Informatique Pour Tous

Cours 15 – Lecture et filtrage des images

A / Codage d'une image matricielle

A.1 / Représentation matricielle d'une image en niveaux de gris

Supposons une image matricielle `Photo` en niveaux de gris, représentée par une matrice *Numpy* :

Remarque : Pour convertir un fichier.bmp (*BitMaP)* contenant une photo, en une matrice (type array), il faut utiliser les commandes suivantes (cf. TD 15) :

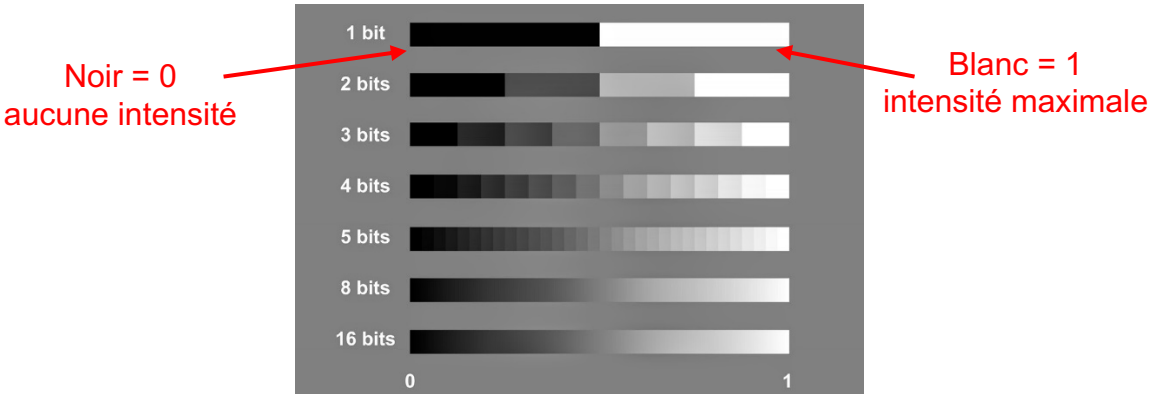
```
import matplotlib.pyplot as plt
Photo = plt.imread("fichier.bmp")
```

Selon le type de codage (8 bits, 16 bits, 24 ou 32 bits), on dispose de plus ou moins de « profondeur » c'est-à-dire de niveaux de gris différents. Exemples, lors de la **rastérisation** (conversion en image matricielle) d'un gradient continu :

```
# Pour récupérer le nombre de lignes et de colonnes de cette matrice, on tape :
Nlig, Ncol = np.shape(Photo)

# Cette image est composée d'un certain nombre de pixels Photo[i,j], au nombre de :
Npixels = np.size(Photo)  # ou Nlig * Ncol
```

Selon le type de codage (8 bits, 16 bits, 24 ou 32 bits), on dispose de plus ou moins de « profondeur » c'est-à-dire de niveaux de gris différents. Exemples, lors de la **rastérisation** (conversion en image matricielle) d'un gradient continu :



Source : Bit – Depth illustration – The Working Man. 2014.

Remarque : Sur une image 8 bits, le niveau de chaque pixel peut être compris entre 0 et 255 ($2^8 = 256$).

Comme les images n'ont pas forcément toutes le même niveau de rastérisation, on préfèrera les **normaliser** pour les traiter informatiquement. On préfère donc travailler avec la matrice $\frac{\text{Photo}}{\max_{i,j} \text{Photo}[i,j]}$ dont les pixels sont compris entre :

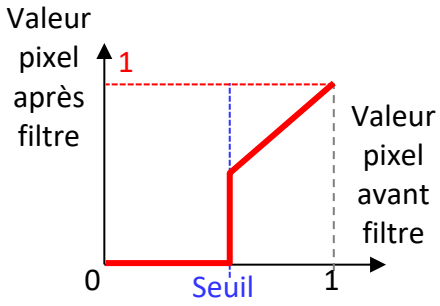
- le noir (aucune intensité d'affichage du pixel) correspond au niveau 0
- et le blanc (intensité maximale) correspond au niveau 1

