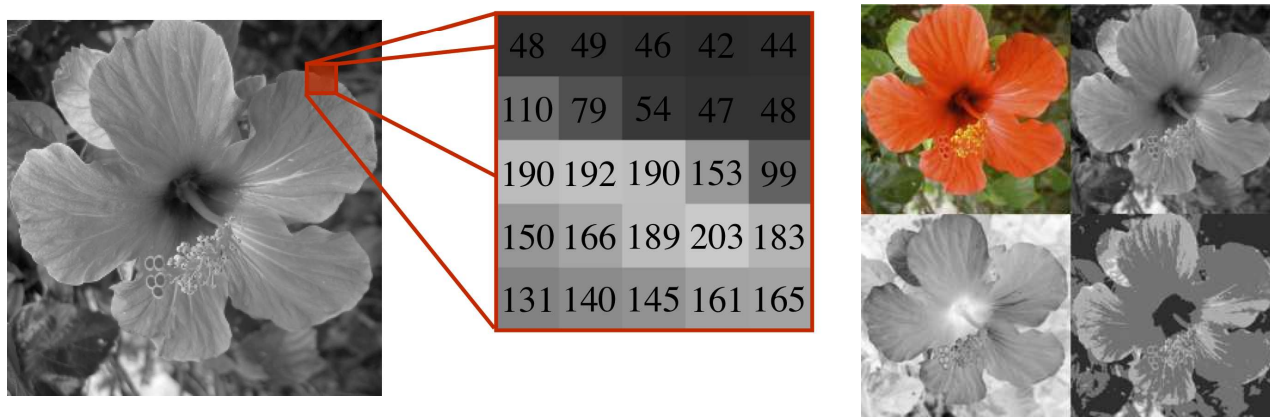


Révisions 4 : Traitements numériques de l'image



PLAN DU CHAPITRE

1	Représentation matricielle d'une image - le module PIL (Python Imaging Library)	3
1.1	Elément d'image : le pixel - caractéristiques	3
1.2	Ouverture et écriture d'un fichier image	4
2	Le traitement d'image	4
2.1	Traitements géométriques de l'image	4
	a - Effet miroir (symétrie par rapport à un axe vertical)	4
	b - Renversement (symétrie par rapport à un axe horizontal)	5
	c - Agrandissement	6
	d - Rotation	7
2.2	Traitements chromatiques de l'image	8
	a - Conversion d'une image couleur en niveau de gris	8
	b - Postérisation - Transformation en image de noir et blanc	9
	c - Inversion d'une image (négatif)	9
	d - Séparation des couleurs	11
	e - Pixellisation	11

2.3	Traitements structurels de l'image : filtrage par convolution	12
	a - Principe de la convolution - noyau de convolution	12
	b - Premier contact : filtre isotrope identité ou "passe-tout"	13
	c - Filtre isotrope passe-haut ou "net"	13
	d - Filtre isotrope passe-bas ou "flou"	14
	e - Filtre isotrope Laplacien	14

1 Représentation matricielle d'une image - le module PIL (Python Imaging Library)




1.1 Élément d'image : le pixel - caractéristiques

A l'instar de la numérisation des signaux, la capture informatique d'une image nécessite de procéder à son échantillonnage ; on réalise pour cela son "découpage" selon une grille dont les dimensions fixeront la **résolution** de l'image numérisée. Chaque élément de cette image porte le nom de **pixel** pour *picture element*.

Ainsi, une image numérique de L lignes et C colonnes sera naturellement représentée par une matrice de taille $[L,C]$ comportant donc $L \times C$ éléments, chacun codant l'information relative au pixel correspondant de la manière suivante :

- POUR UNE IMAGE EN NUANCE DE GRIS : chaque pixel est représenté par un seul nombre entier codé sur N bits représentant son niveau de gris. Nous prendrons $N = 8$ soit un total de $2^8 = 256$ valeurs possibles allant du noir codé par $0_d = 00000000_b$ au blanc codé par $255_d = 11111111_b$. La matrice est alors de dimension 2.
- POUR UNE IMAGE EN COULEURS : il faut cette fois 3 nombres entiers pour représenter chaque pixel afin de coder les contributions respectives des couleurs **rouge**, **vert**, et **bleu** (système RVB ou RGB en anglais). Chaque couleur d'un pixel est codée sur $N = 8$ bits soit une intensité de teinte allant comme pour les nuances de gris de $0_d = 00000000_b$ (absence de cette couleur) à $255_d = 11111111_b$ (contribution maximale de cette couleur) ; la palette des couleurs possibles pour un pixels comporte donc $2^8 \times 2^8 \times 2^8 = 16,8$ millions de couleurs possibles (codage sur 24 bits)

