On s'intéresse aux librairies scientifiques de Python (la syntaxe ressemble beaucoup à celle de certains logiciels que l'on pourrait utiliser : Scilab et Matlab).

- **NumPy** : librairie de calcul numérique permettant notamment de manipuler des tableaux de dimension quelconque.
- **SciPy** : librairie de calcul numérique : intégration numérique, résolution d'équations différentielles, algèbre linéaire, traitement du signal, optimisation…
- **Matplotlib** : librairie pour les représentations graphiques.
- **Sympy** : librairie pour le calcul symbolique ou formel (on travaille avec des expressions, des objets exacts). C'est tout le contraire des librairies précédentes - d'autres logiciels permettent également de travailler de façon formelle : Sage (très lié à Python), Maxima ou XCas pour quelques logiciels gratuits ou Maple, Mathematica en logiciels payants. On n'en parlera pas du tout.

On peut trouver des exemples sur les fonctions de Numpy sur http://wiki.scipy.org/Numpy_Example_List

---

Tableaux avec Numpy

Création de tableaux (ndarray)

On commence par importer le module numpy en mémoire. Afin de ne pas l'importer dans l'espace de noms principal et afin de conserver une syntaxe pas trop lourde, la plupart du temps on l'importe en tant que np

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

On dispose de différents moyens simples de création pour un tableau à plusieurs dimensions (le type est appelé `ndarray` = N-dimensional array) :

```
# à partir de tableaux Python
>>> A=np.array([[1,2,3],[4,5,6]])
>>> print(A)
[[1 2 3]
 [4 5 6]]
# à partir d'une plage (range de numpy)
>>> np.arange(6)
array([0, 1, 2, 3, 4, 5])
>>> np.arange(3,9)
array([3, 4, 5, 6, 7, 8])
>>> np.arange(0,3,0.3)
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8,  2.1,  2.4,  2.7])
# subdivision régulière d'un segment
>>> np.linspace(0,1,11)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
```

On a également quelques fonctions pour créer des tableaux particuliers

```
# tableau de 0
>>> np.zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
# tableau de 1
>>> np.ones((2,2))
array([[ 1.,  1.],
       [ 1.,  1.]])
# identité
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
# matrice diagonale
>>> np.diag([0,1,2,3])
array([[0, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3]])
# on peut aussi utiliser np.diag(np.arange(4))
```

On peut utiliser une fonction pour définir le terme en position (i, j) :

```
>>> def f(i,j):
      return i+2*j
>>> np.fromfunction(f,(4,3), dtype=int)
array([[0, 2, 4],
       [1, 3, 5],
       [2, 4, 6],
       [3, 5, 7]])
```

Création d'un tableau NumPy

`array(list)`	création à partir d'une liste (de listes si besoin) Python (ou d'un itérable)
`arange(debut,fin,pas)`	comme range Python
`linspace(a,b,n)`	subdivision de $[a,b]$ en n points, extrémités comprises - on peut ajouter `endpoint=False` si on ne veut pas du dernier point
`zeros(dim)`	tableau de 0 avec les dimensions données (dim est un tuple)
`ones(dim)`	idem avec des 1
`eye(n)`	matrice carrée identité de taille n
`eye(p,q,n)`	matrice de taille $p \times q$ avec une diagonale de 1 décalée de n crans
`diag(list)`	matrice carrée de diagonale $list$
`fromfunction(f,dim)`	crée le tableau à partir de la fonction f, avec le format dim

Caractéristiques d'un tableau

Quelques propriétés pour un ndarray :

Propriétés d'un ndarray

`ndim`	nombre de dimension du tableau
`shape`	les dimensions du tableau
`size`	le nombre total d'éléments (produit des dimensions)
`dtype`	le type des éléments du tableau

```
>>> A=np.array([[2,5,3],[4,1,2]])
>>> A.ndim
2
>>> A.shape
(2, 3)
>>> A.size
6
>>> A.dtype
dtype('int64')
```