Citons notamment le cas extrême d'une image en noir et blanc, codée sur 2 niveaux : 1 bit : 0 ou 1

Remarque : en traitement d'images, on peut parfois volontairement (détection de motifs ou de contours, élimination ou floutage de fond, …) vouloir :

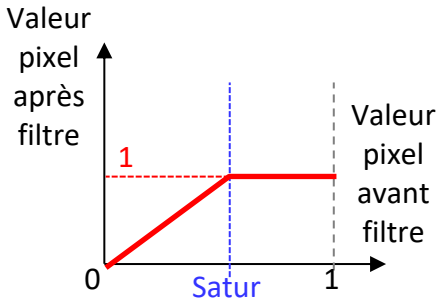
- **Saturer** une partie de l'image
  - **Seuil** (par le bas) : tous les pixels en-dessous d'un certain niveau `Seuil` sont passés noir

```
def Satur_basse(Image, Seuil) :
    N, M = np.shape(Image)
    Im_sortie = Image.copy()
    for i in range(N) :
        for j in range(M) :
            if Im_sortie[i,j] < Seuil :
                Im_sortie[i,j] = 0
    return Im_sortie
```



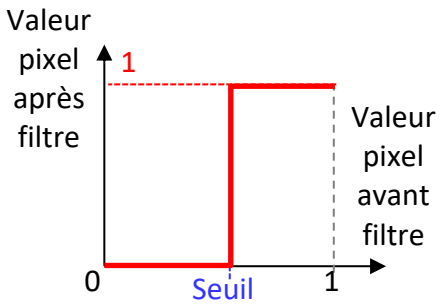
  - **Saturation** (par le haut) : tous les pixels au-dessus d'un certain niveau `Satur` sont passés blancs

```
def Satur_haute(Image, Satur) :
    N, M = np.shape(Image)
    Im_sortie = Image.copy()
    for i in range(N) :
        for j in range(M) :
            if Im_sortie[i,j] > Satur :
                Im_sortie[i,j] = 1
    return Im_sortie
```



- **Seuiller** l'image : tous les pixels en-dessous du `Seuil` sont passés noir, les autres sont passés blancs. 2 bits nécessaires seulement.

```
def Seuillage(Image, Seuil) :
    N, M = np.shape(Image)
    Im_sortie = np.zeros([N,M]).astype(bool)
    for i in range(N) :
        for j in range(M) :
            if Im_sortie < Seuil :
                Im_sortie[i,j] = 0
            else :
                Im_sortie[i,j] = 1
    return Im_sortie
```



Remarque : nous avons vu que pour deux objets *Numpy.array*, `M1*M2` est un produit terme à terme.

En pratique, seule la fonction `Seuillage()` suffit ! En effet, pour une `Photo` donnée, si nous n'avons pas codé la fonction `Satur_basse()`, on peut obtenir la même chose en tapant (prenons par exemple un seuil de 0.4) :

```
Seuillage(Photo,0.4) * Photo
```

