L'essentiel du TD n°6

Le module `numpy` et l'algèbre linéaire

Une matrice = un tableau `numpy` à deux dimensions

1. Créer un tableau
2. Lire dans un tableau : les indices commencent à 0
3. Modifier un tableau : copies et vues
4. Calculer : $*$ n'est pas le produit matriciel
5. Comparer des tableaux
6. Des fonctions sur les matrices : produit, symétrie, quantificateurs
7. `numpy.linalg` : inverse, système, rang

Chaque diapo est illustrée par une section du fichier `EssentielTD6.py`.

Ces documents doivent suffire pour vos révisions sur ce thème.

1. Créer un tableau

```
import numpy as np

A = np.array([[1, 2], [3, 4]])  # liste de listes
np.zeros([2, 3])        # matrice nulle 2 x 3
np.eye(3)               # identité I_3
np.diag([1, 2, 3])      # matrice diagonale
np.linspace(0, 1, 5)    # [0. 0.25 0.5 0.75 1.]

B = np.array([[1, 2], [3, 4]])     # entiers
B[0, 0] = 2.7                      # stocke 2 !
B = np.array([[1., 2.], [3., 4.]]) # flottants
```

À retenir

- `np.zeros`, `np.eye`, `np.linspace` donnent des **flottants** ;
- `np.zeros([n, p])` : le format entre **crochets** ;
- tous les coefficients ont le **même type** : un flottant rangé dans un tableau d'entiers est tronqué. Écrire `1.` pour avoir des flottants.

2. Lire dans un tableau : les indices commencent à 0

```
R = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
R[1, 2]         # 7   (ligne 1, colonne 2)
R[0]            # [1 2 3 4]   ligne 0
R[:, 1]         # [ 2  6 10]  colonne 1
R[-1]           # dernière ligne
R[0:2, 1:3]     # [[2 3] [6 7]]
n, p = np.shape(R)   # n = 3 lignes, p = 4 colonnes
np.size(R)           # 12 coefficients
R[3, 0]         # IndexError !
```

Remarque : c'est différent pour une liste de listes

```
L = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
L[1][2]                    # 7 : syntaxe des listes
L[1, 2]                    # TypeError !
L[1]                       # ligne 1
[ligne[2] for ligne in L]  # colonne 2
len(L), len(L[0])          # 3 4 : pas de np.shape
```

Au concours on manipule souvent des listes de listes.

À retenir

- le coefficient mathématique $a_{i,j}$ est `A[i-1, j-1]` ;
- `A[i]` : ligne i
- `A[:, j]` : colonne j ;
- `A[a:b, c:d]` : lignes a à $b-1$, colonnes c à $d-1$;
- `n, p = np.shape(A)` : nombre de lignes, de colonnes (aussi `A.shape`).

3. Modifier un tableau : copies et vues

```
Z = np.zeros([3, 4])
Z[0, 0] = 5            # un coefficient
Z[1, :] = 1            # toute la ligne 1
Z[:, 3] = [7, 8, 9]    # toute la colonne 3
Z[2, 0:2] = 4          # un morceau de ligne

C = Z                  # même tableau, 2 noms
C[0, 0] = 0            # Z est modifié aussi !
C = np.copy(Z)         # vraie copie indépendante

M[0], M[1] = M[1], M[0]   # NE marche PAS
```

À retenir

- `C = B` **ne copie pas** : utiliser `np.copy(B)` ;
- `M[i]` est une **vue** sur `M`, pas un objet indépendant : pour échanger deux lignes, copier l'une d'elles d'abord.

```
def echange_lignes(M, i, j):
    L = np.copy(M[i])
    M[i] = M[j]
    M[j] = L

def transvection(M, i, j, a):
    M[i] = M[i] + a * M[j]
              # Li <- Li + a Lj
```

4. Calculer : ∗ n'est pas le produit matriciel

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[0, 1], [1, 0]])
A + B, 3 * A   # comme en maths
A * B          # [[0 2] [3 0]]   coef. par coef.
np.dot(A, B)   # [[2 1] [4 3]]   produit AB
A @ B          # idem
A ** 2         # [[1 4] [9 16]]  coef. au carré
np.dot(A, A)   # [[7 10] [15 22]]  vraie A^2
np.transpose(A)    # ou A.T
np.sum(A), np.prod(A), np.max(A), np.min(A)
np.sum(A[:, 0])    # somme de la colonne 0
```

À retenir

- `A * B` et `A ** 2` agissent **coefficient par coefficient** ;
- produit matriciel : `np.dot(A, B)` (ou `A @ B`) ;
- en général $AB \neq BA$;
- puissance : partir de `np.eye(n)` ($A^0 = I_n$) et multiplier avec `np.dot`.

