Le module NumPy

1. Présentation

NumPy est un outil performant pour la manipulation de tableaux à plusieurs dimensions. Il ajoute en effet le type `array`, qui est similaire à une liste, mais dont tous les éléments sont du même type : des entiers, des flottants ou des booléens.

Le module NumPy possède des fonctions basiques en algèbre linéaire, ainsi que pour les transformées de Fourier.

2. Création d'un tableau dont on connaît la taille

```
>>> import numpy as np
>>> a = np.zeros(4)
>>> a
array([ 0., 0., 0., 0.])
>>> nx, ny = 2, 2
>>> a=np.zeros((nx,ny))
>>> a
array( [ [ 0.,  0. ] ,
         [ 0.,  0.] ] )
```

Un tableau peut être multidimensionnel, comme ici, de dimension 3 :

```
>>> a=np.zeros((nx,ny,3))
>>> a
array( [ [ [ 0.,  0.,  0. ],
           [ 0.,  0.,  0. ] ],

         [ [ 0.,  0.,  0. ],
           [ 0.,  0.,  0. ] ] ] )
```

Mais nous nous limiterons dans ce cours aux tableaux uni,bi-dimensionnels.

Il existe également les fonctions `np.ones`, `np.eye`, `np.identity`, `np.empty`, ...
Par exemple `np.identity(3)` est la matrice identité d'ordre 3, `np.empty` est un tableau vide, `np.ones(5)` est le vecteur `[1 1 1 1 1]` et `np.eye(3,2)` est la matrice à 3 lignes et 2 colonnes contenant des 1 sur la diagonale et des zéro partout ailleurs.

Par défaut les éléments d'un tableaux sont des `float` (un réel en double précision) ; mais on peut donner un deuxième argument qui précise le type (`int, complex`, `bool`, ...). Exemple :

```
>>> np.eye(2, dtype=int)
```

3. Création d'un tableau avec une séquence de nombre

La fonction `linspace(premier, dernier, n)` renvoie un tableau unidimensionnel commençant

par `premier`, se terminant par `dernier` avec `n` éléments régulièrement espacés. Une variante est la fonction `arange` :

```
>>> a = np.linspace(-4, 4, 9)
>>> a
array([-4., -3., -2., -1., 0., 1., 2., 3., 4.])
>>> a = np.arange(-4, 4, 1)
>>> a
array([-4, -3, -2, -1, 0, 1, 2, 3])
```

On peut convertir une ou plusieurs listes en un tableau via la commande `array` :

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> b = np.array(range(10))
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> L1, L2 = [1, 2, 3], [4, 5, 6]
>>> a = np.array([L1, L2])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

4. **Création d'un tableau à partir d'une fonction**

```
>>> def f(x, y):
... return x**2 + np.sin(y)
...
>>> a = np.fromfunction(f, (2, 3))
>>> a
array([[ 0. , 0.84147098, 0.90929743],
[ 1. , 1.84147098, 1.90929743]])
```

La fonction `f` a ici été appliquée aux valeurs $x = [0, 1]$ et $y = [0, 1, 2]$, le résultat est

$$[\ [f(0,0), f(0,1), f(0,2)], [f(1,0), f(1,1), f(0,2)]\ ].$$

5. **Manipulations d'un tableau**

   (a) Caractéristiques d'un tableau
   - **a.shape** : retourne les dimensions du tableau
   - **a.dtype** : retourne le type des éléments du tableau
   - **a.size** : retourne le nombre total d'éléments du tableau
   - **a.ndim** : retourne la dimension du tableau (1 pour un vecteur, 2 pour une matrice)

   (b) Indexation d'un tableau

   Comme pour les listes et les chaines de caractères, l'indexation d'un vecteur (ou tableau de dimension 1) commence à 0. Pour les matrices (ou tableaux de dimension 2), le premier index se réfère à la ligne, le deuxième à la colonne. Quelques exemples :

```
>>> L1, L2 = [1, -2, 3], [-4, 5, 6]
>>> a = np.array([L1, L2])
```

```
>>> a[1, 2]  #  extrait l'élément en 2ème ligne et 3ème colonne
6

>>> a[:, 1]  # extrait toute la deuxième colonne
array([-2, 5])

>>> a[1,0:2:2] # extrait les éléments d'indice [début=0, pas=2, fin=2]
               # de la première ligne, le dernier élément n'est pas inclus
array([-4])

>>> a[:, -1:0:-1]  # extrait les éléments d'indice [début=-1, pas=-1, fin=0]
                   # de la première colonne, le dernier élément n'est pas inclus
array([[3, -2],
[6, 5]])

>>> a[a < 0]    # extrait les éléments négatifs
array([-2, -4])
```

(c) Copie d'un tableau

Un tableau est un objet. Si l'on affecte un tableau A à un autre tableau B, A et B font référence au même objet. En modifiant l'un on modifie donc automatiquement l'autre. Pour éviter cela on peut faire une copie du tableau A dans le tableau B moyennant la fonction `copy()`. Ainsi, A et B seront deux tableaux identiques, mais ils ne feront pas référence au même objet.

```
>>> a = np.linspace(1, 5, 5)
>>> b = a
>>> c = a.copy()
>>> b[1] = 9
>>> a; b; c;
array([ 1., 9., 3., 4., 5.])
array([ 1., 9., 3., 4., 5.])
array([ 1., 2., 3., 4., 5.])
```

