Manipulation d'image avec numpy

1 — Format numérique d'image

Une image peut être numérisée sous forme d'*image matricielle* (en anglais « bitmap ») par une matrice de points colorés.

Cette matrice a n lignes (la *hauteur* de l'image) et p colonnes (la *largeur*). L'élément (i, j) représente un *pixel*, c'est-à-dire une portion d'image considérée de couleur constante.

Il existe plusieurs manières de coder la couleur. La méthode accessible avec la bibliothèque `matplotlib.image` est la méthode RGB [1]. Chaque couleur est représentée par une liste à trois entrées `[r,g,b]` où r, g et b sont trois réels représentant respectivement la quantité de rouge, de vert et de bleu que contient la couleur. La méthode de mélange des couleurs est la *synthèse additive* : on peut penser à trois spots de couleurs qui éclairent un fond noir. Le mélange des trois lumières colorées crée la couleur désirée.

Si $r = g = b = 0$, le pixel est noir. Si r, g et b ont leur valeur maximale, le pixel est blanc. Cette valeur maximale dépend de l'image et du format d'image : il peut être de 1 ou de 255.

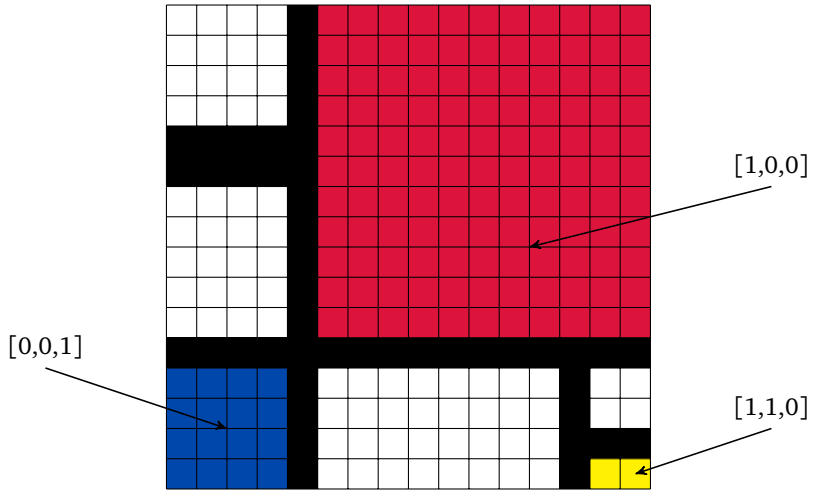
Supposons que la valeur maximale soit à 1 : `[1,0,0]` est du rouge pur, `[0,1,1]` est le cyan (complémentaire du rouge), etc. Les couleurs ayant des proportions identiques de rouge, vert et bleu `[x,x,x]` sont des gris de plus en plus clair lorsque x augmente.

2 — Création d'une image

Avec numpy une image est donc codée avec une matrice A de dimension 3. La première dimension représente le numéro de la ligne, la seconde le numéro de la colonne et la dernière le numéro de la couleur.

Par exemple, si A code l'image ci-dessous, on aura `A[0,0]` qui vaut `[1,1,1]`, `A[0,6,0]` qui vaut `1` et `A[0,6,1]` et `A[0,6,2]` qui valent `0`, etc.

1. pour « Red-Green-Blue », on dirait RVB en bon français.



Créons maintenant une image très simple.

```
import numpy as np
import matplotlib.image as mpimg

# codage des couleurs pour plus de facilité
rouge = [0.807, 0.066, 0.149]
blanc = [1, 1, 1]
bleu =  [0, 0.129, 0.278]

hauteur = 210
largeur = 2*hauteur

# Image de départ, entièrement noire
image = np.zeros((hauteur, largeur,3))

# Remplissage
image[:, 0:largeur//3 ] = bleu
```

```
image[:, largeur//3:2*largeur//3 ] = blanc
image[:, 2*largeur//3:largeur ] = rouge

mpimg.imsave("image2.png", image)
```

Comme d'habitude on commence par inclure les libraires. Ensuite l'image est gérée comme une `array` de `numpy`. Chaque pixel reçoit comme valeur une liste `[r, g, b]` de trois réels pour coder la couleur. On utilise la puissance de `numpy` pour remplir rapidement l'image.

