Chapitre 14

Traitement numérique de l'image. Images en niveau de gris

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) :

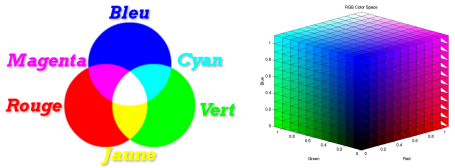
Noir et blanc. Dans ce cas chaque pixel est représenté sur un bit, 0 pour noir, 1 pour blanc. C'est de ce mode que vient le nom "bitmap".

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`.

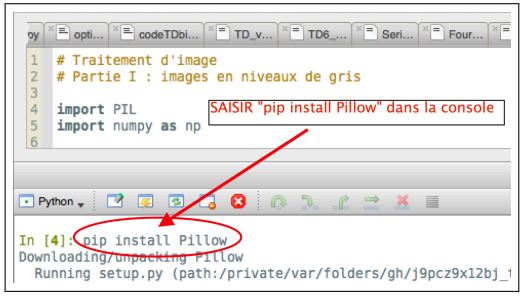
• S'il n'est pas fourni dans la distribution utilisée, il faudra l'installer :

• **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
```



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('image.png')     # 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=RGB size=800x600 at
0x112569400>
>>> img.size
(800, 600)
>>> img.format
'PNG'
>>> img.mode
'RGB'
```

L'image a pour taille 800×600 pixels (largeur × hauteur)

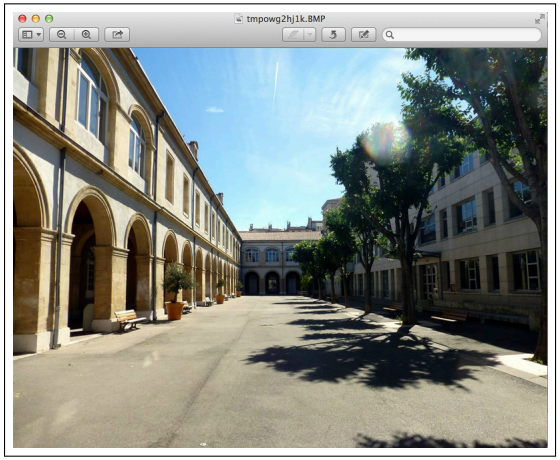
Le fichier est au format `.png`

Le mode de l'image est RGB : chaque pixel est constitué de 3 valeurs (chacune sur 8 bits) des intensités en Rouge, Vert et Bleu.

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[:5]
[(102, 110, 113), (117, 125, 128), (83, 90, 98), (106, 113,
121), (194, 206, 184)]
```

Voila ses 5 premières valeurs : chacun est un triplet de nombres entre 0 et 255 (RGB).

• 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 800 premiers pixels constituent la première ligne, les 800 suivants la deuxième ligne, etc..., les 800 derniers la dernière ligne.

```
>>> len(pixels_img)
480000
>>> 800 * 600
480000
```

14.2.3 Création d'image et sauvegarde

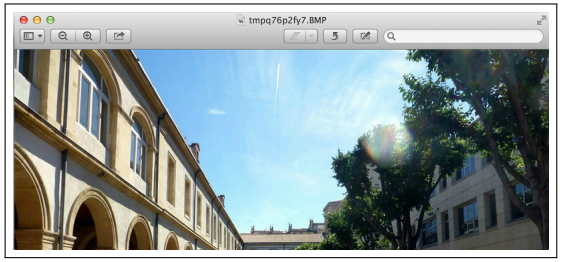
• Copions (slicing) la moitié de l'image du haut pour créer un nouvel objet image :

```
pixels_img_haut = pixels_img[ : 800 * 300]    # Copie de la
moitié des lignes
demiimg = Image.new('RGB', (800, 300))    # Création d'un
nouvel objet image
demiimg.putdata(pixel_img_haut)    # Insertion de son tableau de
pixels
demiimg.show()    # Affichage
```

• La fonction `new()` prend deux paramètres : le mode (ici RGB) 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 Conversion en niveau de gris

Convertissons l'image au mode `'L'` : en niveau de gris :

```
imgGris = img.convert('L')
imgGris.show()
```

On peut aussi la sauvegarder :

```
imgGris.save('imgGris.png', format='PNG')
```

• L'image est convertie du mode RGB au mode niveau de gris selon la formule :

$$L = R \times \frac{299}{1000} + G \times \frac{587}{1000} + B \times \frac{114}{1000}$$

• On obtiendrait donc le même résultat avec le script :