Par exemple :

(255, 0, 0) correspond au rouge le plus intense :	
(0, 255, 0) correspond au vert le plus intense :	
(0, 0, 255) correspond au bleu le plus intense :	

La grille ci-dessous donne quelques exemples d'encodages chromatiques selon la méthode RVB :

rouge (255, 0, 0)	vert (0, 255, 0)	bleu (0, 0, 255)	(255, 127, 127)	(16, 255, 127)	(64, 127, 255)
cyan (0, 255, 255)	magenta (255, 0, 255)	jaune (255, 255, 0)	(127, 255, 127)	(127, 16, 255)	(255, 64, 127)
(0, 0, 0)	(50, 50, 50)	(100, 100, 100)	(150, 150, 150)	(200, 200, 200)	(255, 255, 255)

1.2 Ouverture et écriture d'un fichier image

Le traitement d'une image consiste simplement à intervenir sur les pixels formant celle-ci en modifiant leurs valeurs chromatiques RVB.

Python dispose du module `Image` de la librairie PIL (pour **P**ython **I**maging **L**ibrary) permettant l'ouverture des fichiers graphiques de différents formats (`.jpg`, `.png`, `.eps`, `.gif`, `.pcx` etc..); par ailleurs, le module `numpy` permettra de transformer cette image en matrice :

Listing IV.1 –

```
1 from PIL import Image as Img #chargement du module Image
2 import numpy as np #chargement du module numpy
```

La première étape consiste à ouvrir un fichier contenant une image, par exemple au format `.png` (*Portable Network Graphics*), et la convertir **en sa matrice correspondante** que l'on stockera dans une variable, par exemple `matimage` :

Listing IV.2 –

```
1 image=Img.open("nom_image.png") #ouverture de l'image et stockage dans la variable image
2 matimage=np.array(image) # création de la matrice et stockage dans la variable "matimage"
```

Certains fichiers au format `.png` possède une matrice de dimension 4 et non 3 comme dans le codage RVB classique : le 4^{ième} nombre code pour la luminosité globale et il est inutile pour nous. Il conviendra donc le cas échéant d'éliminer cette 4^{ième} valeur avant le traitement en redimensionnant la matrice par simple slicing comme ci-dessous :

Listing IV.3 –

```
1 .....
2 matimage=matimage[:, :, :3] # élimination de la 4ieme valeur de chaque pixel
3 .....
```

Une fois le traitement de la matrice `matimage` réalisé, il faudra transformer celle-ci en image :

Listing IV.4 –

```
1 ..... # traitement de la matrice matimage
2 ..... # traitement de la matrice matimage
3 image=Img.fromarray(matimage) # création de l'image à partir de la matrice modifiée
4 .....
```

puis enfin enregistrer l'image dans un fichier :

Listing IV.5 –

```
1 .....
2 image.save('nom_image_modifiee.png') # sauvegarde de l'image
```

Enfin, on pourra examiner le résultat du traitement d'image en utilisant la commande `image.show()` qui affiche l'image `image` dans une fenêtre.

2 Le traitement d'image

2.1 Traitements géométriques de l'image

a - Effet miroir (symétrie par rapport à un axe vertical)

Exercice de cours: (2.1) - n° 1. Rédiger un script python renvoyant à l'écran le symétrique de l'image originale par rapport à un axe vertical passant par le "centre" de l'image.

RÉPONSES :

Listing IV.6 –

```
1 matimagesym=np.zeros([L,C]+[3], dtype='uint8')
2 for i in range(L):
3     for j in range(C//2):
4         matimagesym[i,j],matimagesym[i,C-j-1]=matimage[i,j],matimage[i,j]
5 imagesymv=Img.fromarray(matimagesym)
6 image.show()
7 imagesymv.show()
```



FIGURE IV.1 – Effet miroir

b - Renversement (symétrie par rapport à un axe horizontal)

Le principe est en tout point identique à celui de l'effet miroir, avec cette fois la permutation des lignes par rapport à la ligne médiane.