À la fin du script, l'instruction `plt.imshow()` demande à `matplotlib` d'interpréter cet `array` en tant qu'image bitmap. La commande `plt.show()` permet l'affichage et `plt.imsave()` la sauvegarde sur disque. La librairie `matplotlib.image` ne connait que le format d'image PNG [2] : on sauvegarde donc avec l'extension `png`.

3 — Ouverture et manipulation d'une image existante

Voyons un exemple

```
import matplotlib.image as mpimg
img = mpimg.imread('loco.png')
```

L'image `loco.png` est stocké dans le tableau `numpy` nommé `img` (la librairie matplotlib a chargé `numpy` pour nous). Voyons d'un peu plus près l'objet `img`

```
img
array([[[ 0.90588236,  0.91764706,  0.93725491],
        [ 0.90588236,  0.91764706,  0.93725491],
        [ 0.90588236,  0.91764706,  0.93725491],
        ...,
        [ 0.92156863,  0.93333334,  0.95294118],
        [ 0.92156863,  0.93333334,  0.95294118],
        [ 0.92156863,  0.93333334,  0.95294118]],
        ...,
        [ 0.45882353,  0.57647061,  0.32549021],
        [ 0.43921569,  0.55686277,  0.30588236],
        [ 0.43529412,  0.5529412 ,  0.3019608 ]]], dtype=float32)
```

Les couleurs sont codées par 3 flottants entre 0 et 1 (cette information est importante à retenir, car ce n'est pas toujours le cas). La librairie `matplotlib.image` a décodé le fichier et chargé l'image sous forme d'un tableau `numpy`.

Comme `img` est un tableau, il est facile de le manipuler avec les méthodes de `numpy` ou avec des boucles. Pour cela on a besoin d'en connaître les dimensions à l'aide de l'attribut `shape` (sans parenthèses !). Par exemple, modifions l'image ci-dessus pour en saturer la couleur bleue.

```
img[:,:,2] = 1
```

On obtient l'image suivante :

Ex. 1 — Écrire différentes fonctions qui reçoivent par paramètre le nom d'un fichier image, transforment cette image, et sauvegardent le résultat dans un fichier. On les testera avec l'image `loco.png`.

1. Inversion de couleurs : la valeur x de chaque couleur est remplacé par $1 - x$.
2. Séparation en trois images : l'une avec la composante rouge, une deuxième avec la composante verte et la troisième avec la composante bleue.
3. Transformation en niveau de gris. On a plusieurs moyen de le faire :
   a) Dans la couleur grise, les trois niveaux de couleur sont égaux. On remplace donc rouge, vert et bleu par la moyenne des trois couleurs.

2. Portable Network Graphics (PNG) est un format ouvert d'images non destructeur, c'est-à-dire qu'il compresse l'information sans la simplifier (au contraire de `JPEG` par exemple).

b) Mais on peut aussi remplacer chaque niveau par la moyenne pondérée $0,21 \times R + 0,71 \times V + 0,07 \times B$ (la « luminosité » du pixel).

c) Enfin on peut prendre la moyenne entre le minimum et le maximum des trois composantes.

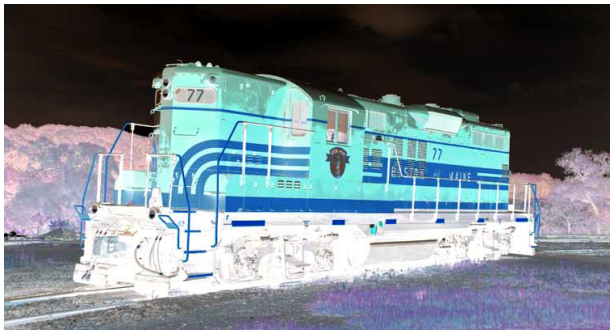
4. Proposer et implémenter une méthode pour transformer l'image en noir et blanc.