```
# Constitution du tableau :
pixels_imgGris = [ ]
for (r,g,b) in pixels_img:
    pixels_imgGris.append(r * 0.299 + g * 0.587 + b * 0.114)
# Conception de l'objet-image :
imgGris = Image.new('L', img.size)
imgGris.putdata(np.array(pixels_imgGris, dtype = np.uint8))
```

L'option `dtype = np.unint8` passé en argument à la conversion `np.array()` impose que les données soient converties en des types d'entiers non signés stockés sur 1 octet.

• Les pixels de l'image ne sont plus constitués que d'une valeur entre 0 et 255, représentées sur un octet (de type : `np.uint8`).

```
import numpy as np
pixels_imgGris = np.array(imgGris.getdata())
```

```
>>> len(pixels_imgGris)
480000
>>> pixels_imgGris[ : 10]
array([107, 122, 88, 111, 199, 226, 238, 174, 53, 55])
```

• Exemple : obtention de l'image "en négatif" :

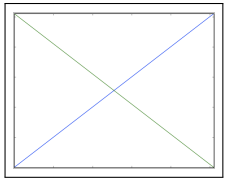
Pour cela changer la valeur de chaque `pixel` en 255-`pixel` : 0 (noir) devient 255 (blanc) et réciproquement : (*i.e.* appliquer la bijection $x \mapsto 255 - x$ de $[[0, 255]]$ sur lui-même).

```
pixels_negatif = 255 - pixels_imgGris
img_negatif = Image.new('L', (800,600))
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 --> 255 - x appliquée à chaque pixel ;
# Le résultat est affecté à un nouvel objet-image
img_negatif = imgGris.point(lambda x: 255 - x)
img_negatif.show()
```

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



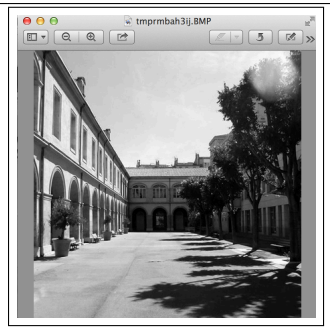
14.3 Traitement d'image

14.3.1 Extraction d'image

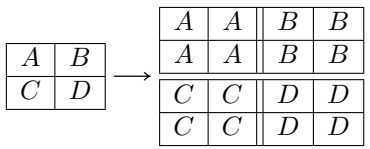
• Extrayons de l'image la partie délimitée par le rectangle allant des pixels 100 à 499 en hauteur et 200 à 599 en largeur (marge de 100 verticalement et 200 horizontalement).

Pour cela il est intéressant de convertir les données à l'aide de `np.reshape()` en un tableau bi-dimensionnel ayant 600 lignes et 800 colonnes :

```
pixels_imgGris = np.reshape(pixels_imgGris ,(600,800))  #
Conversion 2d
pixels_fenetre = pixels_imgGris[100:500, 200:600]    #
Extraction par slicing
fenetre = Image.new('L', (400,400))         # Nouvelle image
400x400
pixels_fenetre = np.reshape(pixels_fenetre, 400*400) # 2d -> 1d
fenetre.putdata(pixels_fenetre)             # Remplissage données
fenetre.show()
```



Nous allons doubler la taille de cette image "grossièrement" : un pixel deviendra un carré de 2×2 pixels :



Algorithme :

• Pour un indice de ligne pair :

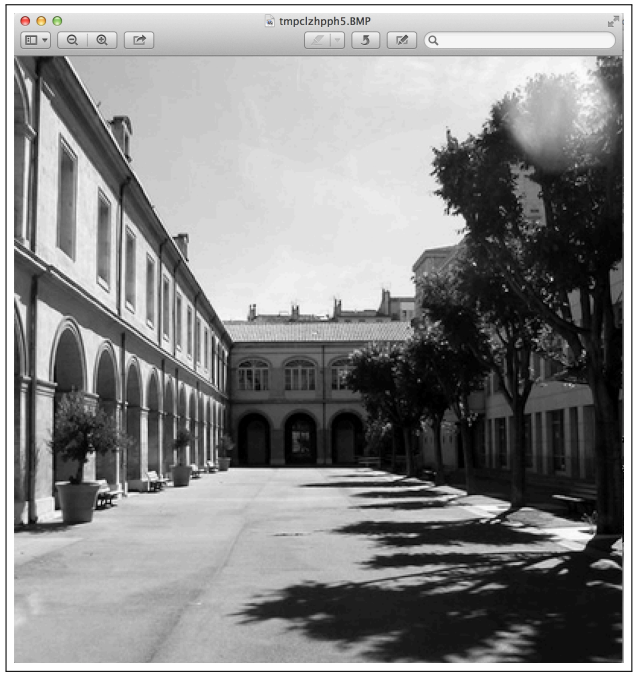
`[A, B, ..., Z]` ⟶ `[A, A, B, B,..., Z, Z]`

• Pour un indice de ligne impair :

Recopier la ligne précédente.

