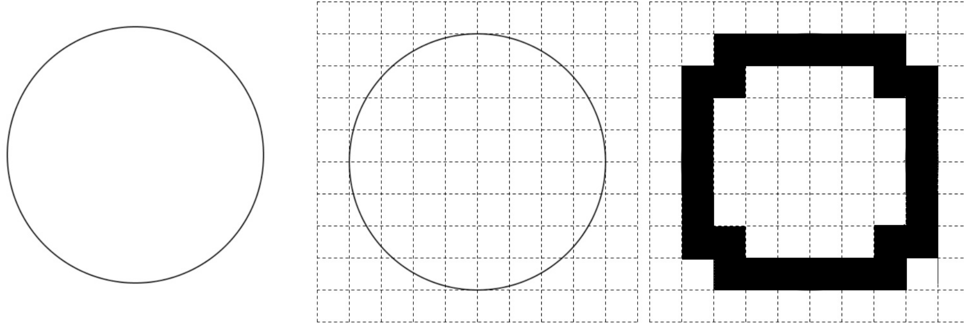
1 Introduction

Pour représenter une image, une possibilité est de lui superposer une grille, dont on appelle chaque case un pixel (*picture element*). On colorie alors chaque case d'une couleur uniforme. L'image pixellisée constitue une approximation plus ou moins fidèle l'image de départ. Plus la grille est fine, plus l'image sera lisse mais plus le nombre de données sera important.



Une image numérique peut donc être vue comme un tableau (matrice) à n lignes et p colonnes dont les coefficients sont les valeurs des pixels.

dans l'exemple précédent :

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

Une image en noir et blanc se traduit directement en une matrice de bits : 0 pour noir et 1 pour blanc. On parle donc de représentation *bitmap* des images (carte des bits).

Si on veut introduire un peu plus de nuances, il faudra attribuer à chaque pixel un peu plus de d'un bit.
Dans le cas d'une image en niveau de gris, la valeur du pixel est généralement un entier entre 0 (noir) et 255 (blanc) (soit 256 valeurs possibles entre $(0000\ 0000)_2$ et $(1111\ 1111)_2$). On utilise donc un octet (8 bits) par pixel.

Et pour la couleur ?
Une image couleur est en fait la réunion de trois images : une rouge, une verte et une bleue. Cette représentation s'apparente au fonctionnement du système visuel de l'homme (nous avons au fond de notre rétine trois types de récepteurs selon la longueur d'onde recue...). Chaque pixel est alors représenté par un triplet d'entiers (Red, Green, Blue) , dont chacun est codé sur un octet. Le codage utilise donc $3 \times 8 = 24$ bits par pixel. Cela permet d'encoder 2^{24} soit un peu plus de 16 millions de couleurs différentes (bien que nous n'en différencions en fait pas autant).

Par exemple, (255,0,0) correspond à un pixel rouge, (0,0,0) noir, (255,255,255) blanc, (100,0,120) ....
remarque : le jaune est un mélange de rouge et de vert.

Si l'image précédente est encodée en Noir et blanc, elle est de taille : ....
Si l'image précédente est encodée en niveau de gris sur un octet, elle est de taille : ....
Si l'image précédente est encodée en RGB, elle est de taille : ....

Dans la suite du chapître, les traitements ne s'appliquent qu'à des images en niveaux de gris. Pour des images en couleur, il est possible d'opérer en appliquant les traitements proposés à chacune des trois couleurs !

Voilà une image en niveau de gris qui va nous servir de base pour les traitements :

La fonction *Image.open(chemin d'accès)* permet de créer un objet python « image » correspondant à l'image du chemin fourni. Cet objet a plusieurs attributs dont deux nous interessent plus particulièrement :
- ses dimensions
- les valeurs attribuées à chaque pixel (en ligne).

Si l'on veut visualiser correctement la matrice des données, il convient donc de mettre cette liste de données en forme.
La fonction *OuvrirImagegris(path)* qui vous sera fournie en TP vous permet d'obtenir directement les données sous forme « matricielle » (ce sont des tableaux numpy !).

Attention, la taille d'une matrice est donnée sous la forme nombre de lignes × nombre de colonnes, c'est à dire hauteur × largeur. Or les dimensions des images sont généralement données sous la forme largeur × hauteur. Il faut donc être vigilant.

Les dimensions de cette image sont : .... Cela représente ....bits.

```
def OuvrirImgGris(path):
    im = Image.open(str(path))   #crée un objet im qui contient l'image située à l'adresse indiquée
    largeur,hauteur = im.size   #méthode size récupère les dimensions
    imdata=im.getdata()  #la methode getdata permet de récuperer les valeurs des pixels en liste
    tab=numpy.array(imdata)  #met les données en tableau
    matrix = numpy.reshape(tab,(hauteur,largeur))  #met les données sous forme de matrice
    return matrix


matrice1=OuvrirImgGris("D:/fichierspython/photogris.bmp")
#print(matrice1)
#print(matrice1.shape)

#créer une image a partir d'une matrice
def CreerImgGris(matrix,path): #matrice de l'image à reconstruire, plus chemin de sortie.
    size=(matrix.shape[1],matrix.shape[0]) #entre matrice et image l'ordre des dimensions est inversé
    im2 = Image.new("L",size)
    im2.putdata(list(matrix.flat))
    im2.save(fp=str(path))
```

2 pixellisation

Afin de réduire l'espace utilisé pour stocker une image, on a vu qu'on pouvait utiliser une grille de pixellisation moins fine. Que fait le code suivant ?

```
#pixelliser

matrice2=numpy.array([[0]*320]*240)

for i in range (240):
    for j in range(320):
        matrice2[i][j]=matrice1[i*2][j*2 ]

CreerImgGris(matrice2,"D:/fichierspython/photogrispixellisee.bmp")
```

3 Un exemple de traitement pixel par pixel : modifier les contrastes

Le contraste peut être modifié par l'application d'une fonction mathématique (de $[\![0;255]\!]$ dans $[\![0:255]\!]$) sur la valeur de chaque pixel.

