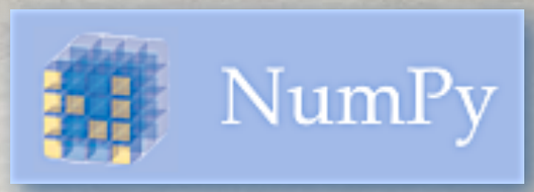
INFORMATIQUE

Programmation Numérique
avec Numpy

Programmation scientifique avec Numpy



Numpy ou numerical python, est un outil de programmation scientifique basé sur les tableaux de données numériques. Il permet :

- **Une syntaxe optimisée pour coder rapidement des opérations abstraites «haut-niveau».**
- **Des appels de fonction pré-compilées (en C) pour optimiser les temps de calcul.**

Objet numpy :

la commande **numpy.array()** permet de le **créer à partir d'une liste**

```
from numpy import *

a = array([1,2,3.14159,"A"])          #definition explicite a partir d'une liste python

print(a)
print("type: ", type(a),"a.dim: ", a.ndim,"a.shape:", a.shape)
```

```
['1' '2' '3.14159' 'A']
type:  <class 'numpy.ndarray'>
a.dim:  1 a.shape: (4,)
```

```
b = array( [[ [0,1], [2,3]], [[4,5], [7,7]]] )
print(b)
print("type: ",type(b),"b.dim: ", b.ndim,"b.shape:", b.shape)
```

```
[[[0 1]                    type:  <class 'numpy.ndarray'>
  [2 3]]                   b.dim:  3 b.shape: (2, 2, 2)
 [[4 5]
  [7 7]]]
```

3D

```
c = arange(15).reshape(3,5)
print(c)
print("type: ", type(c),"c.dim: ", c.ndim,"c.shape:", c.shape)
```

```
[[ 0  1  2  3  4]          type:  <class 'numpy.ndarray'>
 [ 5  6  7  8  9]          c.dim:  2 c.shape: (3, 5)
 [10 11 12 13 14]]
```

2D

Création d'un tableau :

D'autre commandes de base permettent de créer rapidement les briques essentielles pour la programmation numérique : Tableau de 0, de 1, extrapolation linéaires, (compréhension de liste).

```
myZero = zeros((3,4))
print(myZero)

print(ones(10))  #on peut aussi faire des 1 ...

#comprehension de liste
myTab = array([ [i*j for j in range(10)]  for i in range(10) ])
print(myTab)
print(myTab.ndim, myTab.shape)

#abscisse en temps avec arange ~ range(a, b, pas)
time = arange(0., 10., 0.01)
print(time[0:10])

#Echantillonage lineaire
x = linspace( 0, 2*pi, 100 )
print(x)
```

#Pas de temps

Nb. de points

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]                    myZero

[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

```
[[ 0  0  0  0  0  0  0  0  0  0]
 [ 0  1  2  3  4  5  6  7  8  9]
 [ 0  2  4  6  8 10 12 14 16 18]
 [ 0  3  6  9 12 15 18 21 24 27]
 [ 0  4  8 12 16 20 24 28 32 36]
 [ 0  5 10 15 20 25 30 35 40 45]
 [ 0  6 12 18 24 30 36 42 48 54]
 [ 0  7 14 21 28 35 42 49 56 63]
 [ 0  8 16 24 32 40 48 56 64 72]
 [ 0  9 18 27 36 45 54 63 72 81]]

2 (10, 10)                             myTab
```

```
[ 0.  0.06346652  0.12693304  0.19039955  0.25386607  0.31733259

x          .....      6.09278575  6.15625227  6.21971879  6.28318531]
```

```
[ 0.   0.01  0.02  0.03  0.04  0.05 .....

time       ........          0.06  0.07  0.08  0.09]
```

Accès aux éléments :

```
print( myTab[5][7])     # acces esprit Liste :
print( myTab[5,7])      # acces esprit numpy array :
```

```
>>> myTab[5][7]
35
>>> myTab[5, 7]
35
```

Sens identique pour isoler un seul élément

```
>>> print(myTab[0:4, 0:2])

 [[0 0]
  [0 1]
  [0 2]
  [0 3]]
```