```
pixels_zoom = np.empty((800,800))
for i in range(800):
    if i%2 == 0:
        for j in range(800):
            pixels_zoom[i, j] = pixels_fenetre[i//2,j//2]
    else:
        pixels_zoom[i, :] = pixels_zoom[i-1, :]
zoom = Image.new('L', (800,800))
zoom.putdata(np.reshape(pixels_zoom,800*800)) # remplissage
zoom.show()
```

On travaille avec des tableaux 2d ; on convertit en tableau 1d avant d'appliquer les données.

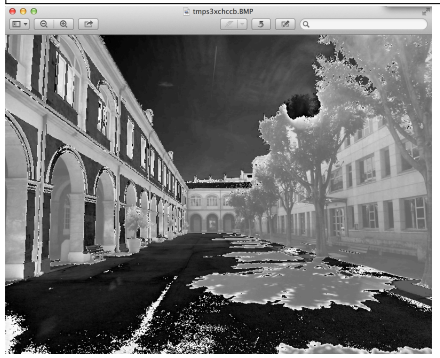


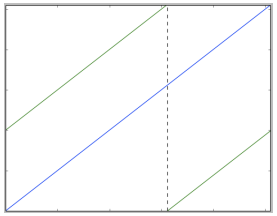
14.3.2 Traitement global

- Pour éclaircir une image il faut augmenter la valeur de chaque pixel.

- Mais il ne faut pas ajouter une même valeur à chaque pixel : se rappeler que les pixels étant représentés sur 1 octet on calcule dans $\mathbb{Z}/256\mathbb{Z}$: un pixel suffisamment clair deviendrait alors sombre :

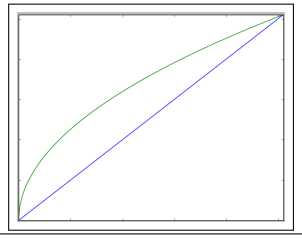
```
# Ajouter 100 à chaque pixel :
imgEclaircie = imgGris.point(lambda x : x + 100)
imgEclaircie.show()
```



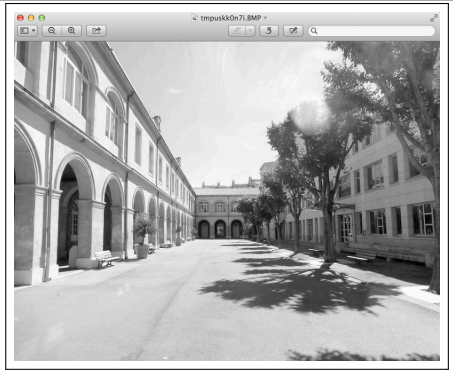


- Il faut appliquer plutôt, par exemple, la bijection de $[0,255]$ dans lui-même :

$$x \mapsto 255 \cdot \sqrt{\frac{x}{255}}$$

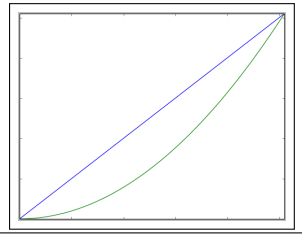


