TP 03 – MATRICES

I Documentation sur les matrices

Pour pouvoir définir et effectuer des opérations sur les **matrices**, on travaille avec les modules `numpy` et `numpy.linalg` que l'on doit donc charger au tout début du TP.

Entrée [1]:
```
import numpy as np
import numpy.linalg as al
```

Pour définir une matrice, on utilise la fonction `array` du module `numpy`. Par exemple, pour créer la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

on procède ainsi :

Entrée [2]:
```
A = np.array([[1,2,3], [4,5,6]])  #attention aux parenthèses/crochets
```

L'attribut `shape` donne la taille d'une matrice (nombre de lignes, nombre de colonnes) sous forme d'un couple.

Entrée [3]:
```
np.shape(A)
```

Out [3]:
```
(2,3)
```

L'accès à un terme de la matrice `A` se fait à l'aide de l'opération d'indexage `A[i,j]` où `i` désigne la ligne et `j` la colonne. Attention, les indices commencent à zéro (comme pour les listes). À l'aide d'intervalles, on peut également récupérer une partie d'une matrice (ligne, colonne, sous-matrice).

`A[i,j]`	Renvoie l'élément en ligne i et colonne j de la matrice A.
`A[i,:]`	Renvoie la ligne i de la matrice A.
`A[:,j]`	Renvoie colonne j de la matrice A.

Les fonctions `zeros` et `ones` permettent de créer des matrices remplies de 0 ou de 1. La fonction `eyes` permet de créer une matrice identité. La fonction `diag` permet de créer une matrice diagonale.

`np.zeros((n,p))`	Création d'une matrice nulle de taille (n, p).
`np.ones((n,p))`	Création d'une matrice composée de 1 de taille (n, p).
`np.eye(n)`	Création de la matrice identité I_n.
`np.diag([a1,..., an])`	Création d'une matrice diagonale.

Les opérations d'ajout et de multiplication par un scalaire se font avec les opérateurs `+` et `*`. Pour effectuer un produit matriciel (lorsque cela est possible), il faut employer la fonction `dot`. La transposée s'obtient avec la fonction `transpose`. Les fonctions `matrix_power` et `inv` de la librairie `numpy.linalg` permettent de calculer respectivement des puissances et l'inverse de matrices.

`A+B`	Effectue la somme des matrices A et B.
`x*A`	Effectue le produit du scalaire x et de la matrice A.
`np.transpose(A)`	Effectue la transposée de A.
`np.dot(A,B)`	Effectue le produit matriciel AB.
`al.inv(A)`	Calcule la matrice inverse de A
`al.matrix_power(A,n)`	Calcule la puissance matricielle A^n.

II Exercices

Exercice 1 [**Produit matriciel**] On considère les matrices

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad \text{et} \qquad B = \begin{pmatrix} 0 & -1 \\ 4 & 7 \\ -8 & 2 \end{pmatrix}.$$

1. (À faire sur le **papier**.) Donner B^T et calculer I_3B, AB et BA.

On obtient

$$B^T = \begin{pmatrix} 0 & 4 & -8 \\ -1 & 7 & 2 \end{pmatrix} \qquad I_3 \times B = B$$

$$AB = \begin{pmatrix} -16 & 19 \\ -28 & 43 \end{pmatrix} \qquad BA = \begin{pmatrix} -4 & -5 & -6 \\ 32 & 43 & 54 \\ 0 & -6 & -12 \end{pmatrix}$$

2. À l'aide des commandes sur les matrices, répondre aux questions suivantes grâce à **Python**.

(a) Définir la matrice B.

```
Entrée [4]: B = np.array([[0,-1], [4,7], [-8,2]])
```

(b) Afficher la transposée de B.

```
Entrée [5]: np.transpose(B)
```

```
Out [5]: array([[ 0,  4, -8],
                [-1,  7,  2]])
```

(c) Créer la matrice identité de taille 3, on l'appelera I.

```
Entrée [6]: I = np.eye(3)
```

(d) Calculer IB avec un produit matriciel.

```
Entrée [7]: np.dot(I,B)
```

```
Out [7]: array([[ 0., -1.],
                [ 4.,  7.],
                [-8.,  2.]])
```

(e) Vérifier à l'aide de Python que l'égalité $IB = B$ est vraie. *Python doit afficher un booléen.*

```
Entrée [8]: np.dot(I,B) == B
```

```
Out [8]: array([[ True,  True],
                [ True,  True],
                [ True,  True]])
```

(f) Calculer AB et BA.

```
Entrée [9]: np.dot(A,B)
```

Out [9]:
```
array([[ -16,  19],
       [-28,  43]])
```

Entrée [10]:
```
np.dot(B,A)
```

Out [10]:
```
array([[ -4,  -5,  -6],
       [ 32,  43,  54],
       [0,  -6,  -12]])
```

Exercice 2 [**Puissances**] On considère la matrice

$$M = \begin{pmatrix} 4 & 2 & 0 \\ 1 & 0 & 1 \\ -1 & 2 & 1 \end{pmatrix}$$

1. (À faire sur le **papier**.) Calculer M^2 et M^3.

