Correction du TP 7

1 Matrices

Exercice 1 — Colonnes d'une matrice

1. La fonction suivante extrait une colonne donnée d'une matrice représentée comme la liste de ses lignes.

```
def colonne(A, j):
    """
    Entrées :
        - une matrice A, donnée comme la liste de ses lignes ;
        - un indice j de colonne.
    Sortie : la liste des éléments de la colonne j.
    """
    n = len(A)          # n est le nombre de lignes de A.
    Col = []            # Col sera la colonne j.
    for i in range(n):
    # Pour i allant de 0 à n - 1, c'est-à-dire
    # pour i parcourant les indices des lignes de la matrice A.
        Col.append(A[i][j])
        # On ajoute à la liste Col le coefficient de A
        # qui est à la ligne i et à la colonne j.
    return Col
```

   On peut aussi construire la colonne en compréhension.

```
def colonne_bis(A, j):
    return [A[i][j] for i in range(len(A))]
```

2. La fonction suivante donne la liste des colonnes d'une matrice représentée comme la liste de ses lignes.

```
def Liste_colonnes(A):
    """
    Entrée : une matrice A, donnée comme la liste de ses lignes.
    Sortie : la liste des colonnes de A.
    """
    p = len(A[0])
    # p est le nombre d'éléments de la première ligne de A,
    # c'est-à-dire le nombre de colonnes de A.
    L = []
    for j in range(p):
    # Pour j allant de 0 à p - 1, c'est-à-dire
    # pour j parcourt les indices des colonnes de la matrice A.
        L.append(colonne(A, j))
        # On ajoute à la liste L la colonne j.
    return L
```

De même, on peut aussi construire la liste des colonnes en compréhension.

```
def Listes_colonnes_bis(A):
    return [colonne(A, j) for j in range(len(A[0]))]
```

Remarquons que la liste des colonnes obtenue représente la transposée de la matrice.

Exercice 2 — Sommes doubles

1. Un coefficient d'une matrice est diagonal si et seulement si son indice de ligne est égal à son indice de colonne.
   La fonction suivante renvoie alors la somme des coefficients diagonaux d'une matrice carrée.

```
def somme_diagonale(A):
    """
    Entrée : une matrice A.
    Sortie : la somme des coefficients diagonaux de A.
    """
    n = len(A) # Nombre de lignes de A.
    p = len(A[0]) # Nombre de colonnes de A.
    if n != p:
        return "La matrice n'est pas carrée."
    else:
        somme = 0
        for i in range(n): # Pour i allant de 0 à n - 1,
        # c'est-à-dire pour i parcourant les indices des lignes de A.
            somme += A[i][i]
        return somme
```

2. Les coefficients situés strictement au-dessus de la diagonale sont ceux dont l'indice de colonne est strictement supérieur à l'indice de ligne.
   La fonction suivante renvoie alors la somme des coefficients d'une matrice carrée situés strictement au-dessus de la diagonale.

```
def somme_triangulaire_sup(A):
    """
    Entrée : une matrice A.
    Sortie : la somme des coefficients strictement au-dessus
             de la diagonale dans la matrice A.
    """
    n = len(A) # Nombre de lignes de A.
    p = len(A[0]) # Nombre de colonnes de A.
    if n != p:
        return "La matrice n'est pas carrée."
    else:
        somme = 0
        for i in range(n): # i va de 0 à n - 1,
        # donc i parcourt les indices des lignes de A.
            for j in range(i + 1, p): # j va de i + 1 à p - 1,
            # donc j parcourt les indices de colonnes de A qui
            # sont strictement supérieurs à l'indice i de ligne.
                somme += A[i][j]
        return somme
```

