TP (S1) 2 Traitement d'images

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

Fichier externe ?

OUI fichier `TP_Images.py` *présent dans le dossier 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

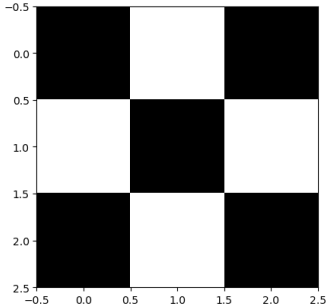
Le pixel est l'élément de base permettant de caractériser la définition d'une image numérique. Une image peut donc être représentée par une matrice T dont chaque terme représente l'état de coloration du pixel correspondant. Par exemple, ci-dessous une image avec 9 pixels [1] :

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]

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, on précise ci-dessous un tableau ainsi que l'image associée.

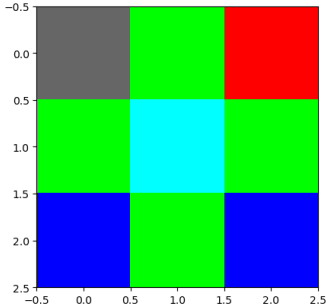
T =	0	1	0
	1	0	1
	0	1	0

codera l'image :



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]

codera l'image :



- 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.

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

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

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]
1
>>> T[0, 1]
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 retourner la moyenne arithhmétique d'un tableau.
```
>>> T
array([[1, 0, 1],
       [0, 1, 0]])
>>> np.mean(T)
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

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

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).

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

Commencez par récupérer et ouvrir le fichier `TPImages.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 Moyenner sur un voisinage [Sol ??]

1) Compléter, dans le code suivant, la définition de la fonction `liste_vois(T:np.array, x:int, y:int)->list` qui retourne la liste des coordonnées des voisins de la position (x, y). On ne renverra bien sûr que des coordonnées dans le champ de `T`. Par exemple,

- si `T` est un tableau de format `(2, 3)`, l'appel `liste_vois(T, 0, 0)` retournera `[(1, 0), (0, 1)]`,
- et l'appel `liste_vois(T, 1, 1)` retournera `[(0, 1), (1, 0), (1, 2), (2, 1)]`.

```
def liste_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
    L = []
    for d in D:
        dx, dy = d[0], d[1]
        a, b = ..........
        if .........:
            L.append((a, b))
    return ..........
```

2) En déduire une fonction `moyenne_vois(T:np.array, x:int, y:int)->float:` qui étant donné un tableau `T` et des coordonnées (x, y) retourne la moyenne des cases voisines de (x, y) dans `T`.

2. OUVRIR, AFFICHER ET CONVERTIR UNE IMAGE

Exercice 2 Afficher une image et extraire des informations [Sol ??]

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 `inverse(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'`

Exercice 3 **Convertir une image couleur en niveau de gris** [Sol **??**]

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,m), zoomer d'un facteur k autour d'un centre C consiste à remplir un tableau (n,m) à l'aide du tableau de taille $(n//k, m//k)$ 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 4 **Zoom sur une image en niveau de gris** [Sol **??**]

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. Interpréter les quantités $x - x_c$, $y - y_c$, $x_1 - \frac{n}{2}$ et $y_1 - \frac{p}{2}$. En déduire que : $\begin{cases} x = x_c + \dfrac{x_1 - \frac{n}{2}}{k} \\ y = y_c + \dfrac{y_1 - \frac{p}{2}}{k} \end{cases}$ où (n,p) désigne la taille commune des deux images, c'est-à-dire `np.shape(Im)` dans la suite.

- 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-n/2)/k)
            if ..............:
                Im1[x_1, y_1] = ..............
    return Im1

P2 = zoom1(Im, Im.shape[0]//2, Im.shape[1]//2,2) #vous pouvez \
↪ choisir un centre différent
plt.figure()
plt.imshow(P2, 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 **??**] 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.

4. TRAITEMENT D'IMAGES PAR CONVOLUTION

Cadre

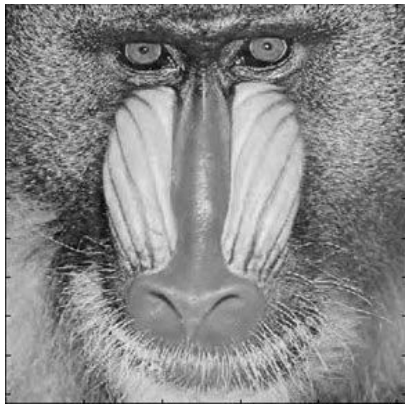
Pour simplifier le code, les images traitées dans cette section seront encore des images en niveau de gris.

Vous trouverez sur le répertoire partagé de la classe une image nommée `Lena.png` que vous copierez dans votre répertoire personnel, et visualiserez sous Python à l'aide des commandes vues à la séance précédente (l'image est en niveaux de gris) :

