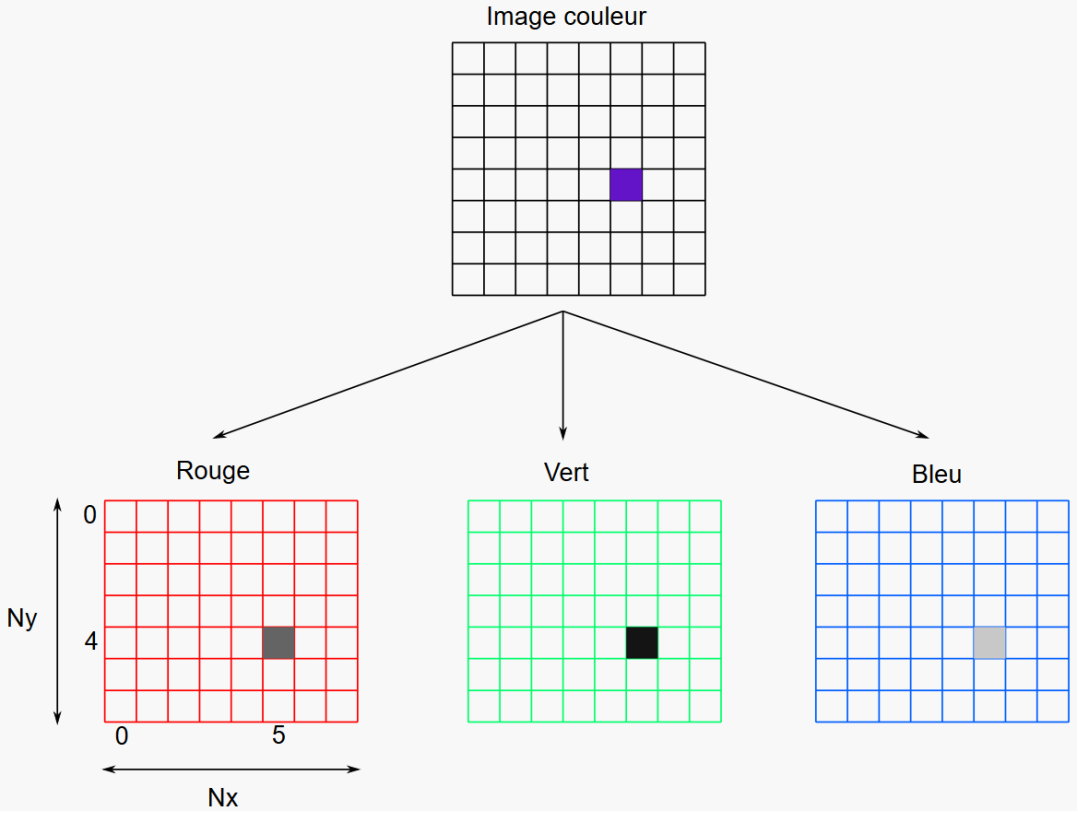
# TD 15 – Lecture et filtrage des images

## A / Contexte

Les données collectées par des caméras ou appareils photos font souvent l'objet de traitements numériques, afin notamment d'aider à la reconnaissance ou au suivi de formes sur les images. On pensera notamment au suivi de motifs (pointage vidéo automatisé), à la reconnaissance de formes (comptage sur une ligne de production, reconnaissance faciale, …) à la retouche d'image automatique (reconnaissance de contours et découpage, filtres type *Instagram*), …

En sciences industrielles, le traitement d'image doit souvent se faire en temps réel, ou très légèrement différé. Exemple : traitement des mesures d'une caméra embarquée. Afin d'alléger les algorithmes, les concepteurs préfèreront généralement travailler en niveaux de gris, même si la caméra est couleur. Prenons par exemple une caméra couleur RVB (Rouge Vert Bleu) en 8 bits :

En couleur, une image correspond à 3 matrices : une matrice des niveaux de rouge, une matrice des niveaux de vert et une matrice des niveaux de bleu.




Chacune des 3 matrices R, V, B, a la même dimension que la matrice de l'image en couleur, par exemple 768 x 1024 pixels (donc 768 lignes, 1024 colonnes), et chaque point de la matrice contient une valeur entre 0 et 255 (s'il s'agit d'une image 8 bits). Ex : violet sur un pixel de l'image en couleur = 128 rouge, 0 vert, 128 bleu.