La modification la plus drastique consiste à obtenir du noir et blanc. (voir code)

Si l'on remplace la valeur de chaque pixel en utilisant la fonction carrée ramenée à $[\![0:255]\!]$ ($f(x) = x^2/255$), l'image est assombrie.
Si on utilise la fonction racine carrée ramenée à $[\![0:255]\!]$ ($g(x) =$....................), l'image est éclaircie.
Ces deux fonctions ont tendance à diminuer les contrastes.

Une autre idée consiste à faire un réajustement linéaire des valeurs des pixels selon :

$$f(x) = \begin{cases} y = x + 0.4(x - 127) & \text{si } y \in [\![0;255]\!] \\ 0 & \text{si } y < 0 \\ 255 & \text{si } y > 255 \end{cases} \tag{1}$$

En TP vous programmerez ce réajustement. Quel est son effet ?

4 Un exemple de traitement local : les filtres

Filtrer une image consite à lui appliquer une transformation mathématique modifiant la valeur des pixels. Un filtre F est défini par une matrice carrée d'ordre n impair, qui se déplace sur l'image en modifiant la valeur du pixel central selon les valeurs de ses voisins.

prenons l'exemple d'un filtre F de taille 3. $F = \begin{pmatrix} f_{0,0} & f_{0,1} & f_{0,2} \\ f_{1,0} & f_{1,1} & f_{1,2} \\ f_{2,0} & f_{2,1} & f_{2,2} \end{pmatrix}$ Appliquons le à la matrice A d'une image.

On superpose le filtre à la matrice carrée extraite de A centrée sur la valeur $a_{i,j}$.
On effectue les produits terme à terme.
On « remplace » $a_{i,j}$ par la somme de ces produits terme à terme.

$a_{i,j} \rightarrow b_{i,j} =$

On parle de convolution discrète (opérateur *) :

remarque : Les bords de l'image est toujours exclu du traitement local !

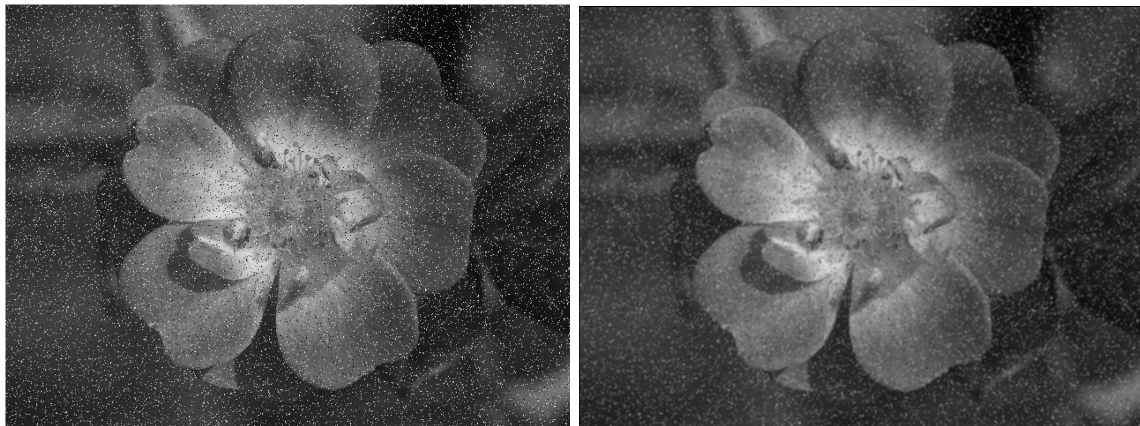
Vous programmerez en TP une fonction filtrage(A,F) prenant en argument deux tableaux numpy (A celui de l'image et F celui du filtre) et renvoyant un tableau numpy B, de même taille que A, contenant les valeurs de A filtrées par F.

A quoi cela peut il servir ?
• restaurer une image
• détecter les contour

4.1 application à la restauration d'image

Parfois certains défauts (poussières, fluctuation de l'intensité électrique, erreurs de transmissions...) conduisent à avoir une image « abimée ». Les valeurs de certains pixels sont « erronées »., d'où des taches de petites tailles à certains endroits aléatoires de l'image. On parle alors de bruit.

exemple :



Le filtre de moyenne d'ordre 3 remplace chaque pixel par la moyenne de ses 9 voisins (lui compris).

La matrice associée est donc :

A droite, l'image obtenue après restauration de l'image abimée proposée par le filtre de la moyenne. On constate que le bruit est bien partiellement réduit mais que le filtrage provoque un certain flou.
Il existe d'autres filtres plus efficaces, comme le filtre médian (qui consiste a prendre la médiane et plus la moyenne) mais qui ne correspond pas à une convolution.

4.2 Détection des contours en TP

5 Pour aller plus loin... jouer avec la couleur en TP