Manipulation d'images

I. Tableau à deux dimensions

En Python, on peut manipuler des tableaux à double entrée comme des tableaux de tableaux, c'est-à-dire un tableau `t` qui contient des tableaux correspondant aux différentes lignes. Ainsi, `t[0]` est la première ligne, `t[1]` la seconde, etc.

```
>>> m = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> m[2] # troisième ligne
[9, 10, 11, 12]
>>> len(m) # nombre de lignes
3
>>> len(m[0]) # taille de la première ligne
4
>>> m[1][2]
7
```

Deux situations très fréquentes où l'on manipule des tableaux à doubles entrées sont le calcul matriciel et le traitement d'images. Dans les deux cas, en effet, on manipule des tableaux homogènes[1] à deux dimensions.

Dans la suite, nous allons nous intéresser à la manipulation d'images. Pour cela, nous allons utiliser deux bibliothèques. Tout d'abord, la bibliothèque `PIL` et plus particulièrement la partie `Image` qui nous servira à charger, manipuler, sauvegarder et visualiser des images. Nous utiliserons de plus la bibliothèque `numpy` (que l'on abregera en `np`). On effectuera donc les importations suivantes :

```
import numpy as np
from PIL import Image
```

II. Chargement et décomposition/recomposition d'une image

Informatiquement, une image de type `bitmap` consiste en une matrice de valeurs, chaque valeur correspondant à un point de l'image que l'on appelle *pixel* (de l'anglais *picture element*), et le mode de l'image permet de déterminer cela.

Les deux principaux modes que nous allons utiliser[2] :

[1] Qui contiennent le même type de données dans toutes les cases.

[2] D'autres modes usuels existent. Ainsi, parfois, on ajoute un 4ème canal pour représenter l'opacité du pixel. C'est le mode `'RGBA'` : on ajoute un canal *alpha* pour la transparence. Pour une image en noir et blanc, on associe un bit à chaque pixel.

- ⋆ Pour une image en niveau de gris, on a associe à chaque pixel un entier non signé sur 8 bits pour une valeur comprise entre 0 (noir) et 255 (blanc) (mode `'L'` pour *Luminance*)
- ⋆ Pour une image en couleur, on aura trois canaux pour le rouge, le vert et le bleu, chaque canal ayant une valeur codée entre 0 et 255 (mode `'RGB'` pour *Red, Green, Blue*).

Nous allons commencer par explorer une image, ici une photo de la *Victoire de Samothrace* exposée au Louvre.

```
>>> victoire = Image.open("victoire.jpg")
>>> victoire.size
(512, 715)
>>> victoire.mode
'RGB'
```

La taille de `victoire` indique qu'il s'agit d'une image de 512 pixels de large et 715 pixels de haut, et qu'il s'agit d'une image en couleurs (mode `'RGB'`).

Pour voir, l'image, il suffit d'exécuter :

```
victoire.show()
```

Comme c'est une image en couleur (sans canal de transparence), elle est composée de trois composantes (dans l'ordre, rouge, vert et bleu) que l'on peut décomposer à l'aide de la méthode `split()`. Les images obtenus lors de la décomposition sont en mode Luminance :

```
>>> r, g, b = victoire.split()
>>> r.mode
'L'
```

Il est bien sûr possible d'afficher une des images en luminance obtenue :

```
>>> r.show()
```

On peut combiner des composantes pour obtenir une image à l'aide de la commande `Image.merge`. Notons que lors de la décomposition, on a perdu l'information indiquant pour chaque canal (on peut les recombiner dans le mauvais ordre, ou bien utiliser une même composante pour plusieurs couleurs). Il faut donc les recombiner en donnant la liste des canaux dans le bon ordre.

```
>>> Image.merge("RGB", [g, b, r]).show() # mauvais ordre
>>> Image.merge("RGB", [r, g, b]).show() # bon ordre
```

Notons que dans l'exemple précédent, on recombine (`merge`) trois images en luminance en une image `RGB` que l'on affiche (`show`) directement.