```
import numpy as np
import matplotlib.pyplot as plt
Im = plt.imread('Lena.png')
plt.figure('Lena.png')
plt.imshow(Im,cmap='gray')
plt.show()
```

Notez que cette image, extrêmement célèbre, a longtemps été utilisée pour illustrer les traitements d'images (pour plus d'information, par exemple l'article wikipédia https://fr.wikipedia.org/wiki/Lenna). Néanmoins, il est difficile de ne pas trouver un certain côté sexiste à cette image, et nous proposons aussi dans le répertoire partagé l'image `mandrill.png` (assez célèbre aussi, mais moins…). Vous pourrez utiliser l'une ou l'autre (ou les deux) de ces deux images par la suite à votre convenance.

Lena.png



mandrill.png

4.1. Convolution d'une image par un masque

- Dans la suite, `A` (plutôt que `Im` pour alléger la présentation) désigne un tableau bidimensionnel contenant les niveaux de gris de chaque pixel d'une image. Pour simplifier le pixel en ligne i et colonne j sera noté $a_{i,j}$ au lieu de `A[i,j]`.
- On considère un autre tableau bidimensionnel `M`, carré et de taille impaire notée $n = 2p+1$, inférieure à celle de `A` (en pratique le nombre de lignes et colonnes de `M` est très petit, souvent égal à 3 - dans ce cas p vaudra 1), et contenant des nombres flottants quelconques. Ce deuxième tableau, appelé le *masque* de la convolution, ne sera pas interprété comme une image dans la suite. Il est impératif de choisir n **impair** pour qu'il y ait bien un pixel central (voir dessin et formule ci-après).
- On appelle *convolution du tableau `A` par le masque `M`* l'image définie par un tableau `B`, de même taille que `A` et vérifiant la formule :

$$b_{i,j} = \sum_{-p\leqslant k,\ell\leqslant p} m_{p+k,p+\ell} \times a_{i+k,j+\ell}\,.$$

Par exemple, si $p = 1$ la formule devient :

$$\begin{aligned} b_{i,j} = {} & m_{0,0} \times a_{i-1,j-1} + m_{0,1} \times a_{i-1,j} + m_{0,2} \times a_{i-1,j+1} \\ & + m_{1,0} \times a_{i,j-1} + m_{1,1} \times a_{i,j} + m_{1,2} \times a_{i,j+1} \\ & + m_{2,0} \times a_{i+1,j-1} + m_{2,1} \times a_{i+1,j} + m_{2,2} \times a_{i+1,j+1} \end{aligned}$$

Une interprétation géométrique de cette formule consiste à imaginer que l'on « centre » le masque sur le pixel courant de l'image `A` et que l'on calcule la somme des pixels à l'intérieur du masque, chaque pixel étant pondéré par la valeur correspondante du masque.

Par exemple, la convolution du masque suivant ci-après

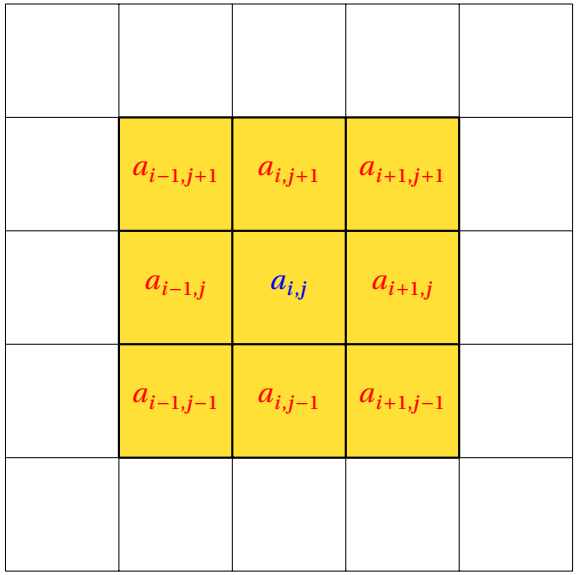
M =

2	-1	3
-4	5	1
0	2	4

produit une image de pixels donnés par :

$$\begin{aligned} b_{i,j} = {} & 2a_{i-1,j-1} - a_{i-1,j} + 3a_{i-1,j+1} \\ & - 4a_{i,j-1} + 5a_{i,j} + a_{i,j+1} \\ & + 0a_{i+1,j-1} + 2a_{i+1,j} + 4a_{i+1,j+1}. \end{aligned}$$

Dans le cas $p = 1$, les pixels de `A` utilisés pour calculer $b_{i,j}$ sont les suivants (colorés en jaune), donc le voisinage de $a_{i,j}$:



Bien sûr, ce calcul ne peut pas se faire sur la bande de p pixels du bord de l'image `A` (car alors la notation $a_{i+k,j+\ell}$ ne correspond pas toujours à un pixel existant). Pour ces pixels on choisira de poser $b_{i,j} = 0$, c'est-à-dire leur donner arbitrairement la couleur noire.

D'autre part, suivant les valeurs placées dans le masque `M`, la matrice `B` calculée par convolution ne contient pas toujours que des valeurs entre 0 et 1 (certains pixels peuvent se retrouver avec des valeurs négatives, ou supérieures à 1). Cela n'empêchera pas d'afficher `B` en tant qu'image, la fonction `plt.imshow` commençant par appliquer une transformation affine sur les valeurs des pixels pour les ramener entre 0 et 1 (ceci de manière transparente pour l'utilisateur).

La fonction principale de convolution vous est donnée dans le fichier externe. La voici.