Opérations sur les tableaux

Toutes les opérations se font élément par élément. Cela peut être très différent de ce qu'on attend (lorsqu'on compare aux matrices usuelles).

```
>>> A=np.array([[2,4],[4,6]])
>>> B=np.array([[-1,4],[5,2]])
>>> A+B
array([[1, 8],
       [9, 8]])
>>> A*B
```

```
array([[-2, 16],
       [20, 12]])
# toutes ces opérations se font case par case
# le produit matriciel se fait avec la fonction dot
>>> np.dot(A,B)
array([[18, 16],
       [26, 28]])
>>> C=np.array([[2,4,5],[2,3,4]])
>>> A+C
# tableaux de tailles différentes
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,2) (2,3)
```

Un peu plus subtil...

```
>>> A=np.array([[2,4],[4,6]]); A
array([[2, 4],
       [4, 6]])
>>> A+1   # on ajoute 1 à chaque élément de A
array([[3, 5],
       [5, 7]])
>>> A=np.array([[2,4],[4,6]]); A
array([[2, 4],
       [4, 6]])
>>> C=np.array([-1,0])
>>> A+C
array([[1, 4],
       [3, 6]])
# On a ajouté à chaque ligne de A le vecteur ligne C (de bonne taille)
>>> D=np.array([[-3],[0]])
>>> A+D
array([[-1,  1],
       [ 4,  6]])
# idem avec un vecteur colonne
#
# un dernier exemple à comprendre... entourez les éléments sélectionnés
>>> A=np.array([[i+2*j for j in range(4)] for i in range(5)])
>>> A
array([[ 0,  2,  4,  6],
       [ 1,  3,  5,  7],
       [ 2,  4,  6,  8],
       [ 3,  5,  7,  9],
       [ 4,  6,  8, 10]])
>>> A[1::2,(2,3)]
array([[5, 7],
       [7, 9]])
```

Pour finir... expliquer :

```
>>> A=np.arange(4)
>>> B=np.array([[3],[2],[5]])
>>> A+B
array([[3, 4, 5, 6],
       [2, 3, 4, 5],
       [5, 6, 7, 8]])
```

Opérations mathématiques

On dispose des fonctions mathématiques usuelles mais qui ont été programmées pour être vectorisées (elles s'appliquent directement à un tableau ndarray)

```
>>> a=np.linspace(0,np.pi,10)
>>> np.sin(a)
array([  0.00000000e+00,   3.42020143e-01,   6.42787610e-01,
         8.66025404e-01,   9.84807753e-01,   9.84807753e-01,
         8.66025404e-01,   6.42787610e-01,   3.42020143e-01,
         1.22464680e-16])
>>> import math
>>> math.sin(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Fonctions usuelles

`square,sqrt`	carré et racine
`sum,prod`	somme et produit
`absolute,fabs`	valeur absolue (`fabs` retourne un flottant)
`minimum,maximum,fmin,fmax`	min et max
`sin,cos,tan`	fonctions trigonométriques
`arcsin,arccos,arctan`	et réciproques
`sinh,cosh,tanh`	fonctions hyperboliques (avec `arc` pour les réciproques
`exp, exp2`	exponentielles (base e et 2)
`log,log10,log2`	logarithmes (base e,10 et 2)
`real,imag,conj,angle`	partie réelle/imaginaire, conjugué et argument

Pour une référence complète : http://docs.scipy.org/doc/numpy/reference/routines.math.html

```
# différence entre fonction et méthode
>>> A=np.array([[0, 1, 2],[3, 4, 5],[6, 7, 8]])
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.sum(A)
36
>>> A.sum()
36
>>> sum(A)
array([ 9, 12, 15])  # pourquoi ? à vous d'expliquer
```

Algèbre linéaire

Dans le module numpy, il existe de nombreuses fonctions liées à l'algèbre linéaire. La plupart se situent dans le sous-module `numpy.linalg` :

```
>>> A=np.array([[2,4],[4,6]])
>>> np.linalg.det(A)  # déterminant de A
-4.0
```

Pour des raisons pratiques, on a parfois envie d'importer ce sous-module dans un nouvel espace de noms :

