

Traitement d'images en niveau de gris

Exercice 1. Télécharger à l'adresse :

<http://www.i2m.univ-amu.fr/perso/jean-philippe.preaux>
le fichier `lenaGris.png`.

Lui appliquer tous les traitements d'image vus dans ce cours.

14.1 Images Bitmap

• Sur les écrans numériques modernes, une image est constituée d'un quadrillage de **pixels**, c'est à dire de petits motifs carrés de couleurs uniformes.



• Par exemple le format Full HD (ou 1080p), est constitué d'une image en 16 :9 de 1080 lignes, soit 1920×1080 pixels colorés.

• La représentation numérique des images la plus employée, nommée **Bitmap**, consiste à stocker le tableau des pixels : sa taille ((L, H) nombres de pixels en largeur et en hauteur), son mode de représentation des couleurs, et le tableau des valeurs de ses $L \times H$ pixels.

• L'image peut être en **mode** (de représentation des couleurs) :

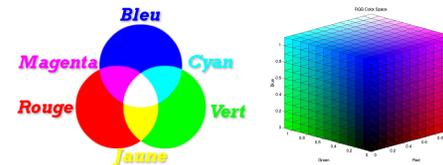
Noir et blanc. Dans ce cas chaque pixel est représenté sur un bit, 0 pour noir, 1 pour blanc. C'est de ce mode que vient le nom "bitmap".

Niveau de gris. Chaque pixel est représenté sur un octet (256 valeurs) dont la valeur varie du plus obscur 0 (noir) au plus clair 255 (blanc) en passant par 254 nuances de gris.

Couleur RGB. Chaque pixel est représenté par 3 octets, chacun donnant l'amplitude des 3 couleurs fondamentales additives : R (rouge), G (vert), B (bleu). Le noir est alors (0,0,0) et le blanc (255,255,255).

Autres modes :

Couleurs CMYK. Couleurs soustractives : Cyan, Magenta, Jaune et Noir ; notamment pour l'impression. **etc...**



• Une image Bitmap peut être stockée sous divers formats de fichiers : BMP, PNG, GIF, JPEG, etc.. compressés ou non.

14.2 Le module Pillow de python

14.2.1 Installation du module Pillow

Sous python, pour le traitement d'images :

- Nous utiliserons le module `Pillow` (compatible python 3). C'est le successeur du module `PIL` qui lui n'est compatible qu'avec python 2.
- S'il n'est pas fourni dans la distribution utilisée, il faudra l'installer :

• **Sous Windows** : package à télécharger avant de l'installer, à l'adresse :

<https://pypi.python.org/pypi/Pillow/2.0.0#downloads>

• **Sous Mac OS-X/Linux** Saisir dans la console de l'EDI :

```
pip install Pillow
```

```

1 # Traitement d'image
2 # Partie I : images en niveaux de gris
3
4 import PIL
5 import numpy as np
6

```

SAISIR "pip install Pillow" dans la console

```

In [4]: pip install Pillow
Downloading/unpacking Pillow
Running setup.py (path:/private/var/folders/gh/j9pcz9x12bj_1

```

14.2.2 Chargement d'une image

Voici une image en couleur au format PNG,



Que l'on stockera, selon son EDI, soit dans le répertoire courant (celui contenant le fichier du programme python), soit dans le répertoire utilisateur (celui portant le nom de l'utilisateur), à la racine.

- On commence par importer le sous-module `PIL.Image` qui contient toutes les fonctions et méthodes qui nous seront nécessaires.

```
from PIL import Image # Importer le sous-module PIL.Image
```

- On crée alors un objet image à partir du fichier `.png` à l'aide de l'instruction :

```
img = Image.open('LenaGris.png') # Sa fonction open() permet le chargement
```

- L'objet-image `img` contient toutes les informations de l'image du fichier `image.png`.

```

>>> print(img)
<PIL.PngImagePlugin.PngImageFile image mode=L size=512x512 at 0x112569400>
>>> img.size
(512, 512)
>>> img.format
'PNG'
>>> img.mode
'L'

