MANIPULATION DES IMAGES

1 PRÉREQUIS

Bibliothèques :

1) **matplotlib.pyplot** pour manipuler les images.

   Chaque image est ouverte et mise sous la forme d'un tableau de pixels à trois dimensions. Deux dimensions pour désigner l'emplacement du pixel, et une troisième dimension pour donner sa couleur.

2) **numpy** pour manipuler les tableaux.

```
from matplotlib.pyplot import *
import numpy as np
```

Image pour les tests :

Vous devez disposer d'une image (en png) pour faire les manipulations.
Il est conseillé de commencer avec des images de petite taille pour éviter les traitements trop longs (environ 400 pixels de côté par exemple).
Nous appellerons **file** le nom du fichier correspondant.

Cela signifie que **file** désigne une chaine de caractères (sans oublier l'extension) correspondant au nom du fichier. Par exemple `file = "monimage.png"`.

Nous utiliserons l'image ci-contre dans les exemples :

```
image = imread(file)          # ouverture - transforme l'image en tableau
imshow(image)                 # affichage - affiche l'image
```

Le mode "RGB" : Les images sont en mode "RGB". Ainsi, chaque pixel est codé par un triplet (R,G,B). R, G et B désignent respectivement la quantité de Rouge, de Vert et de Bleu dans chaque pixel[1]. Cette quantité est codée par des entiers de $[\![0,255]\!]$ (octets en base 2)[2].

Il s'agit d'un système de couleur additif. Ainsi, la couleur $(0,0,0)$ code le noir (absence de couleur), et $(255,255,255)$ désigne le blanc (les trois couleurs à 100%).

Pour travailler avec ce mode, on spécifiera toujours le type `dtype = 'uint8'` (octets en base 2) lors de la création de tableaux images. Si on crée des tableaux d'un autre type, ils risquent d'être interprétés différemment.

```
# enlever la transparence
image = image[...,:3]
#  passer les couleurs en 0-255, si ce sont des flottants
if image.dtype == np.float32:
    image = np.uint8(255*image)
```

Utilisation de numpy :

⚠ Dans les tableaux , la première coordonnée désigne le numéro de la ligne et la seconde désigne le numéro de la colonne. On se déplace donc dans le tableau comme dans une matrice, le pixel de coordonnée $[0,0]$ étant en haut à gauche.

Par exemple, si `image` désigne le tableau numpy, alors `image[i,j,2]` désigne la quantité de bleu dans le pixel de ligne i et de colonne j.

[1] Dans les images png, il y a un 4ème élément correspondant au niveau de transparence. Nous nous en affranchissons dans ce TD. Il existe aussi d'autres modes pour coder les couleurs (noir et blanc, niveau de gris, HSV...)

[2] C'est l'usage *traditionnel*. matplotlib permet aussi de travailler avec des flottants dans $[0,1]$ à la place.

2 MANIPULATIONS GÉOMÉTRIQUES DES IMAGES

Autant que possible, on essayera d'utiliser le slicing pour accélérer les processus.

1) Programmer la fonction **miroir(image)**.

image

miroir(image)

2) Programmer la fonction **rotation(image)**.

image

rotation(image)

3) Programmer la fonction **quart(image)** qui divise la taille de l'image par deux dans la longueur et dans la largeur (on prendra un pixel sur 2 dans chaque direction tout simplement)

(a) image

(b)
quart(image)

4) Programmer la fonction **quatre(image)** qui multiplie la taille de l'image par deux dans la longueur et dans la largeur (on doublera chaque pixel en hauteur et en largeur).

(a) image

(b) quatre(image)

5) Programmer la fonction **identite(image)** qui sépare l'image en 4 (comme les photos d'identité). La taille du fichier ne doit pas être modifiée.

Pour ce faire, on pourra utiliser la parité des lignes et des colonnes. Si le pixel est sur une ligne paire, il ne change pas de ligne, mais s'il est sur une ligne impaire, il passe sur l'autre moitié de l'image. De même pour les colonnes (les quatre images sont donc différentes les unes des autres).

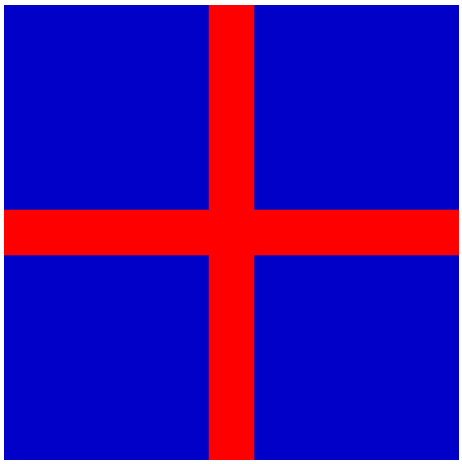
Si vous effectuez cette manipulation un certain nombre de fois sur la même image, vous devez revenir à l'image initiale (par contre, cela peut être très long). Pour le voir, essayez avec une image de taille 256×256.

