

Matrices et tableaux

● Méthode

Exercice flash (10 minutes, sur papier). Écrire une fonction `indice_plus_proche(L: list, v: float)` -> `int` qui renvoie l'indice de l'élément de la liste non vide `L` le plus proche de `v`. On pourra utiliser `abs`. Exemple : `indice_plus_proche([2.0, 7.5, 5.2, 9.0], 6)` renvoie 2.

Corrigé.

```
def indice_plus_proche(L: list, v: float) -> int:
    '''L non vide. Renvoie l'indice du premier élément de L
    le plus proche de v.'''
    i_min = 0
    for i in range(1, len(L)):
        if abs(L[i] - v) < abs(L[i_min] - v):
            i_min = i
    return i_min
```

● Méthode

Pour chaque fonction : **exemple à la main, entrée/sortie, schéma, code puis trace**. Dessiner la matrice de l'exemple sous forme de tableau, avec les numéros de lignes et de colonnes. Les fonctions `sum`, `max`, `min` et `deepcopy` ainsi que les bibliothèques (`numpy...`) sont interdites.

● Exercice 1

Échauffement. Dans tout l'exercice, `M` est une liste de listes non vides, toutes de même longueur. Pour chaque fonction, écrire d'abord un exemple d'appel et le résultat attendu.

1. `somme_colonne(M: list, j: int)` -> `float` renvoie la somme des coefficients de la colonne `j`.
2. `moyennes_lignes(M: list)` -> `list` renvoie la liste des moyennes de chaque ligne.
3. `diagonale(M: list)` -> `list` renvoie la liste des coefficients diagonaux de la matrice carrée `M`.
4. `zeros(p: int, q: int)` -> `list` renvoie une matrice de `p` lignes et `q` colonnes remplie de zéros, dont les lignes sont des listes distinctes.
5. `transposee(M: list)` -> `list` renvoie la matrice dont le coefficient `[i][j]` est `M[j][i]`. Exemple : la transposée de `[[1, 2, 3], [4, 5, 6]]` est `[[1, 4], [2, 5], [3, 6]]`.

Corrigé.

```
def somme_colonne(M: list, j: int) -> float:
    '''Renvoie la somme des coefficients de la colonne j de M.'''
    s = 0
    for i in range(len(M)):
        s = s + M[i][j]
    return s

def moyennes_lignes(M: list) -> list:
    '''Renvoie la liste des moyennes des lignes de M.'''
    res = []
    for ligne in M:
        s = 0
        for x in ligne:
            s = s + x
        res.append(s / len(ligne))
    return res

def diagonale(M: list) -> list:
```

```

'''M carrée. Renvoie la liste des M[i][i].'''
res = []
for i in range(len(M)):
    res.append(M[i][i])
return res

def zeros(p: int, q: int) -> list:
    '''Renvoie une matrice p x q de zéros aux lignes distinctes.'''
    res = []
    for i in range(p):
        ligne = []
        for j in range(q):
            ligne.append(0)
        res.append(ligne)
    return res

def transposee(M: list) -> list:
    '''Renvoie la transposée de M (q lignes, p colonnes).'''
    p = len(M)
    q = len(M[0])
    res = []
    for i in range(q):          # la transposée a q lignes
        ligne = []
        for j in range(p):
            ligne.append(M[j][i])
        res.append(ligne)
    return res

```

● Exercice 2

Trouver l'erreur. Pour chaque fragment, prévoir ce qui se passe sur l'exemple, expliquer l'erreur et corriger.

```

# (a) matrice 2 x 3 de zéros
T = [[0] * 3] * 2
T[0][1] = 5
print(T)

# (b) somme de chaque colonne
# exemple : M = [[1, 2, 3], [4, 5, 6]]
def sommes_colonnes(M):
    res = []
    for j in range(len(M)):
        s = 0
        for i in range(len(M[0])):
            s = s + M[i][j]
        res.append(s)
    return res

```

```

# (c) somme de chaque ligne
# exemple : M = [[1, 2], [3, 4]]
def sommes_lignes(M):
    res = []
    s = 0
    for ligne in M:
        for x in ligne:
            s = s + x
        res.append(s)
    return res

# (d) mettre le rouge à 0
# img : image de triplets (R, G, B)
def sans_rouge(img):
    for i in range(len(img)):
        for j in range(len(img[0])):
            img[i][j][0] = 0

```

Corrigé.

