TP (S1) 8 Traitement d'images

Objectifs

- Utilisation des tableaux multi-dimensionnels.
- Représentation et traitement d'images.

Fichier externe ?

OUI `TP_TableauxImages.py` *(présent(s) dans le répertoire partagé de la classe)*

Ce TP ne sera traité en séance que très partiellement. Des parties seront à faire à la maison, elles sont indiquées par le logo .

1 CODAGE D'UNE IMAGE & TABLEAUX NUMPY

1.1 Pixels

Le pixel est l'élément de base permettant de caractériser la définition d'une image numérique. Une image peut donc être représentée par une matrice T dont chaque terme représente l'état de coloration du pixel correspondant. Par exemple, ci-dessous une image avec 9 pixels [1] :

T =	T[0, 0]	T[0, 1]	T[0, 2]
	T[1, 0]	T[1, 1]	T[1, 2]
	T[2, 0]	T[2, 1]	T[2, 2]

1. bien sûr, dans la pratique ce nombre sera bien plus important

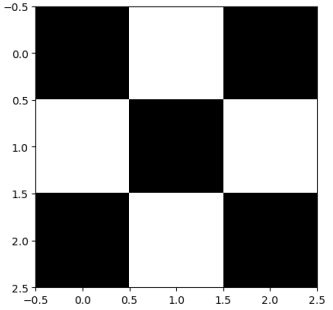
Ainsi, le terme T[0,0] représente l'état du pixel situé en haut à gauche de l'image et de manière générale T[i,j] est le pixel situé sur la ligne d'indice i en partant du haut et sur la colonne d'indice j en partant de la gauche. Le contenu de T[i,j] diffère selon le type d'image :

- Pour une image en noir et blanc, il y a deux états possibles et un bit suffira (0 pour un pixel noir,1 pour le blanc). Par exemple :

Tableau :

T =	0	1	0
	1	0	1
	0	1	0

Image associée :

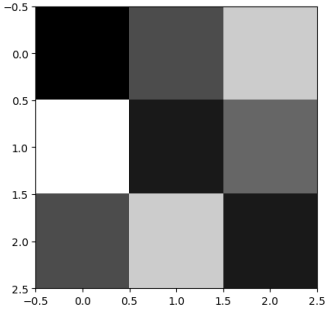


- Pour une image en niveau de gris, chaque pixel est soit un flottant compris entre 0 et 1, soit un entier compris entre 0 et 255. La valeur 0 correspond à un pixel noir, la valeur maximale (1 ou 255) à un pixel blanc. Par exemple :

Tableau :

T =	0	0.3	0.8
	1	0.1	0.4
	0.3	0.8	0.1

Image associée :

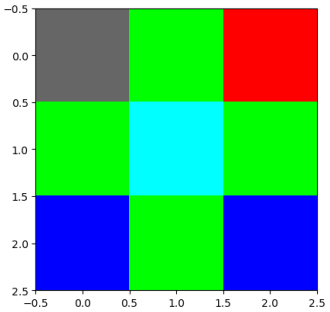


- Enfin, pour une image en couleur, chaque pixel est un triplet `[r,g,b]`, où r, g, b sont soit des flottants entre 0 et 1, soit de entiers entre 0 et 255, codant l'intensité d'une couleur primaire (r pour rouge, v pour vert, b pour bleu) dans le pixel. Par exemple :

Tableau :

T =	[0.4, 0.4, 0.4]	[0,1,0]	[1,0,0]
	[0,1,0]	[0,1,1]	[0,1,0]
	[0,0,1]	[0,1,0]	[0,0,1]

Image associée :



Remarque 1

- Si $r = g = b = 0$, le pixel est noir. Si r, g et b ont leur valeur maximale (donc 1 ou 255, la valeur par défaut étant 255), le pixel est blanc.
- Supposons chaque coordonnée soit un entier entre 0 et 1, alors : `[1,0,0]` est du rouge pur, `[0,1,1]` est le cyan (complémentaire du rouge), etc. Les couleurs ayant des proportions identiques de rouge, vert et bleu `[x,x,x]` sont des gris de plus en plus clair lorsque `x` augmente.

Remarque 2 **(Pourquoi 255 ?)** Le codage RGB de la couleur s'opère informatiquement à l'aide de 3 coordonnées de 8 *bits* chacune ; ainsi chaque coordonnée aura pour valeur maximale $2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0 = 2^8-1 = 255$. Plus de détails sur ces notions seront apportés à la fin de l'année dans le **??**).

