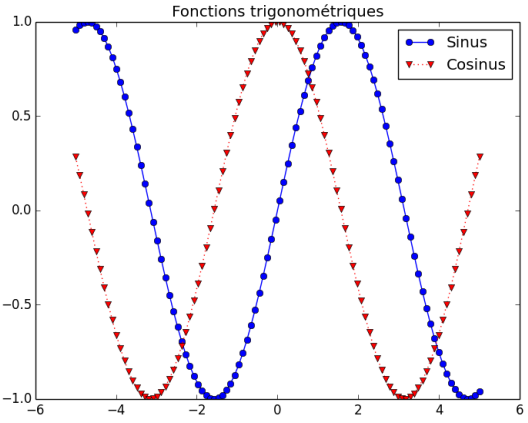
L'aide sur `plt.plot` donne les diverses options possibles pour le tracer. Par exemple, un troisième paramètre à `'r'` donnera une courbe rouge , `'b'` donnera une courbe bleue. Avec `color='#24e5cf'` on peut préciser une couleur en RGB (hexadécimal).

On peut marquer les points avec l'option `marker=` : `o` pour des ronds, `+` pour des plus, `*` pour des étoiles, etc.

Le style de ligne peut aussi être précisé avec `linestyle=` ou `ls=` : `'-'` pour continu, `'--'` pour tirets, `':'` pour pointillés, `'-.'`, etc.

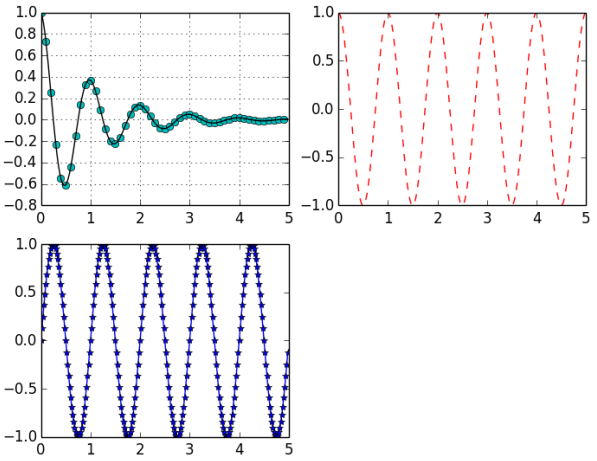
Enfin, on peut utiliser un argument condensé : par exemple `'bo'` ou `'g-v'` ou `'--'`, etc.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
plt.plot(x,np.sin(x),marker='o',label\
 ='Sinus')
plt.plot(x,np.cos(x),'rv:',label='\
 Cosinus')
plt.title("Fonctions trigonométriques\
 ")
plt.legend()
plt.show()
```



On peut aussi séparer le grapheur en plusieurs sous-figures avec `plt.subplot(xyz)` ou `plt.subplot(x,y,z)` où `x` est le nombre de lignes, `y` le nombre de colonnes, `z` le numéro de la figure.

```
f=lambda t : np.exp(-t)*np.cos(2*np.\
 pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.subplot(221)
plt.plot(t1,f(t1),'co',t2,f(t2),'k')
plt.grid(True)

plt.subplot(222)
plt.plot(t2,np.cos(2*np.pi*t2),'r--')

plt.subplot(223)
plt.plot(t2,np.sin(2*np.pi*t2),'b*-')