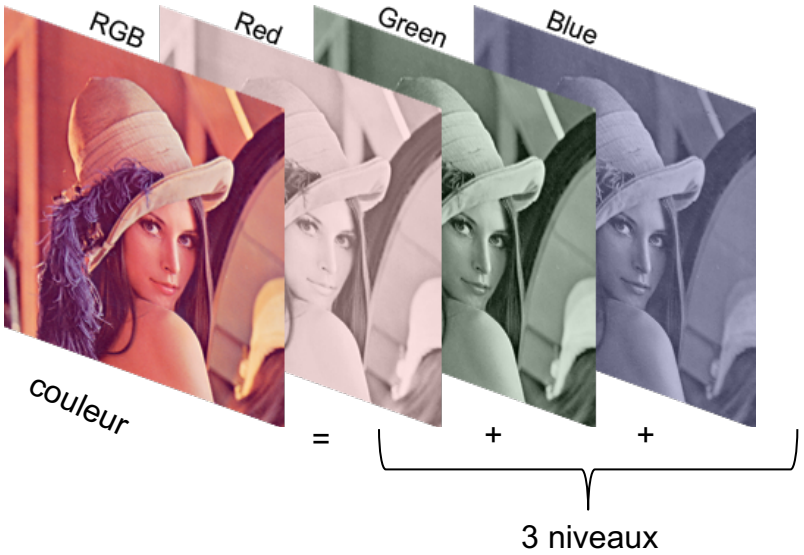
$$\forall\, pixel : \left\{ \begin{array}{c} si\ pixel < seuil, 0 * pixel \rightarrow 0 \\ si\ pixel > seuil, 1 * pixel \rightarrow pixel \end{array} \right\}$$

On parle alors de masque appliqué à l'image. Le complémentaire est obtenu avec :

```
complementaire = np.logical_not(Seuillage(Photo,0.4)).
```

A.2/ Représentation matricielle d'une image en couleurs : matrice à « 3 dimensions »

Une image matricielle en couleurs se décompose en niveaux de chacune de ces couleurs. La décomposition classique est en niveaux de Rouge, Vert et Bleu (on parle de décomposition RVB) :

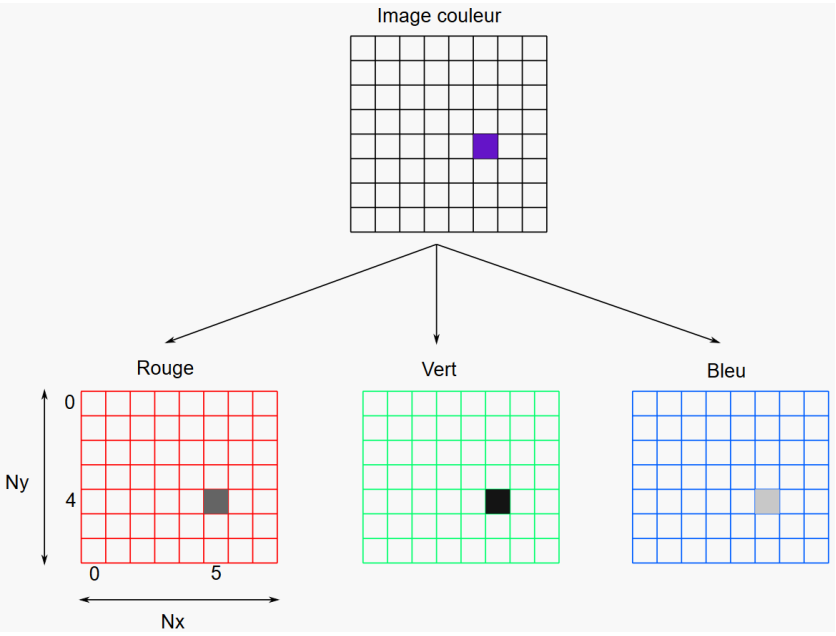



```
# Ainsi, pour une Photo 1024 * 748 en format paysage (largeur = (nombre de colonnes)
* (hauteur = nombres de lignes)), la commande
print( np.shape(Photo) )    # affiche : 748, 1024, 3,
                            # 3 matrices de 748 lignes et 1024 colonnes
# Il faut donc comprendre que chaque pixel de cette image a trois niveaux ! Ex :

print( Photo[10, -2] )      # affiche : array([0.463,  0.278,  0.116])
# ligne 10, avant dernière colonne
# le pixel compte 4*+ de rouge que de bleu et 2*+ de vert que de bleu

# Autre manière de le comprendre :
# Pour une Image donnée, on peut
# par exemple extraire la table
# des niveaux de rouge par :
R = Image[:,:,0]

# Telle que :
print(np.shape(R))
# afficherait 748, 1024
```



Remarque : il arrive qu'une image ait un 4ème niveau de calque. C'est notamment le cas dans la décomposition CMJN (Cyan, Magenta, Jaune, Noir, utile pour l'impression couleur) et pour les images RVB qui ajoutent un niveau de transparence (utile pour les logos, ou les chevauchements d'images notamment).

