Chapitre # (ALGO) 7 Tableaux Numpy. Application au calcul matriciel

Résumé & Plan

Nous revoyons dans ce chapitre certains éléments sur la manipulation de tableaux numpy (déjà un peu abordés pour tracer des courbes), puis leur utilisation pour manipuler des matrices.

- Les chapitres d'Informatique sont composés de cours et d'exercices intégrés. Le cours sera projeté au tableau.
- Il n'est pas attendu que toute la classe aborde tous les exercices. Traitez donc en priorité les exercices présents dans la liste donnée à chaque début de séance.
- Exercices 🌓 / **Pour aller plus loin** : exercices plus difficiles, ou plus techniques. À ne regarder que si les autres sont bien compris.

1. DÉCOUVERTE DU TYPE NDARRAY (« ARRAY » EN ABRÉGÉ)

Tester dans le Shell et compléter les items associés dans la fiche de cours :

```
>>> import numpy as np

>>> L=[1,4,3,2]
>>> type(L)
>>> T=np.array(L)
>>> T
>>> print(T)
>>> type(T)
>>> T[0]
>>> T[-1]
>>> T[1:3]
>>> T[1:]
>>> T[0]=-1
>>> print(T)

>>> U=np.array([[1,-5,0,-1],[2,4,5,10],[-3,7,9,-4]])
>>> U
>>> print(U)
>>> np.shape(U)
>>> a,b = np.shape(U)
>>> a  # le nombre de lignes
>>> b  # le nombre de colonnes
>>> U[0] # Première ligne (décalage d'indices en Python !)
>>> U[1] # Seconde ligne
>>> U[2] # Troisième colonne
>>> len(U)    # le nombre de lignes
>>> len(U[0]) # le nombre de colonnes
>>> np.size(U)
>>> U[0,0]
>>> U[1,3]
>>> U[0:2,3]
>>> U[1,1:3]
>>> U[:,2]

>>> U[1,3]=20
>>> print(U)

>>> L=[1,4,3,2]
>>> D = np.diag(L)
>>> print(D)
>>> M1 = np.zeros((3,2))
>>> print(M1)
>>> M2 = np.ones((2,4))
```


Lycée Camille GUÉRIN – Poitiers

```
>>> print(M2)
>>> M3 = np.eye(7)
>>> print(M3)



>>> import numpy.random as npr
>>> A1 = npr.random((4,2))
>>> print(A1)
>>> A2 = npr.randint(3,8,(5,6))
>>> print(A2)

>>> B1 = np.arange(10,40,5)
>>> print(B1)
>>> B2 = np.arange(10,100,20)
>>> print(B2)
>>> X = np.linspace(0,2*np.pi,101)
>>> Y = np.sin(X)
>>> plt.plot(X,Y)
>>> plt.show()

>>> A = npr.randint(1,5,(2,3))
>>> B = npr.randint(1,5,(2,3))
>>> print(A)
>>> print(B)
>>> A+B
>>> A+3
>>> A*B
>>> 5*A
>>> A**2
>>> A**B
>>> A/B
>>> A==B
>>> A>B
>>> A<=5
>>> np.dot(A,B)
>>> C = npr.randint(1,5,(3,5))
>>> print(C)
>>> np.dot(A,C)

>>> import numpy.linalg as la
```

```
>>> A = np.diag([1,2,4])
>>> B = np.diag([1,2,0])
>>> Am1 = la.inv(A)
>>> print(Am1)
>>> la.matrix-rank(A)     # (tiret du 8) - la notion de rang de \
↪ matrice sera vue plus tard
```

2. PRÉSENTATION SUCCINTE DE LA BIBLIOTHÈQE NUMPY

La librairie `numpy` est consacrée entièrement au calcul numérique en Python. Elle comprend les principales fonctions mathématiques (à l'instar du module `math`).

Elle utilise essentiellement des variables de type `ndarray` (en abrégé `array`), que l'on peut voir comme des tableaux à plusieurs dimensions. Les calculs avec numpy sont particulièrement optimisés car les array sont homogènes (ils ne contiennent que des valeurs d'un même type) et de taille fixée à la création. [1] Traditionnellement on charge la librairie `numpy` avec la ligne :

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

> **Remarque 1** On pourrait utiliser aussi des listes de listes. L'avantage du type `array` est qu'on accède à toute une batterie de fonctions matricielles déjà définies (produit de matrices, recherche d'inverse, *etc.*).