plt.show()
```



Notons qu'il existe un mode interactif : `plt.ion()` pour l'activer et `plt.ioff()` pour le désactiver qui permet de voir en temps réel les modifications sur le grapheur. `plt.clf()` efface la fenêtre graphique.

`plt.figure(n)` (`n` est optionnel) permet de créer plusieurs fenêtres graphiques.

4 Matrices

Les matrices sont en fait des tableaux à deux dimensions. Tout comme les tableaux à une dimension, cette structure est typé : tous les objets à l'intérieur ont un même type déterminé.

a Initialisation

On initialise un tel tableau avec :

- `np.empty((n,p))` ou `np.empty([n,p])` : tableau vide[0] de flottants à n lignes et p colonnes.
- `np.zeros((n,p))` ou `np.zeros([n,p])` : tableau de flottants à n lignes et p colonnes rempli de `0`.
- `np.ones((n,p))` ou `np.ones([n,p])` : tableau de flottants à n lignes et p colonnes rempli de `1`.

0. En fait, il n'est pas vraiment vide ! Essayer !

- `np.random.rand(n,p)` [0] : tableau de flottants à n lignes et p colonnes rempli de nombres aléatoires dans $[0,1[$.
- `np.diag(v)` : tableau carré dont la diagonale est remplie par les éléments de l'itérable `v`.
- `np.diag(v,i)` : tableau dont la i^e diagonale supérieure est remplie par les éléments de l'itérable `v`.
- `np.diag(v,-i)` : tableau dont la i^e diagonale inférieure est remplie par les éléments de l'itérable `v`.

```
>>> np.diag([1,2,3,4])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])

>>> np.diag(range(3),2)
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 2],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

```
>>> np.diag((10,20),-3)
array([[  0,   0, 0, 0, 0],
       [  0,   0, 0, 0, 0],
       [  0,   0, 0, 0, 0],
       [ 10,   0, 0, 0, 0],
       [  0,  20, 0, 0, 0]])
```

- `np.identity(n)` : c'est clair, non ?
- `np.array(L)` : où `L` est une liste de liste **aux dimensions cohérentes** la convertit en tableau à deux dimensions. On peut préciser le type.

```
>>> np.array([[1,2,6],[3,4,5]],'float\
 ')
array([[ 1.,  2.,  6.],
       [ 3.,  4.,  5.]])
```

- `np.reshape(T,(n,p))` ou `T.reshape(n,p)` transforme un tableau de np cœfficients en un tableau à n lignes et p colonnes.

0. Attention, il y a un piège !

```
>>> T=np.linspace(1,5,9)

>>> T
array([ 1. , 1.5, 2. ,  2.5, 3. ,
       3.5,  4. , 4.5,  5. ])

>>> T.reshape(3,3)
array([[ 1. ,  1.5,  2. ],
       [ 2.5,  3. ,  3.5],
       [ 4. ,  4.5,  5. ]])
```

b Mutabilité, slicing

Comme les tableaux à une dimension, les tableaux à deux dimensions sont mutables et supportent le slicing.

On accède à l'élément sur la $i+1$-eme ligne et $j+1$-eme colonne du tableau `T` avec `T[i,j]` (la numérotation commence à 0). Le slicing permet d'obtenir des matrices extraites.

On peut donc modifier des éléments voire des sous-matrices :

```
>>> A=np.linspace(0,121,12).reshape(3\
 ,4)

>>> A
array([[  0.,  11.,  22.,  33.],
       [ 44.,  55.,  66.,  77.],
       [ 88.,  99., 110., 121.]])

>>> A[0,2]=np.pi

>>> A
array([[  0., 11., 3.1416, 33. ],
       [ 44., 55., 66.   , 77. ],
       [ 88., 99., 110.  , 121.]])

>>> A[:,1]=np.linspace(1,6,3)
# remplacement d'une colonne

>>> A
array([[  0., 1. ,  3.1416, 33. ],
       [ 44., 3.5, 66.    , 77. ],
       [ 88., 6. ,110.    ,121.]])

>>> A[0,:]=[1,2,3,4]
# remplacement d'une ligne

>>> A
array([[  1., 2. ,   3., 4. ],
       [ 44., 3.5,  66., 77. ],
       [ 88., 6. , 110., 121.]])

# Attention comme pour tous les types
# mutables...
>>> C=A[:,1]
# référence vers la 2e colonne de A

>>> C
array([ 2. ,  3.5,  6. ])

>>> C[0]=25 # modification de C...

>>> C
array([ 25. ,   3.5,   6. ])

>>> A
# ... entraîne modification de A
array([[  1., 25. ,   3. ,  4. ],
       [ 44.,  3.5,  66. , 77. ],
       [ 88.,  6. , 110. ,121. ]])
```

