INITIATION AU TRAITEMENT D'IMAGE avec NUMPY

par François louis LAILLIER

Date de publication : 04/06/2006

Dernière mise à jour : 19/03/2007

A la fin de ce tutoriel vous saurez (si mon premier tuto est réussi) récupérer les données brutes d'une image, les mettre en forme afin de les traiter et de reconstruire l'image résultat. Tout cela sera fait avec l'aide des librairies ***PIL*** et ***NumPy***. Côté traitement d'images, nous traiterons la segmentation à deux seuils, la dilatation et l'érosion. Avant de commencer je tiens a remercier _GUIGUI sans qui ce tutoriel, alors qu'il était presque terminé, n'aurait jamais été publié.

I - PRÉREQUIS

I-A - Prérequis

Pour effectuer ce tutoriel, vous devez avoir un minimum de connaissances sur les données des images. Il vous faut aussi connaître un minimum la bibliothèque ***Python Image Library (PIL)***. Une connaissance de ***NumPy*** est inutile. Je tiens à souligner que la pluspart des choses acquises au cours de ce tutoriel sont faisables directement avec ***PIL***. *PIL* reste quand même limitée à mon goût. Si vous savez faire ce qui est expliqué dans ce tutoriel, vous saurez faire vos propres opérateurs.

I-B - Version PYTHON et librairies utilisées

- **PYTHON 2.5**
- **PIL 1.1.6**
- **NumPy**
- Système d'exploitation: Windows XP HOME
- Tests effectués avec IDLE 1.2

Vous pouvez également accéder à ces pages depuis notre rubrique **Outils Python**

II - MANIPULATIONS BASIQUES DES IMAGES

Dans cette partie du tutoriel nous apprendrons à manipuler les données brutes issues des images. Nous apprendrons aussi à reconstruire une image grâce à ces données.

II-A - Ouvrir une image et extraire les données

Dans ce paragraphe, le but est d'ouvrir une image en précisant son chemin et remettre en forme les données afin de pouvoir les traiter. Afin de vérifier si votre code fonctionne je vous conseille de créer une image test, une image de petite dimension dont il sera facile d'imprimer à l'écran sa composition.

Exemple d'image:

test.bmp

Ainsi nous aurons quelquechose de ce type comme données brutes.

```
0,0,0,0,0,0
0,1,1,1,1,0
0,0,0,0,0,0
```

Un fois cela fait, nous allons importer les modules nécessaires à ce tutoriel.

Importation des modules

```
IDLE 1.2
>>> import PIL
>>> import Image # on aurait put faire from PIL import Image
>>> import numpy
>>> dir()
['Image', 'PIL', '__builtins__', '__doc__', '__name__']
```

Nous allons la mettre en niveau de gris parce qu'une image est en générale composée de trois composantes de couleurs voir quatre: RGB,RGBA,HSI. Nous travaillerons sur une image grise, image qui pourrait être une image du canal R. (chaque pixel du canal est codé sur un octet ici. D'où l'intérêt de travailler sur du gris. Image monocanal) Pour cela on utilise la fonction ***grayscale*** du module ***ImageOps***:

```
>>> import ImageOps
>>> img=ImageOps.grayscale(img)
```

Nous allons maintenant extraire toutes les données brutes de l'image. On fait cela grâce à la fonction ***getdata()***

```
>>> imgdata = img.getdata()
>>> print list(imgdata)
```

```
[0, 0, 0, 0, 0, 0, 0, 255, 255, 255, 255, 0, 0, 0, 0, 0, 0, 0]
```

On voit bien qu'après le ***getdata*** nous avons les données de l'image sous forme d'une liste 1D avec tous les pixels les uns à la suite des autres. Il faut donc remettre cette liste sous forme de matrice. Pour cela, on a besoin des dimensions de l'image. On utilisera l'attribut **size**

```
>>> larg, haut = img.size
>>> larg
6
>>> haut
3
```