2.1. Généralités

DÉFINIR UN TABLEAU, TAILLE. On définit un tableau avec la fonction `np.array`. Regardons ensuite comment obtenir les dimensions du tableau.

```
>>> A = np.array([[8, 3, 2] , [5, 1, 6]])
>>> np.shape(A) # nb lignes / nb colonnes. On peut aussi \
↪ utiliser np.shape(A)
(2, 3)
>>> A.shape # On peut aussi utiliser  A.shape
(2, 3)
>>> A.dtype # le type des données contenues dans le tableau
dtype('int64')
```

1. C'est donc une différence notable avec les listes de listes

```
>>> B = np.array([[8, 3, 2]])
>>> B.shape # nb lignes / nb colonnes
(1, 3)
```

Attention Récupérer le nombre de lignes et de colonnes

Pour récupérer le format d'un tableau, on peut aussi utiliser l'instruction suivante :

```
>>> A = np.array([[8, 3, 2]])
>>> n = len(A) # nombre de lignes
>>> n
1
>>> p = len(A[0]) # nombre de colonnes
>>> p
3
>>> A = np.array([[8, 3, 2] , [5, 1, 6]])
>>> n = len(A)
>>> n
2
>>> p = len(A[0])
>>> p
3
```

qui fonctionne donc aussi bien sur un vecteur ligne qu'une matrice plus classique (il peut parfois y avoir des soucis avec les vecteurs lignes).

Attention aux indices

Pour `n` lignes et `p` colonnes, la numérotation Python s'effectue entre `0` et `n-1` pour les lignes, et `0` et `p-1` pour les colonnes. Il y a donc un décalage avec l'indice des Mathématiques, source d'erreurs au début.

```
>>> A[1][2] # c'est bon
6
>>> A[1][3] # là ça ne va plus
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 0 with size 3
```

On ne peut pas non plus modifier un coefficient en le remplaçant par une valeur d'un autre type. Par exemple,

```
>>> A[1][1] = [-1, -1]
TypeError: int() argument must be a string, a bytes-like \
↪ object or a real number, not 'list'

The above exception was the direct cause of the following \
↪ exception:

Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: setting an array element with a sequence.
```

Ainsi, les seules modifications autorisées sont celles de type initial que l'on obtient avec la méthode `dtype` :

```
>>> A.dtype # Donc ici, uniquement par un entier.
dtype('int64')
```

Attention Toutes les lignes ont même nombre d'éléments

Par exemple, la définition suivante échoue :

```
>>> M = np.array([[1, 2], [2]])
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: setting an array element with a sequence. The \
↪ requested array has an inhomogeneous shape after 1 \
↪ dimensions. The detected shape was (2,) + inhomogeneous part.
```

On ne peut donc pas convertir n'importe quelle liste de listes en tableau : il faut que chaque ligne ait même nombre d'éléments.

Pour créer une matrice, on peut définir une liste de listes puis la convertir en tableau avec `np.array`. Mais on utilise généralement des fonctions permettant de créer facilement les matrices usuelles.

CONSTRUCTEURS DE TABLEAUX

Operations	Commande	Commentaire
Création	`A = np.array(...)`	On indique une liste de listes en argument
Matrice nulle	`A = np.zeros((n, p))`	Un tuple est demandé en argument, donc deux parenthèses
Matrice qui ne contient que des 1	`A = np.ones((n, p))`	Un tuple est demandé en argument, donc deux parenthèses
Matrice identité	`A = np.identity(n)` ou `A = np.eye(n)`†	

CONSTRUCTEURS DE TABLEAUX

Matrice diagonale	`A = np.diag(L)`	La liste `L` contient la diagonale
Subdivision de pas `h` de $[a,b]$	`np.arange(a, b, h)`	Analogue de `list(range(a, b, h))`
Subdivision à `n` points de $[a,b]$	`np.linspace(a, b, n)`	On s'en est servi pour tracer des suites, fonctions, *etc.*
Copie	`B = A.copy()`	Important pour pouvoir modifier `B` sans modifier `A`, comme pour les listes
Coefficients	`A[i, j]` ou `A[i][j]`	Terme `i,j` du tableau
Ligne `i`	`A[i]`	
Colonne `j`	`A[:, j]`	« `:` » signifie en *slicing* « on prend tout »

† `eye` comme identity en anglais.

Voici quelques exemples.

Attention **Copies**

Comme pour les listes, attention aux copies. En cas de copie en dure souhaitée, on utilisera la syntaxe `N = np.copy(M)` qui réalise une copie indépendante de `M` dans `N`.