III. Manipulation d'images

1. Anatomie d'une image en luminance

Poursuivons notre exploration, et convertissons notre image en niveaux de gris sous forme d'un `np.array`.

```
>>> array_r = np.array(r)
>>> array_r
array([[172, 165, 152, ..., 215, 214, 213],
       [172, 172, 163, ..., 215, 214, 213],
       ...,
       [126, 126, 123, ..., 187, 176, 170]], dtype=uint8)
>>> array_r.shape
(715, 512)
>>> array_r[100][50]
177
```

Quelques remarques importantes concernant les dernières commandes.

- ⋆ Comme on le voit avec l'affichage de `array_r`, les données contenues sont de type `uint8` (c'est le *datatype* ou `dtype`), ce qui signifie que ce sont des entiers non signés codés sur 8 bits, pour des valeurs allant de 0 à $2^8 - 1 = 255$.
- ⋆ En observant la « `shape` » de `array_r` comparé à la « `size` » de `victoire`, on voit qu'il y a une inversion. En effet, le `np.array` est un tableau de lignes, donc on indique en premier la coordonnée `y` avant la coordonnée `x`, puisqu'il s'agit d'un tableau de *lignes*. Ainsi, `array_r[100][50]` est la valeur de la composante rouge du pixel de coordonnées (50, 100). En effet, `array_r[100]` désigne la ligne numéro 100, dans laquelle on prends la 50ème valeur. Il y a donc une inversion entre abscisse/ordonnée d'une part, et ligne/colonne d'autre part.

2. Négatif et contraste

On a vu comment transformer une image en un `np.array`. On peut, à l'inverse, créer une image à partir d'un `np.array` à l'aide de la commande `Image.fromarray`. Voici une expérience amusante à réaliser :

```
# on complémente la valeur des pixels dans un rectangle
for y in range(50, 200):
    for x in range(50, 350):
        array_r[y][x] = 255 - array_r[y][x]
```

```
# on reconvertit le tableau en image
r2 = Image.fromarray(array_r)
# puis on l'affiche
r2.show()
# on peut effectue les deux opérations à la suite sans
# définir de nouvelle variable en exécutant
# Image.fromarray(array_r).show()

# on peut même recombiner le résultat avec les autres canaux
Image.merge("RGB", [r2, g, b]).show()
```

Question 1 Écrire une fonction `negatif(tableau)` qui modifie un tableau représentant une image en **luminance** pour obtenir son négatif. Concrétement, chaque valeur du tableau sera remplacée par son complément à 255.

Question 2 Écrire une fonction `negatif_rgb(image)` qui prends en entrée une image `RGB` et renvoie l'image `RGB` représentant son négatif, que l'on obtiendra en décomposant l'image en trois canaux, inversant chaque composante puis recombinant les trois composantes négatives.

Question 3 Écrire une fonction `puissance(tableau, p)` qui modifie le tableau représentant une image en luminance de la façon suivante :

1. chaque valeur `t[y][x]` du tableau `t`, appartenant à $[\![0,255]\!]$ est transformée en un flottant $f \in [0,1]$ en la divisant par 255 ;
2. on élève f à la puissance $p > 0$ précisée en argument ;
3. on modifie `t[y][x]` en lui attribuant comme valeur la partie entière de $255f$.

Testez votre fonction pour déterminer l'effet de cette fonction suivant la valeur de p, que l'on choisira au départ proche de 1.

3. Conversion en niveaux de gris

Nous allons maintenant convertir une image en niveaux de gris. Pour cela,

1. nous allons créer un `np.array` à la bonne taille à l'aide de `np.zeros` ;
2. remplir chaque case du tableau par la bonne valeur.

Pour créer un tableau aux bonnes dimensions et comportant des `np.uint8`, on pourra utiliser `np.zeros((hauteur, largeur), dtype=np.uint8)`.