(a) image (b) identite(image) (c) identite(identite(image))

Figure 1: Photos d'identité

3 INTERLUDE

Créer une image de 400×400 qui affiche une croix rouge sur fond bleu.
Les branches de la croix font 40 de large et passent par le centre de l'image.



4 GESTION DES COULEURS

A Algorithmes basiques

Dans cette partie, on ne s'occupera pas trop des problèmes de complexité. L'important est d'avoir un algorithme qui donne le bon résultat, même si c'est un peu long.

1) Écrire une fonction **eclaire(image,p)** qui renvoie la même image, mais dont les quantités de chaque couleur ont été augmentées de p%.

(a) eclaire(image,−30) (b) eclaire(image,0) (c) eclaire(image,50) (d) eclaire(image,100)

Figure 2: Éclairage de l'image

2) Écrire une fonction **balance(image, r,g,b)** qui fait la même chose, mais augmente chaque couleur de son propre pourcentage.

3) Écrire une fonction **niveaugris(image)** qui transforme une image en niveaux de gris.

Pour chaque pixel, cette fonction fait la moyenne pondérée des trois couleurs avec les poids respectifs : 0.2989, 0.5870, 0.1140 et attribue cette moyenne aux trois couleurs.

(a) balance(image,100,−20,−20) (b) balance(image,0,0,0) (c) balance(image,−100,0,100)

Figure 3: Balance des couleurs

(a) file (b) niveaugris(image)

Figure 4: Niveaux de gris

4) Écrire une fonction **noirblanc(image, seuil=125)** qui transforme une image en noir et blanc. Le seuil indique à partir de quel niveau d'éclairement (de 0 à 255), le pixel doit être blanc (on pourra utiliser la fonction **niveaugris**).

(a) noiret-blanc(image,100) (b) noiret-blanc(image,125) (c) noiret-blanc(image,150) (d) noiret-blanc(image,200)

Figure 5: Noir et blanc

B Sensibilisation aux problèmes de complexité

Les deux fonctions ci-dessous font la même chose : extraction du canal c de l'image. Ainsi, pour $c = 1$, la fonction n'affiche que le rouge de l'image.

La première fonction utilise des boucles explicites, la seconde utilise le slicing et laisse donc numpy gérer seul les boucles.

Exécuter les deux fonctions et comparer leur temps d'exécution.

```
def canal1(image,c):
    ''' donne le canal c = 0,1,2 de l'image '''
    imageTmp = np.zeros(image.shape,dtype="uint8")
    for y in range(image.shape[0]):
        for x in range(image.shape[1]):
            imageTmp[y,x,c] = image[y,x,c]
    imshow(imageTmp)
def canal(image,c):
    ''' donne le canal c = 0,1,2 de l'image '''
    imageTmp = np.zeros(image.shape,dtype="uint8")
    imageTmp[:,:,c] = image[:,:,c]
    imshow(imageTmp)
```

C Algorithmes optimisés

Dans cette partie, on cherche à optimiser les algorithmes précédents. Cela nous conduit à utiliser davantage de fonctions numpy. C'est plus difficile même si le code tient souvent en très peu de lignes.

Cette partie va au delà des exigences de BCPST. Cela montre ce que l'on peut faire en étudiant un peu la documentation pour optimiser les algorithmes en vue d'un usage plus "*professionnel*".

La partie du cours en ligne sur les fonctions avancées peut être avantageusement utilisée ici. Mais vous avez également tout loisir d'ouvrir la documentation de NumPy et de proposer d'autres méthodes plus rapides ou plus élégantes. Soyez audacieux!

Pour la suite, on définit la fonction `convert`

```
def convert(image):
    return np.uint8(np.minimum(image,255))
```

Cette fonction tronque les coefficients du tableau à 255 avec la fonction `minimum` pour que la couleur reste dans l'intervalle $[\![0,255]\!]$ et convertit le tableau avec le type `uint8` pour l'exploiter comme une image. On pourra l'utiliser dans la suite du TD.

1) Écrire la fonction **eclaire** sans utiliser de boucles explicites, ni par compréhension.

2) Écrire la fonction **balance**, on pourra utiliser une boucle pour parcourir les trois couleurs.

3) Écrire la fonction **balance** en utilisant la notation d'Einstein (voir dans le cours en ligne).

4) Utiliser la notation d'Einstein pour programmer la fonction **niveaugris**.