```
def puissance(A, n):
    P = np.eye(np.shape(A)[0])
    for k in range(n):
        P = np.dot(P, A)
    return P
```

5. Comparer des tableaux

```
A == B   # [[False False] [False False]] : un TABLEAU

def egales(A, B):
    if np.shape(A) != np.shape(B):
        return False
    n, p = np.shape(A)
    for i in range(n):
        for j in range(p):
            if A[i, j] != B[i, j]:
                return False
    return True
```

Pour aller plus loin : `np.all` et `np.any`

```
np.all(A == B)   # True si TOUS les coefs sont égaux
np.any(A == B)   # True si AU MOINS UN l'est
(A == B).all()   # idem np.all(A == B)
# attention : A et B doivent avoir le même format
```

À retenir

- `A == B` compare **coefficient par coefficient** et renvoie un tableau : on ne le met pas dans un `if` ;
- pour tester l'égalité : d'abord le **format**, puis les coefficients.

6. Des fonctions sur les matrices

```
def produit(A, B):      # (AB)_ij = somme a_ik b_kj
    n, p = np.shape(A)
    q = np.shape(B)[1]
    C = np.zeros([n, q])
    for i in range(n):
        for j in range(q):
            for k in range(p):
                C[i, j] = C[i, j] + A[i, k] * B[k, j]
    return C

def est_symetrique(A):
    return egales(A, np.transpose(A))
# aussi possible avec une double boucle : a_ij = a_ji

def est_triangulaire_sup(A):
    n = np.shape(A)[0]
    for i in range(n):
        for j in range(i):      # sous la diagonale
            if A[i, j] != 0:
                return False
    return True
```

À retenir

- $A \in \mathcal{M}_{n,p}$, $B \in \mathcal{M}_{p,q}$: $AB \in \mathcal{M}_{n,q}$, et une **somme** $\sum_k$ devient une **boucle** sur k ;
- « $\forall i, j,\ P(i,j)$ » :

```
for i in range(n):
    for j in range(p):
        if not P(i, j):
            return False
return True
```

- « $\exists i, j,\ P(i,j)$ » : on **inverse**

```
for i in range(n):
    for j in range(p):
        if P(i, j):
            return True
return False
```

7. numpy.linalg : inverse, système, rang

```
import numpy.linalg as la
M = np.array([[1, 2, 1], [2, 3, 1], [1, 1, 2]])
la.inv(M)              # inverse de M
la.matrix_rank(M)      # 3

# système MX = Y
Y = np.array([1, 1, 4])
X = np.dot(la.inv(M), Y)   # [ 1. -1.  2.]
np.dot(M, X)               # vérification : Y

J = np.ones([4, 4])    # rang 1
la.inv(J)              # LinAlgError: Singular matrix
```

À retenir

- `la.inv(M)` : inverse (si M est inversible) ;
- `la.matrix_rank(M)` : rang ;
- $(S) : MX = Y$ avec M inversible a une unique solution $X = M^{-1}Y$;

Remarque : limites de `la.inv` et `la.matrix_rank`

Les calculs se font sur des **flottants** : ils sont **approchés**.

- $N = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ n'est pas inversible, pourtant `la.inv(N)` répond souvent sans erreur (coefficients $\approx 10^{16}$) ;
- $\begin{pmatrix} 1 & 1 \\ 1 & 1+h \end{pmatrix}$ est de rang 2, mais `la.matrix_rank` trouve 1 dès $h = 10^{-15}$.

Un rang, une inversibilité : **se justifient sur papier**.

Le résumé : les instructions du formulaire

```
import numpy as np
np.array(L)          liste de listes -> matrice
np.zeros([n,p])      matrice nulle n x p
np.eye(n)            identité
np.diag(L)           matrice diagonale
np.linspace(a,b,n)   n valeurs de a à b
np.shape(M)          format (n, p)
np.size(M)           nombre de coefficients
np.copy(M)           copie indépendante
np.transpose(M)      transposée (M.T)
np.dot(M,P)          produit matriciel (M @ P)
np.sum, np.prod, np.max, np.min
```

```
import numpy.linalg as la
la.inv(M)            inverse
la.matrix_rank(M)    rang

M[i, j]      coefficient ligne i, colonne j
M[i]         ligne i      (= M[i, :])
M[:, j]      colonne j
M[a:b, c:d]  lignes a..b-1, colonnes c..d-1
```

Les trois pièges

`*` n'est pas `np.dot` · `C = B` n'est pas une copie · `A == B` donne un tableau de booléens