On pourrait croire qu'il faut prendre la moyenne des composantes rouge, bleue et verte de l'image de départ (ce qui donne une version tout à fait acceptable)

mais c'est un peu plus compliqué que cela puisque l'œil humain ne perçoit pas les couleurs primaires avec la même intensité. On utilise en général la combinaison linéaire suivante :

$$0,2126 \times \text{Rouge} + 0,7152 \times \text{Vert} + 0,0722 \times \text{Bleu}.$$

(voir la page Wikipedia « `Niveau de gris` » pour plus d'informations)

Question 4 Écrire une fonction `gris(image)` qui prends en entrée une image en couleurs et retourne l'image convertie en niveaux de gris.

IV. Application de filtres

1. Convolutions

Une méthode usuel de traitement d'image est d'utiliser un produit de convolution entre le tableau `m` représentant une image monochrome, et une matrice de convolution `c` de taille 3×3. L'image obtenue est alors représentée par le tableau `r` défini par :

$$r_{x,y} = \sum_{i=-1}^{1} \sum_{j=-1}^{1} c_{1-i,1-j} m_{x+i,y+j}$$

On note $r = m \otimes c$.

Dans cette formule, la matrice de convolution (que l'on appelle aussi *noyau*) est une matrice 3×3 (les indices vont de 0 à 2 centrés sur 1, d'où les coordonnées $1-i$ et $1-j$ pour i et j allant de -1 à 1). On trouve aussi des noyaux de taille 5×5.

Voici deux matrices de convolution classiques :

$$\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Flou Contraste

On pourra consulter la page Wikipedia « `Noyau (traitement d'image)` » pour plus d'exemples. Lors de l'application d'un noyau de convolution, si la valeur calculée est plus petite que 0 (resp. plus grande que 255), on utilise 0 (resp. 255).

Question 5 Écrire une fonction `convolution(m, c)` qui renvoie le tableau représentant une image monochrome (ou une composante d'une image couleur) obtenue par convolution de `m` par le noyau `c`. On supposera que la convolution sera appliquée à chaque composante indépendemment des autres. On ne traitera pas les pixels en bordure d'image pour éviter le problème de bords.

En fait, pour pouvoir expérimenter plus efficacement, on peut utiliser directement des fonctionnalités de PIL et notamment le module `ImageFilter`, comme illustré ci-après.

```
from PIL import ImageFilter
contraste = ImageFilter.Kernel(
    (3, 3),
    [  0, -1,  0, # ligne 1
      -1,  5, -1, # ligne 2
       0, -1,  0  # ligne 3
    ])
victoire.filter(contraste).show()
```

2. Détection de contours

Nous allons écrire une fonction de détection de contours, mais avant cela, préparons une image avec des contours nets.

Question 6 Créer une image de taille $(200, 200)$ représentant un cercle blanc centré de rayon 75 pixels sur un fond noir.

Pour détecter les contours, on peut s'inspirer des méthodes basées sur les produits de convolution et utiliser des noyaux qui correspondant aux dérivées partielles selon les deux axes.

$$\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Cependant, pour atténuer la sensibilité au bruit, on utilisera les noyaux suivants :

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad \text{et} \quad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

En notant $M_x = m \otimes G_x$ et $M_y = m \otimes G_y$, on peut alors obtenir les contours de l'image m en calculant pour chaque point de coordonnées (x, y) la quantité

$$\sqrt{\left|M_x(x,y)\right|^2 + \left|M_y(x,y)\right|^2}.$$

On reconnaît donc ici la norme euclidienne d'un vecteur à deux composantes qui correspondent aux deux produits de convolution donnant les dérivées selon les abscisses et les ordonnées.

Question 7 Écrire une fonction `contours(m)` qui renvoie un tableau indiquant les contours de l'image représentée par `m`. Vous pourrez éventuellement prendre la partie entière du résultat après l'avoir multiplié par un coefficient approprié.