Corrigé de l'exercice 1:

```
import matplotlib.image as mpimg
import numpy as np


# ---------------------------------------------------------------------
def separe_couleur(nom):
    """ Sépare l'image 'nom' en 3 images contenant chacune des 3 couleurs """

    # Image de départ
    img = mpimg.imread(nom)

    # On initialise les différentes images en noir
    imgrouge = np.zeros(img.shape)
    imgvert  = np.zeros(img.shape)
    imgbleu  = np.zeros(img.shape)

    imgrouge [ : , : , 0] = img [ : , : , 0]
    imgvert  [ : , : , 1] = img [ : , : , 1]
    imgbleu  [ : , : , 2] = img [ : , : , 2]

    mpimg.imsave("rouge-" + nom, imgrouge)
    mpimg.imsave("vert-" + nom, imgvert)
    mpimg.imsave("bleu-" + nom, imgbleu)
    return(imgrouge, imgvert, imgbleu)
```

```
# ---------------------------------------------------------------------
def negatif(nom):
    """Fabrique le négatif de l'image 'nom' """

    img = mpimg.imread(nom)
    imginverse  = 1 - img
    mpimg.imsave("inverse-"+nom, imginverse)

    return(imginverse)

# ---------------------------------------------------------------------
# 3 façons de transformer un pixel coloré en pixel gris
def grismoyen(nom):
    img = mpimg.imread(nom)
    imggris  = (img[ : , : , 0] + img[ : , : , 1] + img[ : , : , 2])/3
    mpimg.imsave("grismoyen-"+nom, imggris, cmap="gray")
    return(imggris)

def grisluminosite(nom):
    img = mpimg.imread(nom)
    imggris  = 0.21 * img[ : , : , 0] + 0.71 *
img[ : , : , 1] + 0.07 * img[ : , : , 2]
    mpimg.imsave("grislum-"+nom, imggris, cmap="gray")
    return(imggris)


def minmax(nom):
    img = mpimg.imread(nom)
    # imggris  = (np.amax(img,2) + np.amin(img,2))/2

    hauteur, largeur, couleur = img.shape
    imggris = np.zeros((hauteur, largeur))
    for i in range(hauteur):
        for j in range(largeur):
```

```
            imggris[i,j] = (max( img[i,j,0], img[i,j,1], img[i,j,2])
                        +  min( img[i,j,0], img[i,j,1], img[i,j,2]))/2


    mpimg.imsave("grisminmax-"+nom, imggris, cmap="gray")
    return(imggris)

def NB(nom, seuil):
    """Transforme une image couleur en niveau de gris"""
    img = mpimg.imread(nom)
    (hauteur, largeur, profondeur) = img.shape

    #On choisit une conversion par gris lumineux
    imggris = grisluminosite(nom)

    # Tout ce qui dépasse seuil vaut 1, le reste vaut 0
    imgNB = (imggris >= seuil)

    mpimg.imsave("NB-" + nom , imgNB, cmap="gray")
    return(imggris)

# Appel aux différentes fonctions
separe_couleur("loco.png")
negatif("loco.png")
grismoyen("loco.png")
grisluminosite("loco.png")
minmax("loco.png")
NB("loco.png", 0.3)
```



Une manière simple pour transformer une image en noir et blanc et de transformer l'image en niveau de gris, puis d'appliquer une fonction de seuil. Si le gris est supérieur à une valeur s alors le pixel est blanchi, sinon il est noirci. Le seuil est à fixer au préalable.

