MATRICES ET IMAGES

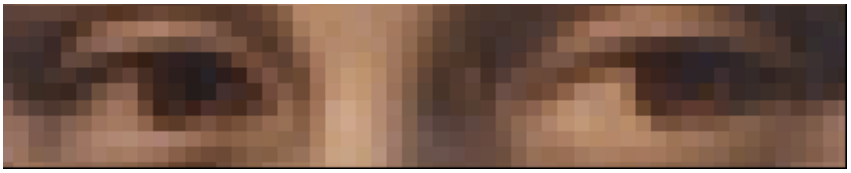
Ce TP utilise des fichiers d'image. Télécharger ces fichiers sur le site, et les mettre dans le dossier courant.

1 Une image, c'est quoi ?

Un pixel, c'est quoi ? Une image est une grille de *pixels* (PICture ELements). Chaque pixel représente un carré de couleur :

- Si l'image est en noir et blanc : un nombre de 0 à 255 (ou 0 à 1) pour la nuance de gris.
- Si l'image est en couleurs : trois nombres de 0 à 255 (ou 0 à 1) pour les couleurs RGB : Red / Green / Blue
- Éventuellement un quatrième nombre de 0 à 1 pour la transparence (RGBA, avec A pour Alpha)

Si on zoome suffisamment sur une image, on peut voir les pixels :



Pour coder un nombre de 0 à 255, une machine aura besoin de 8 *bits* (0 ou 1) :

0 : 0000 0000	4 : 0000 0100	32 : 0010 0000
1 : 0000 0001	...	64 : 0100 0000
2 : 0000 0010	8 : 0000 1000	128 : 1000 0000
3 : 0000 0011	16 : 0001 0000	255 : 1111 1111

Un ensemble de 8 bits est ce qu'on appelle... un *octet* ! Autrement dit, en nuances de gris, il faut 1 octet par pixel, et en couleurs il faut 3 octets par pixel.

Formats des fichiers images. Il existe plusieurs formats pour stocker une image.

- Le format .bmp (pour *bitmap*) stocke l'image en brut. Pour une image bitmap de dimensions $H \times L$ pixels, son poids sera de HL octets (noir et blanc) ou $3HL$ octets (couleurs). Le poids réel est un peu plus grand car le fichier stocke également quelques informations supplémentaires, comme les couleurs utilisées. C'est un format de stockage assez lourd, qui n'est presque plus utilisé de nos jours.
- Les formats .png et .gif compressent les données d'un fichier .bmp pour alléger le fichier, chacun avec une méthode différente. Le fichier est moins lourd, et l'image reste de la même qualité que l'original.
- Le format .jpg réalise une meilleure compression des données que les autres formats, mais c'est une compression *destructive* : la qualité de l'image diminue légèrement après chaque sauvegarde, voir ci-dessous :

Image d'origine  Après 100 sauvegardes

Dans ce TP, on ne travaillera qu'avec du bitmap.

2 Lecture, affichage, enregistrement d'images

Il existe de nombreuses bibliothèques en Python pour travailler avec des images. Nous irons au plus simple : la bien connue `matplotlib` avec ses modules `pyplot` et `image`, ainsi que le module `numpy` pour travailler avec les tableaux numpy.

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

A = img.imread("joconde.bmp")

plt.close(0)
plt.figure(0)  # ouvre une fenêtre "Figure 0" et met un pointeur "plot" dessus
plt.imshow(A)  # tous les plots sont tracés là où le pointeur "plot" pointe
plt.show()
```

Les graduations sur le dessin montrent le nombre de pixels : 321 en largeur et 480 en hauteur. On peut zoomer avec la loupe pour voir les pixels en détail. L'icône maison permet de revenir au cadrage original.

Afficher `A` dans la console. On voit qu'on a affaire à un tableau numpy (ou *array*). Pour voir sa taille, entrer `A.shape`. Il y a 3 dimensions. C'est en fait une liste de listes de listes (ouf) :

- il y a une liste de 480 éléments (qui correspond à la hauteur), dont chaque élément est...
- une liste de 321 élémnets (qui correspond à la largeur), dont chaque élément est...
- une liste de 3 éléments : `[r,g,b]` où $\mathtt{r},\mathtt{g},\mathtt{b} \in [0,255]$ donnent les couleurs du pixel.

L'instruction `A[i][j]` donne les couleurs `[r,g,b]` du pixel de la ligne `i` et de la colonne `j`. Pour un tableau numpy, on peut aussi écrire `A[i,j]`. Ajouter ces lignes au script :