Exercice 3 — Opérations matricielles

1. La fonction suivante effectue la multiplication d'une matrice par un scalaire.

```
def multiplication_scalaire(A, lambada):
    """
    Entrées : une matrice A et un flottant lambada.
    Sortie : la multiplication de la matrice A
             par le scalaire lambada.
    """
    n = len(A) # Nombre de lignes de A.
    p = len(A[0]) # Nombre de colonnes de A.
    M = [p * [0] for ligne in range(n)]
    # La matrice M sera le résultat de la multiplication :
    # c'est une matrice à n lignes et p colonnes,
    # c'est-à-dire de la même taille que la matrice A,
    # initialement nulle.
    for i in range(n):
    # i parcourt les indices de lignes de M.
        for j in range(p):
        # j parcourt les indices de colonnes de M.
            M[i][j] = lambada * A[i][j]
            # Le coefficient de M à la ligne i et à la colonne j
            # est le produit par le scalaire lambada
            # du coefficient à la même position dans A.
    return M
```

2. La fonction suivante transpose une matrice.

```
def transposition(A):
    """
    Entrée : une matrice A.
    Sortie : la matrice transposée de A.
    """
    n = len(A) # Nombre de lignes de A.
    p = len(A[0]) # Nombre de colonnes de A.
    T = [n * [0] for k in range(p)]
    # T sera la transposée de la matrice A :
    # T possède p lignes et n colonnes,
    # et T est initialement nulle.
    for j in range(p):
    # j parcourt les indices de lignes de T,
    # c'est-à-dire les indices de colonnes de A.
        for i in range(n):
        # i parcourt les indices de colonnes de T,
        # c'est-à-dire les indices de lignes de A.
            T[j][i] = A[i][j]
    return T
```

Pour transposer une matrice, on peut aussi procéder comme dans l'exercice 1 (mais c'est moins naturel, en ce sens que l'on perd alors de vue l'indexation des coefficients des matrices).

3. La fonction suivante multiplie deux matrices (si leurs tailles sont compatibles).

```
def produit(A, B):
    """
    Entrées : deux matrices A et B.
    Sortie : le produit matriciel AB s'il est bien défini,
             un message d'erreur sinon.
    """
    nA = len(A) # Nombre de lignes de A.
    pA = len(A[0]) # Nombre de colonnes de A.
    nB = len(B) # Nombre de lignes de B.
    pB = len(B[0]) # Nombre de colonnes de B.
    if pA != nB:
        return "Le produit n'est pas défini."
    else:
        P = [pB * [0] for ligne in range(nA)]
        # P sera le produit AB des deux matrices,
        # P possède nA lignes et pB colonnes.
        for i in range(nA):
        # i parcourt les indices de lignes de P.
            for j in range(pB):
            # j parcourt les indices de colonnes de P.

                # On calcule le coefficient du produit AB
                # à la ligne i et à la colonne j,
                # qui est initialisé à 0.

                for k in range(pA):
                # k parcourt les indices de colonnes de A,
                # qui sont aussi les indices de lignes de B.
                    P[i][j] += A[i][k] * B[k][j]

        return P
```

2 Tracé de courbes

Pour les tracés de fonctions, on importe deux bibliothèques.

```
import numpy as np

import matplotlib.pyplot as plt
```

Exercice 4 — Ils étaient cinq et cent, ils étaient des milliers
Représentons graphiquement la fonction $x \longmapsto x\sin(x)$ sur l'intervalle $[-1;1]$, tantôt avec mille points pour le tracé, tantôt avec cinq points.

Tout d'abord, définissons la fonction à tracer.

```
def f(x):
    return x * np.sin(x)
```

Définissons alors les listes d'abscisses et d'ordonnées que l'on utilisera pour le tracé, dont le premier élément sera $a = -1$ et le dernier $b = 1$.