C'est ici que ***NumPy*** intervient, on va se servir de ***NumPy*** qui est un module qui gère bien les tableaux. Nous allons mettre les données que nous avons recupérées dans un tableau (***array***)

```
>>> tab = numpy.array(imgdata)
```

Si vous tapez dans la console :

```
>>> tab
array([  0,   0,   0,   0,   0,   0,   0, 255, 255, 255, 255,   0,   0,
         0,   0,   0,   0,   0])
```

Alors que si vous tapez:

```
>>> print tab
[  0   0   0   0   0   0   0 255 255 255 255   0   0   0   0   0   0   0]
```

La différence nous montre que si nous appelons ***tab***, le shell nous dit que c'est un tableau qui a les données suivantes, et nous indique le type si nous en avons spécifié un, alors que si nous imprimons ***tab***, la console nous indique seulement les données du tableau (non séparées pas une virgule)

Nous sommes au même point que précédemment nos données se présentent sous une forme 1D. Pour mettre ces données en forme il faut se pencher du côté des ***shape*** (forme).

```
>>> print numpy.shape(tab)
(18,)
```

Cela nous indique les dimensions x,y d'une liste ou tableau. En utilisant la fontion ***reshape***, nous allons mettre la matrice en forme.

```
>>> matrix = numpy.reshape(tab,(larg,haut))
>>> matrix
array([[  0,   0,   0],
       [  0,   0,   0],
       [  0, 255, 255],
       [255, 255,   0],
       [  0,   0,   0],
       [  0,   0,   0]])
>>> matrix= numpy.reshape(tab,(haut,larg))
>>> matrix
```

```
array([[  0,   0,   0,   0,   0,   0],
       [  0, 255, 255, 255, 255,   0],
       [  0,   0,   0,   0,   0,   0]])
```

*On voit bien ici que le souci est dans la compréhension, quand on donne les dimensions d'une image on donne souvent largeur*hauteur (cf plus haut avec la fonction **size**) alors que lorsqu'on travail avec des matrices c'est toujours donné sous le format lignes*colonnes.*

Bon voilà nous avons vu les commandes à connaître pour faire une fonction *OuvrirImg*.

Fonction OuvrirImg

```
def OuvrirImg(path):
    Img = Image.open(str(path))
    Img1 = ImageOps.grayscale(Img)
    largeur,hauteur = Img1.size
    imdata=Img1.getdata()
    tab=numpy.array(imdata)
    matrix = numpy.reshape(tab,(hauteur,largeur))
    return matrix
```

```
>>> a = OuvrirImg("c:\\test.bmp")
>>> print a
[[  0   0   0   0   0   0]
 [  0 255 255 255 255   0]
 [  0   0   0   0   0   0]]
```

II-B - Reconstruire une image à partir d'une matrice

Maintenant nous savons ouvrir une image et mettre ses données sous forme de matrice. Le but de ce paragraphe est de reconstruire une image à partir d'une matrice de type ***ARRAY***.

```
>>> a = OuvrirImg("c:\\test.bmp")
```

Nous avons chargé l'image dans la variable *a*. Il faut créer une nouvelle image afin d'écrire dedans. Cela se fait avec la fonction ***new*** du module ***Image***, il faut que l'image soit de la taille du tableau.

```
>>> Copie = Image.new("L",(a.shape[1],a.shape[0]))
```

Ici nous venons de créer l'objet image *Copie*, une image en niveau de gris. ***(mode='L')***, de dimension (nombre de colonnes x le nombre de lignes). Cette image est vide. Nous allons la remplir. Pour cela la fonction réciproque de ***getdata()*** est utilisée. Comme ***getdata*** nous retourne les données sous forme de liste 1D, il faudra une liste 1D comme argument pour ***putdata***. Nous allons donc mettre à 'plat' la matrice à l'aide de l'attribut **flat**. Pour voir le resultat de façon concrète nous sommes obligés d'utiliser ***list(a.flat)***

```
>>> print a
array([[  0,   0,   0,   0,   0,   0],
       [  0, 255, 255, 255, 255,   0],
       [  0,   0,   0,   0,   0,   0]]

>>> list(a.flat)
[0, 0, 0, 0, 0, 0, 0, 255, 255, 255, 255, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> Copie.putdata(list(a.flat))
```