Exemple 1

```
>>> np.identity(5)
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
>>> np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

OPÉRATIONS. On peut effectuer un grand nombre d'opérations directement sur les array. On peut tout d'abord y appliquer des fonctions coefficient par coefficient.

Exemple 2 Par exemple,

```
>>> A = np.array([[1, 2], [3, 4]])
>>> A**2
array([[ 1,  4],
       [ 9, 16]])
```

Ainsi `A**2` va élever au carré chaque coefficient de `A`. Ce n'est donc pas $A \times A$. Remarquons qu'en utilisant les fonctions de `numpy`, on peut appliquer une fonction coefficient par coefficient.

```
>>> np.log(A)
array([[0.        , 0.69314718],
       [1.09861229, 1.38629436]])
```

La plupart des fonctions mathématiques sont définies par `numpy`. La particularité de ces fonctions est qu'elles peuvent s'appliquer à un réel (comme avec le module `math`) mais aussi sur un tableau (voir l'exemple précédent avec la fonction ln).

OPÉRATIONS

Operations	Commande
Somme de deux matrices compatibles	`A + B`
Produits de deux matrices compatibles	`A @ B` **OU** `np.dot(A,B)`
Transposée	`np.transpose(A)`
Somme de tous les éléments	`np.sum(A)`
Produits de tous les éléments	`np.prod(A)`

Remarque 2 Pensez au symbole `@` pour des expressions matricielles compliquées (bien plus pratique que `np.dot()`).

PARCOURIR UN TABLEAU. Puisqu'on s'y repère comme dans une liste de listes, les parcours se font de la même manière. On imbrique donc deux boucles : l'une dont la variable parcourt les indices des lignes de la matrice et l'autre dont la variable parcourt les indices de ses colonnes. Supposons que `n, p` désignent le nombre de lignes et colonnes d'un tableau A.

Parcours à l'aide des indices

```
n, p = A.shape
for i in range(n):
    for j in range(p):
        ...
```

A l'aide des valeurs