```
def convolution(A : np.array, M : np.array) -> np.array :
    """ renvoie l'image obtenue par convolution de l'image A
    par le masque M """
    q,r = np.shape(A)[0], np.shape(A)[1]
    p = (np.shape(M)[0]-1)//2
    B = np.zeros((q,r))
    for i in range(p,q-p) :
        for j in range(p,r-p) :
            for k in range(-p,p+1) :
                for l in range(-p,p+1) :
                    B[i,j] += M[p+k,p+l]*A[i+k,j+l]
    return B
```

Une autre version plus élégante de la fonction `convolution` utilisant la possibilité de « slicing » des tableaux :

```
def convolution(A : np.array, M : np.array) -> np.array :
    """ renvoie l'image obtenue par convolution de l'image A
    par le masque M """
    q,r = np.shape(A)[0], np.shape(A)[1]
    p = (np.shape(M)[0]-1)//2
    B = np.zeros((q,r))
    for i in range(p, q-p) :
        for j in range(p, r-p) :
            B[i,j] = np.sum(A[i-p:i+p+1,j-p:j+p+1]*M)
    return B
```

Elle retourne l'image `B` obtenue après convolution par le masque `M`.

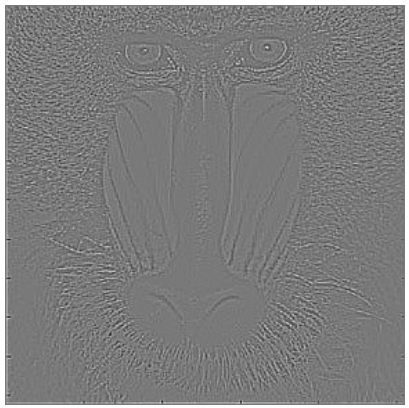
4.2. Effets obtenus par convolution

Certains choix de masques permettent d'obtenir des effets visuels intéressants, nous allons en présenter certains.

4.2.1. Détection de contour

Les contours de l'image (zones où l'intensité lumineuse varie le plus rapidement) peuvent être mis en avant en effectuant une convolution par le masque de taille 3×3 suivant :

$$M = \begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

On constate bien que les contours de l'image initiale apparaissent plus apparents alors que les zones de moindre variation d'intensité lumineuse sont rendues plus homogène (à cause de la petite taille des images reproduites sur votre énoncé papier du TP, et de la qualité limitée des photocopies, vous en trouverez sans doute certaines peu lisible : rassurez-vous, les rendus seront beaucoup plus significatifs à l'écran de votre ordinateur).

Exercice 6 [Sol **??**] Tester la fonction `convolution` en utilisant le masque de contour que l'on pourra définir par la commande `np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1]])` (qui crée un tableau à partir d'une liste de listes, correspondant aux lignes successives).

4.2.2. Effet de lissage

Un effet de « lissage » sur l'image peut s'obtenir par la convolution de l'image par un masque fonctionnant comme un filtre « passe bas », c'est-à-dire réduisant les fortes variations d'intensité lumineuses (contrairement au filtre de contour qui les accentue).

Une première solution consiste à utiliser un filtre dit **moyenneur**, dont toutes cases du masque prennent la même valeur égale à $\frac{1}{n^2}$ où n désigne la taille du masque. Par exemple, pour $n = 3$ le masque sera donc

$$M = \begin{array}{|c|c|c|}\hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline \end{array}$$

Chaque pixel de l'image obtenue par convolution avec ce masque moyenneur sera donc la moyenne de ses n^2 voisins - en incluant lui-même.

Ci-dessous les résultats obtenus pour l'image Lena.png, et pour différentes tailles de masques.


$n = 3$


$n = 7$


$n = 11$

On observe un effet de « flou », d'autant plus prononcé que plus la taille du masque est grande (puisque la moyenne est calculée sur un plus grand nombre de voisins).

Exercice 7 [Sol **??**] Écrire une fonction `masqueFlouM(n : int) -> np.array` renvoyant le masque moyenneur de taille n, puis l'expérimenter pour traiter par convolution les images `Lena.png` et/ou `mandrill.png` pour différentes valeurs de n.

Une autre manière de générer un effet lisseur consiste à utiliser un masque, dit **gaussien**, dont les cases se remplissent en utilisant la formule :

$$\frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

où (x, y) désigne les coordonnées d'un pixel du masque définies **par rapport au pixel central**, et σ un paramètre strictement positif appelé écart-type.