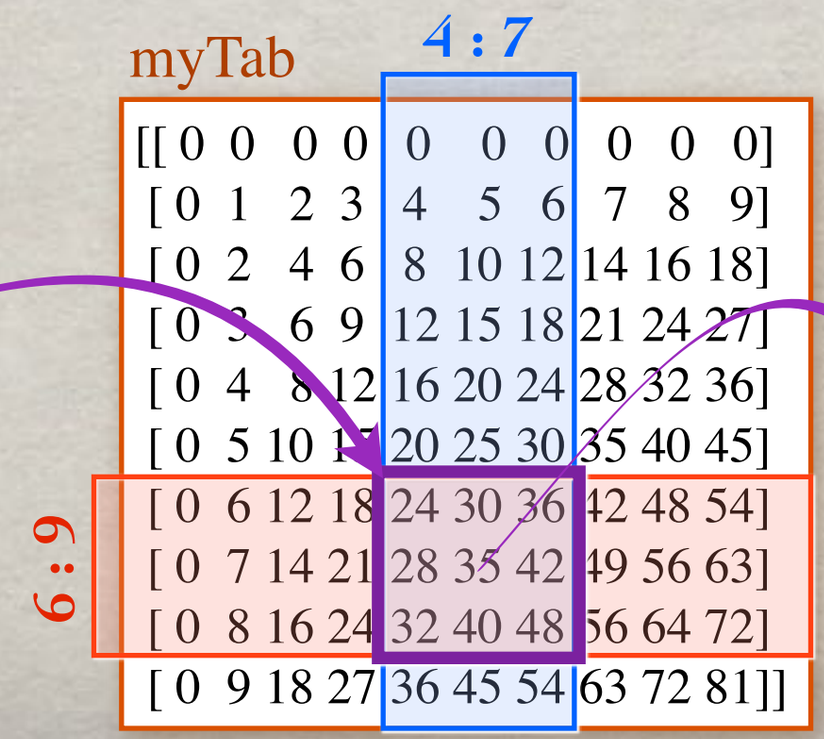
A un sens très différent de :

```
>>> print(myTab[0:4][0:2])

 [[0 0 0 0 0 0 0 0 0 0]
  [0 1 2 3 4 5 6 7 8 9]]

>>> print(myTab[0:4])
[[ 0  0  0  0  0  0  0  0  0  0]
 [ 0  1  2  3  4  5  6  7  8  9]
 [ 0  2  4  6  8 10 12 14 16 18]
 [ 0  3  6  9 12 15 18 21 24 27]]
```

Extraction d'un sous-tableau :

myTab[i1 : i2 , j1 : j2]