**Remarque** : il est classique de voir qu'une image en RVB soit traduite comme quatre matrices, et non trois. La **quatrième** est alors la matrice des niveaux de transparence de l'image.

0 → pixel transparent, quelle que soit sa couleur,

255 → pixel opaque, on ne voit pas ce qui est derrière si l'image est superposée à une autre.

En niveau de gris, une seule matrice suffit pour décrire l'image : 0 → noir, 255 → blanc, et tous les états intermédiaires sont des niveaux de gris, foncés si proches de 0, clairs si proches de 255.

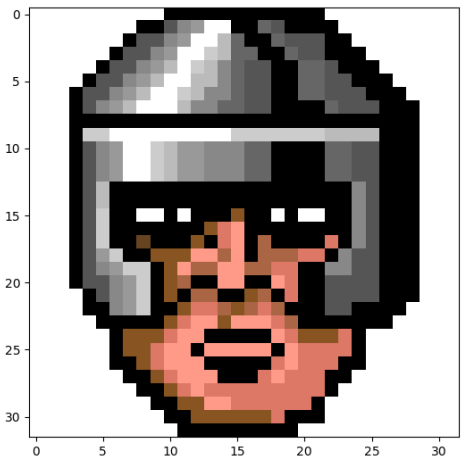
## B / Passage en niveau de gris et seuil

Afin de passer une image RVB en niveaux de gris, nous allons décomposer l'image initiale en 4 calques : un calque de niveaux de rouge, un de vert, un de bleu, et un de niveau d'opacité (que nous fixerons par la suite = 1).


Comme nous l'avons vu en cours, notre algorithme pour créer une image en niveaux de gris à partir d'une image en couleurs est le suivant :

1. Détecter le nombre de lignes et de colonnes de l'image initiale (*N* lignes, *M* colonnes, correspondant au nombre de pixels, donc 512 et 512 dans l'exemple ci-dessus).
2. Créer trois matrices *N* x *M*, $(R_{i,j})_{\substack{0\leq i\leq N-1\\0\leq j\leq M-1}}$ correspondant aux niveaux de rouge, $(V_{i,j})_{\substack{0\leq i\leq N-1\\0\leq j\leq M-1}}$ pour le vert et $(B_{i,j})_{\substack{0\leq i\leq N-1\\0\leq j\leq M-1}}$ pour le bleu.
3. Créer une matrice $(G_{i,j})_{\substack{0\leq i\leq N-1\\0\leq j\leq M-1}}$ vierge de même dimension *N* x *M*, pour laquelle, pour chaque pixel [i, j], le niveau de gris sera calculé comme la moyenne des trois autres niveaux. Donc

$$G_{i,j} = \frac{R_{i,j} + V_{i,j} + B_{i,j}}{3}$$




Soit une image en couleur, de petite taille 32 x 32 pixels. Exactement comme ce que nous avons vu sur les fichiers texte, une image doit être **enregistrée** dans **le même dossier** que celui où est enregistré et exécuté votre code *Python* (ce qui nous épargne d'avoir à donner l'adresse absolue).

**Q1 –** Enregistrer l'image *"test.png"* dans le même dossier que là où se trouve votre script *Python*. Taper alors les commandes :

```
import matplotlib.pyplot as plt
import numpy as np
image_test = plt.imread("test.png") # interprétation comme une matrice
image_test = np.array(image_test).astype(float) # conversion en np.array (matrice)
plt.imshow(image_test)    # commande permettant d'afficher l'image (en couleurs)
```

* * *

**Remarque d'ouverture pour la culture :**

Dans cette première partie du sujet, contrairement au cours et à la seconde partie du sujet, nous travaillons avec un fichier dont l'extension est *.png* (**Portable Network Graphics** , **PNG**, prononcé « ping »).