1.2 Tableaux

La librairie `numpy` *(rencontrée en cours au début du semestre)* est une bibliothèque pour langage de programmation Python, destinée à manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux. C'est le type d'objet crée par les fonctions d'importation d'images que nous détaillerons plus bas. L'objectif de cette sous-section n'est pas un parcours exhaustif de toutes les fonctionnalités de `numpy`, mais uniquement celles qui nous seront utiles pour le traitement d'images. On commence par importer la bibliothèque :

```
import numpy as np
```

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 ces tableaux sont homogènes (ils ne contiennent que des valeurs d'un même type) et de taille fixée à la création : c'est donc une différence importante avec les listes de listes, à garder à l'esprit.

- **Création :** on crée simplement un tel tableau en convertissant une liste de listes.

```
>>> T = np.array([[1, 0, 1], [0, 1, 0]])
>>> type(T)
<class 'numpy.ndarray'>
>>> T.dtype
dtype('int64')
>>> T = np.array([[1, 0, 1], [0, 1, 0]])
>>> T
array([[1, 0, 1],
       [0, 1, 0]])
>>> type(T)
<class 'numpy.ndarray'>
```

- **Taille :** on accède à la dimension de `T` à l'aide de la fonction `np.shape` (existe aussi sous forme de méthode).

```
>>> n,p = np.shape(T)[0], np.shape(T)[1] # ou T.shape
>>> n, p
(2, 3)
```

- **Accession à un élément :** on accède à l'élément de `T` situé à la ligne `i` et dans la colonne `j` par `T[i, j]` (ou `T[i][j]` si on préfère une syntaxe proche des listes de listes).

```
>>> T[0, 0]
np.int64(1)
>>> T[0, 1]
np.int64(0)
```

- **Parcourir :** si `T` possède n lignes et p colonnes, elles sont numérotées de 0 à $n-1$ (*resp.* $p-1$). On dit que (x, y) sont des coordonnées *dans le champ de* T si $0 \leqslant x \leqslant n-1$ et $0 \leqslant y \leqslant p-1$. On peut parcourir un tableau au moyen de deux boucles `for` : `for i in range(n)` puis `for j in range(p)`.
- **Slicing :** on peut extraire facilement des portions de tableaux avec une syntaxe similaire aux listes (quand on extrait avec `a:b` l'indice `b` est toujours exclu). Par exemple :

```
>>> T
array([[1, 0, 1],
```

```
       [0, 1, 0]])
>>> T[1:,1:]
array([[1, 0]])
>>> T[:,1:]
array([[0, 1],
       [1, 0]])
```

- **Tableaux usuels :** on peut créer facilement des tableaux particuliers. Par exemple le tableau nul ou un tableau de 1.

```
>>> np.zeros((2, 3)) # nécessite un tuple en argument
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.ones((2, 3))
array([[1., 1., 1.],
       [1., 1., 1.]])
```

- **Moyenne :** la fonction `np.mean` permet de renvoier la moyenne arithhmétique d'un tableau.

```
>>> T
array([[1, 0, 1],
       [0, 1, 0]])
>>> np.mean(T)
np.float64(0.5)
```

- **Opérations (coefficient par coefficient) :**[2] on peut effectuer un grand nombre d'opérations directement sur les array : elles sont effectuées élément par élément. Ainsi `T**2` va élever au carré chaque coefficient de `T`. La plupart des fonctions mathématiques sont redéfinies par `numpy` et permettent d'agir sur un tableau : par exemple `np.sin(T)` applique le sinus sur chaque élément de `T`.

À retenir **Différences entre tableaux numpy et listes**

Même si les objets `ndarray` et `list` (listes de 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 listes 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, on y accède avec `T.dtype`. Par exemple, `T[0, 0] = 0.001` n'aura aucun effet si `T` est crée avec des entiers (Python transforme alors automatiquement `0.001` en un entier).