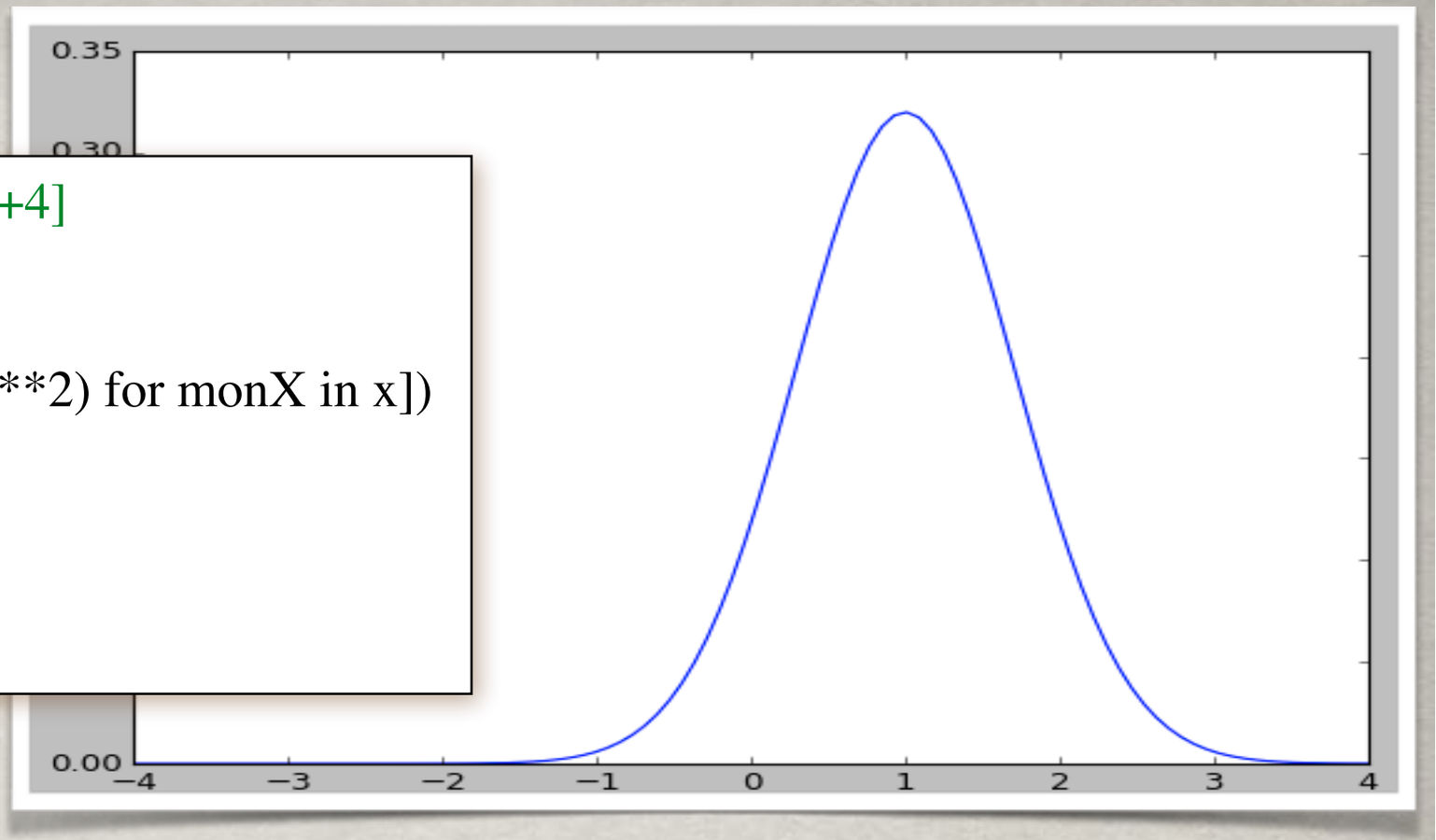
Il n'y a pas de telle syntaxe avec les listes python




```
>>> myTab[6:9 , 4:7]

 array([[24, 30, 36],
        [28, 35, 42],
        [32, 40, 48]])
```