Sauf exception, en traitement d'images, on travaille le plus souvent avec des images en niveaux de gris. Écrivons une fonction `niv_gris(Image)` qui prenne en argument une image en RVB et qui renvoie une seule image en niveaux de gris, où chaque pixel de l'image grise `G` est calculé comme moyenne des niveaux de rouge *R*, vert *V* et bleu *B*

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

L'image initiale étant **non normalisée** (niveaux $R_{i,j}$, $V_{i,j}$, $B_{i,j}$ pouvant dépasser 1), il nous faut une fonction auxiliaire `normalise(Matrice)` qui prend en argument une matrice (un calque) et qui renvoie la matrice normalisée, où toutes les valeurs ont été divisées par la valeur du plus haut pixel (ce qui assure que toutes les valeurs sont ≤ 1) :

$$M'_{i,j} = \frac{M_{i,j}}{\max\limits_{k,l} M_{k,l}}$$

```
def normalise(Matrice):
    N, M = np.shape(Matrice)
    pgelem = - np.inf    # on pourrait aussi prendre pgelem = Matrice[0,0]
    for i in range(N):   # commençons par rechercher la plus grande valeur de Matrice
        for j in range(M):
            if Matrice[i,j] > pgelem :
                pgelem = Matrice[i,j]
    return Matrice / pgelem            # on normalise tous les coeffs


def niv_gris(Image):
    R, V, B = Image[:,:,0],Image[:,:,1], Image[:,:,2]
    Gris = (R+V+B)/3
    return normalise(Gris)
```

A.3 / Ouverture

Un streamer veut partager son écran, à 25 *fps* (images par seconde ou frames per second). Son écran est en 1680 x 1050 pixels, codé en RVB 24 bits (chaque calque R, V ou B est codé en 8 bits). S'il n'encode pas le flux vidéo, de quel débit a-t-il besoin ?

- Chaque image pèse : 1680 * 1050 * 3 = $5.3 \cdot 10^6$ octets = 5.3 Mo
- Il lui faut donc un débit 25 * 5.3 = 132 Mo/s
- Une vidéo de 3:30 pèsera donc : (60*3 + 30)*130 = 27 300 Mo = 27,3 Go … !
- Il va donc encoder le flux !

B / Filtrage et convolution

B.1 / Principe du filtrage par convolution

En traitement d'images ce qu'on appelle *filtre* est un procédé destiné à modifier l'image. En sciences, on utilise généralement des filtres par convolution.

Attention, le produit de convolution de deux matrices **n'est pas** le produit matriciel ! Pour deux matrices M, M' de $M_{n,p}(\mathbb{R})$ de même nombre de lignes et de colonnes, le produit de convolution est :

$$\sum_{i=0}^{n}\sum_{j=0}^{p} M_{i,j}M'_{ij}$$

Par exemple, le produit par convolution de $\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}$ par $\begin{pmatrix}3 & 2 & 4\\ \frac{1}{2} & 2 & 1\end{pmatrix}$ vaut : $3+4+12+2+10+6=37$ c'est-à-dire ici $M_{0,0}M'_{0,0}+M_{0,1}M'_{0,1}+M_{0,2}M'_{0,2}+M_{1,0}M'_{1,0}+M_{1,1}M'_{1,1}+M_{1,2}M'_{1,1}$.