Les données sont intégrées a l'image. Maintenant deux options:

- Soit l'affichage direct de l'image
- Soit la sauvegarde du fichier

Pour l'affichage on utilise la fonction ***show***.

```
>>> Copie.show()
```

Pour la sauvegarde, nous utiliserons la fonction ***save*** en indiquant comme argument le chemin.

```
>>> Copie.save(fp="C:\\Rebuilt.bmp")
```

Il ne nous reste plus qu'à aller voir sur notre disque si l'image *rebuilt.bmp* existe. Nous pouvons programmer la fonction *rebuildImg* grâce aux commandes vues ci dessus.

Fonction RebuildImg

```
def RebuildImg(data,path): #data de limage a reconstruire, plus chemin de sortie.
    Copie = Image.new("L",(data.shape[1],data.shape[0]))
    Copie.putdata(list(data.flat))
    Copie.save(fp=str(path))
```

```
>>> RebuildImg(a,"c:\\IMG.bmp")
```

Après ces deux paragraphes, nous savons prendre les données brutes de l'image et reconstruire une image. Dans les prochains paragraphes, le traitement d'image sera le centre d'intéret.

III - TRAITEMENT D'IMAGE BASIQUE

Pour ce qui est du traitement d'image, nous ferons des choses basiques:

- Une fonction seuillage avec possibilité de donner deux seuils.
- Une érosion compatible sur les images binaires et en niveaux de gris.
- Une dilatation compatible sur les image binaires et en niveaux de gris.

Cette partie sera constituée d'un rappel de cours sur les morphologies mathématiques puis de l'application en programmation.

III-A - Segmentation

III-A-1 - Rappel de Cours

La segmentation est l'opération qui permet de séparer une image en deux ensembles d'objets. Le critère de sélection est la valeur des pixels présents dans l'image. Le but étant de créer deux populations, nous allons séparer l'ensemble des pixels en deux ensembles distincts. Nous mettrons dans une population les pixels ayant pour valeurs des valeurs supérieures au seuil donné, ce sera la population *blanche*, ou *objet*. Dans l'autre population, nous mettrons les pixels ayant des valeurs inférieures au seuil donné, ce sera la population *noire* , ou *fond*. On peut exécuter un double seuillage, c'est à dire établir la population *blanche* étant la population répondant à deux critères de seuils. Par exemple la population blanche sera les pixels ayant une valeur inférieure au seuil 1 mais supérieur au seuil 2.

III-A-2 - Application à la programmation

Je vais écrire un algorithme simple mais qui nous permettra de bien comprendre le but recherché. Ensuite je décrirai les commandes permettant de mettre en oeuvre le seuillage.

Algorithme du seuillage

```
Faire ouvrir Image à seuiller
Créer nouveaux tableau
Faire pour tout i dans l'image a seuiller
    regarder pixel[i]
    si pixel[i] inférieur seuil 1 et si pixel[i] supérieur seuil 2
        mettre tableau[i] à 1
    sinon mettre tableau[i] à zéro.
    Fin si
    Créer l'image résultat
Fin faire
```

Nous admettrons pour la suite du développement que la variable **Img** est le résultat de l'ouverture d'une image par la fonction **OuvrirImg()**.

```
>>> Img = OuvrirImg("c:\\test.bmp")
```

Tout d'abord, je tiens à souligner que faire un seuillage sur *test.bmp* n'est pas judicieux, car il n'y a que deux valeurs de pixels. Donc les pixels sont déjà triés en deux populations. Une fois la fonction programmée, elle marchera pour les images en niveaux de gris. Je vais développer plusieurs solutions afin que vous puissez voir comment on peut faire

la même chose de façon différente. Pour commencer nous allons construire le nouveau tableau. Comme vous avez pu le voir dans la FAQ, il existe plusieurs façons pour créer un tableau, Tout dépend de comment on veut procéder.

```
>>> tab=[0]*2 #créé une liste de 0 de longueur 2
>>> tab
[0, 0]
```