Tracé d'une gaussienne

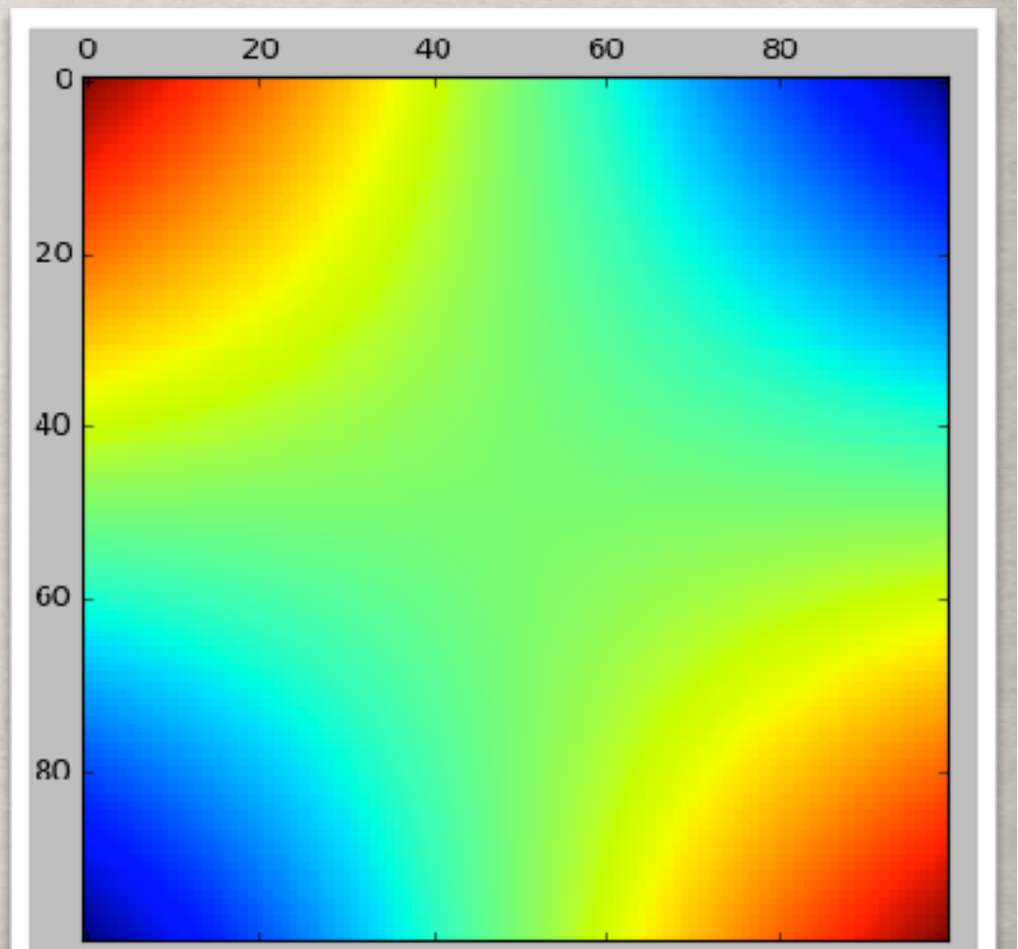
```
x=linspace(-4., +4., 100)      #Domaine [-4, +4]

moy=1.;   sigma=1.25
y = array([exp(-(monX-moy)**2)/(2*sigma**2) for monX in x])
y = y / sum(y)
                    #Gaussienne normalisée.
plt.plot(x, y)
plt.show()
```



Tracé de z(x,y) = xy

```
x = linspace(-1.,1., 100); y=linspace(-1.,1., 100)
     #Domaine [-1, +1]x[-1, +1]

z = array([[monX*monY for monX in x] for monY in y])
```

```
plt.matshow(z)
plt.show()
```



Opérations de base sur les tableaux :

Les opérations usuelles sont réalisées sur chaque composante individuelle.
=> On parle d' "**element wise operation**".

```
X = arange(-10, 10, 2)
print(X)
print(3*X)
print(X**2)
```

Element Wise

1D

```
X    : [ -10  -8  -6  -4  -2   0   2   4   6   8]
3*X  : [ -30 -24 -18 -12  -6   0   6  12  18  24]
X**2 : [100  64  36  16   4   0   4  16  36  64]
```

```
Y = 3*X + ones(len(X))
print(X)
print(Y)
print(X+Y)
print(X*Y)
```

Element Wise

1D

```
X   : [ -10  -8  -6  -4  -2   0   2   4   6   8]
Y   : [ -19. -15. -11.  -7.  -3.   1.   5.   9.  13.  17.]
X+Y : [ -29. -23. -17. -11.  -5.   1.   7.  13.  19.  25.]
X*Y : [ 190. 120.  66.  28.   6.   0.  10.  36.  78. 136.]
```

ATTENTION : les tableaux mis en jeu doivent avoir les <u>mêmes dimensions</u>

```
Z = arange(0,10,2)
print(X+Z)
print(X*Z)
```

PB

```
Traceback (most recent call last):  ValueError:

    operands could not be broadcast together with shapes (10) (5)
```

Le principe est exactement le même avec des tableaux de données en deux dimensions ou «matrice de données». Il se généralise pour un nombre arbitraire de dimensions.

Element Wise 2D

```
#soit une matrix (3,3) :
X = arange(9).reshape(3,3)

Y = X**2
Z = exp(X)

print(X);   print(Y);    print(Z)
```

- La même opération est réalisée sur chacun des éléments du tableau de données.