Si vous tapez la commande :

```
print(np.shape(image_test))
```

l'afficheur doit donner `(32, 32, 4)`. Le premier « 32 » désigne le nombre de lignes de l'image, le second « 32 » désigne le nombre de colonnes. Mais en cours, nous avions vu qu'il devait y avoir 3 niveaux : *R*, *V* et *B*. En fait, l'extension *.png* permet d'ajouter un quatrième calque qui est le niveau de transparence. Pour les pixels [i,j] où ce niveau vaut 0, c'est qu'il n'y a aucune transparence, mais le niveau de transparence peut être compris entre 0 et 1 (totalement transparent). Ceci permet par exemple de faire des logos, des fondus, etc. Notamment, on pourrait, sur cette image :

- Garder les yeux blancs, avec un niveau de transparence 0
- Mais mettre un niveau de transparence 1 à tous les autres pixels blancs, qui sont le « fond ».

Pour une image matricielle dont l'extension serait *.bmp*, il n'y a que 3 niveaux *R, V*, et *B*, pas de niveau de transparence, c'est ce qui distingue de l'extension *.png.* Cette dernière est notamment utilisée pour les insertions de logos (éventuellement en niveaux de transparence), comme maladroitement / ironiquement illustré ci-dessous :

***Insertion 1** : Ajout de la tête sur la photo du lycée, avec niveaux transparence du fond blanc = 1*
***Insertion 2** : Ajout sur la nouvelle photo obtenue (où la tête cache le poteau), de la photo du lycée, avec niveaux transparence = 1 partout sauf = 0 sur le poteau (identifié comme forme parallélipédique)*

Pour la suite, vous l'aurez compris, seuls les trois premiers niveaux de l'image nous intéressent, on fera donc fi (eh oui) du quatrième calque qui correspond aux niveaux de transparence.

* * *

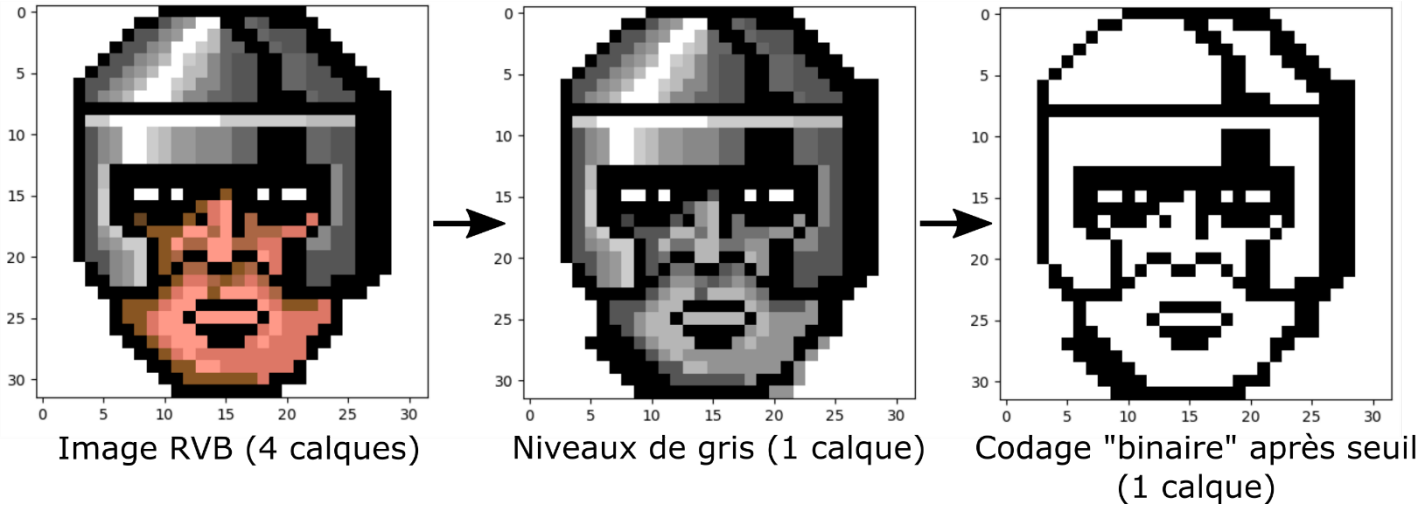
**Q2 –** Écrire une fonction `niv_gris(image)` prenant en argument une matrice représentant une image (qu'elle soit en *.bmp* ou en *.png*, on ne prendra que les 3 premiers calques) et renvoyant une matrice comptant autant de pixels correspondant à l'image en niveaux de gris, quelle que soit la dimension initiale de l'image.