- Affiche `[[0, 5, 0], [0, 5, 0]]` : les deux lignes sont la même liste. Construire les lignes une à une (fonction `zeros`).
- Les bornes sont inversées : `j` va jusqu'à `len(M) - 1 = 1` et `i` jusqu'à `len(M[0]) - 1 = 2`, d'où une `IndexError` pour `M[2][0]`. Écrire `for j in range(len(M[0])):` puis `for i in range(len(M)):`.
- Renvoie `[3, 10]` au lieu de `[3, 7]` : `s` n'est pas remis à 0 entre deux lignes. Placer `s = 0` entre les deux boucles.
- `TypeError` : un tuple n'est pas modifiable. Écrire `(r, g, b) = img[i][j]` puis `img[i][j] = (0, g, b)`.

● Exercice 3

Numériser un document (d'après CCINP PC 2023).

Une image en couleurs de p lignes et q colonnes est une liste de listes dont chaque élément est un triplet (R , G , B) d'entiers entre 0 et 255. Une image en niveaux de gris est une liste de listes d'entiers entre 0 (noir) et 255 (blanc).

1. Combien de bits faut-il pour coder un entier entre 0 et 255 ? Un pouce mesure environ 2,5 cm. Quelles sont les dimensions en pixels d'une page A4 ($21 \times 29,7$ cm) numérisée à 300 pixels par pouce ? En déduire la taille de l'image en couleurs, en bits puis approximativement en mégaoctets.
2. Le niveau de gris d'un pixel est la partie entière de $0,299R + 0,587G + 0,114B$. Écrire une fonction `conversion_gris(imgC: list) -> list` qui renvoie l'image en niveaux de gris.
3. Binariser une image en niveaux de gris consiste à remplacer chaque pixel strictement supérieur à un seuil par 255 et les autres par 0. Écrire `binarisation(imgG: list, seuil: int) -> list`.
4. Écrire une fonction `proportion_noirs(im: list) -> float` qui renvoie la proportion de pixels noirs d'une image binarisée.

Corrigé.

1. 8 bits ($2^8 = 256$). La page mesure $21/2,5 \times 300 = 2520$ pixels par $29,7/2,5 \times 300 = 3564$ pixels, soit 8 981 280 pixels. Chaque pixel utilise $3 \times 8 = 24$ bits : 215 550 720 bits, soit environ 27 Mo.
2. à 4.

```
def conversion_gris(imgC: list) -> list:
    '''imgC : image en couleurs (liste de listes de triplets).
    Renvoie l'image en niveaux de gris correspondante.'''
    res = []
    for ligne in imgC:
        nouvelle = []
        for (r, g, b) in ligne:
            nouvelle.append(int(0.299 * r + 0.587 * g + 0.114 * b))
        res.append(nouvelle)
    return res

def binarisation(imgG: list, seuil: int) -> list:
    '''Renvoie l'image où chaque pixel > seuil vaut 255, les autres 0.'''
    res = []
    for ligne in imgG:
        nouvelle = []
        for v in ligne:
            if v > seuil:
                nouvelle.append(255)
            else:
                nouvelle.append(0)
        res.append(nouvelle)
    return res

def proportion_noirs(im: list) -> float:
    '''im : image binarisée non vide. Renvoie la proportion de pixels noirs.'''
    n_noirs = 0
    for ligne in im:
        for v in ligne:
            if v == 0:
                n_noirs = n_noirs + 1
    return n_noirs / (len(im) * len(im[0]))
```

● Exercice 4

Graphe des galaxies (d'après CCINP PC 2026).

La liste `coords` contient les coordonnées (`x`, `y`, `z`) de N galaxies : c'est une liste de tuples. On relie toutes les galaxies deux à deux ; le poids de l'arête entre les galaxies `i` et `j` est $p_{ij} = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2$.

1. Écrire `poids(coords: list, i: int, j: int) -> float`.
2. Écrire une fonction `matrice_adjacence(coords: list) -> list` qui renvoie la matrice $N \times N$ de coefficients $m_{ij} = p_{ij}$.
3. Calculer la matrice pour `coords = [(0, 1, 5), (-1, 2, -4), (3, 2, 5)]`. Quelles propriétés remarquables possède-t-elle ?
4. Réécrire `matrice_adjacence` en n'appelant `poids` que pour les couples `i < j`.