L'idée ici est de calculer la moyenne en donnant plus d'influence au pixel central du masque, et progressivement de moins en moins quand le pixel s'éloigne du centre. Cette importance plus grande pour les pixels proches du centre est accentuée quand l'écart-type σ est choisi petit. Au contraire, si la valeur de sigma augmente alors l'importance des pixels éloignés du centre s'accroît. En pratique, on propose d'imposer la relation suivante entre l'écart-type et la taille du masque : $\boxed{\sigma = n/4}$. Par exemple pour $n = 3$ le masque obtenu est le suivant (les valeurs de chaque case ayant été arrondies à 10^{-2} près) :

$$M = \begin{array}{|c|c|c|}\hline 0.05 & 0.12 & 0.05 \\ \hline 0.12 & 0.28 & 0.12 \\ \hline 0.05 & 0.12 & 0.05 \\ \hline \end{array}$$

Voici les images obtenues par convolution par des masques de flou gaussien de différentes tailles :


$n = 3$


$n = 7$


$n = 11$

Exercice 8 [Sol ??] Écrire une fonction `masqueFlouG(n : int) -> np.array` renvoyant le masque gaussien de taille n, puis l'expérimenter pour traiter par convolution les images Lena.png et/ou mandrill.png pour différentes valeurs de n.

En comparant les images obtenues par lissage moyenneur et lissage gaussien, on constate que le second à tendance à mieux préserver les contours, ce qui apparaît plus nettement pour un masque de taille plus élevé. Voici les images obtenues avec $n = 21$ pour un masque moyenneur à gauche et gaussien à droite :


moyenneur de taille 21


gaussien de taille 21

Si l'effet de flou peut être utilisé pour des raisons artistiques, il peut être aussi envisagé pour réduire le « bruit » dans une image. En effet il arrive qu'une photo soit endommagée de telle sorte qu'une partie importante de ses pixels aient été modifiés. Par exemple, voici ci-dessous une modification de l'image Lena.png dans laquelle 10% des pixels (choisis au hasard) ont été remplacés par des valeurs choisies au hasard en 0 et 1 :

Ci-dessous, vous pouvez observer l'effet de l'application d'un filtre gaussien de taille croissante sur cette image bruitée :


gaussien de taille 3


gaussien de taille 7


gaussien de taille 11

On constate que les pixels bruités ont tendance à s'estomper sous l'effet du filtre, d'autant plus que sa taille est grande.

Exercice 9 [Sol ??] Écrire une fonction `bruit(A : np.array) -> np.array` renvoyant une copie de l'image `A` dans laquelle 10% des pixels ont été remplacés par des valeurs aléatoire. Pour cela, on pourra utiliser la fonction `randrange(a,b)`, où a et b sont deux entiers tels que $a < b$, renvoyant un **entier** choisi au hasard uniformément parmi les valeurs $a, a+1, \dots, b-2, b-1$, et la fonction `random` renvoyant un **flottant** choisi au hasard uniformément entre 0 et 1. Ces deux fonctions sont à importer du module `random`. Utiliser cette fonction pour obtenir une version bruitée des images `Lena.png` et/ou `mandrill.png`, et lui appliquer une filtre gaussien d'une taille que vous choisirez.

4.2.3. Amélioration de la netteté

Pour compenser la perte de netteté observée à l'étape précédente, on peut utiliser une nouvelle convolution par un masque dont l'effet est de renforcer les contours en augmentant le contraste. Le filtre dit de **différence** a cet effet, et son masque de taille n est défini en donnant la valeur 2 à la case centrale et la valeur $-\frac{1}{n^2-1}$ aux autres cases. L'effet de ce masque est d'ajouter au pixel central la moyenne de ses écarts aux autres pixels, donc conduit à éclaircir celui-ci s'il est déjà plus clair en moyenne que ses voisins et à l'assombrir sinon, ce qui augmente bien le contraste de l'image. Par exemple le masque différence de taille 3 est :

$$\mathrm{M} = \begin{array}{|c|c|c|}\hline -1/8 & -1/8 & -1/8 \\ \hline -1/8 & 2 & -1/8 \\ \hline -1/8 & -1/8 & -1/8 \\ \hline \end{array}$$