2. Les opérations usuelles du calcul matriciel existent aussi mais ne seront pas utiles dans ce TP

```
>>> T = np.array([[1, 2], [3, 4]])
>>> T.dtype
dtype('int64')
>>> T[0, 0] = 0.001
>>> T
array([[0, 2],
       [3, 4]])
```

2 OUVRIR, AFFICHER ET MODIFIER UNE IMAGE

Commencez par récupérer et ouvrir le fichier `TP_TableauxImages.py` disponible dans le répertoire partagé de la classe, ainsi que toutes les images associées, il contient le code des fonctions, généralement à compléter, qui seront étudiées dans les différents exercices de ce TP.

Exercice 1 Afficher une image et extraire des informations [Sol 1]

1. Après avoir copié l'image `'lighthouse.png'` du répertoire de la classe vers votre répertoire perso, recopiez et exécutez le code suivant (avec « Ctrl+Shift+E » après avoir enregistré le code dans le même répertoire que l'image). Vérifiez le type, la dimension (`np.shape(Im)`) et le contenu de `Im` (on se demandera notamment le codage utilisé pour cette image parmi ceux présentés en introduction).

```
import numpy as np
import matplotlib.pyplot as plt
Im = plt.imread('lighthouse.png') # un tableau numpy
plt.figure('gris') # titre à la figure (facultatif)
plt.imshow(Im, cmap = 'gray')
plt.show()
```

(Le paramètre `cmap` permet de régler l'interprétation faite par `imshow` du tableau `numpy` : si chaque coordonnée du tableau est de taille 3, `imshow` utilisera par défaut un codage RGB (voir introduction). Ici on impose un niveau de gris en spécifiant `cmap`)

2. Que fait le code ci-dessous qui est à ajouter à la suite du code de la question précédente ?

```
Imnb = np.zeros(np.shape(Im))
seuil = np.mean(Im)
n, p = np.shape(Im)[0], np.shape(Im)[1]
for i in range(n):
    for j in range(p):
        if Im[i,j] >= seuil:
            Imnb[i,j] = 1

plt.figure('nb') # titre à la figure(facultatif)
plt.imshow(Imnb, cmap = 'gray')
plt.show()
```

3. À partir du code précédent, proposez une fonction `negatif(Im:np.array)->np.array` qui permet d'obtenir le négatif d'une image en niveau de gris, c'est à dire une image où la luminosité est inversée (les zones claires deviennent foncées, les zones foncées deviennent claires). Cette fonction pourra être testée sur l'image `'lighthouse.png'`

4. Sur le même principe, écrire une fonction `miroir_v(Im:np.array)->np.array` qui permet d'obtenir l'image miroir de l'image initiale, c'est à dire l'image symétrique par rapport à un axe vertical passant par le milieu.

5. Ecrire de même une fonction `miroir_h(Im:np.array)->np.array` qui permet d'obtenir l'image symétrique par rapport à un axe horizontal passant par le milieu.

Exercice 2 Convertir une image couleur en niveau de gris

[Sol 2]

1. Le fichier `` `crayons.jpg' `` contient une image couleur qu'il est possible d'afficher en utilisant les mêmes instructions que précédemment mais en supprimant l'argument `cmap='gray'` qui n'est nécessaire que pour les images en niveau de gris ou en noir et blanc. Charger l'image dans la variable `Im2`, examiner le contenu de cette variable puis afficher l'image.

2. Prédire le résultat et analyser la commande :

   `(Im2[200,200][0]+Im2[200,200][1]+Im2[200,200][2])/3.`

3. Il est possible de convertir une image couleur RGB en image en niveau de gris en moyennant chaque pixel, c'est-à-dire en créant un pixel de valeur :

$$\text{Gris} = \frac{1}{3}\text{Rouge} + \frac{1}{3}\text{Vert} + \frac{1}{3}\text{Bleu}.$$

En utilisant `np.mean`, créer une nouvelle image en appliquant cette formule puis affichez-la.

Par ailleurs, il est possible d'enregistrer l'image avec l'instruction `plt.imsave('nom_fichier_image.png',Im3, cmap='gray')`.

3 QUELQUES TRANSFORMATIONS DE BASE SUR LES IMAGES

Cadre

Pour simplifier le code, les images traitées dans cette section seront des images en niveau de gris. Cependant les méthodes proposées s'appliquent également aux images couleurs.

Dans cette partie, on produit à chaque fois une nouvelle image « transformée » à partir de l'ancienne. Chaque coordonnée de pixel de l'image de départ peut donc avoir un pixel image dans l'image d'arrivée, ou aucun (le pixel en question de l'image de départ est donc « perdu »). En cas d'existence, on emploiera le même vocabulaire que pour les applications en Mathématiques en parlant de *pixel image* et de *pixel antécédent*.