Il est peut être judicieux d'utiliser la fonction ***append***.

```
>>> tab.append(12)
>>> tab
[0, 0, 12]
```

Il faut se rendre compte d'une chose, notre image a deux dimensions. Pour le seuillage, travailler à une dimension est suffisant. Donc nous fabriquerons un tableau avec la fonction ***append***. Vous aurez compris que nous travaillerons sur notre matrice au format *"FLAT"*. Nous utiliserons une boucle ***for*** pour parcourir les pixels. Nous ferons un test de comparaison des valeurs des pixels.

```
>>> Img = OuvrirImg("c:\\test.bmp")
>>> ImgF = list(Img.flat)
>>> SeuilHaut = 128
>>> SeuilBas = 25
>>> Tab = []
>>> for i in range(len(ImgF)):
        if ImgF[i]<SeuilBas and ImgF[i] > SeuilHaut:
            Tab.append(1)
        else:Tab.append(0)
>>> print Tab
```

Fonction Seuillage

```
def Seuillage(img,seuil_B,seuil_H):
    imgF=list(img.flat)
    Threshold=[]
    for i in range(len(imgF)):
        if imgF[i] < seuil_H and imgF[i] > seuil_B:
            Threshold.append(1)
        else: Threshold.append(0)
    Threshold=numpy.array(Threshold)
    Threshold=Threshold.reshape(img.shape[0],img.shape[1])
    RebuildImg(Threshold*255,"c:\\seuillage.bmp")
    return Threshold
```

Ci-dessus est implémentée la fonction ***Seuillage()***, j'ai quelques commentaires à faire sur certaines lignes. Dans l'algorithme nous avions vu que nous devions construire l'image résultante de notre traitement. Tout se passe dans cette ligne ***RebuildImg(Threshold*255,"c:\\seuillage.bmp")***, nous multiplions les valeurs contenues dans ***Threshold*** par 255 pour l'affichage. En effet l'image ***Threshold*** est une image binaire, cela veut dire que ses valeurs sont comprises entre 0 et 1. Pour que les objets apparaissent en blanc, je multiplie donc le tableau par 255. Le ***return Threshold*** retourne un tableau de type binaire pour que l'on puisse faire des opérations booléenes dessus (ET, OU, UNION etc etc)

Cependant, dans le souci du détail, Guigui_ m'a fait parvenir une façon de faire un seuillage 100% ***NumPy*** et celui-ci est plus rapide.

Fonction Seuillage

```
import numpy

def seuillage(img, seuil, vbasse = 0, vhaute = 255):
    return numpy.where(img>= seuil, vhaute, vbasse)
```

Voilà, je pense que cela se passe de commentaire, la fonction ***where*** construit un ***array*** qui a pour valeur 255 lorsque les pixels sont supérieurs au seuil, sinon ils sont à zéro. Pour obtenir un objet plus "Booleen", remplacez 255 par 1.

III-A-3 - Exemple de l'application

J'ai effectué un seuillage de 125-250 sur l'image de lenna.GIF. Voici le résultat.