Exercice de cours: (2.1) - n° 2. *Proposer un script python permettant de réaliser un renversement d'image.*

RÉPONSES :

Listing IV.7 –

```
1 matimagesym=np.zeros([L,C]+[3], dtype='uint8')
2 for i in range(L//2):
3     for j in range(C):
4         matimagesym[i,j],matimagesym[L-i-1,j]=matimage[L-i-1,j],matimage[i,j]
5 imagesymv=Img.fromarray(matimagesym)
6 imagesymv.show()
```



FIGURE IV.2 – Renversement vertical de l'image

c - Agrandissement

On va chercher ici agrandir les dimensions d'une image, **en nuances de gris**, tout en conservant ses proportions. En géométrie de \mathbb{R}^2 , l'opération d'agrandissement correspond à une simple homothétie de rapport $k > 1$. Dans le cas d'une image numérique les choses se compliquent un peu car l'image de départ (L lignes et C colonnes) comporte $L \times C$ pixels, et l'idée de simplement calculer l'homothétie de chacun de ses pixels dans l'image d'arrivée ne convient pas puisque celle-ci comporterait $L \times C$ pixels alors qu'elle doit en fait en compter $k^2 L \times C$.

On procèdera plutôt de la manière suivante : la matrice de l'image originelle est toujours `matimage` et celle de l'image agrandie `matimage_homoth` :

- on initialisera un tableau `matimage_homoth` de zéros de dimensions $(k \times L, k \times C)$;
- pour chaque élément de ce tableau `matimage_homoth` correspondant à un pixel de l'image d'arrivée, on va déterminer le pixel le plus proche de son antécédent géométrique dans l'image de départ ;
- on affectera à chaque élément du tableau `matimage_homoth` de l'image d'arrivée la valeur moyenne des pixels voisins du pixel antécédent ;

NB : une autre démarche aurait pu être d'affecter à chaque pixel de l'image d'arrivée la valeur du pixel le plus proche de son antécédent géométrique dans l'image de départ. Cela entraînerait que chaque pixel est reproduit k fois dans l'image d'arrivée, avec des effets de pixellisation peu esthétiques.

Exercice de cours: (2.1) - n° 3. Agrandissement

- 1 Proposer une fonction `moyennepix(matimage:array, i:int, j:int) → uint8` qui prend en argument la matrice de l'image et les coordonnées i (ligne) et j (colonne) d'un élément de celle-ci, et renvoie la valeur moyenne du "bloc" constitué du pixel (i, j) et de ses 8 plus proches voisins.
- 2 Ecrire une fonction `agrandir(matimage:array, k:float) → array` avec $k > 1$ qui renvoie la matrice `matimage_homoth`.

RÉPONSES :

- 1 On propose la fonction suivante :

Listing IV.8 –

```
1 def moyennepix(matimage, i, j):
2     L,C=matimage.shape
3     sommepix=0
4     nbpix=0
5     for ip in range(max(0, i-1), min(L, i+2)):
```

```

6     for jp in range(max(0, j-1), min(C, j+2)):
7         sommepix+=matimage(ip, jp)
8         nbpix+=1
9     return np.uint8(sommepix/nbpix)

```

Ce code gère le fait que le pixel soit sur les bords de l'image ; par exemple pour la boucle d'itération sur les lignes (compteur ip) (bords), la première valeur de l'intervalle du **range** sera le maximum choisi entre 0 : le pixel est sur le bord supérieur, et $i - 1$: le pixel n'est pas sur ce bord, et de même la seconde valeur de l'intervalle du **range** sera le minimum choisi entre C et $i + 2$ (attention : valeur exclue!)

- ② On va maintenant définir la matrice de l'image d'arrivée de dimension $k \times L \times C$, puis calculer **pour chacun de ses pixels** son antécédent dans l'image de départ, et finalement lui affecter la valeur moyenne des voisins en exploitant la fonction **moyennepix**. Cela donne donc :

Listing IV.9 –

```

1 def agrandir(matimage, k):
2     Lp, Cp = int(k*L), int(k*C)
3     matimage_homoth = np.zeros((Lp, Cp), dtype='uint8')
4     for ip in range(Lp):
5         for jp in range(Cp):
6             i, j = int(ip/k), int(jp/k)
7             matimage_homoth[ip, jp] = moyenne(matimage, i, j)
8     return matimage_homoth

```

FIGURE IV.3 – Agrandissement de l'image de 20% ($k=1.2$)

d - Rotation

On se propose enfin d'écrire une fonction réalisant la rotation d'une image. On se limite là-encore à une image en nuances de gris.

Afin de s'assurer que tous les pixels de l'image transformée soient calculés, nous allons, comme dans le cas de l'agrandissement, déterminer pour chacun d'entre eux le pixel antécédent de l'image originelle.