   On a
   $$M^2 = \begin{pmatrix} 18 & 8 & 2 \\ 3 & 4 & 1 \\ -3 & 0 & 3 \end{pmatrix} \quad \text{et} \quad M^3 = \begin{pmatrix} 78 & 40 & 10 \\ 15 & 8 & 5 \\ -15 & 0 & 3 \end{pmatrix}$$

2. Calculer M^2 et M^3 grâce à la commande `np.dot`.

Entrée [11]:
```
M = np.array([[4,2,0], [1,0,1], [-1,2,1]])
np.dot(M,M)
```

Out [11]:
```
array([[18,  8,  2],
       [ 3,  4,  1],
       [-3,  0,  3]])
```

Entrée [12]:
```
np.dot(M,np.dot(M,M))
```

Out [12]:
```
array([[ 78,  40,  10],
       [ 15,  8,  5],
       [-15,  0,  3]])
```

3. En utilisant la bibliothèque numpy.linalg (voir documentation), recalculer M^3.

Entrée [13]:
```
import numpy.linalg as al
al.matrix_power(M,3)
```

Out [13]:
```
array([[ 78,  40,  10],
       [ 15,  8,  5],
       [-15,  0,  3]])
```

4. Écrire une fonction `puisance` qui prend en argument une matrice `A` et un entier `n` et qui renvoie A^n. *On utilisera uniquement la commande* `np.dot`.

Entrée [14]:
```
def puissance(A,n):
    P=A
    for k in range(n-1):
        P = np.dot(A,P)
    return P
```

5. Calculer et afficher M^{10} à l'aide de la fonction précédente. *Vérifier le résultat à l'aide de la commande* `al.matrix_power`.

Entrée [15]:
```
puissance(M,10)
```

Out [15]:
```
array([[2387718, 1263344, 374450],
       [ 444447, 235480, 69997],
       [-444447, -234456, -68973]])
```

Entrée [16]:
```
al.matrix_power(M,10) == puissance(M,10)
```

Out [16]:
```
array([[ True, True, True],
       [ True, True, True],
       [ True, True, True]])
```

Exercice 3 **[Inverse]**

1. Donner l'inverse de la matrice M (on admet qu'elle est inversible).

Entrée [17]:
```
al.inv(M)
```

Out [17]:
```
array([[ 0.16666667, 0.16666667, -0.16666667],
       [ 0.16666667, -0.33333333, 0.33333333],
       [-0.16666667, 0.83333333, 0.16666667]])
```

2. Calculer grâce à Python le produit $M^{-1}M$. Que doit-on obtenir ?

Entrée [18]:
```
np.dot(al.inv(M),M)
```

Out [18]:
```
array([[ 1.00000000e+00, 0.00000000e+00, 2.77555756e-17],
       [ 5.55111512e-17, 1.00000000e+00, -5.55111512e-17],
       [-5.55111512e-17, 0.00000000e+00, 1.00000000e+00]])
```

3. Résoudre à l'aide de la première question, le système linéaire suivant.

$$(S)\begin{cases} 4x + 2y = 1 \\ x + z = 2 \\ -x + 2y + z = 3 \end{cases}$$

Entrée [19]:
```
B = np.array([[1],[2],[3]])
np.dot(al.inv(M),B)
```

Out [19]:
```
array([[0. ],
       [0.5],
       [2. ]])
```

Exercice 4 **[Définir une matrice grâce à des `for`]**

1. Afficher le coefficient en première ligne et première colonne de la matrice M.

Entrée [20]:
```
M[0,0]
```

Out [20]:
```
4
```

2. Afficher la deuxième ligne de M.

Entrée [21]:
```
M[1,:]
```

Out [21]:
```
array([1, 0, 1])
```

3. Afficher la dernière colonne de M.

Entrée [22]:
```
M[:,-1]
```

Out [22]:
```
array([0, 1, 1])
```

4. On souhaite définir la matrice $C = (i-j)_{0\leq i\leq 2, 0\leq j\leq 3}$ (on utilise ici la numérotation Python qui commence à 0).Compléter le programme.

Entrée [23]:
```
# On définit C comme la matrice nulle de taille correcte
C = np.zeros((3,4))

# On va ensuite modifier chacun des cij avec deux boucles for
for i in range(3):
    for j in range(4):
        C[i,j] = i-j

print(C)
```

Out [23]:
```
[[ 0. -1. -2. -3.]
 [ 1. 0. -1. -2.]
 [ 2. 1. 0. -1.]]
```

5. Définir la matrice $D = (ij)_{1\leq i\leq 4, 1\leq j\leq 3}$ (attention, ici la numérotation commence à 1, comme en maths).