Mais attention : le type étant fixé à l'avance, on peut avoir quelques surprises !

```
>>> A=np.arange(15).reshape(3,5)

>>> A
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

>>> B=A/2

>>> B
array([[ 0. , 0.5, 1. , 1.5, 2. ],
       [ 2.5, 3. , 3.5, 4. , 4.5],
       [ 5. , 5.5, 6. , 6.5, 7. ]])

>>> for i in range(3):
        for j in range(5):
            A[i,j]/=2

>>> A
array([[0, 0, 1, 1, 2],
       [2, 3, 3, 4, 4],
       [5, 5, 6, 6, 7]])
```

C Méthodes et fonctions utiles

Voici une liste non exhaustive de fonctions utiles (notons que certaines fonctions spécifiques au calcul matriciel se trouvent dans le sous-module `numpy.linalg`) :

- Dimensions d'un tableau : `T.shape` (noter qu'il n'y a pas de parenthèses).
- Toutes les fonctions mathématiques s'appliquent toujours terme à terme.

```
>>> A.shape
(4, 4)

>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

>>> A*2
array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14],
       [16, 18, 20, 22],
       [24, 26, 28, 30]])

>>> np.sin(A)
array(
[[ 0.    , 0.8414, 0.9092, 0.1411],
 [-0.7568,-0.9589,-0.2794, 0.6564],
 [ 0.9893, 0.4121,-0.5440,-0.9999],
 [-0.5365, 0.4201, 0.9906, 0.6502]])

>>> A*A
# ceci n'est pas un produit
# matriciel !
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121],
       [144, 169, 196, 225]])
```

- `A.max()` donne l'élément maximal, `A.max(axis=0)` et `A.max(axis=1)` donnent respectivement les maxima de chaque colonne et de chaque ligne. On a de la même manière `A.min(),A.sum(),A.prod()`.

```
>>> A=np.random.randint(10,size=(4,4)\
 )
>>> A
array([[2, 1, 3, 1],
       [9, 4, 4, 5],
       [1, 4, 2, 0],
       [4, 5, 8, 0]])

>>> A.max()
9

>>> A.max(axis=0)
array([9, 5, 8, 5])

>>> A.max(axis=1)
array([3, 9, 4, 8])
```

- Conversion de type : `A.astype(type)`.
- Copie : `A.copy()`.
- Produit matriciel : `np.dot(A,B)` ou `A.dot(B)`.
- Puissance d'une matrice : `np.linalg.matrix_power(A,n)`.
- Transposée : `A.T` ou `A.transpose()`.
- Trace d'une matrice : `np.trace(A)`.
- Rang d'une matrice : `np.linalg.matrix_rank(A)`
- Inverse d'une matrice : `np.linalg.inv(A)`.
- Déterminant de matrice : `np.linalg.det(A)`.
- Résolution de système linéaire $Ax = b$: `np.linalg.solve(A,b)`.
- Valeurs propres : `np.linalg.eigvals(A)`.
- Valeurs et vecteurs propres : `np.linalg.eig(A)`.
- Produit scalaire : `np.inner(x,y)`
- Concaténation de matrices (matrices par blocs) : `np.concatenate((A,B,C),axis=i)` crée la matrice $\begin{pmatrix} A & B & C \end{pmatrix}$ si $i = 1$ (horizontal) et la matrice $\begin{pmatrix} A \\ B \\ C \end{pmatrix}$ si $i = 0$ (vertical).

Matrix

Notons qu'il existe même un type `matrix` dans NumPy qui permet de faire des produits matriciels avec $*$ et des puissances de matrices avec $**$:

```
>>> A=np.matrix(np.arange(1,5).\
 reshape(2,2))

>>> A
matrix([[1, 2],
        [3, 4]])

>>> B=np.matrix(np.arange(3,7).\
 reshape(2,2))

>>> B
matrix([[3, 4],
        [5, 6]])
```

```
>>> A*B
matrix([[13, 16],
        [29, 36]])

>>> A**2
matrix([[ 7, 10],
        [15, 22]])

>>> A**(-1)
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```