- Si plusieurs opérandes (tableaux) sont impliqués => ils doivent avoir les mêmes «shapes»

X

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

Y

```
[[ 0  1  4]
 [ 9 16 25]
 [36 49 64]]
```

Z

```
[[  1.00000000e+00   2.71828183e+00   7.38905610e+00]
 [  2.00855369e+01   5.45981500e+01   1.48413159e+02]
 [  4.03428793e+02   1.09663316e+03   2.98095799e+03]]
```

Y+Z

```
[[  1.00000000e+00   3.71828183e+00   1.13890561e+01]
 [  2.90855369e+01   7.05981500e+01   1.73413159e+02]
 [  4.39428793e+02   1.14563316e+03   3.04495799e+03]]
```

```
EXP = zeros(9).reshape(3,3)

for k in range(10):
    EXP += X**k / factorial(k)

print(EXP)
```

```
[[  1.00000000e+00   2.71828183e+00   7.38905610e+00]
 [  2.00855369e+01   5.45977739e+01   1.48283184e+02]
 [  3.90468618e+02   1.03327369e+03   2.37346778e+03]]
```

ATTENTION : cela n'a rien à voir avec le produit matriciel, chaque élément est indépendant *Element Wise*

Quelques mots sur les générateurs de nombres aléatoires numpy :

On utilise le module **numpy.random** :

- random.**rand**(N) -> génère un **array** de N éléments entre 0 et 1 (distribution porte)
- random.**randn**(N) -> génère un **array** de N éléments de moyenne 0. et écart-type 1. (distribution gauss-normale)

- **mean** -> moyenne d'un tableau numpy [pour toute dimension]
- **var** -> variance d'un tableau numpy (soit l'écart-type au carré).

```
#Générateur de valeurs aléatoires :

x = random.rand(10000)  #Bruit blanc entre 0 et 1

print("moyenne = ", mean(x))
print("sigma   = ", var(x)**0.5)
print("1/(2*sqrt(3)) = ", 1/(2*3**0.5) )


#Loi Gauss normale :

y = 5.*random.randn(10000)+10.
print("moyenne = ",mean(y))
print("sigma   = ", var(y)**0.5 )
```

```
moyenne =  0.503749106271
  sigma   =  0.288125883746

1/(2*sqrt(3)) =  0.288675134 ...
```