lenna.gif

seuillage.gif

III-B - Dilatation

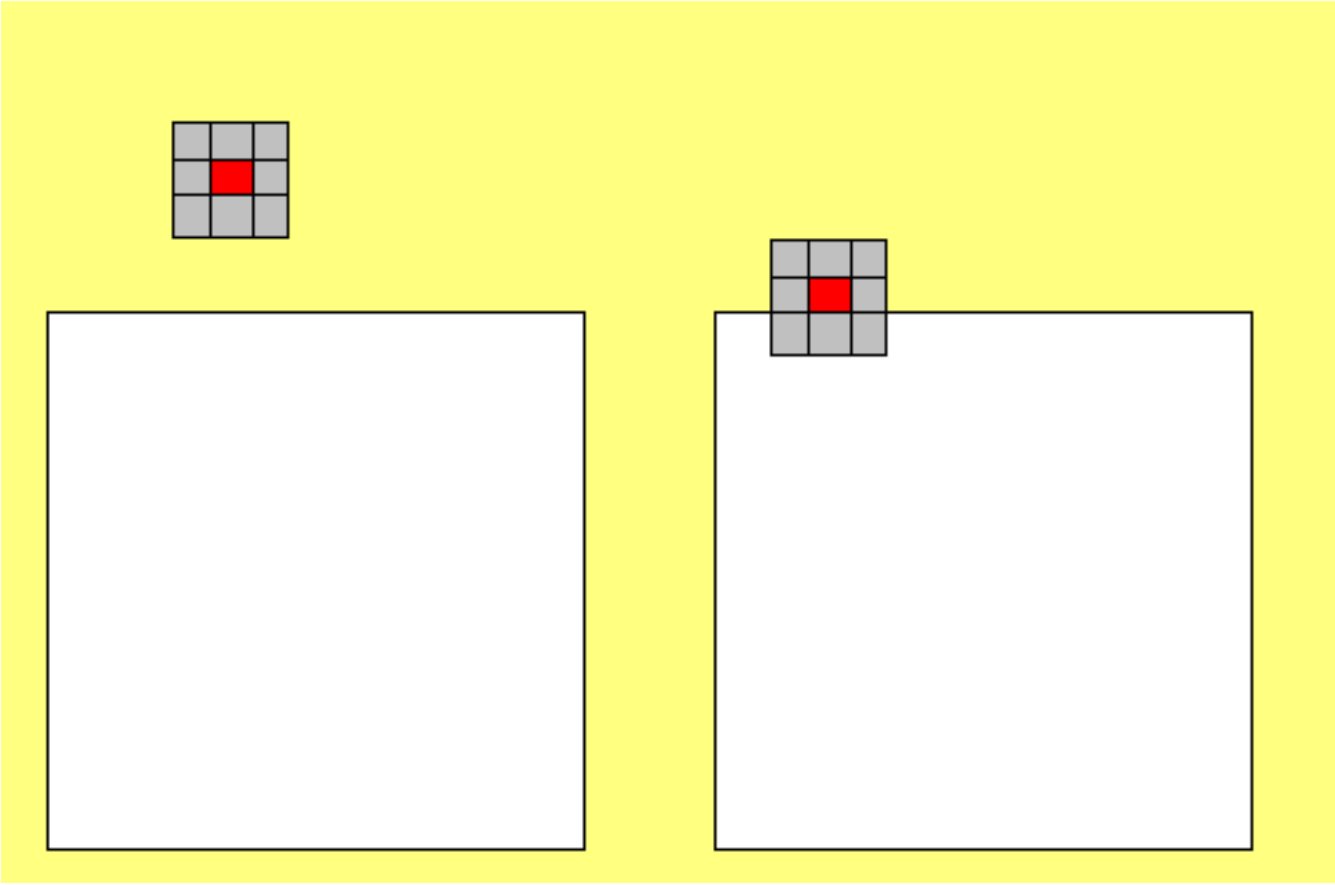
III-B-1 - Rappel de Cours

La dilatation est avec l'érosion et quelques autres opérateurs, un opérateur de base en morphologie. Nous pouvons créer plusieurs filtres morphologiques avec la dilatation et l'érosion. L'érosion est une analogie de la dilatation, quand vous aurez compris la dilatation, il sera aisé pour vous de comprendre l'érosion. Admettons un ensemble d'objets. Posons nous la question: "Est ce que en le parcourant dans l'image mon élément structurant touche mon objet ?" Les réponses positives à cette question s'appelle le DILATÉ de mon objet.

Qu'est ce que l'élément structurant ?

L'élément structurant est un masque que nous faisons parcourir sur toute l'image. A chaque position on se pose la question posée précédemment, sachant que la reponse sera appliquée à l'endroit où est centré l'objet.