```
from math import sqrt
imgEclaircie = imgGris.point(lambda x: 255*sqrt(x/255))
imgEclaircie.show()
```

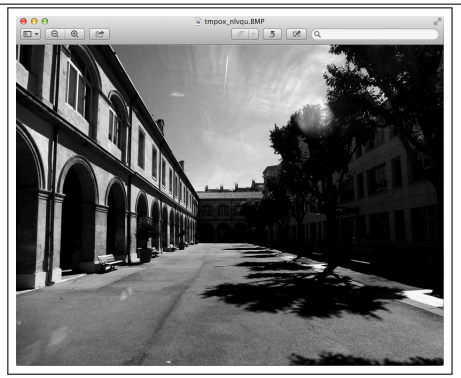


• Pour obscurcir l'image appliquons à ses pixels :

$$x \mapsto 255 \cdot \left(\frac{x}{255}\right)^2$$

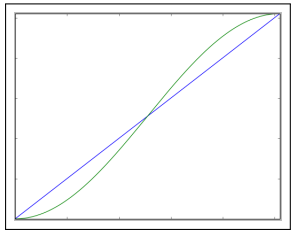


```
imgObscurcie = imgGris.point(lambda x: 255*(x/255)**2)
imgObscurcie.show()
```



• Pour augmenter le contraste de l'image il faut éclaircir les clairs et obscurcir les foncés : en partant de la bijection de $[0,1]$ sur lui-même : $x \mapsto \frac{1}{2}+\frac{1}{2}\cdot\sin\left(\left(x-\frac{1}{2}\right)\cdot\pi\right)$, on construit la bijection de $[0,255]$ sur lui-même :

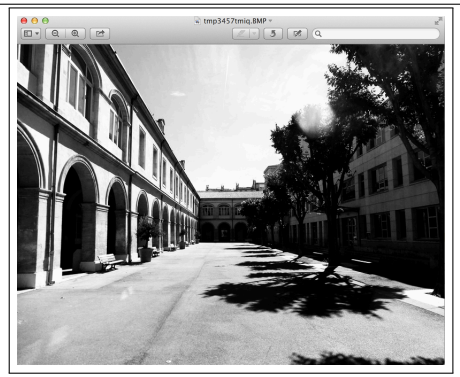
$$x \mapsto 255 \cdot \left(\frac{1}{2} + \frac{1}{2} \cdot \sin\left(\left(\frac{x}{255} - \frac{1}{2}\right)\cdot\pi\right)\right)$$



Foncés Clairs

• Augmentation du contraste :

```
f = lambda x : 0.5 + 0.5 * np.sin((x-0.5)*np.pi)
F = lambda x : 255 * f(x/255)
imgContraste = imgGris.point(F)
imgContraste.show()
```



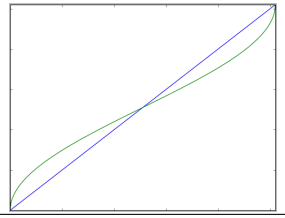
• Plus grande augmentation du contraste :

```
f = lambda x : 0.5 + 0.5 * np.sin((x-0.5)*np.pi)
F = lambda x : 255 * f(x/255)
imgContraste = imgGris.point(lambda x: F(F(x)))
imgContraste.show()
```

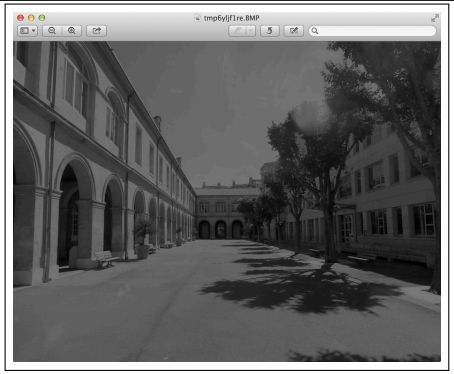
• Pour diminuer le contraste appliquer sa bijection réciproque :

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

$$x \mapsto 255 \cdot \left(\frac{1}{2} + arcsin\left(\frac{2x}{255} - 1\right)\right)$$



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



14.3.3 Traitement local

• 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 :

```
( pixel[i-1,j-1]   + pixel[i-1,j]   + pixel[i-1,j+1]
+ pixel[i,j-1]     + pixel[i,j]     + pixel[i,j+1]
+ pixel[i+1,j-1]   + pixel[i+1,j]   + pixel[i+1,j+1]) / 9
```

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

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

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

```
def convolution(pixels, noyau):
    # S'assurer que le noyau est une matrice carrée
    # avec nbre impair de lignes :
    (n1, n2) = np.shape(noyau)
    assert n1 == n2 and n1 % 2 == 1
    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.

On ne convertit pas en `uint8` : la conversion en `uint8` transformerait `pixel` en `pixel % 256`.
Tandis que lors du remplissage par `putdata()` de l'image, tout pixel > 255 sera changé en 255 (blanc) et toute pixel < 0 sera changé en 0. Ce qui facilite le traitement local.