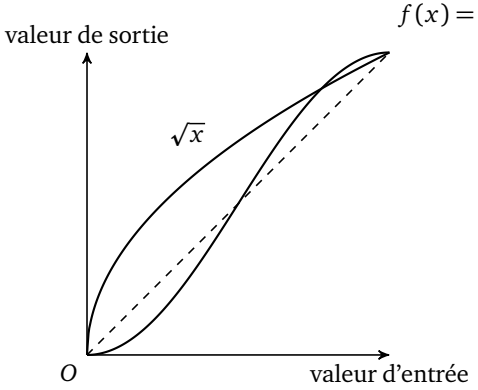
Pour les exercices suivants utiliser l'image `hibiscus.png`. Elle est en niveau de gris : chaque pixel est donc associé à un seul flottant qui est le niveau de gris.
Il est nécessaire de sauvegarder les images avec une option supplémentaire :

```
mpimg.imsave("nomdelimage.png", cmap = "gray")
```

Ex. 2 — Amélioration du contraste Pour améliorer le contraste, on transforme le niveau de gris d'entrée en un nouveau niveau de gris dans le but d'exagérer les différences entre les niveaux. Pour cela on applique une fonction « bien choisie ».

1. Améliorer le contraste en utilisant une fonction racine carrée : on remplace chaque niveau d'entrée x par $\sqrt{x}$.
2. Comparer avec l'usage d'un courbe « en S » (un morceau de sin convenablement transformé).

$$f(x) = \frac{1}{2} + \frac{1}{2} \times \sin\left(\pi(x - \frac{1}{2})\right)$$



3. Expliquer en quoi la transformation précédente améliore le contraste de l'image.
4. Quel type de fonction est susceptible d'être utile ?

Corrigé de l'exercice 2: 1.

```
import matplotlib.image as mpimg
import numpy as np
import math as m

def contraste1(nom):
    """ Améliore le contraste de l'image 'nom'
        Avec une racine carrée """
    # Image de départ
    img = mpimg.imread(nom)

    # On initialise les différentes images en noir
    img_contraste = np.sqrt(img)

    mpimg.imsave("contraste1-" + nom, img_contraste)

def contraste2(nom):
    """ Améliore le contraste de l'image 'nom'
        Avec une courbe en S (fonction sinus) """
    # Image de départ
    img = mpimg.imread(nom)

    img_contraste =  0.5+0.5*np.sin(m.pi*(img-0.5))

    mpimg.imsave("contraste2-" + nom, img_contraste)

# contraste1("hibiscus.png")
# contraste2("hibiscus.png")
contraste1("grislum-loco.png")
contraste2("grislum-loco.png")
```

2. La fonction précédente diminue les faibles niveaux et augmente les forts. Elle a donc tendance à exagérer les écarts entre les niveaux.
3. Il est préférable d'utiliser une fonction continue (pour utiliser toute la gamme de niveau), croissante (pour éviter l'effet « négatif »), avec une pente importante dans la zone de niveau dont on veut augmenter le contraste.

4 — Convolution

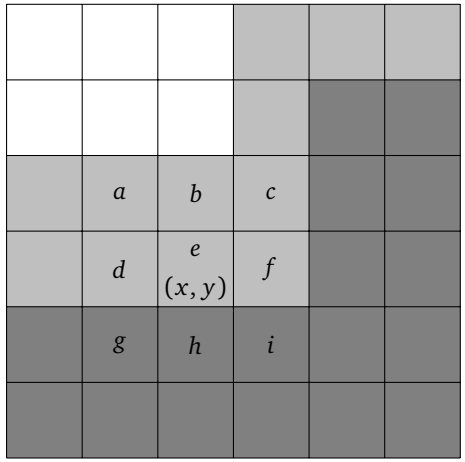
La convolution 2D est un traitement d'image où la valeur de chaque pixel est recalculé en fonction des valeurs des pixels voisins.

Cherchons par exemple à « flouter » une image. La valeur du pixel `[i,j]` est remplacé par la moyenne pondérée des valeurs des pixels voisins.

$$\text{pixel}[x,y] = \frac{1}{8}\text{pixel}[x-1,y-1] + \frac{1}{8}\text{pixel}[x-1,y] + \\ \frac{1}{8}\text{pixel}[x-1,y+1] + \cdots + \frac{1}{8}\text{pixel}[x+1,y+1]$$

Évidemment, cette formule n'est pas valable pour les pixels du bord de l'image, une autre formule doit être appliquée. On ne va pas se soucier dans ce TD : l'image de sortie sera donc un peu plus petite que l'image d'entrée.