Exemple avec un objet, un élément structurant carré 3x3 et deux positions de celui ci:



Dans le premier cas: Est ce que l'élément structurant touche l'objet ?

Réponse: **NON** donc le pixel sous le centre de l'élément structurant (case rouge) n'est pas un pixel du dilaté de l'objet

Dans le deuxieme cas: Est ce que l'élément structurant touche l'objet ?

Réponse: **OUI** donc le pixel sous le centre de l'élément structurant appartient au **dilaté** de l'objet.

III-B-2 - Application à la programmation

Nous avons vu les bases de la dilatation mais lorsqu'il faut coder tout cela, on utilise une autre définition de la dilatation. En programmation le dilaté d'un pixel c'est la valeur **MAX** au voisinage de ce pixel compris dans l'élément structurant. La définition citée plus haut ne marche que pour les images binaires, cependant on peut dilater une image en niveau de gris. La définition que je viens de citer fonctionne avec les images binaires comme en niveau de gris.

Un autre problème intervient en programmation, c'est l'effet de bord. La gestion des bords en traitement d'images est assez subjective, pour certain comme pour la pluspart des cas nous supprimons les objets touchant les bords, la gestion des bords est alors une chose obsolète. Pour nous la gestion des bords est respectée. Pour cela nous allons agrandir notre image. Nous allons ajouter une bordure d'un pixel, car comme vous l'avez deviné, nous aurons des problèmes du genre **index out of range** .

Exemple d'une dilatation sur une matrice de chiffre. RAPPEL DE DEFINITION: Le dilaté est la valeur **MAX** comprise au voisinage d'un pixel compris dans l'élément structurant.

12	4	63	156	2
84	42	21	84	84
232	14	24	8	9
1	3	161	94	5
9	66	8	96	33
8	5	369	8	48

12	4	63	156	2
84	42	21	84	84
232	14	161	8	9
1	3	161	94	5
9	66	8	96	33
8	5	369	8	48

La valeur surlignée est bien la valeur **MAX** comprise dans l'élément structurant. **La valeur surlignée est bien le dilaté des valeurs à cet endroit.**

Algorithme de la dilatation

```
Ouvrir Image à Dilater
Ajouter une bordure a l'image pour gérer l'effet de bord
Créer un nouveau tableau pour stocker les resultats de la taille de l'image ORIGINALE
x=largeur de l'image+bordure
y=hauteur de l'image+bordure
pour tous les i allant de 1 a x-1:
    pour tous les j allant de 1 a j-1:
        tableau[i-1][j-1]=max(voisinage 3x3 de ce pixels)
Reconstruire l'image dilaté
```

Pour gérer les bords nous allons créer une fonction. Mais quel type de pixels allons nous mettre dans cette bordure ? Nous allons mettre des pixels **NOIR** car ils n'interagirons pas avec les valeurs.

```
                              0,0,0,0,0
1,1,1                         0,1,1,1,0
1,1,1    --->fonction()---->  0,1,1,1,0
```

```
1,1,1                          0,1,1,1,0
                               0,0,0,0,0
```

Pour cela il suffit de créer un tableau de dimensions (x+2,y+2).

Algorithme de la fonction Bord

```
Ouvrir Image à Dilater
Créer un tableau (x+2,y+2) remplit de 0
Pour i de 1 a x-1:
    Pour j de 1 a y-1:
        tableau[i][j]=Image[i-1][j-1]
    fin Pour
fin Pour

Faire DILATATION
```

Voici la fonction ***bord()***.

La fonction Bord

```
def Bord(data):
    data=numpy.array(data)
    x=data.shape[1]+2
    y=data.shape[0]+2
    new=[x*[0]]*y  #création du tableau
    new=numpy.array(new)

    h=1
    for i in range(1,y-1):
        for j in range(1,x-1):
            new[i][j]=data[i-1][j-1] #remplissage du tableau

    return new
```

Il suffit maintenant de parcourir ce tableau de 1 à x-1 et de 1 à y-1, de regarder dans un voisinage 3x3 les valeurs et appliquer la valeur max au point d'ancrage de l'élement structurant.