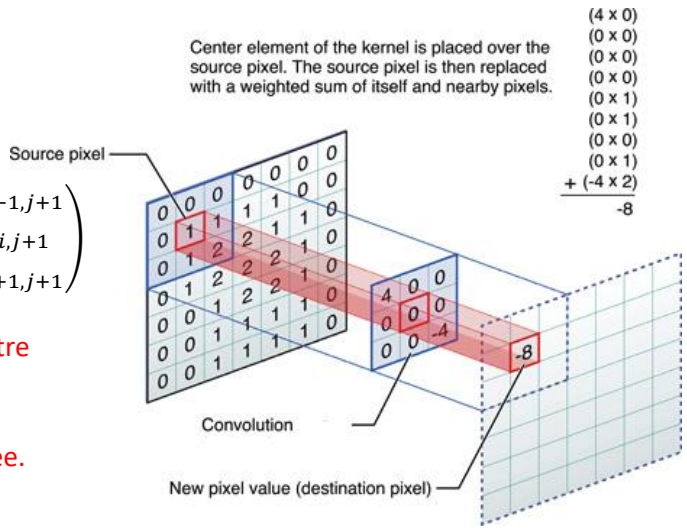
Écrivons une fonction `produit_convol(M1, M2)` prenant en argument deux matrices *np.array* et renvoyant la valeur du produit de convolution :

```
def produit_convol(M1, M2) :
    N1, M1 = np.shape(M1)
    N2, M2 = np.shape(M2)
    assert N1 == N2 and M1 == M2, "matrices non convoluables"
    s = 0
    for i in range(N1):
        for j in range(M1):
            s += M1[i,j]*M2[i,j]
    return s
```

Un filtre est représenté par une matrice de petite taille, en général carrée et comptant un nombre impair d'éléments. Dans ce cours, nous prendrons des filtres de taille 3×3. Une telle matrice est constituée **d'un cœur** (élément central).

Pour chaque pixel (i,j) de l'image à filtrer

- On extrait la sous-matrice $E_{i,j}=\begin{pmatrix}P_{i-1,j-1} & P_{i-1,j} & P_{i-1,j+1}\\ P_{i,j-1} & \boldsymbol{P_{i,j}} & P_{i,j+1}\\ P_{i+1,j-1} & P_{i+1,j} & P_{i+1,j+1}\end{pmatrix}$
- On calcule le produit de convolution $s_{i,j}$ entre E et le filtre (qui est indépendant de *i* et *j*)
- On affecte la valeur de $s_{i,j}$ au pixel (i,j) de l'image filtrée. (c'est-à-dire qu'on l'affecte à la position du cœur du filtre)



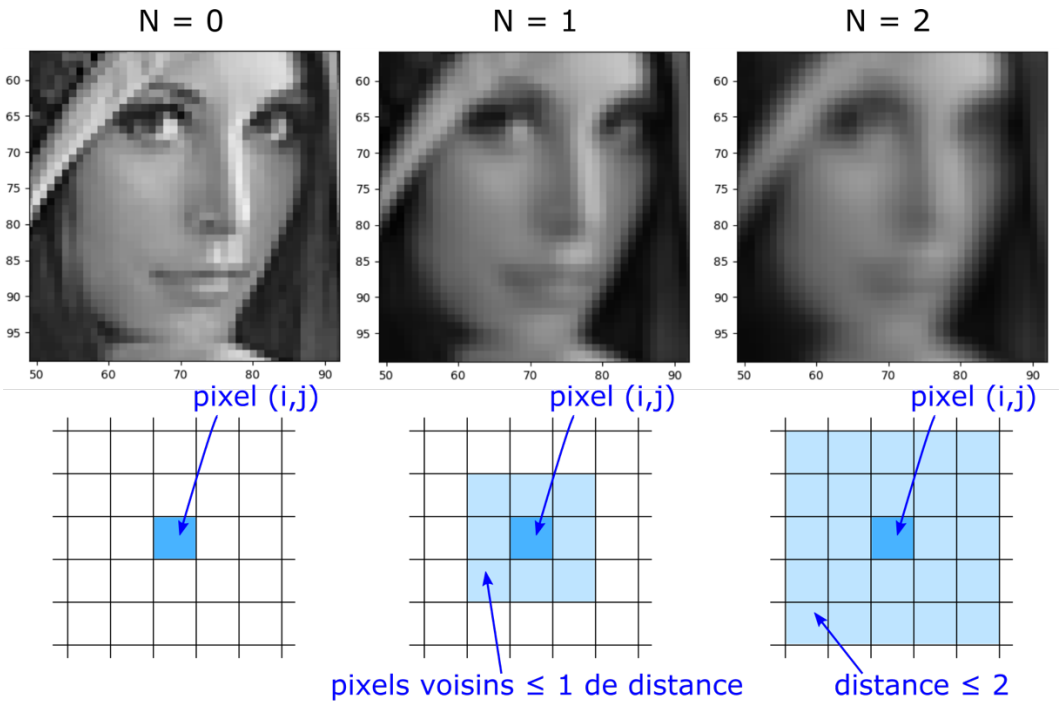
Attention : l'extraction et le produit de convolution doivent se faire sur l'image **non filtrée**. Or, nous allons balayer (comme pour la moyenne glissante) le filtre, et les éléments voisins vont donc être pris en compte plusieurs fois au cours de ces balayages. Il est donc **impossible** de faire le filtrage **en place** (deux matrices distinctes sont nécessaires) !