Voici l'effet obtenu pour une taille croissante sur l'image Lena.png initiale :


différence de taille 3


différence de taille 7


différence de taille 11

Maintenant, voici l'effet obtenu en appliquant à l'image initiale successivement un filtre gaussien de taille 11 puis le filtre différence de taille 11 :


image initiale


gaussien de taille 11


puis différence de taille 11

On constate bien un regain en netteté (mais pas jusqu'à retrouver celle de l'image initiale...). Enfin, voici le même traitement sur l'image bruitée :


image bruitée


gaussien de taille 11


puis différence de taille 11

Exercice 10 [Sol **??**] Écrire une fonction `masqueDifference(n : int) -> np.array` renvoyant le masque de différence de taille n. Expérimenter cette fonction sur différentes images, bruitées ou non, et associée ou non à un filtre de flou.

4.2.4. Filtre médian

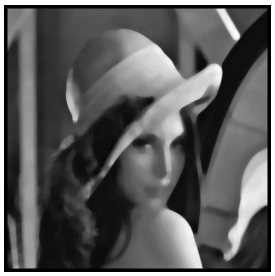
Pour éliminer le bruit dans une image, il existe en fait un autre traitement de lissage, dit **filtre médian**, généralement plus efficace. Il ne s'agit pas ici d'une convolution par un masque, même si le principe s'en rapproche. On choisit une taille, comme pour un masque mais sans définir des coefficients dans les cases, et on centre de nouveau cette zone sur un pixel de l'image initiale. Le principe est alors de définir la valeur du pixel de l'image résultat comme étant la **médiane** des valeurs des pixels contenus dans cette zone (au lieu de la **moyenne** pour le filtre du même nom). L'intérêt est que la médiane est globalement moins sensible que la moyenne aux variations de quelques pixels isolés.

On rappelle que la médiane d'une liste de valeurs est définie comme valeur « centrale » de cette liste quand elle est triée dans l'ordre croissant ou décroissant, peu importe (il y a alors autant de pixels ayant une valeur inférieure à cette médiane que de valeur supérieure).

Voici l'effet obtenu pour une taille croissante sur l'image `Lena.png` initiale :


médian de taille 3


médian de taille 7


médian de taille 11

On constate, sans surprise, un effet de flou assez similaire à ceux des filtres de moyenne ou gaussien. Par contre, la différence est sensible sur une image bruitée :


image bruitée


médian taille 3


médian taille 7

On constate qu'un filtre médian de taille 3 suffit à éliminer la presque totalité du bruit, avec une perte très limitée de netteté. Par contre, appliquer un filtre médian de taille supérieur ne fait plus qu'augmenter le flou sans gain au niveau du bruit.

Exercice 11 [Sol **??**] Écrire une fonction `filtreMedian(A : np.array, n : int) -> np.array` renvoyant l'image obtenue à partir de `A` en appliquant un filtre médian de taille n. Expérimenter cette fonction sur différentes images, en particulier bruitées bien sûr. *On pourra se servir de la fonction `np.sort` permettant de trier certains tableaux dans l'ordre croissant*

À retenir

Les différents filtres présentés dans ce TP ne sont bien sûr pas à connaître, mais vous retiendrez le principe de stockage des images sous la forme de tableaux bi-dimensionnels de pixels, ainsi que la manière de travailler sur ces derniers (parcours boucles for imbriqués, « slicing », …).

SOLUTIONS DES EXERCICES

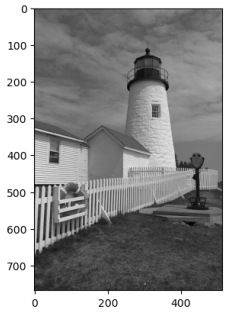
Solution ??

1)
```python
def liste_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
    L = []
    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:
            L.append((a, b))
    return L
```

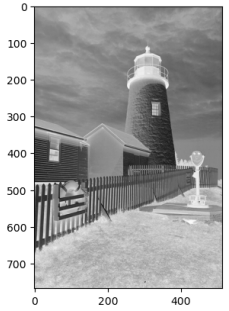
2)
```python
def moyenne_vois(T:np.array, x:int, y:int)->float:
    moy = 0
    L = liste_vois(T, x, y)
    for c in L:
        moy += T[c[0], c[1]]
    return moy/len(L)
```

Solution ??

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.
2. 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.
```python
import numpy as np
import matplotlib.pyplot as plt