Voici la fonction **dilatation()**

La fonction Dilatation

```
def Dilatation(img):

    Dilate=[0]*(img.shape[1]-2)*(img.shape[0]-2)
    h=0
    for i in range(1,img.shape[0]-1):
        for j in range(1,img.shape[1]-1):

 Dilate[h]=max([img[i-1][j-1],img[i][j-1],img[i+1][j-1],img[i-1][j],img[i][j],img[i+1][j],img[i-
1][j+1],img[i][j+1],img[i+1][j+1]])
     #Mise de la valeur max du voisinage 3x3 au point(x,y)
           h+=1

    Dilate=numpy.array(Dilate)
    Dilate=numpy.reshape(Dilate,(img.shape[0]-2,img.shape[1]-2))
    RebuildImg(Dilate,"c:\\dilate.bmp")
```

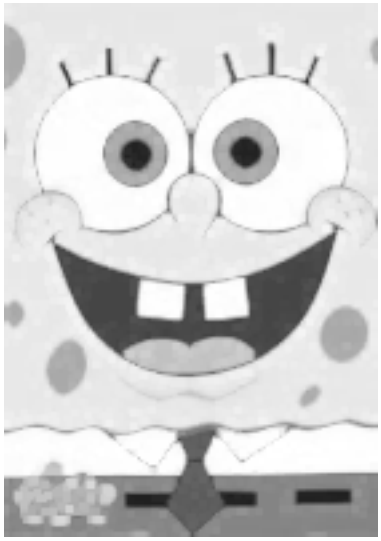
Voila, nous avons programmé la dilatation. Pour l'érosion vous verrez trés peu de changement sont à apporter.

III-B-3 - Exemple de l'application

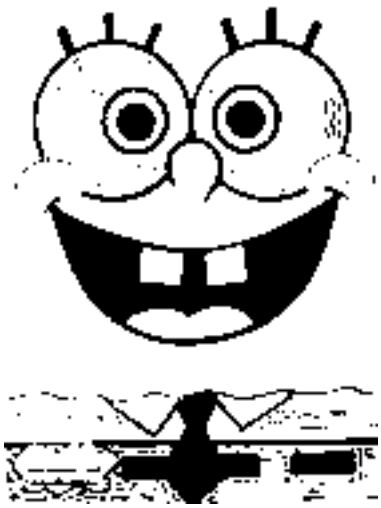
```
>>> a = OuvrirImg("c:\\bob.bmp")
>>> b = Bord(a)
>>> c = Dilatation(b)
```

bob.bmp

bob dilaté



bob seuillé 128-255



bob seuillé puis dilaté

On voit bien que les objets en blanc gagnent du "terrain" alors que les objets en noir en perdent. Nous avons dilaté.