3.1 Zoomer

Zoomer consiste à agrandir une partie d'une image autour d'un point particulier en lui associant plus de pixels qu'il n'y en avait pour cette partie dans l'image initiale. De cette façon, elle apparait plus grosse. Généralement, la taille de l'image ne varie pas au cours d'un zoom, seule la zone affichée est différente. Ainsi, si on considère une image initiale de taille (n, p), zoomer d'un facteur k autour d'un centre C consiste à remplir un tableau (n, p) à l'aide du tableau de taille $(n//k, p//k)$ centré autour du pixel C.

Comme dans tous les problèmes de transformation d'image, la nouvelle image sera stockée dans un tableau `Im1`.

Pour chaque point de la nouvelle image, il s'agit de déterminer (à l'aide de k) à quel point de l'ancienne il correspond (voir la première question du prochain exercice). À partir de là, plusieurs options se présentent pour colorer ce pixel :

- reprendre directement la couleur de l'ancien pixel (dans `Im`);
- ou reporter une valeur moyenne des pixels avoisinants (dans `Im`).

La première option produit des images de qualité médiocre puisqu'en moyenne un pixel de l'ensemble de départ sera reproduit k fois à l'identique dans l'image d'arrivée, ce qui produira des effets d'escalier. La seconde option, pour laquelle il y a de nombreuses variantes, est plus coûteuse mais produit un meilleur lissage. Cette option ajoute cependant du flou à l'image.

Exercice 3 Moyenner sur un voisinage [Sol 3]

Compléter le code suivant relatif à une fonction `moyenne_vois(T:np.array, x:int, y:int)->float:` qui, étant donné un tableau `T` et des coordonnées (x,y), renvoie la moyenne des valeurs des cases voisines de (x,y) dans `T`.

```
import numpy as np
def moyenne_vois(T:np.array, x:int, y:int)->list:
    n, p = np.shape(T)[0], np.shape(T)[1]
    D = [(1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), \
    ↪ (-1, 1), (0, 1)] # déplacements élémentaires
    S, nb = 0, 0
    for d in D:
        dx, dy = d[0], d[1]
        a, b = __________, __________
        if 0 <= a and a < n and 0 <= b and b < p:
            S += ________________ # somme des valeurs
            nb += _____ # nombre de cases voisines
    return ________________
```

Exercice 4 Zoom sur une image en niveau de gris [Sol 4]

L'objectif de cet exercice est de proposer une fonction `zoom(Im:np.array,x_c:int,y_c:int,k->float:)->np.array` qui, à partir du tableau `Im` d'une image initiale en niveau de gris, renvoie une nouvelle matrice de même taille que `Im` correspondant au zoom de l'image initiale centrée sur le pixel de coordonnées (x_c,y_c) avec un « grossissement » de facteur k. Ce qui signifie qu'un pixel de l'image initiale occupera environ k^2 pixels dans l'image renvoyée (k n'étant pas nécessairement un entier).

1. On note (x_1,y_1) les coordonnées d'un point dans l'image zoomée et (x,y) celles du point correspondant dans l'image initiale. Nous allons construire l'image nouvelle (de pixels (x_1,y_1)) telle que les relations ci-dessous soient satisfaites :

$$\begin{cases} x = x_c + \frac{x_1 - \frac{n}{2}}{k} \\ y = y_c + \frac{y_1 - \frac{p}{2}}{k} \end{cases}$$

où (n,p) désigne la taille commune des deux images, c'est-à-dire `np.shape(Im)` dans la suite.
Justifier la pertinence de ces relations, en comparant la distance entre (x_1,y_1) et $\left(\frac{n}{2},\frac{p}{2}\right)$ d'une part, et la distance entre (x,y) et $\left(\frac{n}{2},\frac{p}{2}\right)$ d'autre part.

- Notez que ces formules ne renvoient en général pas des valeurs **entières** pour x et y, et qu'il faudra donc arrondir à l'entier le plus proche les résultats, ce qui peut se faire par la fonction `round`.
- D'autre part, il se peut que (x,y) corresponde, si on s'éloigne trop du centre du zoom, à des valeurs hors du champ des indices de l'image initiale. Dans ce cas le pixel d'indices (x_1,y_1) de l'image zoomée sera placé à 0 ce qui a pour effet de colorier ce point en noir.