• Appliquons au floutage de l'image :

```
pixels_img = np.array(imgGris.getdata())
pixels_img = np.reshape(pixels_img, (600,800))
noyau = np.ones((5,5)) / 25
pixels2 = convolution(pixels_img, noyau)
pixels_flou = np.reshape(pixels2, 600*800)

imgFloute = Image.new('L',(800,600))
imgFloute.putdata(pixels_flou)
imgFloute.show()
```

• 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(imgGris.getdata())
pixels_img = np.reshape(pixels_img, (600,800))
# 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',(800,600))
imgNet.putdata(np.reshape(pixels_net, 600*800))
imgNet.show()
```

• Pour la détection de contour :

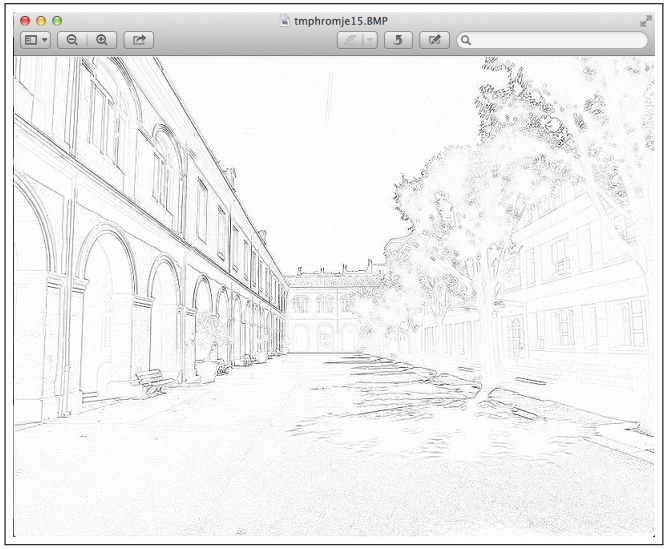
Appliquer une convolution avec pour noyau :

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

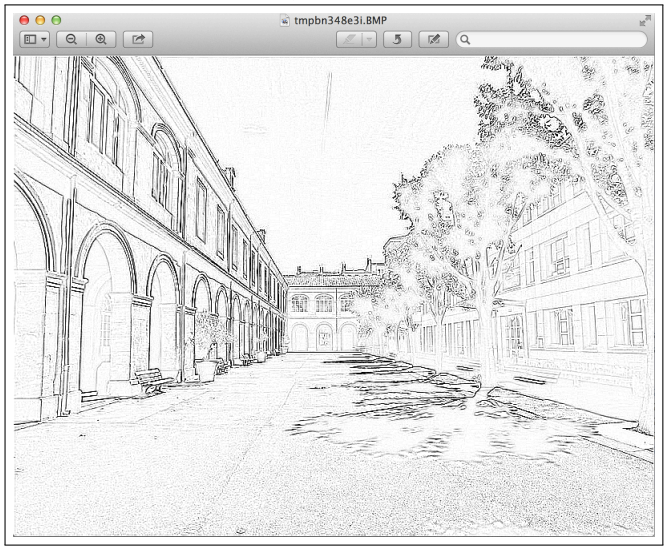
Les pixels ne variant pas de leurs voisins directs deviendront noirs (0), les pixels sombres sur fond clair deviendront plus clairs.

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

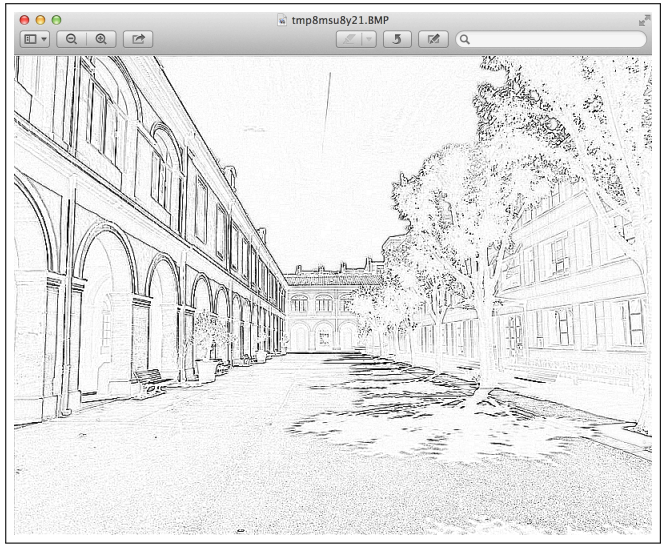
```
# Récupération du tableau des pixels :
pixels_img = np.array(imgGris.getdata())
pixels_img = np.reshape(pixels_img, (600,800))
# Convolution :
noyau = np.array([[0,1,0],[1,-4,1],[0,1,0]])
pixels2 = convolution(pixels_img, noyau)
pixels2 = 255 - pixels2    # Passer au négatif
pixels_contour = np.reshape(pixels2, 600*800)
imgContour = Image.new('L',(800,600))
imgContour.putdata(pixels_contour)
imgContour.show()
```



avec le noyau : $\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$: détecte les sombres sur fond clair.



avec le noyau : $\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$: détecte les clairs sur fond sombre.



Exercice : Améliorer l'algorithme permettant de doubler la taille de l'image qui évite le phénomène de "pixellisaton".

Pour cela constituer d'abord le tableau :

$$\begin{matrix} A & B \\ C & D \end{matrix} \longrightarrow \begin{matrix} A & 0 & B & \dots \\ 0 & 0 & 0 & \dots \\ C & 0 & D & \dots \\ 0 & 0 & 0 & \dots \end{matrix}$$

avant de lui appliquer une convolution avec un noyau 3×3 bien choisi.

On rappelle d'abord les instructions ayant permis d'extraire une vignette :

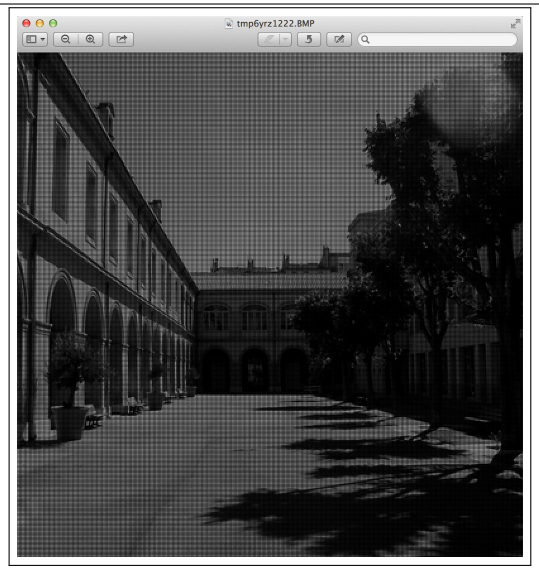
```
pixels_imgGris = np.array(imgGris.getdata())
pixels = np.reshape(pixels_imgGris,(600,800))
pixels_fenetre = pixels[100:500, 200:600]
```

Création du premier tableau de pixel, avant traitement :