```
for L in A: # L est une ligne
    for x in L:
        ...
```

Méthode **Créer une matrice**

Plusieurs options s'offrent à vous.

- Si la matrice est de petite taille, on écrit directement les coefficients.
- Si la matrice est de grande taille (typiquement dépendant d'un certain entier `n`), on peut :
  - ◇ soit utiliser des commandes existantes si la matrice est proche d'une matrice usuelle. Par exemple, `np.eye`, `np.zeros`, `np.ones`, *etc.*
  - ◇ Soit partir d'une matrice nulle initialisée à la bonne taille (avec `np.zeros`), puis la compléter des bons coefficients à l'aide d'une boucle `for`.

Exercice 1 | **Matrices à créer** Solution On considère les matrices $A, B \in \mathcal{M}_n(\mathbb{R})$ définies par :

1. $\forall (i,j) \in \{1,\dots n\}^2, \quad A_{ij} = ij,$
2. $\forall (i,j) \in \{1,\dots n\}^2, \quad B_{ij} = i^2 - j^2$ si $\quad i \leq j, \quad B_{ij} = 0$ sinon.

Créer ces deux matrices en Python dans deux fonctions d'en-têtes `creer_matrice_A(n)`, `creer_matrice_B(n)`. *Indication : On pourra partir de la matrice nulle, que l'on complètera à l'aide de deux boucles for, en prenant garde aux décalages des indices*

Exercice 2 | **Somme** Solution Écrire une fonction d'en-tête `somme(M)` qui retourne la somme des éléments de `M`. *On s'interdira bien entendu d'utiliser `np.sum`.*

Exercice 3 | **Matrice des entiers consécutifs** Solution

1. Créer fonction d'en-tête `creer_mat_entiers(n)` qui retourne une matrice de format $n \times n$ contenant tous les entiers entre 1 et n^2 (de gauche à droite et haut en bas). Par exemple, `creer_mat_entiers(3)` retournera `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.
2. Que vaut $\sum_{k=1}^{n^2} k$? Le retrouver à l'aide de la question précédente et de l'exercice précédent. *On exécutera les fonctions sur plusieurs valeurs de n*

2.2. Différences entre tableaux numpy et listes

Même si les objets `ndarray` (listes de listes) et `list` (listes) semblent être très proches, il y a néanmoins quelques différences à bien garder en tête.

- La méthode `append` n'existe pas sur les tableaux, même unidimensionnels. Ainsi, un tableau a une certaine taille lors de sa création et conservera sa taille tant qu'il existe. Ce qui n'empêche pas de construire une liste de liste avec `append`, puis de convertir le tout en tableau avec `np.array()`.
- Une liste peut contenir des objets de natures différentes, alors que tous les éléments d'un tableau sont de même type. Type là encore défini lors de sa création et fixé jusqu'à la fin. Ainsi, la conversion en array d'une liste de listes ne respectant pas cela échouera.

3. APPLICATION AU CALCUL MATRICIEL

Nous avons vu pour l'instant comment créer un tableau, le parcourir, modifier ses éléments *etc.* et qu'un tableau permettait de coder une matrice en Mathématiques. L'objectif de cette section est de traiter des problèmes du cours de calcul matriciel à l'aide du module `numpy`.

Cadre

Dans toute cette section, l'ensemble $\mathbb{K}$ désignera $\mathbb{R}$ ou $\mathbb{C}$, et n, p désignent deux entiers supérieurs ou égaux à 1.

3.1. Calcul de puissances

Méthode **Calcul des puissances d'une matrice avec Python**

Soit `M` un tableau carré correspondant à une matrice M carrée. Il n'y a pas de fonction toute faite dans numpy pour calculer M^n. Rappelons également que `M**n` élève les coefficients de M à la puissance n mais n'effectue pas le produit matriciel. On procède comme suit :

- on initialise un tableau `P` à l'identité.
- On effectue n fois l'affectation `P = P @ M`.
- On retourne `P`.

Exercice 4 | **Puissances et suites** Solution On considère trois suites $(x_n),(y_n),(z_n)$ vérifiant :

$$\forall n\in\mathbb{N},\quad \begin{cases} x_{n+1} &= -x_n - 3y_n + 3z_n \\ y_{n+1} &= 3x_n - 7y_n + 3z_n \\ z_{n+1} &= 6x_n - 6y_n + 2z_n \end{cases} \qquad x_0 = y_0 = 1, \quad z_0 = 2.$$

On note par ailleurs : $\forall n\in\mathbb{N},\quad \mathrm{X}_n = \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix}$.

1. Donner une matrice A telle que : $\forall n\in\mathbb{N},\quad \mathrm{X}_{n+1} = \mathrm{AX}_n$. Créer cette matrice dans Python sous forme de tableau numpy. On peut montrer par récurrence que : $\forall n\in\mathbb{N},\quad \mathrm{X}_n = \mathrm{A}^n\mathrm{X}_0$.
2. Créer une fonction d'en-tête `puissance_mat(A, k)` qui retourne le tableau correspondant à A^k pour un entier k. En déduire une fonction `val_xyz(n)` qui retourne x_n, y_n, z_n étant donné un entier `n`. Conjecturer leur nature en exécutant pour plusieurs valeurs jusqu'à $n = 50$.
3. Proposer une version récursive `puissance_mat_rec` de la fonction `puissance_mat`.

Exercice 5 | **Indice de nilpotence** Solution En cas d'existence, on dit qu'une matrice $\mathrm{A}\in\mathfrak{M}_n(\mathbb{K})$ est *nilpotente* s'il existe $p\in\mathbb{N}$ tel que $\mathrm{A}^p = 0_n$. On appelle *indice de nilpotence* le plus petit entier p vérifiant cette propriété.

1. Soit $\mathrm{M} = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix}$. Montrer que M est nilpotente, préciser son indice.
2. Écrire une fonction d'en-tête `indice_nilpo(M)` prenant en argument une matrice carrée `M` et renvoyant l'indice en question. *On pourra constater que la commande* `np.all(P == np.zeros(P.shape))` *teste si une matrice* `P` *est nulle.*

3.2. Propriétés

Exercice 6 | **Triangulaire ou pas ?** Solution On rappelle qu'une matrice carrée $\mathrm{M}\in\mathfrak{M}_{n,n}(\mathbb{K})$ est triangulaire supérieure si tous ses coefficients strictement en-dessous de la diagonale sont nuls, *i.e.* :

$$\forall (i,j)\in[\![1\,,\,n]\!]^2,\quad j<i \implies m_{i,j} = 0.$$

1. Écrire une fonction d'en-tête `est_triangulaire_sup(M)` qui renvoient `True` si la matrice `M` est triangulaire supérieure, `False` sinon. Même chose pour tester si une matrice est triangulaire inférieure, en utilisant la fonction `est_triangulaire_sup`.
2. En déduire une fonction `est_diagonale(M)` qui renvoient `True` si la matrice `M` est diagonale, `False` sinon.

Exercice 7 | **Matrices stochastiques** Solution On dit qu'une matrice $\mathrm{M}\in\mathfrak{M}_n(\mathbb{R})$ est *stochastique* si la somme sur chaque ligne vaut 1 et chaque coefficient est entre 0 et 1, *i.e.* :

$$\forall i\in[\![1\,,\,n]\!],\quad \sum_{j=1}^{n} m_{i,j} = 1,\quad \forall (i,j)\in[\![1\,,\,n]\!]^2,\quad m_{i,j}\in[0,1].$$

Ce type de matrice apparait souvent en probabilités, ce qui explique l'hypothèse sur les coefficients dans $[0,1]$.

1. Écrire une fonction d'en-tête `est_stochastique(M)`, qui étant donnée une matrice carrée retourne `True` si elle est stochastique, et `False` dans le cas contraire.
2. Écrire une fonction d'en-tête `est_stochastique_prod(M, N)`, qui étant données deux matrices carrées (de formats compatibles) retourne `True` si MN est stochastique, et `False` dans le cas contraire. Tester sur plusieurs couples de matrices stochastiques. Que conjecturer ?
3. Une matrice stochastique est dite *bistochastique* si en plus la somme sur chaque colonne vaut 1. En utilisant la fonction `stochastique`, écrire une fonction d'en-tête `est_bistochastique(M)` qui retourne `True` si `M` est bistochastique.

4. SOLUTIONS DES EXERCICES

Solution (exercice 1) Énoncé Les matrices ont des grands formats, donc on part d'une matrice nulle que l'on complète correctement. Attention au décalage entre les indices Maths / Python.

