Feuille de TD n° 14

Introduction

Dans ce TP, nous allons manipuler les images numériques dans leur représentation la plus simple.

Une image en couleurs est usuellement représentée comme un tableau rectangulaire (hauteur x largeur) dont chaque donnée est un triplet de nombres entiers (code RGB : teintes de couleur Rouge, Vert, Bleu, nombres entre 0 et 255). Chacun de ces triplets de nombres entiers est appelé **pixel**, c'est le plus petit élément coloré d'une image.
Lorsque l'on charge une image sur python avec numpy, celle-ci est alors stockée sous forme d'un tableau à 3 dimensions, de paramètres `(hauteur, largeur, couleur)`. C'est-à-dire que pour `img` une image, `img[i,j,0]` est l'entier correspondant à la teinte de rouge du pixel à la position (i, j). Les valeurs `img[i,j,1]` et `img[i,j,2]` sont respectivement pour les teintes de vert et de bleu.
On peut ainsi manipuler une image plus ou moins comme une liste, en faisant attention aux indices.
Beaucoup de formats d'images différents existent (dans le but de stocker une image donnée en prenant le moins de place possible), mais ce TP se focalisera sur ce format, qui est entre autres représenté par les fichiers de type `png`.

Chargez les librairies suivantes :

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

Récupérez les images `chat.png`, `coucher.png` et `ratel.png`. Elles devront être dans le même dossier que le document python utilisé.
On peut retrouver le dossier du document python actuel via :

```
import os
print(os.getcwd())
```

Exercice 1 (**Chargement d'une image avec `plt.imread`**).

1. Charger l'image `chat.png` :

```
img_chat = plt.imread("chat.png")
print(img_chat.shape)   # (h, w, 3) ou (h, w, 4) si RGBA
print(img_chat.dtype)   # en général float32, float64 OU uint8
print(img_chat.max())   # souvent 1.0
```

2. Certaines images ont leurs teintes de pixels stockées sous forme de nombres flottants entre 0 et 1 au lieu d'entiers entre 0 et 255.
   Expliquer le sens des abréviations $float32$, $float64$, $uint8$.
3. Quelle différence principale y a-t-il entre ces trois types ?
4. L'instruction `max` fonctionne exactement comme pour une liste unidimensionnelle : elle renvoie la valeur maximale présente dans le tableau. (donc souvent 1.0 ou 255 pour une teinte de couleur maximale).
5. Si le type de pixels de l'image est en `float`, on peut le convertir en `uint8` avec les instructions :

```
if img_chat.dtype == np.float32 or img_chat.dtype == np.float64:
    img_chat = (img_chat * 255).astype(np.uint8)
```

6. On peut aussi afficher l'image sur python :

```
plt.figure(figsize=(8,6))
plt.imshow(img_chat)
plt.axis('off')
plt.show()
```

Modification manuelle d'une image

4. Modifier le pixel $(0, 0)$ de l'image `img_chat` pour le rendre noir, et afficher le résultat.
   Un tableau numpy possède les mêmes subtilités d'usage qu'une liste. Lorsqu'on le manipule, il est bon d'en créer une copie afin d'éviter des effets de bord.

5. On peut créer une copie du tableau et ajouter une bande rouge verticale via :

```
img_mod = img_chat.copy()
img_mod[:, 100:150, 0] = 255   # Rouge
img_mod[:, 100:150, 1] = 0
img_mod[:, 100:150, 2] = 0
plt.imshow(img_mod)
plt.show()
```

   Indiquer les coordonnées de tous les points colorés en rouge par ces instructions.
6. Créer une bande bleue horizontale large de 50 pixels au milieu de l'image.

Création d'une image

6. Puisqu'une image sur numpy est un tableau tri-dimensionnel, on peut créer une image en définissant un tableau tri-dimensionnel.
   On utilise les instructions suivantes :

```
h, w = 300, 400 #largeur, hauteur
img = np.zeros((h, w, 3), dtype=np.uint8)
```

   De quelle couleur est une telle image ? Pourquoi ?
7. Quel est le code couleur du noir ? Et celui du blanc ?
8. On peut alors modifier les couleurs de l'image en modifiant la valeur de chaque élément du tableau. Cela se fait souvent via deux (ou trois) boucles for.
   Créer une image `img2` de taille 300×400, entièrement blanche.
9. Créer une image `img3` de taille 400×500, telle que le pixel i, j a une teinte de rouge valant $2 * |i - j|$, une teinte de vert valant $255 - |i - j|$, et une teinte de bleu valant 100.
   On fera attention à ce que ces valeurs soient des entiers entre 0 et 255.

10. Afficher le résultat avec `plt.imshow(img2)`. Quel effet obtient-on ?
11. Créer une image `img4` de plateau d'échecs, dont chaque case noire/blanche fait 20 pixels de côté.
    On pourra s'aider de divisions euclidiennes par deux entiers bien choisis.

Exercice 2 (**Inversion chromatique**)**.**
L'inversion chromatique consiste à modifier la teinte de couleur de chaque pixel par sa valeur complémentaire (complément à 255).