Notons tout d'abord qu'un pixel $[i, j]$ de l'image originelle n'est pas rigoureusement un point mais une surface s'étirant du point (i, j) au point $(i + 1, j + 1)$. On adoptera l'algorithme suivant :

- on initialisera un tableau **matimage_rotatee** de zéros de dimensions (L, C) identiques à celles de l'image d'origine ;
- on choisit un point P' centre d'un pixel (i', j') de l'image que l'on va construire (la "rotatée") et on va déterminer son point antécédent P dans l'image originelle en calculant la rotation inverse ;

- Si le point P appartient à l'image originelle, alors on détermine le pixel (i, j) qui est le plus proche de ce point, et la valeur moyenne du bloc composé de ce pixel et ses plus proches voisins est affectée au pixel (i', j') ; si le point P est en dehors de l'image d'origine, laisser la valeur du pixel à 0 (ou le passer à 255, au choix!);

Commençons par déterminer la formule analytique de la rotation inverse permettant de remonter à l'antécédent de P' dans l'image originelle; la formule analytique de la rotation de centre C et d'angle θ est :

$$P' = P_c + R(P - P_c) \Leftrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \end{bmatrix} + \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} x - x_c \\ y - y_c \end{bmatrix}$$

En inversant cette relation il vient : $P = P_c + R^{-1}(P' - P_c)$ avec $R^{-1} = \frac{{}^t \text{com}(R)}{\det(R)}$ soit après calcul :

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \end{bmatrix} + \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} x' - x_c \\ y' - y_c \end{bmatrix}$$

ce qui permet de calculer l'antécédent $P(x, y)$ de $P'(x', y')$.

On commence par implémenter la fonction `moyennepix(matimage:array,i:int,j:int,p:int) → uint8` qui permet de renvoyer la valeur moyenne des pixels du bloc de dimension $(2p+1, 2p+1)$ autour du pixel (i, j) , le paramètre p étant ajustable :

Listing IV.10 –

```
1 def moyennepix(matimage,i,j,p):
2     sommepix=0
3     nbpix=0
4     for k in range(max(0,i-p),min(L,i+p+1)):
5         for l in range(max(0,j-p),min(C,j+p+1)):
6             sommepix+=matimage(k,l)
7             nbpix+=1
8     return np.uint8(sommepix/nbpix)
```

puis on implémente la fonction `rotation(matimage:array,centre=array,alpha:float) → array` :

Listing IV.11 –

```
1 def rotation(matimage,centre,alpha):
2     L,C=matimage.shape
3     matimage_rotatee=np.zeros((L,C), dtype='uint8') #tableau final avec des pixel à 0, donc noirs
4     cos,sin=np.cos(alpha),np.sin(alpha)
5     Rinv=np.array([[cos,sin],[-sin,cos]])
6     for ip in range(L):
7         for jp in range(C):
8             Pp=np.array([[ip+0.5],[jp+0.5]],dtype=float) # P'=centre du pixel (i',j')
9             P=centre+Rinv*(Pp-centre) # pour calculer la position du point antécédent
10            i,j=int(P[0,0]),int(P[1,0])
11            if 0<=i<L and 0<=j<C: #on vérifie que (i,j) est dans l'image originelle
12                matimage_rotatee[ip,jp]=moyennepix(matimage,int(P[0,0]),int(P[1,0]),1) # p=1-> bloc 9 pixels
```

En choisissant le centre de l'image comme centre d'une rotation de 60° , voici ce que l'on obtient :

2.2 Traitements chromatiques de l'image

a - Conversion d'une image couleur en niveau de gris

Exercice de cours: (2.2) - n° 4. Proposer un script permettant de transformer une image couleur en image en nuances de gris en affectant à chaque pixel gris la valeur moyenne des trois couleurs de ce même pixel dans l'image originale. Attention : on notera que pour les entiers codés sur 8 bits, la valeur affectée se calcule modulo 256 avec par exemple `uint8(256)=0`, et `uint8(280)=24`, ainsi, on fera "prudemment" le calcul de valeur moyenne.

RÉPONSE :



FIGURE IV.4 – Rotation de l'image de 60° autour de son centre

Listing IV.12 –

```

1 matimagegrise=np.zeros([L,C],dtype='uint8')
2 for i in range(L):
3     for j in range(C):
4         matimagegrise[i,j]=int(matimage[i,j][0]/3+matimage[i,j][1]/3+matimage[i,j][2]/3)
5 imagegrise=Img.fromarray(matimagegrise)
6 imagegrise.show()

```

b - Postérisation - Transformation en image de noir et blanc

La postérisation consiste à transformer une image grise en image de pixels ne contenant que deux valeurs de gris. On fixe arbitrairement le critère de conversion d'un pixel. Par exemple, les pixels de valeur inférieure à 128 prendront la valeur 50 et les pixels de valeur comprise entre 127 et 256 prendront la valeur 200.