```
def creer_matrice_A(n):
    A = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            A[i, j] = (i+1)*(j+1)
    return A

def creer_matrice_B(n):
    B = np.zeros((n, n))
    for i in range(n):
        for j in range(i, n):
            # j >= i uniquement
            B[i, j] = (i+1)**2-(j+1)**2
    return B
>>> creer_matrice_A(4)
array([[ 1.,  2.,  3.,  4.],
       [ 2.,  4.,  6.,  8.],
       [ 3.,  6.,  9., 12.],
       [ 4.,  8., 12., 16.]])
>>> creer_matrice_B(4)
array([[  0.,  -3.,  -8., -15.],
       [  0.,   0.,  -5., -12.],
       [  0.,   0.,   0.,  -7.],
       [  0.,   0.,   0.,   0.]])
```

Solution (exercice 2) Énoncé On propose deux solutions (l'une à l'aide des indices, l'autre sans indices).

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

```
def somme(M):
    S = 0
    for L in M:

# L est une ligne de M
        for x in L:
            S += x
    return S
```

```
>>> A = np.array([[1, 2], [3, 4]])
>>> somme(A)
10
```

Solution (exercice 3) Énoncé

```
def creer_mat_entiers(n):
    M = np.zeros((n, n))
    m = 1
    for i in range(n):
        for j in range(n):
            M[i][j] = m
            m += 1
    return M
>>> creer_mat_entiers(3)
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

D'après le cours de Maths, $\sum_{k=1}^{n^2} k = \frac{n^2(n^2+1)}{2}$. Donc par exemple 10 pour $n = 2$, et 45 pour $n = 3$.

```
>>> somme(creer_mat_entiers(2))
10.0
>>> somme(creer_mat_entiers(3))
45.0
```

Solution (exercice 4) Énoncé

1. La matrice $A = \begin{pmatrix} -1 & -3 & 3 \\ 3 & -7 & 3 \\ 6 & -6 & 2 \end{pmatrix}$ convient. On la code en python dans la question suivante.
2. Pour les besoin de l'exercice 4 qui suit, on code une fonction

`puissance_mat(A, k)` qui calcule A^k pour une matrice A quelconque (pas nécessairement pour seulement la matrice A de cet exercice).

```
A = np.array([[-1, -3, 3], [3, -7, 3], [6, -6, 2]])

def puissance_mat(A, k):
    n, p = A.shape
    P = np.eye(n)
    for _ in range(k): # on répète k fois :
        P = P @ A
    return P
>>> puissance_mat(A, 3)
array([[ -28.,  -36.,   36.],
       [  36., -100.,   36.],
       [  72.,  -72.,    8.]])
```

```
def val_xyz(n):
    X0 = np.transpose([[1,1,2]])
    return puissance_mat(A,n) @ X0
>>> val_xyz(1)
array([[2.],
       [2.],
       [4.]])
>>> val_xyz(2)
array([[4.],
       [4.],
       [8.]])
>>> val_xyz(3)
array([[ 8.],
       [ 8.],
       [16.]])
>>> val_xyz(7)
array([[128.],
       [128.],
       [256.]])
>>> val_xyz(10)
array([[1024.],
       [1024.],
       [2048.]])
>>> val_xyz(50)
array([[1.12589991e+15],
       [1.12589991e+15],
       [2.25179981e+15]])
```

Les suites semblent diverger vers $+\infty$.

