Traitement d'images

1 Introduction

Une image de type *bitmap* (« carte de points » en français) est stockée sous la forme d'une matrice (tableau de tableaux) de *pixels* (contraction de *picture element*). La couleur de chaque pixel est définie par trois entiers (R pour rouge, V pour vert et B pour bleu), la synthèse *additive* de ces trois couleurs par les trois cellules luminophores d'un élement d'écran permet de reproduire approximativement l'ensemble des couleurs visibles.

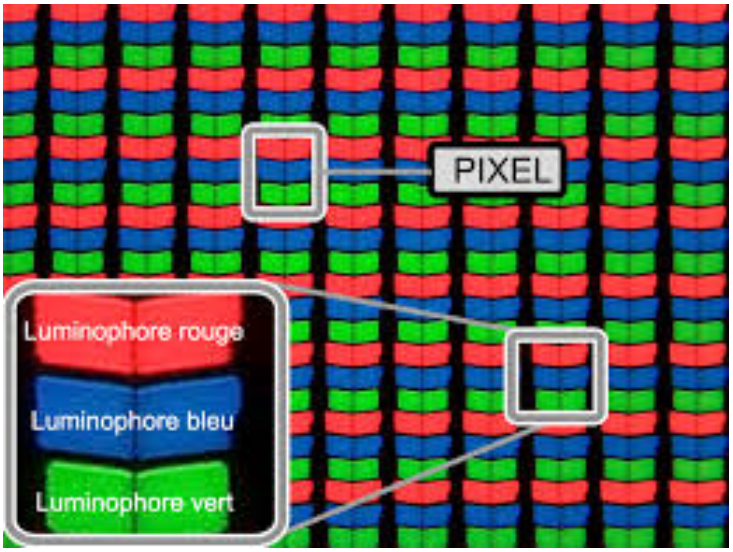

Figure 1. Luminophores d'un écran

Chaque entier RVB est codé sur 8 bits (1 octet) pour une image dite en « couleurs vraies ». Une image en niveaux de gris est une image pour laquelle $R = V = B$. Notons les deux cas particuliers du noir ($R = 0, V = 0, B = 0$) et du blanc ($R = 255, V = 255, B = 255$).

Question 1. Calculer approximativement le nombres de couleurs possibles chaque pixel d'une image en couleurs vraies. Calculer l'ordre de grandeur du poids en Mo d'une image HD (définition minimum de 1280×720 pixels).

2 Opérations de base

2.1 Imports

Pour cette séance, il sera nécessaire d'importer trois modules dans un nouveau script dont la structure sera la suivante :

```
import numpy as np
import matplotlib.pyplot as plt
import PIL.Image as Image
```

2.2 Ouverture d'un fichier image

Question 2. Dans Capytale créer un nouveau script `Images` et cliquer sur Gérer les fichiers annexes pour importer le fichier image `exemple.png` qui se trouve dans le dossier réseau de votre classe dans le même dossier que votre script. Ouvrir ensuite le fichier sous forme d'un objet de type `File` :

```
im = Image.open('exemple.png')
```

Question 3. Afficher la définition de l'image :

```
print(im.size)
```

Quelle seront les dimensions de la matrice correspondant à cette image ?

2.3 Matrice des pixels

Question 4. Après ouverture du fichier `exemple.png`, convertir l'objet `File` obtenu en tableau de tableaux (matrice `M`) :

```
M = np.asarray(im)
```

Afficher ensuite les dimensions de cette matrice (`M.shape`), le contenu du pixel situé à la ligne 10 et à la colonne 25, et le type choisi pour le codage des entiers RVB (`M.dtype.name`).

2.4 Affichage de l'image

Question 5. Convertir la matrice `M` en objet `File` :

```
im2 = Image.fromarray(M)
```

Afficher ensuite cet objet image `im2` dans la fenêtre graphique de `matplotlib.pyplot` à l'aide des lignes suivantes :

```
fig,ax = plt.subplots()    # fenêtre d'affichage et zone de tracé
ax.imshow(im2)              # affichage de l'image dans la zone de tracé
plt.show()
```

Remarque. Si on travaille sous Spyder, on peut enregistrer enfin l'objet `im2` sous forme d'un nouveau fichier image :

```
im2.save('exemple2.png')
```