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

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

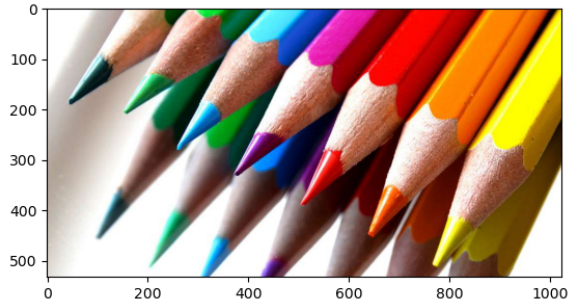



```python
def inverse(Im):
    n, p = np.shape(Im)[0], np.shape(Im)[1]
    Imneg = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            Imneg[i,j] = 1-Im[i,j]
    return Imneg

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



Solution ??

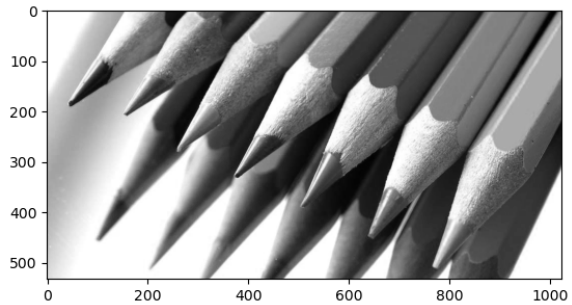
1)
```python
plt.figure('couleurs')
Im2 = plt.imread('crayons.jpg')
np.shape(Im2) # Dimension du tableau
Im2[200, 200]  # Contenu d'un élément du tableau
plt.imshow(Im2)
```



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` :

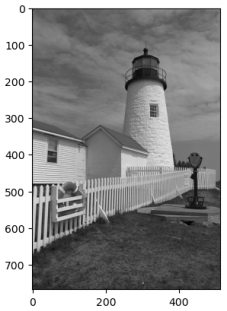
```
>>> (Im2[200,200][0]+Im2[200,200][1]+Im2[200,200][2])/3
<input>:1: RuntimeWarning: overflow encountered in scalar add
77.33333333333333
```

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')
```

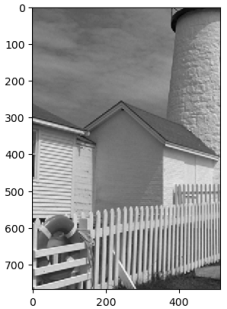


Solution ??

1.
```
Im = plt.imread('lighthouse.png')
plt.figure('gris')
plt.imshow(Im, cmap="gray")
```



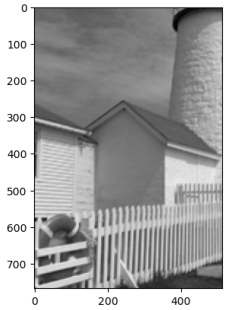
```
def zoom1(Im:np.array,x_c:int,y_c:int,k:float)->np.array:
    n, p = np.shape(Im)[0], np.shape(Im)[1]
    Im1 = np.zeros((n, p))
    for x_1 in range(n):
        for y_1 in range(p):
            x = x_c + round((x_1-n/2)/k)
            y = y_c + round((y_1-n/2)/k)
            if (x > 0 and x < n)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



2. Le pixel de coordonnées $(n/2, p/2)$ (dans la nouvelle image) a pour antécédent (x_c, y_c), le centre du zoom (dans l'ancienne image).
Le zoom multiplie les distances entre le centre et les autres pixels par un facteur k. On a donc : $n/2 - x_1 = k(x_c - x)$, $p/2 - y_1 = k(y_c - y)$. Ce qui conduit à la

relation demandée.

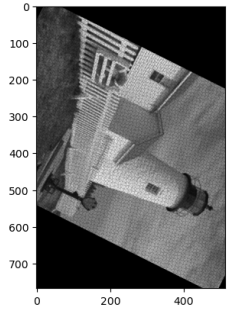
3.
```
Im = plt.imread('lighthouse.png')

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 ??