Par exemple, l'opération de « floutage » (ou de filtrage passe-bas) est un filtre moyen où tous les éléments valent 1. On retrouve ici la moyenne glissante : si on définit le filtre comme étant le cœur + N éléments de chaque côté :

$$(\mathbf{1})$$

(pas de filtrage)

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



Comme pour la moyenne glissante, on voit que le filtrage par convolution présente des problèmes aux bords de l'image (effets de bord). Pour un filtre de taille 3×3, l'image en sortie perd en taille, par rapport à l'image en entrée :

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

La table (*a*) correspond à l'image initiale, représentée en (*c*) (**remarque :** les 0 correspondant au blanc sont ici représentés par des cases vierges afin de ne pas surcharger le dessin).

La table (*b*) est à remplir en appliquant une fenêtre glissante avec $N = 1$ à la table (*a*). Le résultat correspondant en niveaux de gris est représenté en (*d*).

Remarques :

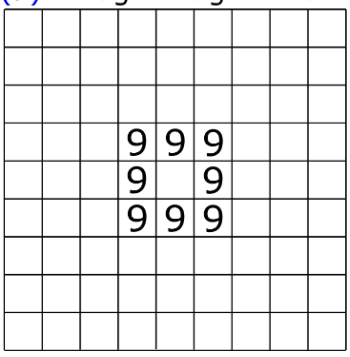
Ici le $s_{i,j}$ maximal correspondrait à la convolution des deux matrices suivantes :

$\begin{pmatrix} 1 & 1 & 1 \\ 1 & \mathbf{1} & 1 \\ 1 & 1 & 1 \end{pmatrix}$ avec $\begin{pmatrix} 9 & 9 & 9 \\ 9 & 9 & 9 \\ 9 & 9 & 9 \end{pmatrix}$ qui donne $s_{i,j}^{max} = 81$.

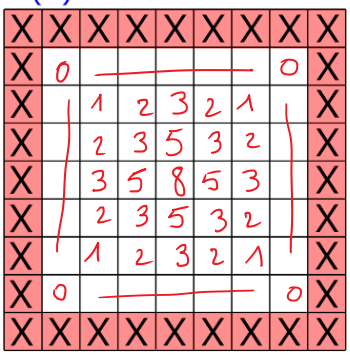
On normalise ensuite les résultats entre 0 et 9 (codage sur 10 niveaux).