Une façon de faire une copie de tableau sans utiliser la méthode `copy()` est la suivante :

```
>>> d = np.zeros(b.shape, a.dtype)
>>> d[:] = b
>>> d
array([ 1., 2., 3., 4., 5.])
```

(d) Redimensionnement d'un tableau

Voici comment modifier les dimensions d'un tableau :

```
>>> a = np.linspace(1, 10, 10)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> a.shape = (2, 5)
>>> a
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a.shape = (a.size,)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

```
>>> a.reshape(2, 5)
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

Attention : la fonction `reshape` ne modifie pas l'objet, elle crée une nouvelle vue.
La fonction `flatten` renvoie une vue d'un tableau bi-dimensionnel en tableau uni-dimensionnel.

```
>>> a = np.linspace(1, 10, 10)
>>> a.shape = (2, 5)
>>> a
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a.flatten
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

(e) Boucles sur les tableaux

```
>>> a = np.zeros((2, 3))
>>> for i in range(a.shape[0]):
...     for j in range(a.shape[1]):
...         a[i, j] = (i + 1)*(j + 1)
>>> print a
[[ 1. 2. 3.]
[ 2. 4. 6.]]

>>> for e in a:
... print e
[ 1. 2. 3.]
[ 2. 4. 6.]
```

6. **Calculs avec des tableaux**

Pour bien des calculs il est possible d'éviter d'utiliser des boucles (qui sont très consommatrices de temps de calcul). On peut faire directement les calculs sur des tableaux. Ceux-ci sont faits via des fonctions C, un langage de programmation bas niveau, et sont donc plus rapides. C'est ce qu'on appelle «vectoriser» un programme.

Voici un exemple, dans lequel on veut calculer $b = 3a - 1$:

```
>>> a = np.linspace(0, 1, 1E+06)
>>> %timeit b = 3*a -1
100 loops, best of 3: 10.7 ms per loop
>>> b = np.zeros(1E+06)
>>> %timeit for i in xrange(a.size): b[i] = 3*a[i] - 1
10 loops, best of 3: 1.31 s per loop
```

Voici un autre exemple de calcul fait directement sur un tableau :

```
>>> def f1(x):
...   return np.exp(-x*x)*np.log(1+x*np.sin(x))
...
>>> x = np.linspace(0, 1, 1e6)
>>> a = f1(x)
```

7. **Produit matriciel**

Le calcul matriciel se fait avec la fonction `dot()`.
Attention, si le deuxième argument est un vecteur ligne, il sera transformé si besoin en vecteur colonne, par transposition. Ceci est du au fait qu'il est plus simple de définir un vecteur ligne, par exemple `v=np.array([0,1,2])`, qu'un vecteur colonne, ici `v=np.array([[0],[1],[2]])`.
Exemple :

```
>>> A=np.array([[1,1,1],[0,1,1],[0,0,1]]
>>> v=np.array([0,1,2])
>>> np.dot(v,A)
array([0, 1, 3])
>>> np.dot(A,v)
array([3, 3, 2])
```

On peut aussi transposer préalablement un vecteur ligne :

```
>>> v=np.array([[0,1,2]])
>>> v=np.transpose(v)
>>> v
```

Attention, l'opération `A**2` correspond à une élévation au carré terme à terme. Pour élever une matrice au carré il faut taper `np.dot(A,A)`.

Pour élever une matrice carrée à une puissance n il faut faire une boucle :

```
>>> B=A
>>> for i in range(1,n):
...   B = np.dot(A,B)
```

A la sortie de cette boucle B contiendra A^n.

8. **Produit scalaire**

Il faut utiliser la fonction `vdot()`.

9. `all`, `any` **et** `where`

```
>>> a = np.array([[1, 2, 4], [5, 6, 9]])
>>> np.any(a > 2)
True
>>> np.all(a > 2)
False
>>> np.where(a <= 1)
(array([0]), array([0]))
>>> np.where(a > 1)
(array([0, 0, 1, 1, 1]), array([1, 2, 0, 1, 2]))
```

10. **Autres opérations sur un tableau**

- `a.argmax` : renvoie un tableau d'indices des valeurs maximales selon un axe
- `a.max` : renvoie le maximum
- `a.astype` : renvoie un tableau de a converti sous un certain type
- `a.conj` : renvoie le conjugué de a
- `a.sum` : renvoie la somme des éléments de a
- `a.prod` : renvoie le produit des éléments de a
- `a.transpose` : renvoie le transposé de a

11. **Le sous-module linalg**

(a) Inversion d'une matrice carrée : la fonction `inv()`

Exemple :

```
>>> A=np.array([[1,1,1],[0,1,1],[0,0,1]]
>>> np.linalg.inv(A)
array([[ 1., -1., 0.],
[ 0., 1., -1.],
[ 0., 0., 1.]])
```

(b) Déterminant d'une matrice carrée : la fonction `det()`

(c) Résolution d'un système linéaire $Ax = b$: la fonction `solve(A,b)`

Exemple : on veut résoudre le système linéaire :

$$\begin{cases} x - y + z &= 1 \\ -x + y + z &= 1 \\ 2x - y - z &= 0 \end{cases}$$

```
>>> A=np.array([[1,-1,1],[-1,1,1],[2,-1,-1]])
>>> b= np.array([1,1,0])
>>> np.linalg.solve(A,b) # avec solve()
array([ 1., 1., 1.])
>>> np.dot(np.linalg.inv(A),b) # en inversant A
array([ 1., 1., 1.])
```