**Test :**

```
image_gris = niv_gris(image_test)
plt.imshow(image_gris, cmap = 'gray')
# affiche l'image en niveaux de gris.
Il existe d'autres couleurs ('ocean', 'hot'…)
print(np.shape(image_gris)) # affiche (32,32), il n'y a qu'un seul calque
```

Nous repartirons désormais de la fonction **niv_gris** obtenue en **Q2**. Nous souhaitons désormais appliquer un seuil à cette image, afin de coder cette image en binaire : noir, ou blanc. Pour cela, il s'agit de tester la valeur de chaque niveau de gris de pixel, et appliquer la règle suivante après avoir défini un seuil compris entre 0 et 1.

- Si sa valeur est ≤ seuil, alors le pixel est fixé à **0 (noir)**
- Sinon, le pixel est fixé à **1 (blanc).**




**Q3 –** Écrire une fonction `NB(imgG, val_seuil)`, prenant pour arguments une image en niveaux de gris et un seuil compris entre 0 et 1, et renvoyant une image de même dimension que l'image d'entrée `imgG`, mais seuillée.

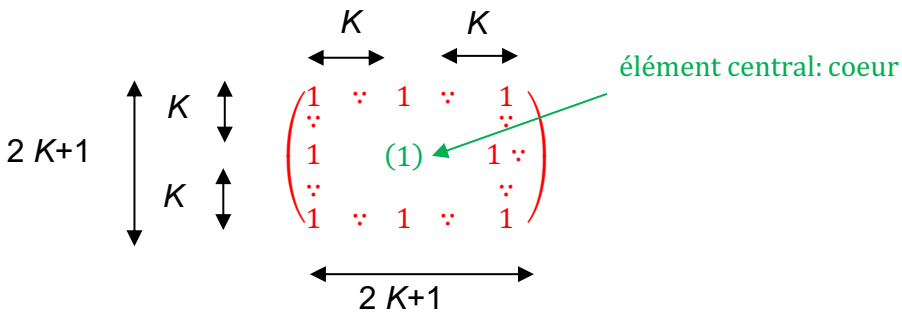
**Tests** :
```
image_NB1 = NB(image_gris, 0.1)
image_NB9 = NB(image_gris, 0.9)
plt.imshow(image_NB1, cmap='gray')   # affiche :
```

```
plt.imshow(image_NB9, cmap= 'gray')   # affiche :
```

## C / Flou variable

La plupart des opérations de filtrage utilisées en traitement d'image numérique sont réalisées à l'aide d'opérateurs matriciels appliqués à chaque pixel, appelés filtres. Nous allons ici voir l'exemple simple du « flou », opéré par une moyenne glissante sur l'image.

Pour flouter une image en niveaux de gris, nous allons **pour chaque pixel (i, j)** créer une fenêtre centrée sur ce pixel, et calculer la moyenne des niveaux de gris sur cette fenêtre. La valeur de la moyenne sera alors affectée au pixel central. Nous noterons $K$ la taille de la demi-fenêtre, donc que la moyenne sera réalisée en incluant tous les pixels voisins du pixel (i, j) situés à une distance ≤ $K$ de ce pixel central, c'est-à-dire que la fenêtre sera carrée de côté $2K + 1$ pixels, centrée sur (i, j). Le filtre est donc :