```
>>> import numpy.linalg as lin
>>> lin.det(A)
-4.0
```

Algèbre linéaire

`dot(A,B)`	produit matriciel
`trace(A)`	trace
`linalg.matrix_power(M, n)`	puissance d'une matrice
`linalg.det(A)`	déterminant de A
`linalg.inv(A)`	inverse de A
`linalg.solve(A,B)`	résolution de $Ax = B$ (B vecteur ou matrice)
`linalg.eig(A)`	valeurs propres et vecteurs propres de A
`linalg.eigvals(A)`	valeurs propres de A
`linalg.norm(A)`	norme 2 de la matrice

Pour les quelques fonctions supplémentaires : http://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Sélection, extraction

On dispose d'une syntaxe assez proche de celle de l'extraction de certaines parties d'un tableau

- **pour un élément**

```
>>> A=np.array([[2,4,5],[6,8,1]])
>>> A
array([[2, 4, 5],
       [6, 8, 1]])
>>> A[0,0]
2
>>> A[1,2]=0
>>> A
array([[2, 4, 5],
       [6, 8, 0]])
```

- **sur les lignes**

```
>>> A=np.array([[2,4,5],[6,8,1]])
>>> A
array([[2, 4, 5],
       [6, 8, 1]])
>>> A[1]
array([6, 8, 0])
```

- **une colonne**

```
>>> A[:,0]
array([2, 6])
>>> A[:,0].shape
(2,)
# cela devient un tableau à une seule dimension
>>> A[:,(0,2,1,2,1,0)]
array([[2, 5, 4, 5, 4, 2],
       [6, 0, 8, 0, 8, 6]])
# on crée un tableau avec différentes colonnes de A
```

- **comme en Python**

```
>>> A=np.arange(12)
>>> A
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> A[1:9:2]
array([1, 3, 5, 7])
```

Opérations élémentaires

On peut utiliser ces opérations pour modifier directement toute une ligne ou une colonne d'une matrice (et même plusieurs lignes ou colonnes en même temps si on le souhaite) et ainsi programmer facilement les opérations élémentaires sur les lignes d'une matrice. On va l'appliquer sur la matrice A définie par

```
>>> A=np.arange(12).reshape(3,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

On repart de cette matrice à chaque fois.

- **dilatation** : rien de plus simple

```
>>> A[2]=3*A[2]
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [24, 27, 30, 33]])
```

- **permutation** : ne pas se tromper dans les notations. Si on veut permuter les lignes 0 et 2

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

Cela a remplacé le coefficient en position $(0,2)$ de la matrice par celui en position $(2,0)$, et pas les lignes 0 et 2. Pour le faire correctement :

```
>>> A[(0,2),:]=A[(2,0),:]
>>> A
array([[ 8,  9, 10, 11],
       [ 4,  5,  6,  7],
       [ 0,  1,  2,  3]])
```

- **transvection** : si on veut effectuer l'opération $L_2 \leftarrow L_2 + 2L_1$, il suffit d'écrire

```
>>> A[1]=A[1]+2*A[0]
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  7, 10, 13],
       [ 8,  9, 10, 11]])
```

Vues et copies

On retrouve le même principe qu'en Python, mais en encore plus poussé :

```
>>> A=np.array([[2,4,5],[6,8,1]])
>>> A
array([[2, 4, 5],
       [6, 8, 1]])
>>> B=A[1]
>>> B
array([6, 8, 1])
>>> B[0]=9
>>> B
array([9, 8, 1])
>>> A
array([[2, 4, 5],
       [9, 8, 1]])
```

Dans l'exemple précédent, on remarque que B n'est pas un nouvel objet - ses éléments sont toujours ceux de A. La sélection d'une partie d'une matrice ne crée pas un nouvel objet mais ce qu'on appelle **une vue** sur le tableau de départ. Si on veut vraiment créer un nouveau tableau, on doit en faire une copie.

```
>>> A=np.array([[2,4,5],[6,8,1]])
>>> B=A.copy()
>>> B[0,0]=-1
>>> B
```

```
array([[-1,  4,  5],
       [ 6,  8,  1]])
>>> A
array([[2, 4, 5],
       [6, 8, 1]])
```

Cela fonctionne également sur les vues

```
>>> A=np.array([[2,4,5],[6,8,1]])
>>> A
array([[2, 4, 5],
       [6, 8, 1]])
>>> B=A[:,(0,2)].copy()
>>> B
array([[2, 5],
       [6, 1]])
>>> B[1,1]=-1
>>> B
array([[ 2,  5],
       [ 6, -1]])
>>> A
array([[2, 4, 5],
       [6, 8, 1]])
```