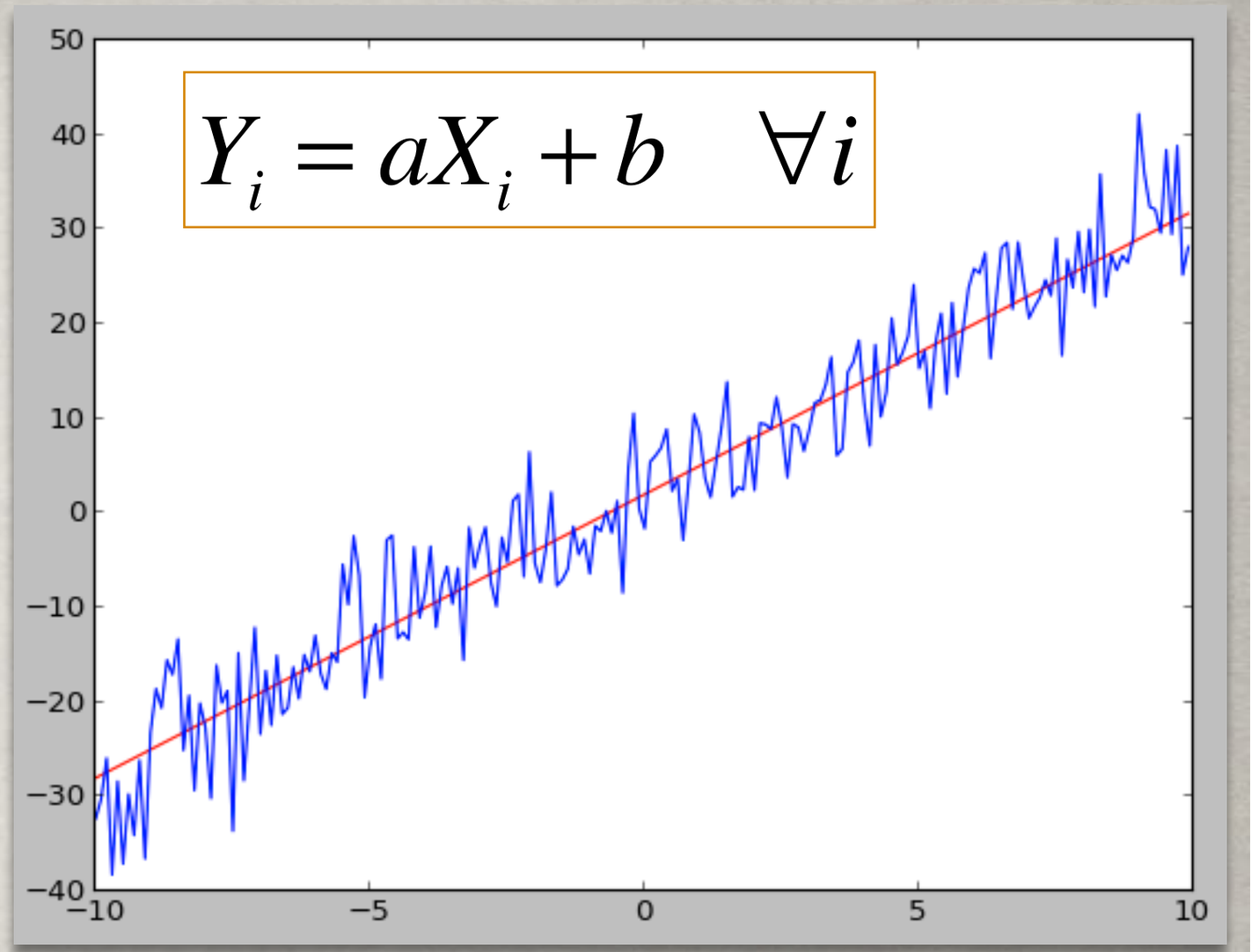
```
>>> x[5000:5010]
array([ 0.30662069,  0.14817712,  0.76268679,  0.23822813,  0.12438644,
        0.80035323,  0.34590838,  0.45425313,  0.93250598,  0.03796143])
```

```
>>> y[5000:5010]
array([ 20.55226167,  12.27504108,  15.79964709,   2.14197313, 6.34268206,
        11.68634653,   9.89063696,   8.37077617, 9.66910074,  12.7879308 ])
```

```
moyenne =  10.084358077
sigma     =    5.25082807314
```

Exemple 1D : Tracé d'une droite

```
X = arange(-10, 10, 0.1)    ;      (a, b) = (3, 2)

Yth = a*X + b   #une droite.

#bruit gaussien additionnel :
epsilon = 5*random.randn(len(Yth))
Ynoise=Yth+epsilon



plt.plot(X,Yth,'r')
plt.plot(X,Ynoise,'b')
plt.show()
```



Algèbre linéaire avec Numpy

Numpy permet de réaliser les opérations matricielles et vectorielles avec des tableaux de données numpy.

Il ne s'agit en aucun cas de faire ici une liste exhaustive des possibilités de numpy ou sciPy, mais de donner des exemples de base de son utilisation. Toutes les opérations usuelles d'algèbre linéaire sont possibles :

- www.SciPy.org ! —> Numpy
- https://docs.scipy.org/doc/numpy/reference/routines.linalg.html
- https://docs.scipy.org/doc/scipy-0.18.1/reference/tutorial/linalg.html (des exemples)

On peut représenter les vecteurs colonnes comme une matrice de n lignes et 1 colonne :

```
U = np.array( [[1], [0], [2]])
V = np.array(  [[1], [1], [1]])
```

```
>>> U
array([[1],
       [0],
       [2]])
```

shape (3, 1)

```
>>> V
array([[1],
       [1],
       [1]])
```

shape (3, 1)

Transposition :

.T ou .transpose()

```
>>> U.T
array([[1, 0, 2]])
```

shape (1, 3)

Bien des opérations ne sont accessibles que via le module **linalg** (linear algebra) de numpy.

Norme euclidienne :

```
>>> np.linalg.norm(U)
2.2360679774997898
```

Norme « 1 » :

```
>>> np.linalg.norm(U, 1)
3
```

Norme infinie :

```
>>>np.linalg.norm(U, np.inf)
2
```

Le choix de la norme est même possible en option !