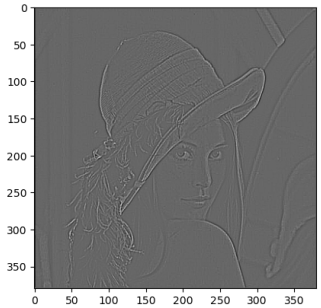
1)
```
def rotation(Im:np.array, x_c:int,y_c:int,a:float)->np.array:
    n, p = Im.shape[0], Im.shape[1]
    Im1 = np.zeros((n, p))
    for x in range(n):
        for y in range(p):
            x_1 = round(x_c+np.cos(a)*(x-x_c)+np.sin(a)*(y-y_c))
            y_1 = round(y_c-np.sin(a)*(x-x_c)+np.cos(a)*(y-y_c))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



Solution ??

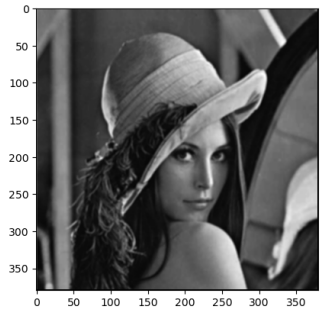
```
masqueC = np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1]])
Im = plt.imread('Lena.png')
ImC = convolution(Im, masqueC)
plt.figure('Lena contour')
plt.imshow(ImC,cmap='gray')
```



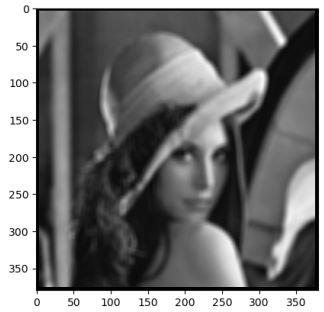
Solution ??

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

def masqueFlouM(n : int) -> np.array :
    """
    renvoie le masque moyenneur de taille n
```

```
    """
    M = 1/n**2*np.ones((n, n))
    return M

ImE = convolution(Im, masqueFlouM(3))
plt.figure('Lena flou moyenneur taille 3')
plt.imshow(ImE,cmap='gray')
```

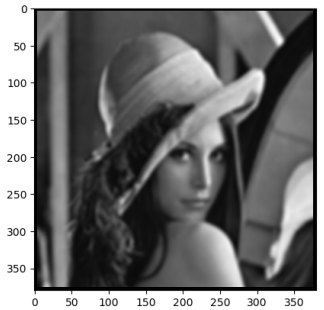


```
ImE = convolution(Im, masqueFlouM(10))
plt.figure('Lena flou moyenneur taille 10')
plt.imshow(ImE,cmap='gray')
```



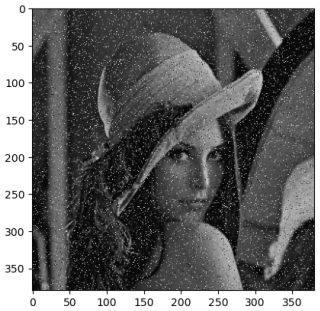
Solution ??

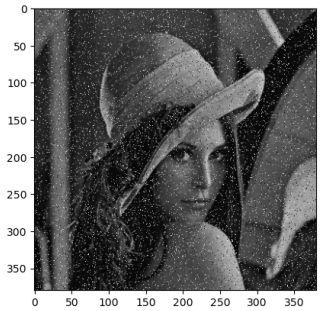
```
def masqueFlouG(n : int) -> np.array :
    """ renvoie le masque gaussien de taille n """
    p = (n-1)//2
    M = np.zeros((n,n))
    s = n/4
    p = (n-1)/2
    for i in range(n) :
        for j in range(n) :
            x, y = i-p, j-p
            M[i, j] = \
            ↪ 1/(2*np.pi*s**2)*np.exp(-(x**2+y**2)/(2*s**2))
    return M
ImG = convolution(Im, masqueFlouG(10))
plt.figure('Lena flou gaussien taille 10')
plt.imshow(ImG, cmap='gray')
```



Solution ??