Exercice de cours: (2.2) - n° 5. Proposer un script python permettant une postérisation.

RÉPONSE :

Listing IV.13 –

```

1 matimageposter=np.zeros([L,C],dtype='uint8')
2 for i in range(L):
3     for j in range(C):
4         if matimagegrise[i,j]<128:
5             matimageposter[i,j]=50
6         else:
7             matimageposter[i,j]=200
8 imageposter=Img.fromarray(matimageposter)
9 imageposter.show()

```

NB : on peut également transformer l'image en noir et blanc en imposant des valeurs de conversion de 0 (noir) ou 255 (blanc)

c - Inversion d'une image (négatif)

On cherche ici à obtenir le négatif d'une image.

Exercice de cours: (2.2) - n° 6. Négatif d'une image

- ❶ Comment calculer la valeur du "négatif" d'un pixel ?
- ❷ Proposer un script python réalisant le négatif d'une image.

RÉPONSES :



FIGURE IV.5 – Postérisation d'une image

- ❶ Sachant que la valeur p affectée à un pixel appartient à l'intervalle $\llbracket 0, 255 \rrbracket$, la valeur de son négatif est simplement $p' = p - 255$
- ❷ Proposition de Script :

Listing IV.14 –

```
1 matimagenegat=np.zeros([L,C]+[3],dtype='uint8')
2 for i in range(L):
3     for j in range(C):
4         matimagenegat[i,j]=[255-matimage[i,j][0],255-matimage[i,j][1],255-matimage[i,j][2]]
5 imagenegat=Img.fromarray(matimagenegat)
6 imagenegat.show()
```



FIGURE IV.6 – Négatif d'une image

d - Séparation des couleurs

Exercice de cours: (2.2) - n° 7. Proposer un script python permettant à partir d'une image en couleur de former 3 images identiques, mais chacune résultant de l'extraction d'une seule des 3 couleurs rouge, vert, bleu de l'image d'origine.

RÉPONSE :

Listing IV.15 –

```
1 matimagerouge=np.zeros([L,C]+[3],dtype='uint8')
2 matimagevert=np.zeros([L,C]+[3],dtype='uint8')
3 matimagebleu=np.zeros([L,C]+[3],dtype='uint8')
4 for i in range(L):
5     for j in range(C):
6         matimagerouge[i,j]=[matimage[i,j][0],0,0]
7         matimagevert[i,j]=[0,matimage[i,j][1],0]
8         matimagebleu[i,j]=[0,0,matimage[i,j][2]]
9 imagerouge=Img.fromarray(matimagerouge)
10 imagevert=Img.fromarray(matimagevert)
11 imagebleu=Img.fromarray(matimagebleu)
12 imagerouge.show()
13 imagevert.show()
14 imagebleu.show()
```

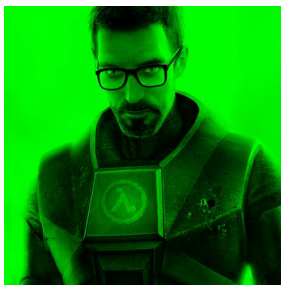


FIGURE IV.7 – Séparation des couleurs

e - Pixellisation

La pixellisation d'une image consiste à transformer celle-ci en une image de même dimension découpée en blocs $p \times p$ de p^2 pixels auxquels on attribue la valeur moyenne des p^2 pixels originels.

La méthode est la suivante :

- on définit une matrice "moyenne" de dimension $[L//p, C//p]$ qui contiendra donc autant d'éléments qu'il y aura de blocs dans l'image découpée
- chaque élément de la matrice "moyenne" prendra la valeur moyenne des pixels du bloc $p \times p$ de l'image d'origine.
- Enfin, on construit la matrice de sortie pixellisée en affectant à chaque pixel la valeur moyenne du bloc auquel il appartient à l'aide de la matrice "moyenne".

Exercice de cours: (2.2) - n° 8. Bâtir une fonction `pixellisation(M,p)` prenant en argument la matrice d'image M et l'entier p et qui renvoie la matrice de l'image pixellisée par blocs de p^2 pixels.