Corrigé.

```
def poids(coords: list, i: int, j: int) -> float:
    '''Renvoie le carré de la distance entre les galaxies i et j.'''
    s = 0
    for k in range(3):
        s = s + (coords[i][k] - coords[j][k]) ** 2
    return s

def matrice_adjacence(coords: list) -> list:
    '''Renvoie la matrice N x N des poids entre galaxies.'''
    N = len(coords)
    M = []
    for i in range(N):
        ligne = []
        for j in range(N):
            ligne.append(poids(coords, i, j))
        M.append(ligne)
    return M

def matrice_adjacence2(coords: list) -> list:
    '''Même résultat, en exploitant la symétrie.'''
    N = len(coords)
    M = zeros(N, N)          # diagonale nulle
    for i in range(N):
        for j in range(i + 1, N):
            p = poids(coords, i, j)
            M[i][j] = p
            M[j][i] = p
    return M
```

Question 3 : `[[0, 83, 10], [83, 0, 97], [10, 97, 0]]`. La matrice est symétrique ($p_{ij} = p_{ji}$) et sa diagonale est nulle.

● Exercice 5

Nettoyer une image (*voisins d'un pixel*).

Une image binarisée scannée contient des pixels noirs isolés (poussières). Les *voisins* du pixel (`i`, `j`) sont les pixels (`i + di`, `j + dj`) avec `di` et `dj` dans `[-1, 0, 1]`, sauf (`i`, `j`) lui-même, à condition qu'ils soient dans l'image. On prendra pour exemple :

```
im = [[ 0, 255, 255, 255],
      [255, 255,  0,  0],
      [255, 255, 255, 255]]
```

1. Combien un pixel du coin, du bord et de l'intérieur a-t-il de voisins ?
2. Écrire une fonction `nb_voisins_noirs(im: list, i: int, j: int) -> int`. Sur l'exemple, que renvoient les appels pour `(0, 0)` et `(1, 2)` ?

- Un pixel noir est *isolé* s'il n'a aucun voisin noir. Écrire une fonction `pixels_isoles(im: list) -> list` qui renvoie la liste des couples `(i, j)` des pixels noirs isolés.
- Écrire une fonction `nettoyer(im: list) -> list` qui renvoie une nouvelle image où les pixels noirs isolés sont devenus blancs, sans modifier `im`.

Corrigé.

```
def nb_voisins_noirs(im: list, i: int, j: int) -> int:
    '''Renvoie le nombre de voisins noirs du pixel (i, j) dans im.'''
    p, q = len(im), len(im[0])
    n = 0
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            a, b = i + di, j + dj
            if (di, dj) != (0, 0) and 0 <= a < p and 0 <= b < q:
                if im[a][b] == 0:
                    n = n + 1
    return n

def pixels_isoles(im: list) -> list:
    '''Renvoie la liste des (i, j) des pixels noirs sans voisin noir.'''
    res = []
    for i in range(len(im)):
        for j in range(len(im[0])):
            if im[i][j] == 0 and nb_voisins_noirs(im, i, j) == 0:
                res.append((i, j))
    return res

def nettoyer(im: list) -> list:
    '''Renvoie une copie de im où les pixels noirs isolés sont blancs.'''
    res = []
    for ligne in im:
        res.append(ligne.copy())          # copie ligne par ligne
    for (i, j) in pixels_isoles(im):
        res[i][j] = 255
    return res
```

Question 1 : 3 voisins pour un coin, 5 pour un bord, 8 à l'intérieur. Question 2 : 0 pour (0, 0), 1 pour (1, 2). Sur l'exemple, `pixels_isoles(im)` renvoie `[(0, 0)]`.

● Exercice 6

Tuples et tableaux 3D.

- `imgC` est une image en couleurs (liste de listes de triplets). Écrire `canal(imgC: list, k: int) -> list` qui renvoie la matrice des composantes numéro `k` (0 pour le rouge, 1 pour le vert, 2 pour le bleu).
- Écrire `echanger_rouge_bleu(imgC: list) -> list` qui renvoie l'image où chaque pixel `(R, G, B)` devient `(B, G, R)`.
- Écrire `vers_3D(imgC: list) -> list` qui renvoie la même image sous forme d'un tableau 3D de listes : l'élément `[i][j][k]` est la composante `k` du pixel `(i, j)`. Quel est l'avantage de cette représentation ? Quel est son inconvénient ?