Mais on va faire plus fort : on va généraliser cette formule avec des coefficients différents.



$$\text{pixel}[x,y] = a \times \text{pixel}[x-1,y-1] + b \times \text{pixel}[x-1,y] + \\ c \times \text{pixel}[x-1,y+1] + \cdots + i \times \text{pixel}[x+1,y+1]$$

Ex. 3 — Convolution – Floutage 1. Écrire une fonction `convolution2D( image, matrice )` qui prend en paramètre une image et une matrice 3×3 de la forme

$$M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

et qui effectue l'opération de convolution décrite précédemment.

2. Flouter l'image hibiscus en lui appliquant une convolution convenable.

Corrigé de l'exercice 3:

```
import matplotlib.image as mpimg
import numpy as np
import random

def convolution2D (img, matrice):
```

```
    """ Effectue l'opération de convolution 2D sur l'image img """

    (hauteur, largeur) = img.shape
    img_conv = np.zeros((hauteur, largeur))

    for x in range(1, hauteur-1):
        for y in range(1, largeur-1):
            img_conv[x,y] = np.sum(matrice *
img [x-1:x+2 , y-1:y+2 ])

    return img_conv

def flou(nom):
    """Floutte l'image 'nom'
       Par moyennage sur les 8 voisins """

    img = mpimg.imread(nom)

    img_flou = convolution2D( img, [[1/8,1/8,1/8],[1/8,0,1/8],[1/8,1/8,1/8]])

    mpimg.imsave("flou-" + nom, img_flou, cmap="gray")

# flou("hibiscus.png")

# Mais ça existe déjà

from scipy.ndimage import convolve

img = mpimg.imread("hibiscus.png")
kernel = np.array([[1/8,1/8,1/8],[1/8,0,1/8],[1/8,1/8,1/8]])

img_conv = convolve(img, kernel, mode='constant')

mpimg.imsave("flourapide-hibiscus.png", img_conv, cmap="gray")
```

Ex. 4 — Détection des bords – Méthode de la convolution Dans une image, les bords des objets correspondent à des zones où les valeurs des pixels changent rapidement. C'est le cas par exemple lorsque l'on passe d'un objet clair (avec des valeurs grandes) à un arrière plan sombre (avec des valeurs petites).
Considérons une image à laquelle on applique une convolution avec la matrice

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

1. Que se passe-t-il pour les pixels dans des zones uniformes ? Et pour des pixels dans des zones à fort contraste.
2. Appliquez cette convolution à l'image `cells.png`. Commentez.
3. Dans certaines images, comme `coupe.png` on s'intéresse à une détection de bord dans une certaine direction (ici horizontale). Proposez une convolution pour se faire.

Corrigé de l'exercice 4:

```
import matplotlib.image as mpimg
import numpy as np
import random

def convolution2D (img, matrice):
    """ Effectue l'opération de convolution 2D sur l'image img """

    (hauteur, largeur) = img.shape
    img_conv = np.zeros((hauteur, largeur))

    for i in range(1, hauteur-1):
        for j in range(1, largeur-1):
            img_conv[i,j] = np.sum( img[i-1:i+2, j-1:j+2] *
matrice)

    return img_conv
```

```
def detectionbord(nom):
    img = mpimg.imread(nom)

    img_bord = convolution2D( img, [[1,1,1],[1,-8,1],[1,1,1]])

    mpimg.imsave("bord-" + nom, img_bord, cmap="gray")


def detectionhorizon(nom):
    img = mpimg.imread(nom)

    img_bord = convolution2D( img, [[-1,-2,-1],[0,0,0],[1,2,1]])

    mpimg.imsave("bordh-" + nom, img_bord, cmap="gray")


# detectionbord("grislum-loco.png")
# detectionhorizon("grislum-loco.png")
detectionbord("coupe.png")
detectionhorizon("coupe.png")
```