RÉPONSE :

Listing IV.16 –

```
1 def pixellisation(M,p):
2     matmoy=np.zeros([L//p,C//p], dtype='uint8') #définition de la matrice moyenne
3     matimagepixel=np.zeros([L,C], dtype='uint8') # définition de la matrice de sortie
4     for i in range(0,L-p,p): #balayage des lignes par saut de p
5         for j in range(0,C-p,p): #balayage des colonnes par saut de p
6             moy=0.
```

```

7         for k in range(p):
8             for l in range(p):
9                 moy=moy+(1./p**2)*M[i+k,j+l] #calcul de la valeur moyenne des pixels du blocs p*p
10            matmoy[i//p,j//p]=moy #affectation de la valeur moyenne au bloc de la valeur moyenne
11    for i in range(L-p): #boucles de construction de la matrice de sortie
12        for j in range(C-p):
13            matimagepixel[i,j]=matmoy[i//p,j//p] #le pixel i,j prend la valeur moyenne de son bloc p*p
14    return matimagepixel
    
```

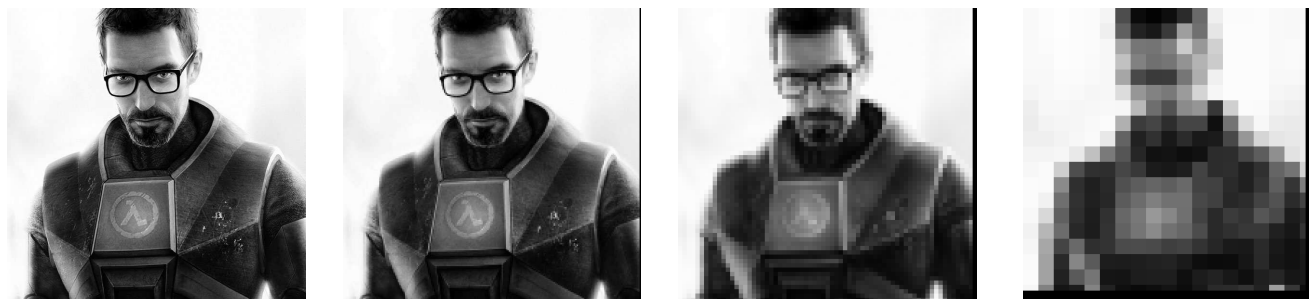


FIGURE IV.8 – Image d’origine puis effet de la pixellisation pour les valeurs respectives $p = 3$, $p = 9$, et $p = 32$

2.3 Traitements structuraux de l’image : filtrage par convolution

a - Principe de la convolution - noyau de convolution

La plupart des filtres de traitement d’image s’appuient sur la méthode de convolution qui consiste à modifier la valeur de chaque pixel d’une image en fonction de la valeur de ses voisins.

Ainsi, on peut définir formellement l’action de la convolution sur le pixel (i, j) par une matrice N appelée *noyau de convolution* de taille $(2p + 1) \times (2p + 1)$ avec $p \in \mathbb{N}$ et telle que :

$$nv_pixel[i, j] = \sum_{k=0}^{2p} \sum_{l=0}^{2p} N[k, l] \times pixel[i + k - p, j + l - p]$$

Nous n’envisagerons ici que des actions de convolution sur les plus proches voisins du pixel (i, j) soit 8 pixels au total, en prenant simplement $p = 1$.

$$\begin{pmatrix} \dots & \dots & \dots & \dots & \dots \\ \dots & (i-1, j-1) & (i-1, j) & (i-1, j+1) & \dots \\ \dots & (i, j-1) & (\mathbf{i}, \mathbf{j}) & (i, j+1) & \dots \\ \dots & (i+1, j-1) & (i+1, j) & (i+1, j+1) & \dots \\ \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

Ainsi tous les noyaux de convolution de ce cours seront des matrices 3×3 :

$$N = \begin{pmatrix} N[0, 0] & N[0, 1] & N[0, 2] \\ N[1, 0] & N[1, 1] & N[1, 2] \\ N[2, 0] & N[2, 1] & N[2, 2] \end{pmatrix}$$

Exercice de cours: (2.3) - n° 9. FONCTION DE CONVOLUTION Rédiger une fonction `convolution(matimage, N)` de paramètres la matrice en nuance de gris `matimagegrise` ainsi que le noyau de convolution N (3×3) et renvoyant la nouvelle matrice résultant de l’opération de convolution décrite par N .

