MATRICES ET IMAGES

Dans ce TP, on va travailler sur un fichier image nommé « `joconde.bmp` ». Dans la console, taper l'instruction : `cd`. Ceci renvoie le « current directory », i.e. le dossier courant. Quand on souhaite importer un fichier dans Python, le current directory est le premier lieu qui sera examiné pour trouver ce fichier. C'est donc dans le « current directory » qu'il faudra placer le fichier `joconde.bmp`. On peut changer ce dossier en tapant : `cd Chemin\Complet\Du\Nouveau\Dossier`.

Télécharger les fichiers du TP (accessibles tout en bas sur le site), et les mettre dans le dossier courant.

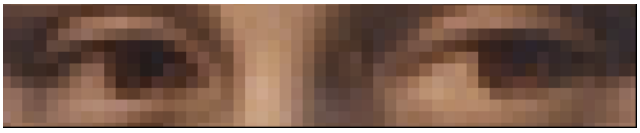
Note : on peut aussi utiliser le module `os`. Avec `os.getcwd()` on peut obtenir le « current (working) directory » et avec `os.chdir("Chemin\...")` on peut changer ce dossier. Attention à bien mettre des guillemets !

1 Une image, c'est quoi ?

Une image est une grille de *pixels* (PICture ELements). Chaque pixel représente un carré de couleur, auquel on associe :

- *(si l'image est en noir et blanc :)* un nombre de 0 à 255 (ou 0 à 1) pour la nuance de gris du pixel.
- *(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 en binaire, une machine aura besoin de 8 *bits* (0 ou 1) :

0 : 0000 0000	3 : 0000 0011	8 : 0000 1000	64 : 0100 0000
1 : 0000 0001	4 : 0000 0100	16 : 0001 0000	128 : 1000 0000
2 : 0000 0010	...	32 : 0010 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. Pour stocker les pixels d'une image dans un fichier, plusieurs formats existent :

- Le format .bmp (pour *bitmap*) stocke l'image en brut. Une image bitmap de dimensions $H \times L$ pixels aura un poids de HL octets (noir et blanc) ou $3HL$ octets (couleurs). Le poids réel est un peu plus grand car le fichier stocke aussi d'autres informations (des métadonnées). Ce format assez lourd est presque inutilisé 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

2 Lecture et affichage d'images

Il existe de nombreux modules en Python pour travailler avec des images. Nous irons au plus simple en utilisant le module `matplotlib` avec ses sous-modules `pyplot` et `image`, ainsi que le module `numpy`. Recopier et compiler le code suivant :

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

A = img.imread("joconde.png") # ici chaque couleur "varie" entre 0 à 1

A = A*255              # maintenant chaque couleur "varie" entre 0 et 255
A = A.astype("int")    # et est codée par un entier et non un flottant

def affichage(M):
    plt.close()                              # ferme toutes les figures
    plt.figure()                             # ouvre une nouvelle figure
    plt.imshow(M,interpolation="nearest") # dessine l'image sans "fondu"
    plt.show()                               # affiche l'image
    return None

affichage(A)   # permet d'afficher l'image en une ligne
```

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 la variable `A` dans la console. On voit que c'est 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). On y trouve...

- une liste de 480 éléments (qui correspond à la hauteur), dont chaque élément est .....
- une liste de 321 éléments (qui correspond à la largeur), dont chaque élément est .....
- une liste de 4 éléments : `[r,g,b,a]` où `r,g,b,a` sont des entiers de 0 à 255 qui donnent les couleurs et l'opacité du pixel à cet endroit.

Ajouter ces lignes au script :

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

3 Traitements d'image locaux

Note : l'instruction `A[i][j]` récupère donc les couleurs du pixel de la ligne `i` et de la colonne `j`, càd une liste du type `[r,g,b,a]`. Pour un *tableau numpy*, on peut aussi écrire `A[i,j]` au lieu de `A[i][j]` .

Changer la couleur. Un traitement *local* est un traitement pixel par pixel : à chaque élément du tableau numpy, on applique une fonction $f : [0,255]^3 \to [0,255]^3$. Par exemple, on peut supprimer la couleur verte de chaque pixel.