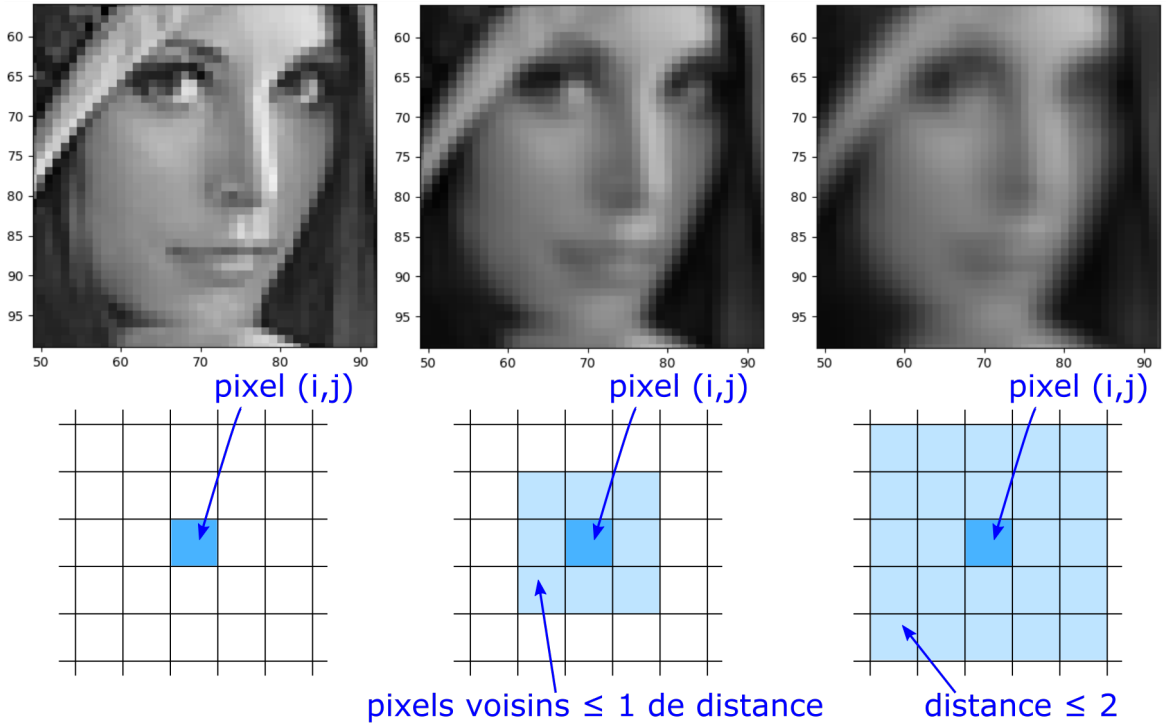
$K = 0$ | $K = 1$ | $K = 2$

$(1)$

(pas de filtrage)

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & \mathbf{1} & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \mathbf{1} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$




Moyenne sur 1 élément ⟶ image initiale | Moyenne sur 9 éléments | Moyenne sur 25 éléments

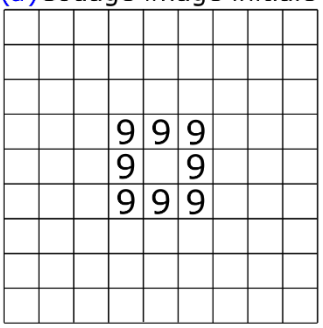
**Remarque** : la moyenne est égale à la somme des niveaux de gris de tous les pixels de la fenêtre, divisée par le nombre de pixels contenus dans la fenêtre, c'est-à-dire $(2K+1)^2$.

Soit, ci-contre, une image où nous supposerons que les niveaux de gris sont codés sur 10 niveaux (de 0 = blanc à 9 = noir).

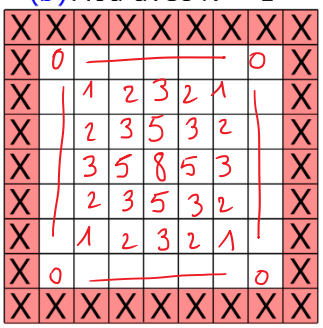
Comme nous l'avons vu en cours, l'application d'un flou avec une taille *K* = 1 fait :

- Perdre les bords (*K* lignes ou colonnes à chaque bord)
- Et génère une image « floutée »

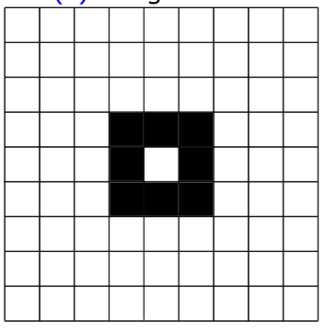
(*a*) Codage image initiale



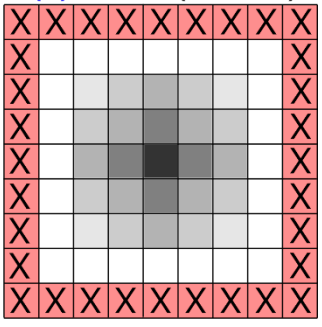

(*b*) Flou avec N = 1




(*c*) Image initiale



(*d*) Floutée (1 voisin)



Contrairement au cours, où nous avions codé une convolution entre un filtre et une sous-matrice extraite de l'image autour de chaque pixel, nous allons ici, pour chaque pixel (i, j), simplement calculer la moyenne des pixel voisins (verticalement et horizontalement) distants de $K$ ou moins.