RÉPONSE :

Listing IV.17 –

```

1 def convolution(M,N,p):
2     Mmodif=np.zeros([L,C],dtype='uint8') #création de la matrice modifiée "nulle"
3     for i in range(1,L-1): #attention aux bords de l'image qui ne peuvent être traités par la convolution
4         for j in range(1,C-1):
5             pixel=0 #initialisation du nouveau pixel
6             for k in range(3):
7                 for l in range(3):
8                     pixel=pixel+N[k,l]*M[i+k-p,j+l-p]
9             if pixel<0: #attention de définir des valeurs de pixels comprises entre 0 et 255
10                 pixel=0
11             elif pixel>255:
12                 pixel=255
13             Mmodif[i,j]=pixel #écriture du nouveau pixel
14     return Mmodif

```

b - Premier contact : filtre isotrope identité ou "passe-tout"

Pour commencer, on peut imaginer une convolution par un noyau laissant l'image inchangée; on parle de noyau identité.

Exercice de cours: (2.3) - n° 10. Définir le noyau identité.

RÉPONSE :

L'application du noyau identité doit laisser le pixel (i, j) inchangé; il faut donc que ses plus proches voisins soient sans effet sur ses valeurs; le noyau est donc

$$N = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Listing IV.18 –

et le code python correspondant :

```

1 N=np.zeros([3,3],dtype='uint8')
2 N[1,1]=1

```

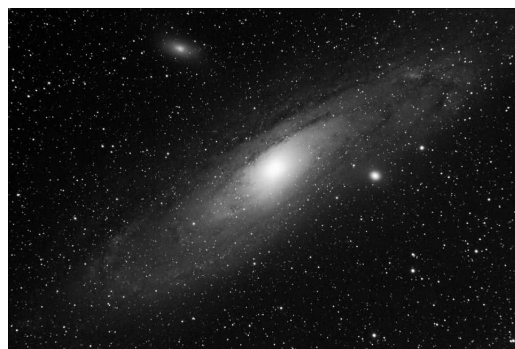
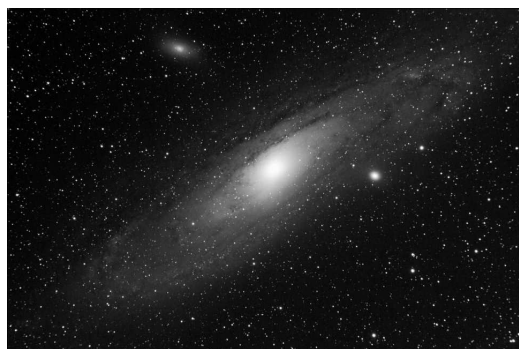


FIGURE IV.9 – Image d'origine à gauche et image filtrée à droite : le noyau identité est sans effet

c - Filtre isotrope passe-haut ou "net"

L'application principale des produits de convolution est la création des filtres « passe haut » et « passe bas ». Un filtre « passe haut » favorise les hautes fréquences spatiales, comme les détails, et de ce fait, il améliore le contraste. Un filtre « passe haut » est caractérisé par un noyau comportant des valeurs négatives autour du pixel central, comme dans l'exemple ci-dessous :

$$N = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

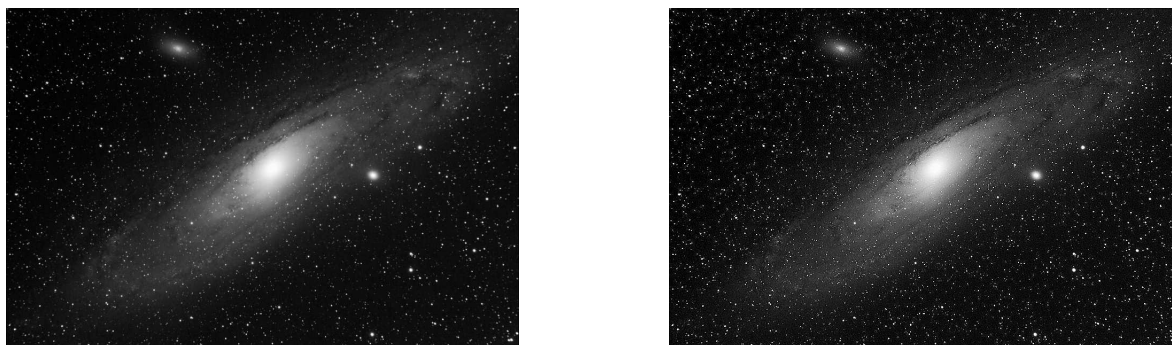


FIGURE IV.10 – Effet du filtrage passe-haut

d - Filtre isotrope passe-bas ou "flou"

Le filtrage passe-bas permet un adoucissement des détails et une réduction de l'aspect "granuleux" de certaines images ; il est par exemple utile lorsque l'on souhaite améliorer la qualité d'une photographie numérique prise dans des conditions de luminosité insuffisante, les capteurs CCD générant alors un fort "bruit".