2. Compléter, dans le code suivant, la définition de la fonction `zoom1(Im:np.array,x_c:int,y_c:int,k:float)->np.array` en utilisant les formules précédentes pour colorier la nouvelle image. Tester cette fonction avec l'image `'lighthouse.png'`.

```
Im = plt.imread('lighthouse.png')

plt.figure('gris')
plt.imshow(Im, cmap = "gray")
plt.show()

def zoom1(Im:np.array,x_c:int,y_c:int,k:float)->np.array:
    n, p = np.shape(Im)[0], np.shape(Im)[1]
    Im1 = .......... #on créé la matrice destination
    for x_1 ..........: #on traite chacun des pixels (x_1,y_1) \
    ↪ de la destination Im1
        for y_1 ..........:
            x = x_c + round((x_1-n/2)/k)
            y = y_c + round((y_1-p/2)/k)
            if ............:
                Im1[x_1, y_1] = ............
    return Im1

Z1 = zoom1(Im, Im.shape[0]//2, Im.shape[1]//2,2) #vous pouvez \
↪ choisir un centre différent
```

```
plt.figure()
plt.imshow(Z1, cmap = "gray")
plt.show()
```

3. À partir de la fonction `zoom1` et de la fonction `moyenne_vois`, créer une nouvelle fonction `zoom2(Im:np.array,x_c:int,y_c:int,k:float)->np.array` qui utilise cette fois la moyenne des valeurs des pixels avoisinant le pixel antécédent. Tester cette fonction avec l'image `'lighthouse.png'`.

3.2 Rotation

On cherche désormais à appliquer une rotation à une image à partir d'un centre et d'un angle. La méthode adoptée est similaire à celle proposée pour le zoom : on commence par créer la matrice de zéros qui accueillera les données de l'image après rotation. Pour chaque pixel (x_1, y_1) de l'image finale, on calcule son antécédent (x, y) dans l'image initiale (que l'on obtient après rotation de l'angle opposé), qui est donné par les formules suivantes :

$$\begin{cases} x = x_c + \cos(\alpha)(x_1 - x_c) + \sin(\alpha)(y_1 - y_c) \\ y = y_c - \sin(\alpha)(x_1 - x_c) + \cos(\alpha)(y_1 - y_c), \end{cases}$$

là encore, les résultats seront arrondis aux entiers les plus proches, et si (x, y) n'appartient pas à l'image initiale le point correspondant sur l'image finale sera de couleur noire.

Exercice 5 **Algorithme de rotation** [Sol 5] Proposez une fonction

```
rotation(Im:np.array, x_c:int,y_c:int,a:float)->np.array
```

qui renvoie un tableau correspondant à l'image originale après une rotation de centre (x_c, y_c) et d'angle a par l'approche détaillée ci-dessus. On remplira le pixel à l'aide de la fonction `moyenne_vois` codée précédemment.

À retenir

Une image est représentée en mémoire comme un tableau dont la dimension correspond à la taille de l'image en pixels. Les éléments du tableau décrivent le pixel soit par sa luminosité, soit par sa couleur décomposée en 3 couleurs primaires (rouge, vert, bleu) avec éventuellement sa transparence. L'instruction `Im = plt.imread('lighthouse.png')` permet de récupérer le tableau de type `np.array` descriptif de l'image. L'instruction `plt.imshow(Im,cmap="gray")` (suivie de `plt.show()`) permet d'afficher l'image (l'option "gray" permettant de préciser lorsqu'il s'agit d'une image en niveau de gris).

Lors d'opération sur les images, la meilleure méthode consiste souvent à :

- créer un tableau de zéros à la bonne taille pour recevoir l'image après traitement
- déterminer l'antécédent de chacun des pixels par la transformation à appliquer
- calculer la ou les valeurs des pixels cibles à partir de la valeur de l'antécédents et éventuellement de ses voisins.

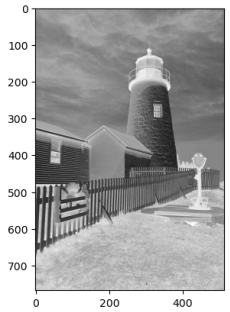
Solution 1

1. `Im` est un tableau numpy (np.array) dont la dimension correspond à la taille de l'image en pixels et qui contient des flottants entre 0 et 1.