Entrée [24]:
```
# On définit D comme la matrice nulle de taille correcte
D = np.zeros((4,3))

# On va ensuite modifier chacun des dij avec deux boucles for
for i in range(4):
    for j in range(3):
        D[i,j] = (i+1)*(j+1) #attention au décalage

print(D)
```

Out [24]:
```
[[ 1. 2. 3.]
 [ 2. 4. 6.]
 [ 3. 6. 9.]
 [ 4. 8. 12.]]
```

Exercice 5 *Les questions de cet exercice sont indépendantes.*

1. Sans rentrer les coefficients un à un, définir en Python la matrices suivante

$$\begin{pmatrix} 5 & 3 & 3 \\ 3 & 5 & 3 \\ 3 & 3 & 5 \end{pmatrix}$$

Entrée [25]:
```
A = 3*np.ones((3,3))+2*np.eye(3)
print(A)
```

Out [25]:
```
[[5. 3. 3.]
 [3. 5. 3.]
 [3. 3. 5.]]
```

2. Écrire une fonction, appelée `trace`, qui prend en argument une matrice carrée `A` et qui renvoie la trace de la matrice, c'est-à-dire la somme des ses coefficients diagonaux. *On vérifiera que* `trace(M)` *(où M est la matrice définie à l'Exercice 2) renvoie* `5`*.)*

Entrée [26]:
```
def trace(A):
    (n,p)=np.shape(A)
    trace=0
    for i in range(n):
        trace = trace + A[i,i]
    return(trace)
```

Entrée [27]:
```
trace(M)
```

Out [27]:
```
5
```

3. Écrire une fonction, appelée `somme`, qui prend en argument une matrice `A` et qui renvoie la somme de tous ces coefficients. *On vérifiera que* `somme(M)` *(où M est la matrice définie à l'Exercice 2) renvoie* `10`*.)*

Entrée [28]:
```
def somme(A):
    (n,p) = np.shape(A)
    S=0
    for i in range(n):
        for j in range(p):
            S = S + A[i,j]
    return S
```

Entrée [29]:
```
somme(M)
```

Out [29]:
```
10
```

4. On dit qu'une matrice est stochastique si tous ses coefficients sont positifs, et la somme de chacune de ses lignes vaut 1. Écrire une fonction, appelée `stoch`, qui prend en entrée une matrice `A` et renvoie `True` si la matrice est stochastique, et `False` sinon. *On vérifiera que* `stoch(M)` *(où M est la matrice définie à l'Exercice 2) renvoie* `False` *et que* `stoch(np.array([[1,0],[0,1]]))` *renvoie* `True`*.)*

Entrée [30]:
```
def stoch(A):
    (n,p)=np.shape(A)
    #On passe en revue toutes les lignes de A
    for i in range(n):
        #On calcule la somme des éléments de la ligne i
        S = 0
        for j in range(p):
            S = S + A[i,j]
        #On teste si la somme de la ligne 1 vaut 1
        if S !=1 :
            return False
    return True
```

Entrée [31]:
```
stoch(M)
```

Out [31]:
```
False
```

Entrée [32]:
```
B = np.array([[1,0],[0,1]])
stoch(B)
```

Out [32]:
```
True
```

5. On dit qu'une matrice $A \in \mathcal{M}_n(\mathbb{R})$ est à diagonale strictement dominante si

$$\forall i \in \{1, \dots, n\}, \qquad |a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$$

Écrire une fonction, appelée `diagdom`, qui prend en entrée une matrice `A` et renvoie `True` si la matrice est à diagonale strictement dominante, et `False` sinon. *On vérifiera que* `diagdom(M)` *renvoie* `False` *et que* `diagdom(np.array([[-4,2,1],[1,6,2],[1,-2,5]]))` *renvoie* `True`.*)*

Entrée [33]:
```
def diagdom(A):
    (n,p)=np.shape(A)
    #On passe en revue toutes les lignes de A
    for i in range(n):
    #On calcule la somme de la v.a. des éléments de la ligne i
    #sauf le terme diag
        S = 0
        for j in range(p):
            if j !=i:
                S = S + np.abs(A[i,j])
        #On teste la condition
        if np.abs(A[i,i]) <= S:
            return False
    return(True)
```

Entrée [34]:
```
diagdom(M)
```

Out [34]:
```
False
```

Entrée [35]:
```
B = np.array([[-4,2,1],[1,6,2],[1,-2,5]])
diagdom(B)
```

Out [35]:
```
True
```