Pour balayer tous les pixels centraux (i, j), il faut 2 boucles *for* imbriquées

→ Mais pour chacun des couples (i, j) ainsi balayer, il faut balayer la fenêtre glissante, en vertical et horizontal.

→ Nous partirons donc sur 4 boucles for imbriquées.

Je vous conseille vivement de revoir l'exercice de la moyenne glissante (TD 6 sur les listes). Le flou n'est qu'une version 2D de la moyenne glissante (qui est 1D).

Soit une image initiale, en niveaux de gris, de taille $N \times M$ sur laquelle on souhaite appliquer un flou de taille $2K + 1$.

**Q4 –** ⌨ Avant de commencer à coder, déterminer sur feuille les bornes des quatre boucles *for* :

Pour *i* allant de _____ à _____

Pour *j* allant de _____ à _____

Pour *k1* allant de _____ à _____

Pour *k2* allant de _____ à _____

→ *On veut calculer la moyenne (donc la somme) des* `img[i + k1, j + k2]`

*(qu'on voudra alors affecter à l'*`image_sortie[i,j]` *)*

**Remarque** : ne pas faire attention au niveau d'indentation de ce qui est écrit en noir.

**Q5 –** ⌨ Écrire alors une fonction `flou(img, K)` prenant en argument une image en **couleur** et un entier *K* correspondant à la taille de la demi-fenêtre. Cette fonction renverra une matrice des niveaux de gris floutée par cette fenêtre glissante.

**Convention :** pour éviter décalages d'index, on se facilite la vie en imposant ici que l'image de sortie a **la même taille** que l'image en niveaux de gris bords. Les bords (sur lesquels on ne peut pas appliquer la moyenne glissante) sont simplement fixés à blanc (valeur 1).

**Aide** : plutôt que d'initialiser l'image de sortie à `np.zeros([N, M])` vous la fixerez à `255*np.ones([N, M])`. La fonction `np.ones([N, M])` renvoie une matrice de taille *N* x *M* remplie de 1. Ainsi, les bords valent ainsi déjà 255 (l'image étant sur 8 bits, les pixels ne sont pas compris entre 0 et 1 mais entre 0 – noir et 255 – blanc).

**Test :** Télécharger et déposer dans le dossier courant la photo *"lenna.bmp"*

```
Photo = np.array(plt.imread("lenna.bmp")).astype(float)
Photo_ext = Photo[::4, ::4, :]  # on ne garde qu'1 pixel sur 4, pour alléger
print( np.shape(Photo_ext) )  # Affiche (166, 320, 3) : couleur, 3 calques

non_floutee = flou(Photo_ext, 0)  # K = 0 : on convertit en niv. de gris mais
print(np.shape(non_floutee)) # on n'applique aucun flou : affiche (166, 320)
plt.imshow(non_floutee, cmap = "gray")   # affichera l'image en niveaux gris

Photo_floue = flou(Photo_ext,10)    # bon gros flou.
print(np.shape(Photo_floue))  # Affiche (166, 320) puisqu'on ne modifie pas
                              # la taille de l'image (on garde les bords blancs)
plt.imshow(Photo_floue, cmap = "gray")   # Affichera l'image très floue
```

**Remarque :** vous remarquerez que le calcul est assez long… c'est pour cela qu'on a sous-échantillonné l'image en ne prenant qu'un pixel sur 4 (on perd donc ¾ de l'information) ! Mais, juste pour voir, essayez de calculer la photo floue sur la photo complète (`Photo` plutôt que `Photo_ext`)… → le temps de calcul est multiplié par… 16 !!! Ceci nous emmène vers la complexité des algorithmes, que nous aborderons peu après la rentrée. Bonnes vacances à vous !