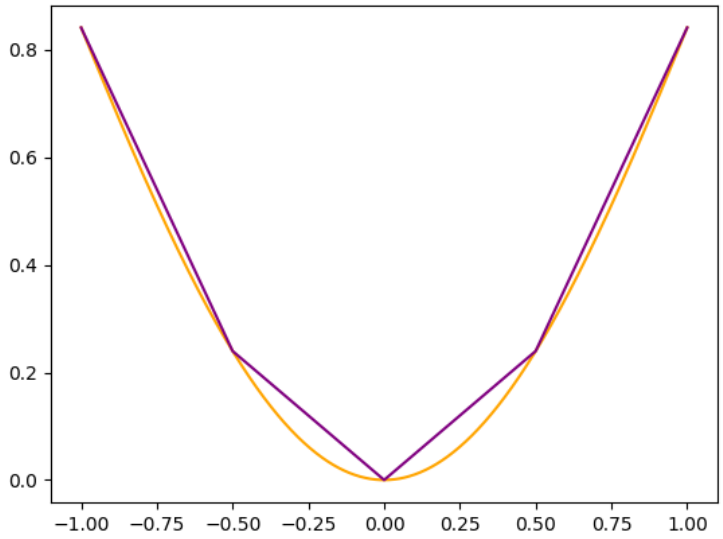
```
a = -1
b = 1

# Avec cinq points :
h5 = (b - a) / 4
abscisses5 = [a + k * h5 for k in range(5)] # Une liste de 5 points.
ordonnees5 = [f(x) for x in abscisses5]

# Avec mille points :
h1000 = (b - a) / 999
abscisses1000 = [a + k * h1000 for k in range(1000)]
ordonnees1000 = [f(x) for x in abscisses1000]
```

Traçons enfin les deux courbes sur une même figure.

```
plt.figure()
plt.plot(abscisses5, ordonnees5)
plt.plot(abscisses1000, ordonnees1000)
plt.show()
```



Exercice 5 — Intersection de deux courbes

- Commençons par justifier que la droite d'équation $y = x$ et la courbe de la fonction cosinus s'intersectent exactement une fois,
  c'est-à-dire montrons que la fonction cosinus admet un unique point fixe sur $\mathbb{R}$.

  ⋆ La fonction cosinus est à valeurs dans $[-1; 1]$.
  Donc pour tout $x \in \mathbb{R} \setminus [-1; 1]$, on a $\cos(x) \neq x$.
  Ainsi, $\boxed{\text{la fonction cosinus n'admet aucun point fixe dans } \mathbb{R} \setminus [-1; 1]}$.

⋆ Montrons à présent que la fonction cosinus admet un unique point fixe dans $[-1;1]$.
Pour cela, étudions la fonction suivante :

$$\begin{aligned} g\colon [-1;1] &\longrightarrow \mathbb{R} \\ x &\longmapsto \cos(x) - x. \end{aligned}$$

— La fonction g est continue sur l'intervalle $[-1;1]$.
— La fonction g est strictement décroissante sur $[-1;1]$ comme somme de la fonction cosinus, strictement décroissante sur $[-\pi;\pi]$ et donc sur $[-1;1]$, et de la fonction $x \longmapsto -x$, strictement décroissante sur $\mathbb{R}$ donc sur $[-1;1]$.

Le théorème de la bijection continue assure alors que
— l'image de la fonction g est

$$\begin{aligned} g([-1;1]) &= [g(1); g(-1)] \\ &= [\cos(1) - 1; \cos(-1) + 1] \\ &= [\cos(1) - 1; \cos(1) + 1]; \end{aligned}$$

— la fonction g réalise une bijection de $[-1;1]$ sur son image $[\cos(1) - 1; \cos(1) + 1]$.

Comme $-1 \leqslant \cos(1) \leqslant 1$, on a :

$$\cos(1) - 1 \leqslant 0 \leqslant \cos(1) + 1,$$

donc le réel 0 admet un unique antécédent par la fonction g dans l'intervalle $[-1;1]$,
c'est-à-dire qu'il existe un unique $x \in [-1;1]$ tel que $g(x) = 0$,
c'est-à-dire qu'il existe un unique $x \in [-1;1]$ tel que $\cos(x) = x$.
Ainsi, la fonction cosinus admet un unique point fixe dans l'intervalle $[-1;1]$.

Finalement, la fonction cosinus admet bien un unique point fixe dans $\mathbb{R}$.