Une vidéo en niveaux de gris est un tableau 3D `V : V[t]` est l'image numéro `t` et `V[t][i][j]` un pixel. Toutes les images ont `p` lignes et `q` colonnes. Exemple à deux images : `V = [[[0, 100], [200, 50]], [[100, 200], [0, 150]]]`.

- Écrire `luminosites(V: list) -> list` qui renvoie la liste des luminosités moyennes de chaque image. Résultat sur l'exemple ?

5. Écrire `image_moyenne(V: list) -> list` qui renvoie l'image dont chaque pixel est la moyenne dans le temps des pixels correspondants. Résultat sur l'exemple ?
6. Écrire `image_max(V: list) -> int` qui renvoie le numéro de l'image la plus lumineuse, en réutilisant les fonctions précédentes.

Corrigé.

```
def canal(imgC: list, k: int) -> list:
    '''Renvoie la matrice des composantes k des pixels de imgC.'''
    res = []
    for ligne in imgC:
        nouvelle = []
        for pixel in ligne:
            nouvelle.append(pixel[k])
        res.append(nouvelle)
    return res

def echanger_rouge_bleu(imgC: list) -> list:
    '''Renvoie l'image où (R, G, B) devient (B, G, R).'''
    res = []
    for ligne in imgC:
        nouvelle = []
        for (r, g, b) in ligne:
            nouvelle.append((b, g, r))    # nouveau tuple
        res.append(nouvelle)
    return res

def vers_3D(imgC: list) -> list:
    '''Renvoie l'image sous forme de tableau 3D de listes.'''
    res = []
    for ligne in imgC:
        nouvelle = []
        for (r, g, b) in ligne:
            nouvelle.append([r, g, b])    # liste modifiable
        res.append(nouvelle)
    return res

def luminosite(V: list) -> list:
    '''Renvoie la liste des moyennes des pixels de chaque image de V.'''
    res = []
    for img in V:
        s = 0
        for ligne in img:
            for v in ligne:
                s = s + v
        res.append(s / (len(img) * len(img[0])))
    return res

def image_moyenne(V: list) -> list:
    '''Renvoie l'image des moyennes temporelles des pixels de V.'''
    nt, p, q = len(V), len(V[0]), len(V[0][0])
    res = []
    for i in range(p):
        ligne = []
        for j in range(q):
            s = 0
            for t in range(nt):    # moyenne dans le temps
                s = s + V[t][i][j]
```

```

        ligne.append(s / nt)
    res.append(ligne)
return res

def image_max(V: list) -> int:
    '''Renvoie le numéro de la première image de luminosité maximale.'''
    lum = luminositees(V)
    t_max = 0
    for t in range(1, len(lum)):
        if lum[t] > lum[t_max]:
            t_max = t
    return t_max

```

Question 3 : les composantes deviennent modifiables (`img[i][j][0] = 0` est possible), mais une liste occupe plus de mémoire qu'un tuple et rien n'empêche d'y ajouter une quatrième composante par erreur.

Question 4 : `[87.5, 112.5]`. Question 5 : `[[50.0, 150.0], [100.0, 100.0]]`. Dans `image_moyenne`, la boucle sur `t` est la plus intérieure : l'ordre des boucles n'est pas forcément celui des indices.

● Méthode

Travail personnel : deux exercices flash (15 minutes chacun).

1. `position_max(M: list) -> tuple` renvoie le couple (`i`, `j`) des indices du premier plus grand coefficient de la matrice non vide `M`.
2. `est_symetrique(M: list) -> bool` renvoie `True` si la matrice carrée `M` est symétrique, c'est-à-dire si `M[i][j] == M[j][i]` pour tous `i` et `j`.

Corrigé.

```

def position_max(M: list) -> tuple:
    '''Renvoie (i, j) tel que M[i][j] est le premier maximum de M.'''
    i_max, j_max = 0, 0
    for i in range(len(M)):
        for j in range(len(M[0])):
            if M[i][j] > M[i_max][j_max]:
                i_max, j_max = i, j
    return (i_max, j_max)

def est_symetrique(M: list) -> bool:
    '''M carrée. Renvoie True si M est symétrique, False sinon.'''
    n = len(M)
    for i in range(n):
        for j in range(i + 1, n):      # inutile de tester deux fois
            if M[i][j] != M[j][i]:
                return False
    return True

```