Manipulation d'images

Cyril Charignon

Pour travailler l'utilisation de matrices et de boucles imbriquées, nous allons faire un peu de manipulation d'image.

Table des matières

1 Représentation d'une image

Une manière simple d'enregistrer une image est de la découper en petits carrés monochromes, appelés pixels, et d'enregistrer la couleur de chaque pixel dans une case d'une matrice.

Il reste à décider de la manière de coder une couleur.

- Pour une image en niveaux de gris, on choisit un entier N et on code chaque pixel par un élément de $[\![0, N[\![$. On décide que ce nombre représente la quantité de lumière émise par ce pixel, ainsi, 0 signifie noir et $N-1$ blanc. L'entier N est en général une puissance de 2, de sorte qu'un nombre entier de bits permet d'enregistrer chaque pixel.
- Pour une image en couleur, on enregistre pour chaque pixel trois nombres qui indiquent respectivement les quantités de rouge, bleu et vert.

De nombreuses autres possibilités existent, par exemple remplacer les entiers par des flottants entre 0 et 1, ajouter un quatrième nombre pour la transparence...

2 En Python

Voici quelques éléments de syntaxe que nous allons utiliser pour les exemples et exercices. Ils ne sont pas exigibles aux concours ni aux DS.

Nous allons utiliser pour charger et afficher des images la bibliothèque matplotlib. La commande suivante charge cette bibliothèque, ou plutôt sa sous-bibliothèque « pyplot » qui est celle qui nous intéresse, en mémoire sous l'alias « plt ».

```
1 import matplotlib.pyplot as plt
```

On peut alors charger une image du disque dur grâce à `plt.imread(adresse_du_ficher_image)`. Et on peut afficher une image via `plt.imshow(image)` pour créer le graphe à afficher suivi d'un `plt.show()` pour afficher effectivement ce graphe à l'écran. Ces deux opérations sont séparées pour permettre de superposer plusieurs images, et de fixer des paramètres entre les deux.

La bibliothèque matplotlib utilise en fait la bibliothèque numpy, et en particulier, pour des questions de performances, les tableaux non redimensionnables de cette dernière. C'est pourquoi le type d'une image chargée via `imread` n'est pas `list` mais `ndarray`.

Comme les ndarray ne sont pas redimensionnables, il n'y a pas de `pop` ni de `append`. En outre, ces tableaux sont typés : il est impossible de mélanger des éléments de différent type dans un tableau numpy, et il est impossible de changer le type des éléments une fois le tableau créé. Ces limitations exclues, un tableau numpy peut être manipulé de la même manière qu'un tableau ordinaire. Signalons une commande utile : une fois chargée la bibliothèque numpy via `import numpy as np`, la commande `np.zeros((n,p), dtype=...)` renvoie une matrice de format (n, p) remplie de zéros du type précisé après le `dtype=`.

3 Exemple

3.1 Passage en niveaux de gris

Si R, G, et B sont les intensités de rouge, vert et bleu d'un pixel, on estime en général que l'intensité lumineuse totale perçue par l'œil est $0.2125 * R + 0.7154 * G + 0.0721 * B$ (Norme CIE 709)

Cela signifie que le vert est une couleur perçu beaucoup plus lumineuse que les deux autres, la moins lumineuse étant le bleu.

En appliquant cette formule à chaque pixel, on transforme aisément une image couleur en image en niveaux de gris :

```
11 def en_niveaux_de_gris(im):
12     n,p,_= im.shape
13     res=np.zeros((n,p), dtype=np.uint8)  # 256 niveaux de gris
14     for i in range(n):
15         #p <print(i)
16         for j in range(p):
17             r, g, b = im[i][j]
18             res[i][j] = int(0.2125 * r + 0.7154 * g + 0.0721 * b)
19     return res
```

3.2 Histogramme

Écrivons une fonction pour calculer le nombre de pixels pour chaque intensité lumineuse. Cela nous permettra par exemple de détecter puis de corriger une image sous ou sur exposée.

```
33 def histo(im):
34     n,p,_= im.shape
35     res = np.zeros( (256,), dtype=int)
36     for i in range(n):
37         for j in range(p):
38             r, g, b = im[i][j]
39             intensité = int(0.2125 * r + 0.7154 * g + 0.0721 * b)
40             res[intensité] += 1
41     return res
```

3.3 Améliorer le contraste

Pour changer l'exposition et le contraste, commençons par écrire une fonction générique qui permet d'appliquer une transformation quelconque à chaque pixel.

```
50 def ramène_à_256(x):
51     if x<0:
52         return 0
53     elif x>255:
54         return 255
55     else:
56         return int(x)
```

```
1 ↦
72 def fonction_appliquée(im, f):
73     """
74     Applique f à chaque canal de chaque pixel.
75     """
76     n,p,_= im.shape
77     res=np.zeros((n,p,3), dtype=np.uint8)
78     for i in range(n):
79         for j in range(p):
80             for canal in range(3):
81                 res[i][j][canal] = ramène_à_256(f(im[i][j][canal]))
82     return res
```