Illustration avec le flou de taille 3 (N = 1)

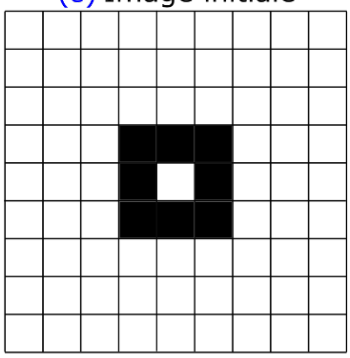
(*a*) Codage image initiale



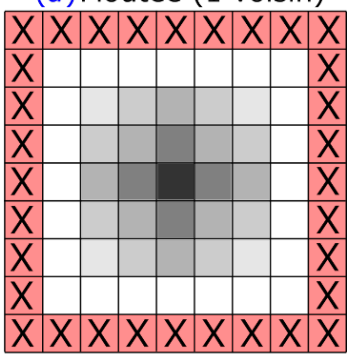
(*b*) Flou avec N = 1



(*c*) Image initiale



(*d*) Floutée (1 voisin)



Écrivons donc une fonction `convolution(Filtre, ImageG)` qui prend en argument un Filtre de taille 3×3 et une Image $N \times N$ en niveaux de gris, et qui renvoie l'image $(N-2) \times (N-2)$ filtrée.

```
def convolution(Filtre, ImageG):
    # on supposera ici un filtre de taille 3x3
    N, M = np.shape(ImageG)
    Imsortie = np.zeros([N-2,M-2])
    for i in range(1,N-1):
        for j in range(1,M-1):
            Extraction = ImageG[i-1:i+2, j-1:j+2]
            Imsortie[i-1,j-1] = produit_convol(Filtre, Extraction)
    return Imsortie
```

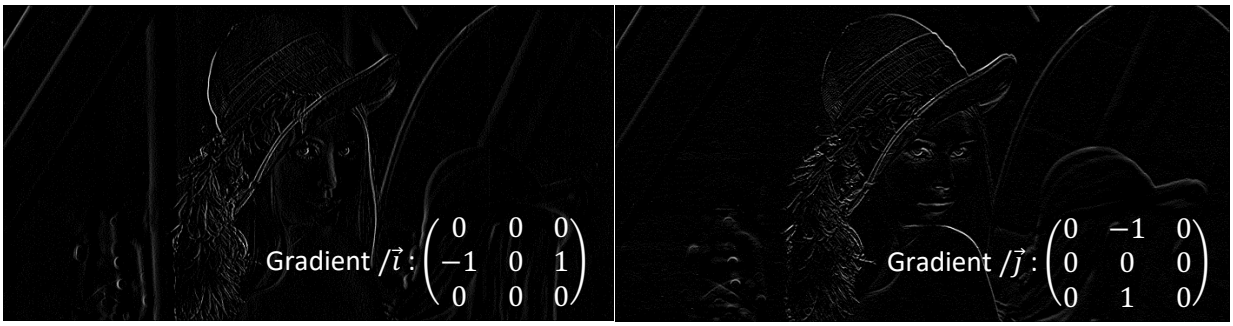
$\{0 \leq i \leq N-1\}$ $\{0 \leq j \leq N-1\}_{image\ initiale}$

$\{1 \leq i \leq N-2\}$ $\{1 \leq j \leq N-2\}_{zone\ à\ filtrer\ =\ image\ finale}$

$\#\begin{pmatrix} P_{i-1,j-1} & P_{i-1,j} & P_{i-1,j+1} \\ P_{i,j-1} & \boldsymbol{P_{i,j}} & P_{i,j+1} \\ P_{i+1,j-1} & P_{i+1,j} & P_{i+1,j+1} \end{pmatrix}$

$(i,j)_{image\ initiale} = (i-1, j-1)_{image\ finale}$

B.2 / Quelques filtres classiques



➔ On voit que les images ici obtenues sont très sombres (peu d'information, car 0 partout là où le contraste, donc le gradient est faible). On applique généralement un seuil (assez bas) sur ces images, notamment pour les détections de contours (filtres de détection de visage, de formes, pour le suivi de trajectoires, etc).