2.5 Résumé des opérations de base

Sur le diagramme suivant sont résumées les principales opérations de conversion, du fichier de départ au fichier d'arrivée, que vous venez d'apprendre. Au niveau de la matrice, on peut appliquer les traitements :

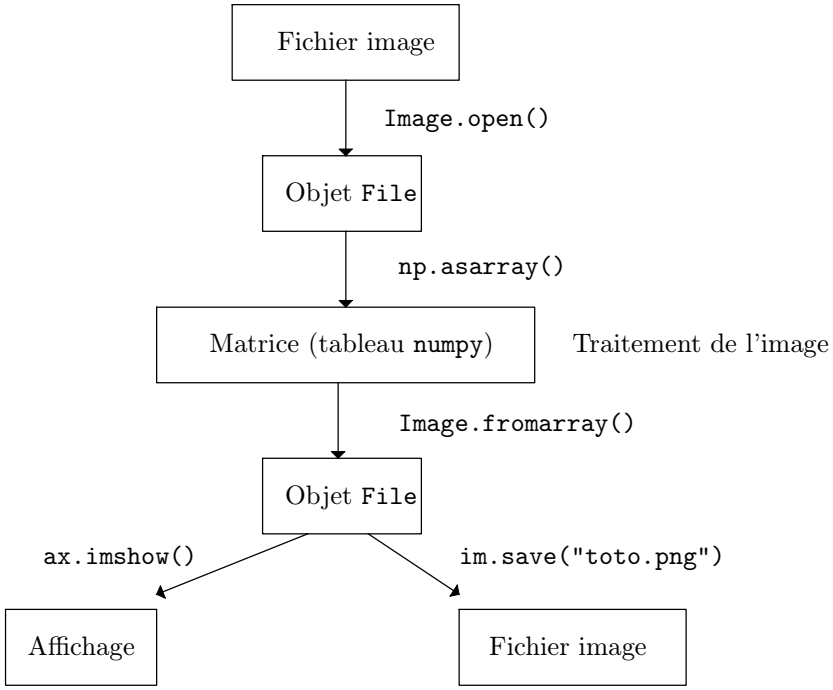

Figure 2. Du fichier image au fichier image

2.6 Fonction de test des traitements

Question 6. Écrire une fonction `test(fic, f)` qui prend un fichier image `fic` et une fonction de traitement `f` en argument et qui affiche côte à côte l'image originale et l'image traitée par le traitement `f`.

Tester cette fonction avec une fonction de traitement identité `Id` qui ne fait rien (qui prend en argument une matrice d'image `M` et qui retourne la même matrice).

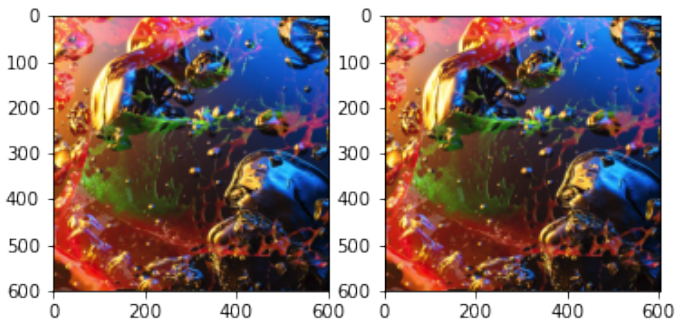

Figure 3. Test de la fonction `test`

3 Quelques traitements d'image

3.1 Niveaux de gris

Question 7. Répondre d'abord aux questions suivantes :

- Comment faire en sorte que chaque entier RVB de chaque pixel ait la même valeur ?
- Comment faire en sorte que le calcul soit compatible avec le `dtype` (`uint8`) ?

Ecrire une fonction `ng` qui prend en argument une matrice représentant une image couleur et qui retourne la matrice correspondant à l'image en niveaux de gris. Tester la fonction à l'aide de la fonction `test`.

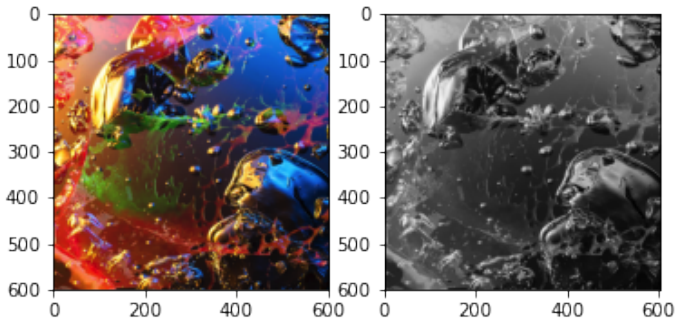

Figure 4. Test de la fonction `ng`

3.2 Négatif

Question 8. Quelle opération faudrait-il réaliser sur chaque pixel pour obtenir le *négatif* d'une image ? Écrire une fonction `neg` puis la tester.