Produit scalaire :

C'est le produit matriciel **.dot()** d'un vecteur transposé avec un vecteur colonne. On obtient un **scalaire** => shape (1, 1)

```
>>> U.T.dot(V)
array([[3]])
```

Produit vectoriel :

```
W = np.cross(U, V, axis = 0)
```

On doit préciser la direction car U.shape : (**3**,1)

```
>>> W
array([[-2],
       [ 1],
       [ 1]])
```

```
>>> W = cross(U.T, V.T)

>>> W
array([[-2,  1,  1]])
```

Attention : par défaut il faut des vecteurs en ligne

Produit matricielle :

```
A = array([[1,0,2],[2,1,0]])
B = array([[1,2],[1,2],[1,2]])

AB = dot(A,B)
BA = dot(B,A)
```

A • B = AB

```
A                B                AB
[[1 0 2]    •    [[1 2]      =    [[3 6]
 [2 1 0]]         [1 2]           [3 6]]
                  [1 2]]
```

B • A = BA

```
B                A                BA
[[1 2]      •    [[1 0 2]    =    [[5 2 2]
 [1 2]            [2 1 0]]         [5 2 2]
 [1 2]]                            [5 2 2]]
```

```
theta = pi/6   #Matrice rotation +30° / (Oz)

R = np.array([[ cos(theta),   -sin(theta),      0],
              [  sin(theta),   cos(theta),      0],
              [       0     ,       0     ,     1]])
```

```
>>> R
array([[  0.866,    - 0.5   ,     0.   ],
       [  0.5   ,     0.866,     0.   ],
       [  0.    ,     0.    ,     1.   ]])
```

Matrice inverse :

```
Rinv = np.linalg.inv(R)
```

np.linalg.inv()

—> rotation de -30°/ (Oz)

```
>>> Rinv
array([[  0.866,      0.5   ,     0.   ],
       [ - 0.5   ,     0.866,     0.   ],
       [  0.    ,     0.    ,     1.   ]])
```

Applications simples :

```
#Matrices rotations de 90° autour d'un axe
theta = pi/2
Rz90 = array([[cos(theta), -sin(theta),      0      ],
              [sin(theta),  cos(theta),      0      ],
              [    0      ,      0      ,      1      ]])
```

```
Rx90 = array([[     1      ,      0      ,      0      ],
              [     0      , cos(theta),  -sin(theta)],
              [     0      , sin(theta),   cos(theta)]])
```

```
Ry90 = array([[cos(theta),      0      ,+sin(theta)],
              [     0      ,      1      ,      0      ],
              [-sin(theta),      0      , cos(theta)]])
```

Soit l'enchaînement des 3 rotations en 3D :

```
>>> Rot = Ry90.dot( Rx90.dot( Rz90 ))      #soit   Ry90 . Rx90 . Rz90
...
array([[ 1.00000000e+00,  0.00000000e+00,  6.12323400e-17],
       [ 6.12323400e-17,  3.74939946e-33, -1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  3.74939946e-33]])
```

```
>>> np.around(Rot)
array([[ 1. ,  0. ,  0.],
       [ 0. ,  0. , -1.],
       [ 0. ,  1. ,  0.]])
```

Inverse :

```
>>> around(linalg.inv(Rot))
array([[ 1. ,  0. , -0.],
       [-0. ,  0. ,  1.],
       [ 0. , -1. ,  0.]])
```

On est tjrs. sous le joug des erreurs numériques
=> **np.around()**

```
>>> cos(pi/2)
6.123233995736766e-17
```

Déterminant :

```
>>> linalg.det(Rot)
1.0
```

Valeurs propres : (chercher « eigenvalues »)

```
>>> linalg.eigvals(Rot)
array([ 1.  ,  3.74939946e-33+1.j,  3.74939946e-33-1.j])
```

Résolution d'un système linéaire

Implémentation du pivot de Gauss avec numpy

1 - La première étape consiste à combiner les lignes pour former un système d'équations avec une matrice triangulaire

On réalise une boucle, de sorte que pour chaque colonne on doit :