Exercice de cours: (2.3) - n° 11. Proposer un noyau permettant de réaliser ce type de filtrage.

RÉPONSES : $N = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$

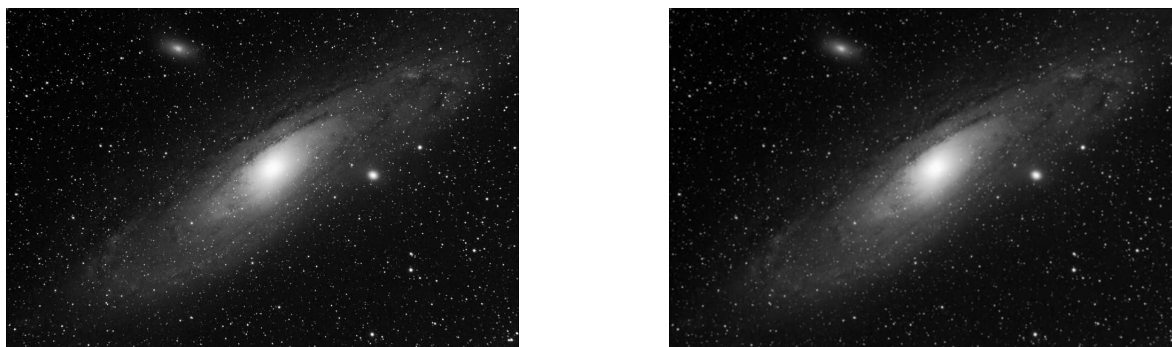


FIGURE IV.11 – Effet du filtrage passe-bas

e - Filtre isotrope Laplacien

Le filtre Laplacien est un filtre de convolution particulier utilisé pour mettre en valeur les détails qui ont une variation rapide de luminosité. Le Laplacien est donc idéal pour rendre visible **les contours des objets** ; il est donc souvent employé dans la reconnaissance de formes. En outre, les contours d'une image constituant les parties les plus informatives de celle-ci, le filtrage laplacien permet de ne garder que ce qui est fondamental dans l'image et d'accélérer ainsi son traitement (par exemple dans la reconnaissance de forme en temps réel).

Exercice de cours: (2.3) - n° 12. Proposer un noyau de convolution 3×3 pour le filtrage Laplacien.

RÉPONSE :

En faisant deux développements limités au second ordre d'une fonction quelconque $f(x, y)$ en $(x + dx, y)$, et $(x - dx, y)$ autour du point $M(x, y)$, puis en discrétisant "en pixels" les positions de ces mêmes points en respectivement $(i + 1, j)$, et $(i - 1, j)$, autour du pixel (i, j) , on tire facilement les deux relations suivantes :

$$\begin{cases} f_{i+1,j} \simeq f_{i,j} + \left. \frac{\partial f}{\partial x} \right|_{i,j} + \frac{1}{2} \left. \frac{\partial^2 f}{\partial x^2} \right|_{i,j} \\ f_{i-1,j} \simeq f_{i,j} - \left. \frac{\partial f}{\partial x} \right|_{i,j} + \frac{1}{2} \left. \frac{\partial^2 f}{\partial x^2} \right|_{i,j} \end{cases}$$

En sommant ces deux relations et en réorganisant les termes, cela donne donc :

$$\left. \frac{\partial^2 f}{\partial x^2} \right|_{i,j} \simeq -2f_{i,j} + f_{i+1,j} + f_{i-1,j}$$

En faisant de même avec selon l'axe y il vient :

$$\left. \frac{\partial^2 f}{\partial y^2} \right|_{i,j} \simeq -2f_{i,j} + f_{i,j+1} + f_{i,j-1}$$

Finalement le Laplacien discrétisé peut s'écrire :

$$\Delta = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \equiv -4f_{i,j} + f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1}$$

d'où le noyau de convolution suivant :

$$N = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



FIGURE IV.12 – Effet du filtre laplacien

REMARQUE : on peut encore "amplifier" l'effet du Laplacien en considérant deux axes supplémentaires (et donc deux dimensions supplémentaires) selon les deux diagonales passant par le pixel (i, j) . Le noyau de convolution du Laplacien "amplifié" s'écrit alors :

$$N_{ampl} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$