3. La version récursive est basée sur la relation suivante :

$$\forall k \in \mathbb{N}, \quad A^{k+1} = A \times A^k.$$

```
def puissance_mat_rec(A, k):
    n, p = A.shape
    if k == 0:
        return np.eye(n)
    else:
        return A @ puissance_mat_rec(A, k-1)
>>> puissance_mat(A, 3)
array([[ -28.,  -36.,   36.],
       [  36., -100.,   36.],
       [  72.,  -72.,    8.]])
>>> puissance_mat_rec(A, 3)
array([[ -28.,  -36.,   36.],
       [  36., -100.,   36.],
       [  72.,  -72.,    8.]])
```

Solution (exercice 5) Énoncé Un calcul simple montre que $M^3 = 0_{3,3}$ alors que $M^2 \neq 0_{3,3}$, elle est donc nilpotente d'ordre 3.

```
def indice_nilpo(M):
    p = 1
    while np.all(M == np.zeros(M.shape)) == False:
        M = M @ M
        p = p + 1
    return p
>>> M = np.array([[0, 1, 2], [0, 0, 3], [0, 0, 0]])
>>> indice_nilpo(M)
3
```

Solution (exercice 6) Énoncé L'idée est de parcourir la matrice et de renvoyer `False` dès qu'on trouve un coefficient non nul strictement en-dessous de la diagonale.

```
def est_triangulaire_sup(M):
    n = M.shape[0] # matrice supposée carrée donnée en entrée
    for i in range(n):
```

```
        for j in range(i):
            if M[i, j] != 0:
                return False
    return True

def est_triangulaire_inf(M):
    return est_triangulaire_sup(np.transpose(M))

def est_diagonale(M):
    return est_triangulaire_sup(M) and est_triangulaire_inf(M)
>>> A = np.array([[1, 1], [0, 1]])
>>> est_triangulaire_sup(A)
True
>>> est_triangulaire_inf(A)
False
>>> est_diagonale(A)
False
>>> B = np.eye(3)
>>> B
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> est_diagonale(B)
True
>>> est_triangulaire_sup(B)
True
>>> est_triangulaire_inf(B)
True
```

Solution (exercice 7) Énoncé

1. L'idée est la suivante : on parcourt la somme des coefficients de chaque ligne et on retourne `False` dès que l'une des sommes est différente de 1, ou qu'un des coefficients n'est pas dans $[0,1]$. Sinon, on retourne `True` et la matrice sera bien stochastique.

```
def est_stochastique(M):
    n, p = M.shape
    for i in range(n):
        somme = 0
        for j in range(p):
            somme += M[i, j]
            if not 0 <= M[i, j] <= 1:
                return False
        if somme != 1:
            return False
    return True
>>> M = np.array([[1/2, 1/2], [0.1, 0.9]])
>>> est_stochastique(M)
True
>>> M = np.array([[0, 1/2], [0.1, 0.9]])
>>> est_stochastique(M)
False
```

2.

```
def est_stochastique_prod(M, N):
    P = M @ N
    return est_stochastique(P)
>>> M = np.array([[1/2, 1/2], [0.1, 0.9]])
>>> N = np.array([[0, 1], [1, 0]])
>>> est_stochastique_prod(M, N) # c'est gagné
True
>>> M = np.array([[1/2, 1/2], [0.1, 0.9]])
>>> N = np.array([[1, 0], [0.1, 0.9]])
>>> est_stochastique_prod(M, N) # encore gagné
True
```

On conjecture raisonnablement que le produit de deux matrices stochastiques est encore stochastique. Pour savoir si une matrice stochastique est bistochastique, il suffit de regarder si la transposée est stochastique.

```
def est_bistochastique(M):
    return est_stochastique(M) and \
    ↪ est_stochastique(np.transpose(M))
>>> M = np.array([[0, 1], [1, 0]])
>>> est_bistochastique(M)
True
>>> M = np.array([[0, 1/2], [0.1, 0.9]])
>>> est_bistochastique(M)
False
```