Redimensionnement

Les matrices ne sont pas toujours de la bonne taille... on a parfois envie de les redimensionner. Cela se réalise grâce à la méthode `reshape` :

```
>>> A=np.arange(12)
>>> A
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> A.reshape((3,4))
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> A   # A n'est pas modifiée
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> B=A.reshape((4,3))
>>> B
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> B[0,0]=-1
>>> A  # A a été modifiée - normal puisque B est une vue sur A
array([-1,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Si on veut modifier la matrice, on doit utiliser la méthode `resize` :

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

Quelques cas particuliers : ne pas mélanger $(n,1)$ et $(n,)$

```
>>> A=np.arange(12)
>>> B=A.reshape((4,3))
>>> C=B[:,1]
>>> C
array([ 1,  4,  7, 10])
>>> C.shape
(4,)  # une seule dimension
>>> C.reshape(2,2) # ou C.reshape((2,2))
array([[ 1,  4],
       [ 7, 10]])
>>> C.reshape(1,4)
array([[ 1,  4,  7, 10]])
>>> A=np.array([0,1,2])
>>> A.reshape(3,1) # 3 colonnes, 1 ligne
```

```
array([[0],
       [1],
       [2]])
>>> A.reshape(1,3) # 1 ligne, 3 colonnes mais 2 dimensions
array([[0, 1, 2]])
>>> B=A.reshape(1,3)
>>> B
array([[0, 1, 2]])
>>> B.reshape(3) # plus qu'une dimension
array([0, 1, 2])
>>> B.reshape((3,)) # autre écriture
array([0, 1, 2])
```

Empilement

On peut empiler (horizontalement ou verticalement) deux tableaux. C'est par exemple utile pour appliquer un pivot de Gauss en même temps sur une matrice et un vecteur colonne (on colle le vecteur colonne à droite et on effectue ainsi les opérations en même temps). Il y a pas mal de possibilités (notamment puisque les tableaux peuvent être de dimensions plus grandes que 1 ou 2) mais on va rester simple ici

```
>>> a=np.array([1,2,3])
>>> b=np.array((2,3,4))
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
>>> A=np.eye(4)
>>> B=np.array([1,2,3,4])
>>> B.resize(4,1)
>>> np.hstack((A,B));
array([[ 1.,  0.,  0.,  0.,  1.],
       [ 0.,  1.,  0.,  0.,  2.],
       [ 0.,  0.,  1.,  0.,  3.],
       [ 0.,  0.,  0.,  1.,  4.]])
```

Commentaires sur le type

Les tableaux sont des tableaux avec un seul type de données (cela permet d'optimiser les calculs, les accès), qu'on obtient par `dtype`. Lors de la création les données sont converties dans un même type :

```
>>> a=np.array([1,2,3.])
>>> a
array([ 1.,  2.,  3.])
>>> a.dtype
dtype('float64')
```

On peut forcer le type des données lors de la création d'un tableau (dans toutes les fonctions décrites précédemment)

```
>>> a=np.array([1,2,3], dtype='complex64')
>>> a
array([ 1.+0.j,  2.+0.j,  3.+0.j], dtype=complex64)
```

On peut aussi modifier le type de tous les éléments d'un tableau

```
>>> a=np.array([1,2,3])
>>> a.dtype
dtype('int64')
>>> a
array([1, 2, 3])
>>> a.astype('float64')
array([ 1.,  2.,  3.])
>>> a.dtype
dtype('int64')  # le tableau n'a pas été modifié.
>>> a=a.astype('float64')
>>> a
array([ 1.,  2.,  3.])
>>> a.dtype
dtype('float64')
```

Quelques types numériques usuels

`int8,int16,int32,int64,int128`	entiers sur le nombre de bits précisé
`float32,float64,float128`	nombres flottants
`d,f`	`d`=float64 (double) et `f`=float32
`complex64, complex128`	nombres complexes (`D`=complex128)
`bool`	booléens (`True, False`)

Nombres aléatoires

On peut tirer aléatoirement des nombres (avec différentes distributions possibles - pour une référence complète : http://docs.scipy.org/doc/numpy/reference/routines.random.html). On n'en présente ici que quelques unes :