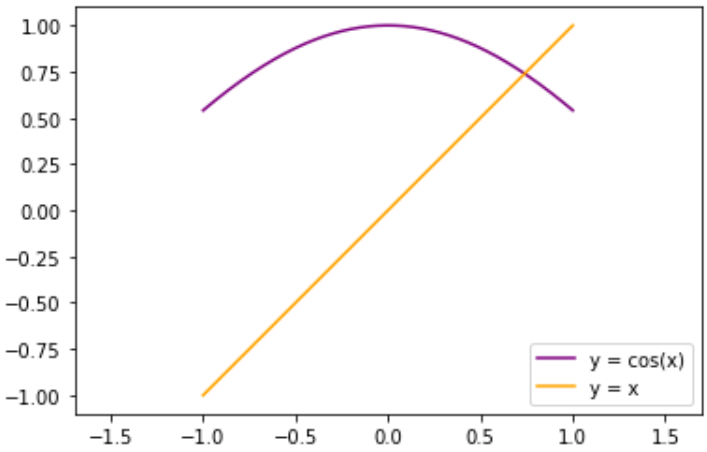
- Pour déterminer graphiquement la valeur de ce point fixe, c'est-à-dire le point d'intersection entre la droite d'équation $y = x$ et la courbe de la fonction cosinus, on va tracer les deux courbes sur un même graphique.
On trace les deux courbes sur $[-1;1]$, puisque d'après l'étude précédente, le point fixe de la fonction cosinus appartient à $[-1;1]$.

```
a = -1
b = 1
h = (b - a) / 999

# Liste des mille abscisses des points des deux courbes
abscisses = [a + k * h for k in range(1000)]

# Listes des ordonnées correspondantes pour les deux fonctions
ordonnees_cos = [np.cos(x) for x in abscisses]
ordonnees_identite = [x for x in abscisses]
    # On pourrait juste garder la même liste abscisses.
```

```
# Tracé
plt.figure()
plt.plot(abscisses, ordonnees_cos)
plt.plot(abscisses, ordonnees_identite)
plt.show()
```

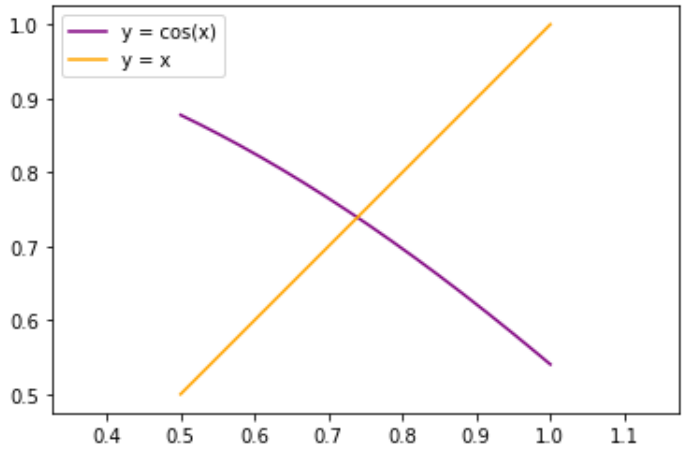


Pour avoir une estimation plus précise du point fixe, on peut alors recommencer le tracé sur un intervalle plus petit.

```
a = 0.5
b = 1
h = (b - a) / 999

abscisses_bis = [a + k * h for k in range(1000)]
ordonnees_cos_bis = [np.cos(x) for x in abscisses]

plt.figure()
plt.plot(abscisses_bis, ordonnees_cos_bis)
plt.plot(abscisses_bis, abscisses_bis)
plt.show()
```



On lit graphiquement que le point fixe de cosinus est compris entre $0,7$ et $0,8$.