- Trouver le pivot (getPivot)
- Combiner les lignes pour mettre le pivot sur la diagonale et n'obtenir que des zéros en dessous du Pivot

```
def sysTriangulaire(A,B):

    N, M, NB = len(A), len(A[0]), len(B)
    if not(N==M==NB): return "PB de dimension du système"  #test dimension du system

    for i in range(N):  #boucle sur les colonnes

        p = getPivot(A, i)
        if (p==-1): return "Système non solvable !"

        A,B = combiner(A,B,i,p)  #place le pivot et combine les lignes
                                 #pour former les zéros.
    return A, B
```

On s'appuie donc sur les deux fonctions suivantes :

Renvoie l'indice p du Pivot et -1 si il n'y en a pas.

```
def getPivot(M, i):  #renvoie l'indice p du pivot pour M[i:, i:]
    p=i
    while(M[p,i]==0 and p<len(M)-1): #descend jusqu'à trouver
        p+=1                         #un pivot non nul.
    if (p==len(M)-1 and M[p,i]==0):
        return -1    #on ne trouve pas de pivot non nul
    return p
```

Combine les lignes de la matrice et du vecteur colonne :
On obtient un système équivalent mais avec des zéros sous le Pivot.

```
def combiner(M,C,i,p):  #combine les lignes de M[i:, i:] et B[i:]
    N=len(M)            #autour d'un pivot en p et de valeur pVal

    M[i], M[p] = M[p], M[i] #met la ligne du pivot sur la position diagonale i en cours
    p=i; pVal=M[p,i]        #soit p=i

    for k in range(i+1, N):
        coeff = M[k][i]                    #Numpy permet d'éviter les boucles :
        M[k]  = pVal*M[k] - coeff*M[p]     #Combine les lignes pour former les zéros.
        C[k]  = pVal*C[k] - coeff*C[p]     #idem vecteur colonne.

    return M, C
```

Exemple de déroulement

A

```
[[ 4.    1.    2.    3.]
 [ 5.    2.    0.    3.]
 [ 0.    9.    1.    4.]
 [ 2.    1.    3.    4.]]
```

B

```
[[    1.]
 [    2.]
 [    3.]
 [    4.]]
```

```
[[  4.   1.   2.   3.]
 [  0.   3. -10.  -3.]
 [  0.  36.   4.  16.]
 [  0.   2.   8.  10.]]
```

```
[[   1.]
 [   3.]
 [  12.]
 [  14.]]
```

```
[[  4.   1.   2.   3.]
 [  0.   3. -10.  -3.]
 [  0.   0. 372. 156.]
 [  0.   0.  44.  36.]]
```

```
[[   1.]
 [   3.]
 [ -72.]
 [  36.]]
```

T

```
[[  4.   1.   2.    3.]
 [  0.   3. -10.   -3.]
 [  0.   0. 372.  156.]
 [  0.   0.   0. 6528.]]
```

C

```
[[    1.]
 [    3.]
 [  -72.]
 [16560.]]
```

2 - On résout alors le système triangulaire de bas en haut de façon à n'avoir plus qu'une inconnue à trouver à la fois :

```
def resolTriangulaire(T,C):
    N=len(T); X=np.array([0.]*N)
    for k in range(0, N):
        i   = N-k-1
        val = C[i]

        for j in range(i+1, N):
            val -= X[j]*T[i][j]

        X[i] = val/T[i][i]

    return np.array([[X[k]] for k in range(N)])
```

AX=B

```
Y = np.linalg.solve(A,B)
print("Solution linalg : ")
print(Y); print()


T,C = sysTriangulaire(A,B)
print("Solution pivot de Gauss : ")
print(resolTriangulaire(T,C))
```

A | B

```
[[ 4.  1.  2.  3.]     |    [[ 1.]
 [ 5.  2.  0.  3.]     |     [ 2.]
 [ 0.  9.  1.  4.]     |     [ 3.]
 [ 2.  1.  3.  4.]]    |     [ 4.]]
```

```
Solution linalg :          Pivot de Gauss :
[[-0.86029412]             [[-0.86029412]
 [-0.65441176]              [-0.65441176]
 [-1.25735294]              [-1.25735294]
 [ 2.53676471]]             [ 2.53676471]]
```