```
# on charge le module numpy correspondant
>>> import numpy.random as random
>>> random.rand(3,2)  # distribution uniforme dans [0,1]
array([[ 0.02855742,  0.52332929],
       [ 0.78417361,  0.84822666],
       [ 0.89726416,  0.04540198]])
>>> random.randn(3,2) # distribution normale (gaussienne)
array([[ 0.28568179, -1.01267956],
       [ 0.06839141, -1.45589655],
       [-1.37344087,  0.13437214]])
```

Dans le tableau suivant, `format` est un entier ou un tuple représentant donnant les dimensions du tableau

Tableaux aléatoires (numpy.random)

`rand(d1,d2,..)`	distribution uniforme dans $[0,1]$, matrice à plusieurs dimensions
`random(format)`	distribution uniforme dans $[0,1]$, dimension donnée par le tuple `format`
`randn(d1,d2,..)`	loi normale centrée réduite, matrice à plusieurs dimensions
`randint(max)`	entier entre 0 et $max-1$
`randint(min,max,format)`	tableau d'entiers entre min et max (non compris)

Compléments : tests et sélection

Les opérations sur un tableau se font case par case... on peut en profiter pour effectuer des tests :

```
>>> A=(np.arange(12)).reshape(3,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> B=A>5
>>> B
array([[False, False, False, False],
       [False, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
>>> A*B  # True est considéré comme 1 et False comme 0
array([[ 0,  0,  0,  0],
       [ 0,  0,  6,  7],
       [ 8,  9, 10, 11]])
>>> A[B] # on sélectionne les éléments de A à partir du tableau de booléens B
```

On peut exploiter l'indexage par tableau de booléens afin de réaliser un filtrage de la matrice : on veut par exemple mettre à 0 tous les termes d'une matrice aléatoire (loi uniforme sur $[0,1]$) qui sont inférieurs à $0,2$ ou supérieurs à $0,8$:

```
>>> A=np.random.rand(5,5)
>>> A
array([[ 0.48566809,  0.57685161,  0.40003415,  0.07051508,  0.99748014],
       [ 0.94857656,  0.28916706,  0.43693069,  0.34616586,  0.33395492],
       [ 0.08702542,  0.17393928,  0.31909167,  0.7410378 ,  0.02276524],
       [ 0.19723913,  0.28455439,  0.49688163,  0.17100071,  0.48735244],
       [ 0.21675362,  0.08592115,  0.88847285,  0.03770435,  0.81880245]])
>>> B=(A<0.2) | (A>0.8)
>>> A[B]=0
>>> A
array([[ 0.48566809,  0.57685161,  0.40003415,  0.        ,  0.        ],
       [ 0.        ,  0.28916706,  0.43693069,  0.34616586,  0.33395492],
       [ 0.        ,  0.        ,  0.31909167,  0.7410378 ,  0.        ],
       [ 0.        ,  0.28455439,  0.49688163,  0.        ,  0.48735244],
       [ 0.21675362,  0.        ,  0.        ,  0.        ,  0.        ]])
```

Matrices

Numpy définit également un type matrice (un tableau à 2 dimensions) :

```
>>> A=np.matrix([[1,4,3],[2,1,2]])
>>> B=np.matrix([-1,1,0])
>>> B
matrix([[-1,  1,  0]]) # conversion en 2 dimensions automatique
>>> B.T
matrix([[-1],
        [ 1],
        [ 0]])
>>> B.transpose()
matrix([[-1],
        [ 1],
        [ 0]])
>>> np.transpose(B)
matrix([[-1],
        [ 1],
        [ 0]])
>>> A*(B.T) # produit matriciel standard
matrix([[ 3],
        [-1]])
>>> C=np.array(B)  # quelques conversions
>>> C
array([[-1,  1,  0]])
>>> C.reshape(3)
array([-1,  1,  0])
```

Opérations sur les matrices

`*`	produit matriciel
`transpose`	transposée (fonction ou méthode)
`A.T`	transposée de A