Exercice 6 — Courbes paramétrées

1. **Ellipse.**
   Traçons la courbe paramétrée définie par : pour tout $t \in [0; 2\pi]$,

$$\begin{cases} x(t) &= 4\cos(t) \\ y(t) &= 3\sin(t), \end{cases}$$

   c'est-à-dire l'ensemble suivant

$$\Big\{ (x(t), y(t)) \;\Big|\; t \in [0; 2\pi] \Big\}.$$

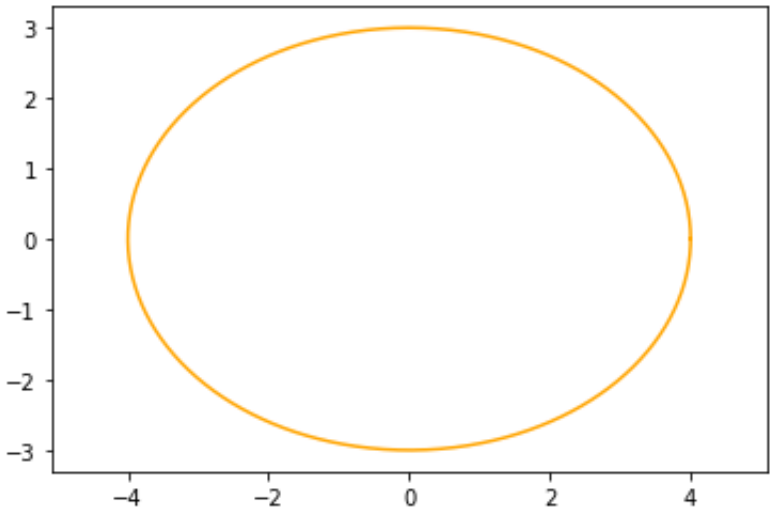
```
# Liste des valeurs de t, c'est-à-dire des paramètres :
a = 0
b = 2 * np.pi
h = (b - a) / 999

parametres = [a + k * h for k in range(1000)]

# Liste des valeurs de x, qui seront en abscisses :
abscisses = [4 * np.cos(t) for t in parametres]

# Liste des valeurs de y, qui seront en ordonnees :
ordonnees = [3 * np.sin(t) for t in parametres]

# Tracé de l'ellipse
plt.figure()
plt.plot(abscisses, ordonnees)
plt.show()
```



2. **Cercle.**
3. **Spirale.**
4. **Courbe de Lissajous.**

3 Traitement d'images

Dans toute cette partie, on se concentrera davantage sur les manipulations matricielles que sur l'aspect graphique.

Exercice 7 — Sens dessus dessous

Pour tourner de 180 degrés une image à n lignes et p colonnes, on doit, pour tous indices i et j, envoyer le pixel de la ligne i et de la colonne j à la ligne $(n-1)-i$ et à la colonne $(p-1)-j$.

```
def retournement(M):
    """
    Entrée : une matrice M de pixels.
    Sortie : la matrice M retournée.
    """
    n = len(M) # Nombre de lignes de M.
    p = len(M[0]) # Nombre de colonnes de M.
    R = [p * [0] for ligne in range(n)]
    # R sera l'image retournée : c'est une matrice
    # à n lignes et p colonnes, comme M.
    for i in range(n): # i parcourt les indices de lignes de M.
        for j in range(p): # j parcourt les indices de colonnes de M.
            R[n - 1 - i][p - 1 - j] = M[i][j]
    return R
```

Exercice 8 — Seuillage

1. La fonction suivante seuille une matrice de pixels à un seuil donné, c'est-à-dire que tous les coefficients de la matrice qui sont inférieurs au seuil seront des pixels noirs.

```
def seuillage(seuil, M):
    """
    Entrées :
        - un flottant seuil compris entre 0 et 1 ;
        - une matrice M de pixels.
    Sortie : la matrice M seuillée au seuil seuil.
    """
    n = len(M) # Nombre de lignes de M.
    p = len(M[0]) # Nombre de colonnes de M.
    S = [p * [0] for ligne in range(n)]
    # S sera la matrice seuillée, de même taille que M,
    # dont tous les pixels sont initialement noirs.
    for i in range(n): # i parcourt les indices de lignes de S.
        for j in range(p): # j parcourt les indices de colonnes de S.
            if M[i][j] >= seuil:
                S[i][j] = M[i][j]
            # Sinon, le pixel reste noir.
    return S
```

2.