Ce code transforme une image en niveau de gris en une image noir et blanc à partir d'un seuillage ici égal à la valeur moyenne.

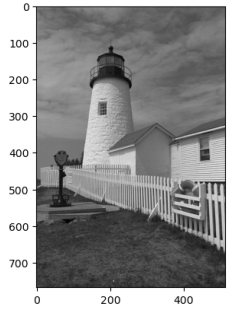
3.

```
def negatif(Im:np.array)->np.array:
    n, p = np.shape(Im)[0], np.shape(Im)[1]
    Imneg = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            Imneg[i,j] = 1-Im[i,j]
    return Imneg

Imneg = negatif(Im)
plt.figure()
plt.imshow(Imneg, cmap ="gray")
```



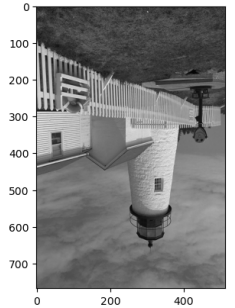
4.

```
def miroir_v(Im:np.array)->np.array:
    n, p = np.shape(Im)[0], np.shape(Im)[1]
    Im_mirv = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            Im_mirv[i,j] = Im[i,p-1-j]
    return Im_mirv

Im_mir_v = miroir_v(Im)
plt.figure()
plt.imshow(Im_mir_v, cmap ="gray")
```



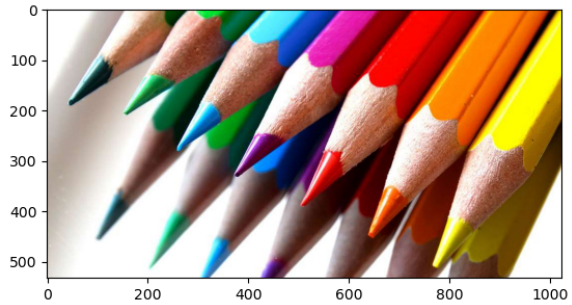
5.

```
def miroir_h(Im:np.array)->np.array:
    n, p = np.shape(Im)[0], np.shape(Im)[1]
    Im_mir_h = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            Im_mir_h[i,j] = Im[n-1-i,j]
    return Im_mir_h

Im_mir_h = miroir_h(Im)
plt.figure()
plt.imshow(Im_mir_h, cmap ="gray")
```



Solution 2

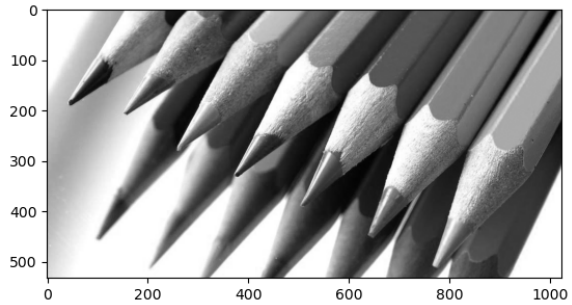
1.
```
plt.figure('couleurs')
Im2 = plt.imread('crayons.jpg')
np.shape(Im2) # Dimension du tableau
Im2[200, 200]  # Contenu d'un élément du tableau
plt.imshow(Im2)
```



2. L'instruction devrait renvoyer la moyenne des trois pixels. Elle renvoie une erreur, due au fait que les trois coordonnées ne sont pas du type `int`, mais `uint8` :

```
>>> (Im2[200,200][0]+Im2[200,200][1]+Im2[200,200][2])/3
<input>:1: RuntimeWarning: overflow encountered in scalar add
np.float64(77.33333333333333)
```

3.
```
n, p = np.shape(Im2)[0], np.shape(Im2)[1]
Im3 = np.zeros((n, p))
for i in range(n):
    for j in range(p):
        Im3[i, j] = np.mean(Im2[i, j])

plt.figure('gris')
plt.imshow(Im3, cmap = 'gray')
```