```

L'image a pour taille `512 × 512` pixels (largeur × hauteur)

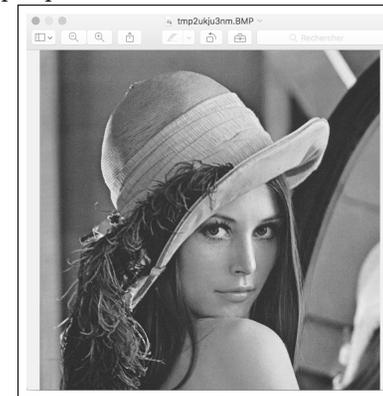
Le fichier est au format `.png`

Le mode de l'image est niveau de gris (`greyscaLe`) : chaque pixel est constitué d'une valeur entre 0 et 255.

Pour afficher l'image à partir de l'objet-image `img` :

```
>>> img.show()
```

affiche la fenêtre graphique :



Les pixels de l'image sont obtenus grâce à la méthode `getdata()` ; il faut effectuer une conversion pour récupérer le tableau des valeurs de ses pixels :

```

>>> pixels_img = list(img.getdata())
>>> pixels_img[:10]
[160, 166, 162, 161, 167, 166, 164, 167, 162, 164]

```

Voilà ses 10 premières valeurs : chacun est un entier entre 0 et 255 (Niveau de Gris).

- Les pixels sont représentés dans un tableau unidimensionnel, ligne après ligne. Le premier pixel est celui en haut à gauche de l'image, et le dernier est celui en bas à droite.

Ainsi les 512 premiers pixels constituent la première ligne, les 512 suivants la deuxième ligne, etc..., les 512 derniers la dernière ligne.

```
>>> len(pixels_img)
262144
>>> 512 * 512
262144
```

14.2.3 Création d'image et sauvegarde

• Copions (slicing) la moitié de l'image du bas pour créer un nouvel objet image :

```
pixels_img_bas = pixels_img[ 512 * 256 : ] # Copie de la
moitié des lignes
demiimg = Image.new('L', (512, 256)) # Création d'un nouvel
objet image
demiimg.putdata(pixels_img_bas) # Insertion de son tableau de
pixels
demiimg.show() # Affichage
```

- La fonction `new()` prend deux paramètres : le mode (ici L) et la taille ; à ce stade elle ne contient encore aucun pixel.
- La méthode `putdata()` permet de définir les pixels d'une image.
- L'affichage produit :



- On peut alors sauvegarder un nouveau fichier image, `Demi_img.png` :

```
demiimg.save('Demi_img.png', format='PNG')
```

14.2.4 Obtention de l'image en négatif

Obtention de l'image "en négatif" :

Pour cela changer la valeur de chaque pixel en $255 - \text{pixel}$: 0 (noir) devient 255 (blanc) et réciproquement : (i.e. appliquer la bijection $x \mapsto 255 - x$ de $[[0, 255]]$ sur lui-même).

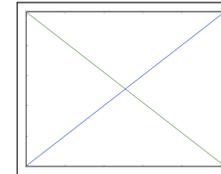
```
pixels_negatif = pixels_img[:]
for k in range(len(pixels_negatif)):
    pixels_negatif[k] = 255-pixels_negatif[k]
img_negatif = Image.new('L', (512,512))
img_negatif.putdata(pixels_negatif)
img_negatif.show()
```

- ou plus simplement en appliquant la méthode `point()` qui permet d'appliquer au tableau de ses pixels une fonction mathématique pixel par pixel :

```
# Application  $x \mapsto 255 - x$  appliquée à chaque pixel ;
def inversion(x): return 255-x
# Le résultat est affecté à un nouvel objet-image
img_negatif = img.point(inversion)
img_negatif.show()
```



Image originale

Image en négatif : $x \mapsto 255 - x$ 

14.3 Traitement d'image

14.3.1 Traitement global

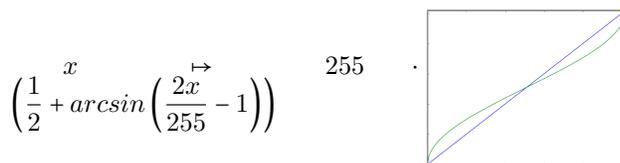
Eclaircissement de l'image

- Pour éclaircir une image il faut augmenter la valeur de chaque pixel.
- Il faut considérer une bijection de $[0, 255]$ dans lui-même qui soit croissante et concave (c'est à dire à dérivée positive et décroissante), par exemple :