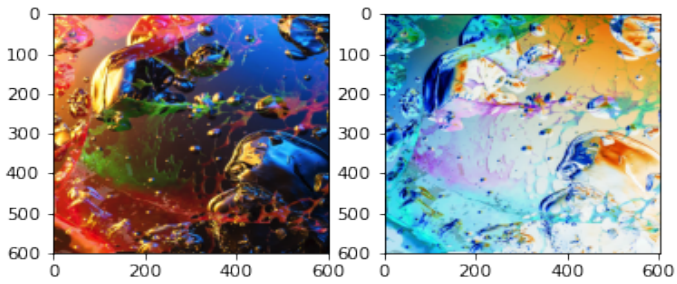

Figure 5. Test de la fonction `neg`

3.3 Floutage

On veut écrire une fonction `flou` qui à partir d'une matrice image et d'un entier `N` renvoie une matrice image floutée `N` fois (on définira `N=10` comme valeur par défaut). On floute une image en calculant pour chaque pixel une *moyenne locale* des valeurs des quatre pixels voisins :

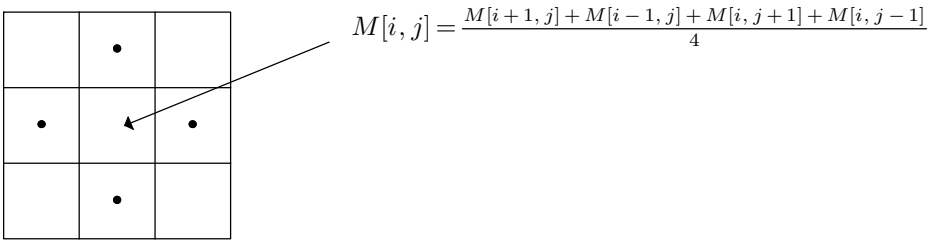

Figure 6. Principe du floutage

Question 9. Ecrire la fonction `flou` et l'appliquer sur `exemple.png` à l'aide de la fonction `test`.

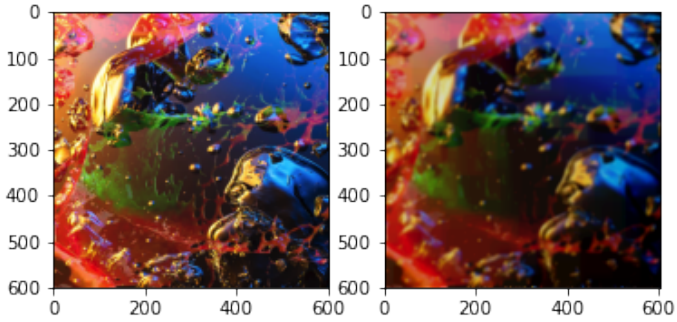

Figure 7. Test de la fonction `flou`

3.4 Seuillage

Seuiller une image consiste à transformer une image en niveaux de gris en image noir et blanc (une image où les pixels ne peuvent prendre que les valeurs $(0,0,0)$ ou $(255,255,255)$). Le principe est simple : en dessous d'un certain seuil `s` de niveau de gris, le pixel de l'image d'arrivée est noir, sinon il est blanc.

Question 10. Utiliser ce qui a été vu en introduction du TP pour créer une version niveaux de gris de l'image `exemple.png` (on pourra par exemple l'appeler `exemple_ng.png`).

Question 11. Écrire une fonction `seuil(M, s)` qui prend en argument la matrice `M` d'une image en niveaux de gris et un seuil `s` et qui retourne la matrice de l'image convertie en noir et blanc.

Question 12. A votre avis, quel peut être l'intérêt du seuillage ?