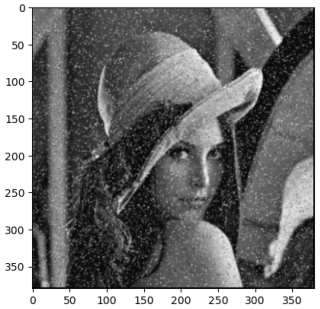
```
from random import randrange,random

def bruit(A : np.array) -> np.array :
    """ renvoie l'image obtenue en ajoutant un bruit aléatoire
    à l'image A """
    n, p = np.shape(A)
    B = np.copy(A)
    for i in range(n):
        for j in range(p):
            if randrange(10) == 0:
                B[i,j] = random()
    return B

ImB = bruit(Im)
plt.figure('Lena bruitée')
plt.imshow(ImB,cmap='gray')
```

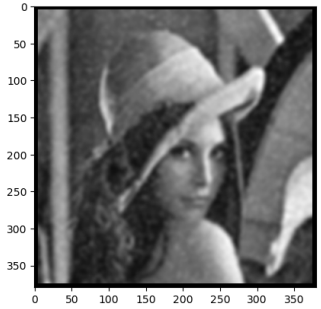




```
ImE = convolution(ImB, masqueFlouG(3))
plt.figure('Lena bruit+flouGaussien3')
plt.imshow(ImE, cmap='gray')
```

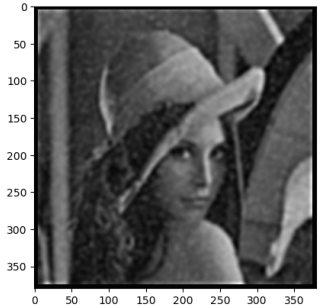
```
ImF = convolution(ImB, masqueFlouG(11))
plt.figure('Lena bruit+flouGaussien11')
plt.imshow(ImF,cmap='gray')
```





Solution ??

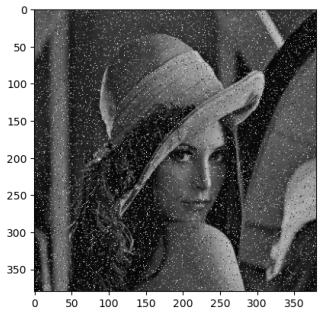
```
ImFD = convolution(ImF, masqueDifference(11))
plt.figure('Lena bruit+flouGaussien11+Différence11')
plt.imshow(ImFD,cmap='gray')
```



```
def masqueDifference(n : int) -> np.array :
    """ renvoie le masque différence de taille n """
    M = -1/(n**2-1)*np.ones((n,n))
    p = (n-1)//2
    M[p,p] = 2
    return M

ImB = bruit(Im)
plt.figure('Lena bruitée')
plt.imshow(ImB,cmap='gray')
```

Solution ?? On décide ici de créer une fonction intermédiaire transformant un tableau bi-dimensionnel en un tableau unidimensionnel, sur lequel on applique la fonction `sort` du module `numpy` afin de le trier (on aurait pu aussi reprogrammer cette fonction par l'un des algorithmes vus au TP précédent).

```python
def conv2D1D(M : np.array) -> np.array :
    """ convertit un tableau bidimensionnel
    en un tableau unidimensionnel ayant les
    mêmes cases """
    n, p = np.shape(M)[0], np.shape(M)[1]
    T = np.zeros(n*p)
    for i in range(n) :
        for j in range(p) :
            T[i*p+j] = M[i,j]
    return T

def filtreMedian(A : np.array, n : int) -> np.array :
    """ renvoie l'image obtenue par filtrage median
    de taille n de l'image A """
    q,r = np.shape(A)[0], np.shape(A)[1]
    p = (n-1)//2
    m = (n**2-1)//2
    B = np.zeros((q, r))
    for i in range(p,q-p) :
        for j in range(p,r-p) :
            T = conv2D1D(A[i-p:i+p+1,j-p:j+p+1])
            Ttrie = np.sort(T)
            B[i,j] = Ttrie[m]
    return B

ImB = bruit(Im)
plt.figure('Lena bruitée')
plt.imshow(ImB,cmap='gray')
```




```python
ImM = filtreMedian(ImB,3)
plt.figure('Lena bruit + médian 3')
plt.imshow(ImM, cmap='gray')
```