- Pour diminuer le contraste appliquer sa bijection réciproque :

$$\forall x \in [0, 1], \quad y = \frac{1}{2} + \frac{1}{2} \cdot \sin\left(\left(x - \frac{1}{2}\right) \cdot \pi\right) \iff x = \frac{1}{2} + \frac{1}{\pi} \cdot \arcsin(2y - 1)$$



```
def f(x): return 0.5 + np.arcsin(2*x - 1)/np.pi
def F(x): return 255 * f(x/555)
imgDecontraste = img.point(F)
imgDecontraste.show()
```



14.3.2 Traitement local

Traitement local de l'image : floutage

- Prenons le cas du floutage d'une image :

On remplace la valeur de chaque pixel par la moyenne des valeurs de ses pixels voisins :

A l'aide d'une "convolution avec le noyau" :

$$\begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

C'est la matrice des pondérations appliquées aux pixels voisins.

- Pour une image de taille (N, M) : Pour tout $0 < i < N - 1$ et $0 < j < M - 1$:
Changer `pixel[i, j]` en :

$$\left(\begin{array}{lll} \text{pixel}[i-1, j-1] & + \text{pixel}[i-1, j] & + \text{pixel}[i-1, j+1] \\ + \text{pixel}[i, j-1] & + \text{pixel}[i, j] & + \text{pixel}[i, j+1] \\ + \text{pixel}[i+1, j-1] & + \text{pixel}[i+1, j] & + \text{pixel}[i+1, j+1] \end{array} \right) / 9$$

- Pour un floutage plus important appliquer plutôt une convolution avec le noyau 5×5 :

$$\begin{pmatrix} 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \end{pmatrix}$$

Module numpy (pour la manipulation de tableaux)

Pour faciliter la programmation on utilise le module numpy : il permet de manipuler des tableaux bidimensionnels :

La fonction `np.array()` permet de convertir une liste en tableau numpy unidimensionnel, une liste de listes en tableau numpy bidimensionnel :

```
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
```

a défini le tableau A bidimensionnel (matrice) $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.

L'élément ligne i colonne j s'obtient grâce à :

```
>>> A[0,0]
1
>>> A[1,2]
6
```

La fonction `np.zeros((n,p))` renvoie un tableau de taille (n,p) rempli de zéros :

```
>>> np.zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

La fonction `np.ones((n,p))` renvoie un tableau de taille (n,p) rempli de uns :

```
>>> np.ones((2,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

La fonction `np.shape(A)` renvoie la taille de A sous forme d'un couple :

```
>>> np.shape(A)
(2,3)
```

La fonction `np.reshape()` permet de redimensionner un tableau : son deuxième argument est un couple d'entiers ou un entier :

```
>>> A
A = np.array([[1,2,3],[4,5,6]])

>>> np.reshape(A, (3,2))
np.array([[1, 2],
         [3, 4],
         [5, 6]])

>>> np.reshape(A, 6)
np.array([1,2,3,4,5,6])

>>> A
A = np.array([[1,2,3],[4,5,6]])
```

Fonction de traitement local d'image

• Ecrivons la fonction qui effectue la convolution de l'image (tableau de pixels) avec un noyau :

```
import numpy as np    # Module pour utiliser des tableaux 2d
                      (matrices)

def convolution(pixels, noyau):
    # S'assurer que le noyau est une matrice carrée avec nbre
    impair de lignes :
    (n1, n2) = np.shape(noyau)    # Taille de noyau
    k = n1 // 2                    # n1 = n2 = 2 * k + 1
    (n, m) = np.shape(pixels)     # Taille du tableau
    pixels2 = np.zeros((n, m))    # Nouveau tableau de pixels

    # Parcours des pixels de l'image
    for i in range(k, n-k):
        for j in range(k, m-k):
            # Nouvelle valeur du pixel (i,j) :
            for iN in range(n1):
                for jN in range(n1):
                    pixels2[i,j] += noyau[iN,jN] *
                                pixels[i-k+iN, j-k+jN]

    return pixels2
```

La fonction retourne un tableau de pixels après convolution par le noyau. L'image obtenue sera bordée par une bande de k pixels noirs.