Solution 3

1.
```
def moyenne_vois(T:np.array, x:int, y:int)->list:
    n, p = np.shape(T)[0], np.shape(T)[1]
    D = [(1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), \
    ↪ (-1, 1), (0, 1)] # déplacements élémentaires
    S, nb = 0, 0
    for d in D:
        dx, dy = d[0], d[1]
        a, b = x+dx, y+dy
        if 0 <= a and a < n and 0 <= b and b < p:
            S += T[a, b]
            nb += 1 # nombre de cases voisines
    return S/nb
>>> T = np.array([[1, 2], [3, 4]])
>>> T
array([[1, 2],
       [3, 4]])
>>> moyenne_vois(T, 0, 0)
np.float64(3.0)
>>> moyenne_vois(T, 1, 1)
np.float64(2.0)
```

Solution 4

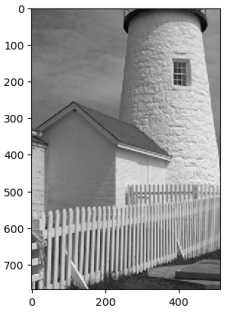
1. Notons d la première distance, et d_1 la seconde. Alors en tenant compte des relations on a :

$$\begin{aligned} d_1 &= \sqrt{\left(x_1 - \frac{n}{2}\right)^2 + \left(y_1 - \frac{p}{2}\right)^2} \\ &= \sqrt{k^2(x - x_c)^2 + k^2(y - y_c)^2} \\ &= k \times \sqrt{(x - x_c)^2 + (y - y_c)^2} = \boxed{k \times d}. \end{aligned}$$

La distance d'origine a bien été multipliée par k, c'est ce que l'on voulait.

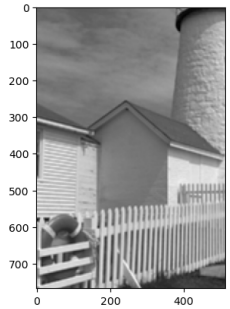
2.
```
Im = plt.imread('lighthouse.png')
plt.figure('gris')
plt.imshow(Im, cmap="gray")
```

```
def zoom1(Im:np.array,x_c:int,y_c:int,k:float)->np.array:
    n, p = np.shape(Im)[0], np.shape(Im)[1]
    Im1 = np.zeros((n, p))
    for x_1 in range(n):
        for y_1 in range(p):
            x = x_c + round((x_1-n/2)/k)
            y = y_c + round((y_1-p/2)/k)
            if (x >= 0 and x < n) and (y >= 0 and y < p):
                Im1[x_1, y_1] = Im[x, y]
    return Im1

Z1 = zoom1(Im, Im.shape[0]//2, Im.shape[1]//2, 2)
plt.figure()
plt.imshow(Z1, cmap ="gray")
```



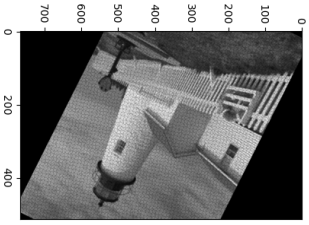
3.

```
Im = plt.imread('lighthouse.png')

def zoom2(Im:np.array,x_c:int,y_c:int,k:float)->np.array:
    n, p = np.shape(Im)[0], np.shape(Im)[1]
    Im1 = np.zeros((n, p))
    for x_1 in range(n):
        for y_1 in range(p):
            x = x_c + round((x_1-n/2)/k)
            y = y_c + round((y_1-n/2)/k)
            if (x > 0 and x < n) and (y > 0 and y < p):
                Im1[x_1, y_1] = moyenne_vois(Im, x, y)
    return Im1

Z2 = zoom2(Im, Im.shape[0]//2, Im.shape[1]//2,2)
plt.figure()
plt.imshow(Z2, cmap ="gray")
```



Solution 5

```
def rotation(Im:np.array, x_c:int,y_c:int,a:float)->np.array:
    n, p = Im.shape[0], Im.shape[1]
    Im1 = np.zeros((n, p))
    for x in range(n):
        for y in range(p):
            x_1 = round(x_c+np.cos(a)*(x-x_c)+np.sin(a)*(y-y_c))
            y_1 = round(y_c-np.sin(a)*(x-x_c)+np.cos(a)*(y-y_c))
            if (x_1 > 0 and x_1 < n) and (y_1 > 0 and y_1 < p):
                Im1[x_1, y_1] = moyenne_vois(Im, x, y)
    return Im1

Im = plt.imread('lighthouse.png')
R = rotation(Im, Im.shape[0]//2, Im.shape[1]//2, 90)
plt.figure()
plt.imshow(R, cmap ="gray")
```


700
600
500
400
300
200
100
0
0
200
400