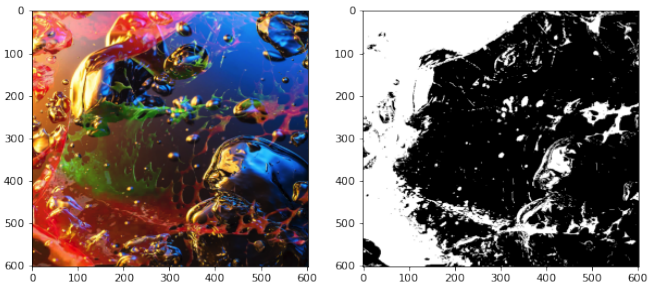

Figure 8. Test de la fonction `seuil` avec `s=120`

3.5 Contours

Voici le code d'une fonction qui calcule le gradient d'une matrice image `M` selon la direction `i` (verticale) :

```
def gradi(M):
    Mc = np.array(M, dtype=np.int16) # conversion du type
    Mg = np.zeros(M.shape) # matrice de retour
    n, m, k = M.shape
    for i in range(1, n-1):
        for j in range(m):
            Mg[i, j] = Mc[i+1, j] - Mc[i-1, j] # ?
    return np.array(np.abs(Mg), dtype=np.uint8) # valeur absolue et conversion
```

Question 13. Pourquoi est-il nécessaire de convertir le type de la matrice initiale ? Quel est l'opération mathématique appliquée à la ligne commentée par `# ?` ? Pourquoi est-il nécessaire de prendre la valeur absolue et de convertir la matrice de retour ?

Question 14. Par analogie avec le code de `gradi`, écrire une fonction `gradj` qui retourne le gradient d'une matrice image `M` selon la direction `j` (horizontale). Tester les fonctions `gradi` et `gradj`.

Question 15. Utiliser les deux fonctions précédentes pour écrire une fonction `cont(M)` qui retourne la matrice image des contours de `M`.

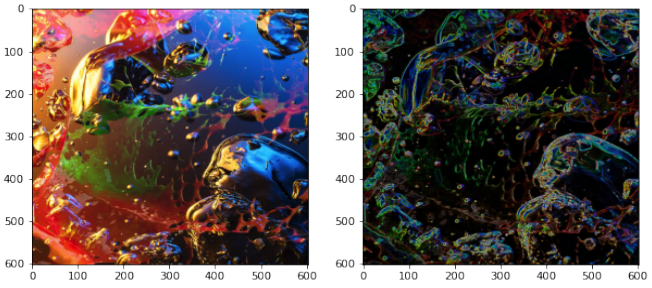

Figure 9. Test de la fonction `cont`

3.6 Réduction

Réduire une image consiste à remplacer plusieurs pixels de l'image de départ pour un seul pixel de l'image d'arrivée pour passer d'une définition initiale (n, m) à une définition (n_2, m_2) plus petite (donc $n_2 < n$ et $m_2 < m$).

Question 16. Écrire une fonction `avg(M, i1, i2, j1, j2)` qui retourne la valeur moyenne (un triplet RVB) de la sous-matrice `M[i1:i2, j1:j2]`.

Question 17. Écrire une fonction `red(M, n2, m2)` qui prend en argument une matrice image `M` et une définition d'arrivée définie par `n2` et `m2` et qui retourne la matrice image qui correspond à cette nouvelle définition.

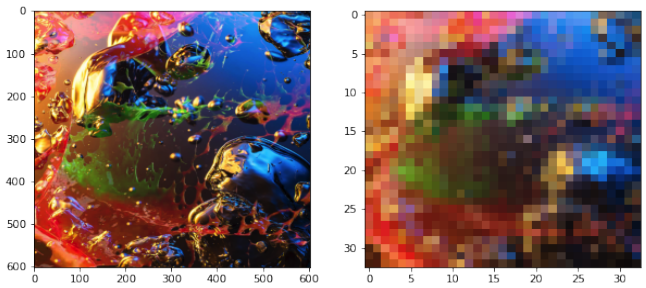

Figure 10. Test de la fonction `red` avec `n2=33` et `m2=33`.

4 Annexe : création de matrices avec **numpy**

- Créer un tableau `M` de dimensions $(n \times m \times p)$ contenant des entiers naturels égaux à zéro codés sur 8 bits :

```
M = np.zeros((n, m, p), dtype=np.uint8)
```

- Créer une copie `M2` d'un tableau `M` :

```
M2 = np.array(M)
```

- Convertir un tableau dans un autre type (ici le type de sortie est `np.float32`) :

```
M2 = np.array(M, dtype = np.float32)
```