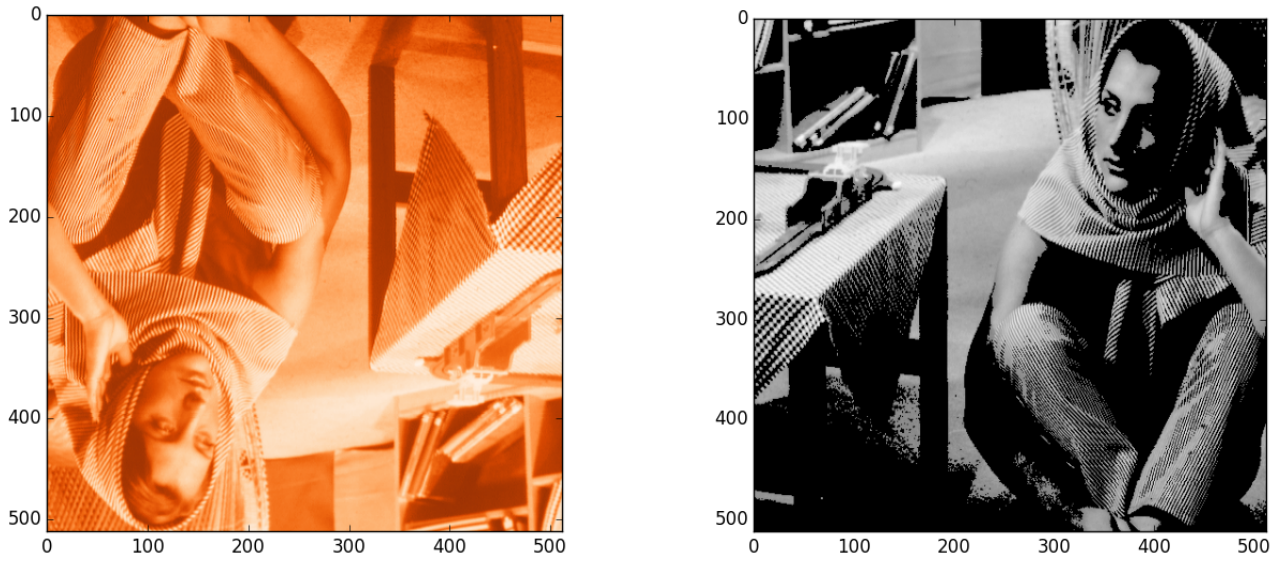


FIGURE 1 – Barbara à l'envers (à gauche) et Barbara seuillée à 0.5 (à droite)

Exercice 9 — Agrandissement

La fonction suivante agrandie une matrice de pixels d'un facteur d, c'est-à-dire que chaque coefficient de la matrice sera répété d^2 fois dans la matrice agrandie.

```
def agrandissement(M, d):
    """
    Entrées :
        - une matrice M de pixels ;
        - un entier naturel d non nul.
    Sortie : la matrice M agrandie d'un facteur d.
    """
    n = len(M) # Nombre de lignes de M.
    p = len(M[0]) # Nombre de colonnes de M.
    G = [d * p * [0] for k in range(d * n)]
    # G sera la matrice agrandie,
    # elle possède d * n lignes et d * p colonnes.
    for i in range(d * n): # i va de 0 à d * n - 1,
    # c'est-à-dire que i parcourt les indices de lignes de G.
        for j in range(d * p): # j va de 0 à d * p - 1,
        # c'est-à-dire que j parcourt les indices de colonnes de G.
            G[i][j] = M[i // d][j // d]
            # Ainsi, les coefficients des d premières lignes de G,
            # c'est-à-dire des lignes 0 à d  - 1 de G,
            # sont ceux de la première ligne (ligne 0) de M,
            # car le quotient de la division euclidienne
            # de 0, ..., de d - 1 par d est nul.
            # Et ainsi de suite.
    return G
```

Exercice 10 — Réduction moyennée