```
Dim = A.shape
hauteur,largeur = Dim[0],Dim[1]
```

3 Traitements d'image locaux

Un traitement *local* est un traitement pixel par pixel : on applique une fonction $f : [0,255]^3 \to [0,255]^3$ à chaque élément du tableau numpy.

La couleur c'est cool. On va supprimer la couleur verte de chaque pixel.

```
B = np.copy(A)      # A est en lecture seule par défaut. La copie B ne l'est pas
for i in range(hauteur):
    for j in range(largeur):
        B[i,j,1] = 0

plt.close(1)
plt.figure(1)  # maintenant le pointeur "plot" pointe sur Figure 1
plt.imshow(B)
plt.show()
```

Exercice 1. Appliquer le code suivant. Modifier ensuite la couleur à supprimer, ou deux, ou la mettre au maximum ; apprécier le résultat.

Astuce mnémotechnique : si le pixel est `[99,111,222]`, il n'est pas toujours évident de se rappeler à quelle couleur correspond la valeur `99`. Mémorisez que c'est le système *RGB*, donc *Red* en premier, *Green* en deuxième et *Blue* en dernier (c'est aussi le même ordre que l'arc-en-ciel ou les longueurs d'onde).

La couleur blanche correspond à `[255,255,255]` tandis que la couleur noire est `[0,0,0]`.

Exercice 2. Tracer un trait blanc entre les points de coordonnées $(200,50)$ et $(200,250)$. Faire de même entre $(0,50)$ et $(200,100)$.

50 nuances de noir et blanc. On va maintenant convertir l'image en niveaux de gris. Pour cela, on doit appliquer une fonction $f_{gris} : [0,255]^3 \to [0,255]$. Plusieurs choix sont possibles. On va prendre la suivante, et ajouter une fonction de « traitement » :

```
def fgris(pixel):
    return 0.299*pixel[0] + 0.587*pixel[1] + 0.114*pixel[2]
def traitement(p):
    return p
```

Maintenant :

```
def noir_et_blanc(image):
    hauteur,largeur,_ = image.shape
    newImage = np.zeros((hauteur,largeur))  # tableau de zéros de taille HxL
    for i in range(hauteur):
        for j in range(largeur):
            newImage[i][j] = fgris(image[i][j])
            newImage[i][j] = traitement( newImage[i][j] )
    return(newImage)

B = A.copy()
B = noir_et_blanc(B)

plt.close(3)   # permet d'effacer ce qu'il y a déjà sur la figure 3 en la fermant
plt.figure(3)
plt.imshow(B, cmap = 'Greys_r')   # précise que l'image est en niveau de gris
plt.show()
```

Exercice 3. Compiler et regarder l'image obtenue. Essayer d'autres traitements, tel que $p \mapsto p^2$, ou encore $p \mapsto \ln p$ (avec la fonction `np.log`).

Dans `plt.imshow(...)`, on peut aussi utiliser d'autres *colormaps* : essayer `cmap='hot'` ou `cmap='jet'`.

Exercice 4. Adapter ce qui précède pour afficher le négatif d'un tableau image. Le négatif s'obtient en appliquant le traitement $p \mapsto 255 - p$ à chaque pixel.

Noter que cela marche aussi pour une image en couleur. Il faut alors appliquer la fonction f sur chaque couleur de chaque pixel.

Modifier le contraste. Jusqu'à présent, on a travaillé sur des pixels qui prenaient leurs valeurs dans $[0,255]$. Il est souvent plus pratique de travailler dans $[0,1]$, alors dans cette section on va commencer par