III-C - Erosion

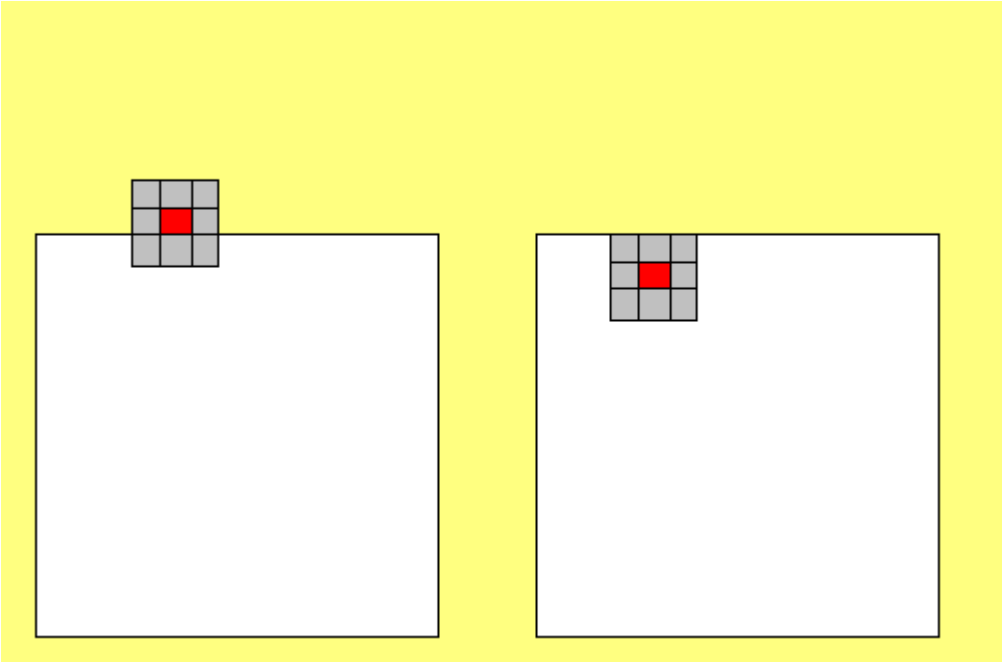
III-C-1 - Rappel de Cours

Comme je l'ai dit précédemment, l'érosion est un opérateur de base en traitement d'image. Je pense que vous avez compris ce qu'est la DILATATION, alors je ne vais pas m'étendre et me perdre dans les définitions complexes. Rappelez vous que nous nous posions une question pour savoir si un pixel appartenait au dilaté de cet objet. Il existe une question pour l'érosion.

Admettons un ensemble d'objets.

Posons nous la question : Est ce que mon élément structurant est complètement inclus dans mon objet ?

Les réponses positives constituerons l'érodé de l'objet.



Dans le premier cas posons nous la question : Est ce que l'élement structurant est complètement inclus dans mon objet?

- Dans le premier cas, la réponse est négative
- Dans le second, elle est positive donc le pixel situé sous le centre de l'élément structurant sera mis a un. C'est aussi simple que cela.

III-C-2 - Application à la programmation

Comme pour la dilatation, il existe une astuce pour calculer l'érodé d'un objet. Pour la dilatation, nous prenions le max dans un voisinage 3x3, pour l'érosion nous prendrons le minimum dans un voisinage 3x3. Pour la gestion des bords, nous aurons une fonction **bord2** proche de la fonction **bord** sauf que l'on ajoutera un bord de pixels blancs pour qu'il ne fausse pas le résultat.

La fonction Bord2

```
def Bord2(data):
    data=numpy.array(data)
    x=data.shape[1]+2
    y=data.shape[0]+2
    new=[x*[1]]*y  #création du tableau
    new=numpy.array(new)
```

La fonction Bord2

```
        h=1
        for i in range(1,y-1):
            for j in range(1,x-1):
                new[i][j]=data[i-1][j-1] #remplissage du tableau

        return new
```

Voici le code de l'érosion.

La fonction Erosion

```
    def Erosion(img):

        Erode=[0]*(img.shape[1]-2)*(img.shape[0]-2)
        h=0
        for i in range(1,img.shape[0]-1):
            for j in range(1,img.shape[1]-1):

     Erode[h]=min([img[i-1][j-1],img[i][j-1],img[i+1][j-1],img[i-1][j],img[i][j],img[i+1][j],img[i-
    1][j+1],img[i][j+1],img[i+1][j+1]])
         #Mise de la valeur max du voisinage 3x3 au point(x,y)
               h+=1

        Erode=numpy.array(Erode)
        Erode=numpy.reshape(Erode,(img.shape[0]-2,img.shape[1]-2))
        RebuildImg(Erode,"c:\\erode.bmp")
```

Voila l'érosion est programmée. C'est très similaire à la dilatation à quelques détails près.

III-C-3 - Exemple de l'application

```
    >>> a = OuvrirImg("c:\\bob.bmp")
    >>> b = Bord2(a)
    >>> c = Erosion(b)
```

bob.bmp

bob érodé