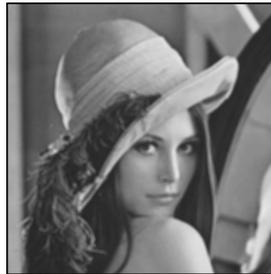
Traitement local de l'image : floutage

```

pixels_img = np.array(img.getdata())
pixels_img = np.reshape(pixels_img, (512,512))
noyau = np.ones((5,5)) / 25
pixels2 = convolution(pixels_img, noyau)
pixels_flou = np.reshape(pixels2, 512*512)

imgFloute = Image.new('L', (512,512))
imgFloute.putdata(pixels_flou)
imgFloute.show()

```



Sur la gauche l'image obtenue ; sur la droite celle obtenue avec un floutage plus important, utilisant comme noyau : `noyau = np.ones((7,7)) / 49`

Netteté - Accentuation de contours

• Pour augmenter la netteté de l'image on peut appliquer un traitement local à l'aide d'une convolution avec le noyau :

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

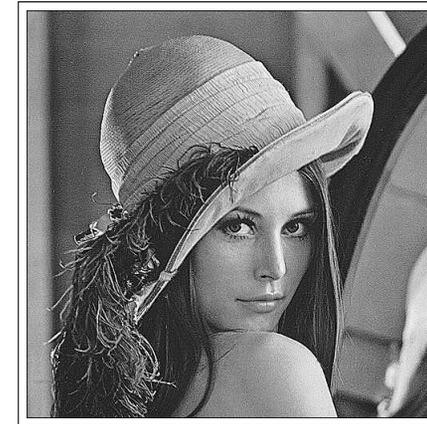
```

# Récupération du tableau des pixels :
pixels_img = np.array(img.getdata())
pixels_img = np.reshape(pixels_img, (512,512))
# Convolution :
noyau = np.array([[0,-1,0],[-1,5,-1],[0,-1,0]])
pixels_net = convolution(pixels_img, noyau)
# Construction de l'image nette :
imgNet = Image.new('L', (512,512))
imgNet.putdata(np.reshape(pixels_net, 512*512))
imgNet.show()

```

On peut aussi appliquer le noyau :

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$



Détection de contour (Hors programme)

- Pour la détection de contour :

Appliquer une convolution avec pour noyau :

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Les pixels ne variant pas de leurs voisins directs deviendront noirs (0), les pixels sombres sur fond clair deviendront plus clairs.

Puis passer l'image obtenue en négatif grâce à l'application $x \mapsto 255 - x$.

```
# Récupération du tableau des pixels :
pixels_img = np.array(img.getdata())
pixels_img = np.reshape(pixels_img, (512,512))
# Convolution :
noyau = np.array([[1,1,1],[1,-8,1],[1,1,1]])
pixels2 = convolution(pixels_img, noyau)
pixels2 = 255 - pixels2 # Passer au négatif
pixels_contour = np.reshape(pixels2, 512*512)
imgContour = Image.new('L', (512,512))
imgContour.putdata(pixels_contour)
imgContour.show()
```



Reéssayer avec les noyaux :

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -24 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 24 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

Exercice 2. Télécharger à l'adresse :

<http://www.i2m.univ-amu.fr/perso/jean-philippe.preaux>
le fichier `pelote-pollenx30.png`.

qui représente des grains de pollen vues au microscope électronique, agrandis 30 fois.

1. Lui appliquer tous les traitements d'image vus dans ce cours.
2. Essayer de trouver vos propres fonctions pour éclaircir/obscurcir l'image, ou changer le contraste.
3. Lui appliquer de la détection de contour avec différents noyaux.