```
B=A.copy()/255
```

Un traitement local consistera maintenant à appliquer une fonction $g : [0,1]^3 \to [0,1]^3$ sur chaque pixel, ou encore à appliquer une fonction $g : [0,1] \to [0,1]$ à chaque couleur de chaque pixel.

Augmenter le contraste consiste à amplifier les différences de couleur. Cela améliore la visibilité des formes mais cela peut entraîner une perte d'information car les pixels trop extrêmes deviennent impossibles à distinguer (cf les `if` de la fonction `contrPlus` ci-dessous). Pour modifier le contraste, on doit appliquer les fonctions `contrPlus` et `contrMoins` à chaque élément de `image`, donc faire 3 boucles (ligne / colonne / couleur). Fort heureusement, si on veut appliquer une même fonction à tous les éléments d'un tableau numpy, on peut utiliser `np.vectorize` (cf page suivante)

Exercice 5. Avec les fonctions `CP` et `CM` (cf ci-dessous), augmenter ou baisser le contraste de l'image en niveaux de gris et l'afficher.

```
def contrPlus(color):  # fonction de [0,1] dans [0,1] qui augmente le contraste
    if color < 0.05:
        return 0
    if color > 0.5:
        return 1
    return 2*color-0.1

CP = np.vectorize(contrPlus)
print( CP( [0.01 0.9 0.3] )) # pour voir ce que permet np.vectorize
```

```
def contrMoins(color):  # fonction de [0,1] dans [0,1] qui diminue le contraste
    return 0.25+0.5*color

CM = np.vectorize(contrMoins)  # on peut appliquer CM au tableau image directement
```

Les valeurs telles que `0.05` ci-dessus sont purement arbitraires. On peut s'amuser à modifier `contrMoins` avec des fonctions trigo ou des fonctions comme $x \mapsto x - x^2$.

4 Traitements d'image globaux

Les traitements d'image globaux peuvent changer un pixel en fonction d'un ou plusieurs autres pixels. C'est beaucoup plus général qu'un traitement local.

Floutage. Pour flouter une image, chaque couleur d'un pixel sera la moyenne de ses 9 voisins (lui-même plus les 8 qui l'entourent), ou 25 voisins, ou plus. On dispose donc d'un paramètre d pour faire la moyenne des $(2d+1)^2$ voisins. Sur les bords, un pixel a moins de voisins, donc pour que ce soit plus facile on ne va flouter que les pixels des positions $[d, H-d-1] \times [d, L-d-1]$.

Exercice 6. Commenter et appliquer le code suivant (garder $d \leq 3$ sinon c'est très long).

```
d = 1
B = A.copy()
for i in range(d,hauteur -d):
    for j in range(d,largeur -d):
        for k in range(3):
            test[i,j,k]=np.sum(B[i-d:i+d+1,j-d:j+d+1,k])/(2*d+1)**2
```

Dilatation. L'objectif est de déformer l'image de façon à zoomer autour d'un point (x_0, y_0) tout en contractant les pixels les plus éloignés. On verra quand même toute l'image, mais les points proches de (x_0, y_0) sembleront plus gros et ceux éloignés sembleront plus petits. L'idée est d'utiliser une fonction de déformation

$$\begin{aligned} f : \mathbb{R}^2 &\to \mathbb{R}^2 \\ (x,y) &\mapsto f(x,y) \end{aligned}$$

Le pixel de coordonnées (x,y) de `B` sera identique à celui de coordonnées $f(x,y)$ de `A`.

Exercice 7. Ouvrir le fichier TP8_dilatation.py, le compiler puis commenter. Que valent $f(0,0)$ et $f(\texttt{lg}, 0)$? Changer (x_0, y_0) et le grossissement.

5 Exercices d'approfondissement

Exercice 8 (Une surprise vous attend !)**.** Télécharger l'image « joconde_inversee » sur le site, et écrire une fonction qui réalise une rotation à 180° pour la remettre à l'endroit. *Indication : si un pixel se trouve à la position (i, j), où doit-il se trouver lorsqu'on retourne l'image ?*