1. Ecrire une fonction `inversion(img)` qui prend en entrée une image `img` et réalise une inversion chromatique de l'image.
   On fera attention à créer une copie de l'image, et on utilisera `img.shape` pour récupérer la hauteur h et la largeur l de l'image.
2. Quelle est l'inversion chromatique de l'inversion chromatique ?
3. Tester l'instruction

```
img4 = 255 - img_chat
```

   Que cela fait-il ? et pourquoi ?
4. Créer une fonction `invers_part(img)` d'inversion chromatique partielle (seuls rouge et bleu sont inversés).

Exercice 3 (**Floutage d'une image**)**.**
Le floutage d'une image consiste à appliquer à chaque pixel une moyenne des couleurs des pixels voisins. La forme de la moyenne et du voisinage de pixels définissent le floutage effectué.

1. On peut extraire une sous-image d'une image `img` avec l'instruction :

```
img_extr = img[a:b,c:d]
```

   Charger l'image `coucher.png` sous le nom `img_coucher`, et tester cette instruction sur l'imageavec des valeurs pertinentes de a, b, c, d.
   Quelle portion de l'image est extraite ?
2. Créer une fonction `moyenne(img)` qui prend en entrée une image `img` et qui renvoie la valeur moyenne des couleurs de tous les pixels de l'image. Cette fonction renverra donc un triplet d'entiers.
3. Ecrire une fonction `flou_3x3(img)` qui prend en entrée une image `img` et qui renvoie l'image obtenue en floutant chaque pixel (i, j) par la coloration moyenne du bloc de pixels $3x3$ centré en (i, j) (sauf pour les pixels en bordure d'image, qui seront intouchés).

4. Charger l'image `ratel.png` sous le nom de `img_ratel`. Appliquez le flou sur l'image `img_ratel`.
   Que se passe-t-il si on l'applique plusieurs fois ?
5. Comment pourrait-on appliquer aussi un flou aux bords de l'image ?

Exercice 4 (Stéganographie).
La stéganographie consiste à cacher des une image dans une autre image. Un peu de précision est perdue entre les étapes, mais on obtient un processus assez efficace pour dissimuler un message dans une image qui est "invisible" à l'oeil. Une façon de réaliser cela numériquement est d'utiliser l'écriture en binaire des couleurs de chaque pixel.
Une couleur est représentée par un entier entre 0 et 255, c'est-à-dire par un nombre binaire à 8 chiffres (un entier sur 8 bits). Sur ces 8 chiffres, si l'on conserve les 4 chiffres significatifs (les 4 bits de poids fort), l'image sera modifiée mais seulement légèrement. On peut ainsi récupérer les 4 bits de poids fort d'une image A, et "cacher" ces 4 bits dans les 4 bits de poids faible d'une image B. Les couleurs de l'image B ne seront que légèrement changées, donc la modification sera peu visible, et il suffira de récupérer les 4 bits de poids faibles des couleurs de B pour reconstruire l'image A.

1. Ecrire deux fonctions `4bits_poids_fort(x)` et `4bits_poids_faible(x)` qui prennent un entier x entre 0 et 255 et qui renvoient l'entier y obtenu en conservant les 4 bits de poids fort de x (respectivement les 4 bits de poids faible) dans son écriture en binaire.
   Par exemple, pour $x = 200 = (1100\,1000)_2$ la première fonction renverra $192 = (1100\,0000)_2$ et la seconde renverra $8 = (0000\,1000)_2$.
   On utilisera les opérateurs // et % avec des entiers bien choisis.
2. Tester les fonctions sur 201, 75, 255.
3. Ecrire une fonction `Cacher(a,b)` qui prend en entrée deux entiers a et b entre 0 et 255 et qui renvoie en sortie l'entier c obtenu en "cachant" les 4 bits de poids fort de a dans les 4 bits de poids faible de b.
   Par exemple, pour $a = 200 = (1100\,1000)_2$ et $b = 70 = (0100\,0110)_2$, la fonction renverra $c = (0100\,1100)_2 = 76$.
   On utilisera les fonctions précédentes et l'opérateur // avec un entier bien choisi.
4. Ecrire une fonction `DeCacher(c)` qui prend en entrée un entier c entre 0 et 255 et qui renvoie en sortie l'entier a obtenu en "récupérant" les 4 bits de poids fort de a dans les 4 bits de poids faible de c.
   Par exemple, pour $c = (0100\,1100)_2 = 76$, la fonction renverra $a = (1100\,0000)_2 = 192$.
5. L'avantage des tableaux numpy est le fait qu'on peut leur appliquer des opérations arithmétiques simples (ou des opérations de la librairie numpy) et que cela affectera directement chaque coefficient du tableau.
   Créer une fonction `CacherImage(imgA,imgB)` qui prend en entrée deux images `imgA` et `imgB` de mêmes tailles et qui renvoie l'image `imgC` obtenue en cachant `imgA` dans `imgB` par stéganographie.
   On commencera par vérifier que `imgA` et `imgB` ont la même taille, puis par convertir leur type en `uint8` afin de manipuler des entiers entre 0 et 255.
   On s'aidera des fonctions précédentes.
6. Créer une fonction `DeCacherImage(imgC)` qui prend en entrée une image `imgC` et qui renvoie l'image `imgA` obtenue en décachant par stéganographie.
7. Cacher l'image `ratel.png` dans l'image `coucher.png`, puis la décacher. Comparer l'image cachée à `coucher.png` et l'image décachée à `ratel.png`.