```
pixels_zoom = np.zeros((800,800))
for i in range(800):
    if i%2 == 0:
        for j in range(800):
            if j%2 == 0:
                pixels_zoom[i, j] = pixels_fenetre[i//2,j//2]
```

Afichage de l'image parcellaire :

```
zoom = Image.new('L', (800,800))
zoom.putdata(np.reshape(pixels_zoom,800*800))
# Affichage image parcellaire
zoom.show()
```



On applique pour noyau de convolution :

$$\begin{pmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{pmatrix}$$

0	0	0
0	**A**	0
0	0	0

$\longrightarrow$

0	0	0
0	**A**	0
0	0	0

0	0	0
A	**0**	**B**
0	0	0

$\longrightarrow$

0	0	0
A	$(\mathbf{A}+\mathbf{B})/\mathbf{2}$	**B**
0	0	0

0	**A**	0
0	**0**	0
0	**C**	0

$\longrightarrow$

0	**A**	0
0	$(\mathbf{A}+\mathbf{C})/\mathbf{2}$	0
0	**C**	0

A	0	**B**
0	**0**	0
C	0	**D**

$\longrightarrow$

A	0	**B**
0	$(\mathbf{A}+\mathbf{B}+\mathbf{C}+\mathbf{D})/4$	0
C	0	**D**

```
noyau = np.array([[0.25,0.5,0.25],[0.5,1,0.5],[0.25,0.5,0.25]])
pixels_zoom2 = convolution(pixels_zoom, noyau)

zoom.putdata(np.reshape(pixels_zoom2,800*800))
# Affichage image de taille double
zoom.show()
```