```
B = A.copy()      # A est en lecture seule. La copie B ne l'est pas

for i in range(hauteur):
    for j in range(largeur):
        B[i,j,1] = 0  # La composante 1 code la couleur verte

affichage(B)
```

Exercice 1. Appliquer le code suivant. Modifier le code pour saturer la couleur verte (i.e. la mettre à sa valeur maximum en tout point), puis faite varier les plaisirs : supprimer deux couleurs, saturer la troisième, etc.

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 $(200,250)$, puis 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 considérer la fonction 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 nuanceGris(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 = nuanceGris(B)

plt.close()
plt.figure()
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. Tester le code ci-dessus avec une fonction de traitement qui permet d'obtenir le négatif de l'image. Le négatif d'un pixel correspond à une fonction affine $f : [0,255] \to [0,255]$ telle que $f(255) = 0$ et $f(0) = 255$.

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.

```
def contrPlus(color):  # fonc° de [0,1] dans [0,1] qui augmente le contraste
    if color < 0.05:
        return 0       # on ne distingue plus les couleurs entre 0 et 0.05
    if color > 0.55:
        return 1       # on ne distingue plus les couleurs entre 0.55 et 1
    return 2*color-0.1

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

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 $x \mapsto x - x^2$.

Pour modifier le contraste, on doit appliquer les fonctions `contrPlus` et `contrMoins` à chaque élément de `B`, 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 ci-dessous).

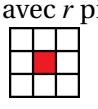
```
cpVect = np.vectorize(contrPlus) # cpVect est maintenant une FONCTION...
print(cpVect( [0.01 0.9 0.3] ))  #... qui applique contrPlus à chaque élément

cmVect = np.vectorize(contrMoins)
```

Exercice 5. Avec les fonctions `cpVect` et `cmVect`, modifier le contraste de l'image en niveaux de gris et l'afficher.

4 Traitements d'image globaux

Un traitement d'image global peut changer un pixel en fonction d'un ou plusieurs autres pixels (en général proches du pixel qu'on considère). C'est donc beaucoup plus général qu'un traitement local.

Floutage. Pour flouter une image, chaque pixel de l'image aura une nouvelle couleur, obtenue par une moyenne des couleurs de ses « voisins ». Pour déterminer les voisins, on se fixe un paramètre $r \in \mathbb{N}^*$. Ce paramètre représente le « rayon » du floutage : le pixel de départ est placé au centre d'un carré avec r pixels à sa droite, à sa gauche, en bas et en haut. Par exemple, avec $r = 1$ et $r = 2$, cela donne respectivement :  et  .

Question 1. Étant donné $r \in \mathbb{N}^*$, combien de pixels contient le carré avec un rayon de floutage r ?

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 $[r, H-r-1] \times [r, L-r-1]$.

Exercice 6. Compléter et appliquer la fonction suivante à l'image (garder $r \leq 3$ sinon c'est très long).

```
def floutage(A,r):
    B = A.copy()
    for i in range(r, hauteur - r):
        for j in range(r, largeur - r):
            for k in range(3):
                B[i,j,k] = np.sum( A[ i-r:i+r+1, j-r:j+r+1, k ] ) / ...
```

5 Exercices d'approfondissement

Ci-dessous, lorsqu'on dit qu'une fonction prend en argument une « image », on entend par là un tableau numpy correspondant à une image. Idem pour le fait qu'elles retournent une image. On testera alors la fonction sur une image comme « joconde.bmp » et on affichera le résultat avec `plt.imshow`.

Exercice 7. Écrire une fonction `noirBlanc` qui prend en argument (un tableau numpy correspondant à) une image et retourne la même image avec seulement deux couleurs : noir et blanc. On regardera, pour chaque pixel, si la moyenne des valeurs des couleurs est plus proche du noir ou du blanc : si cette moyenne est plus proche du noir, le pixel deviendra noir. Sinon, il sera blanc.

Exercice 8. Écrire une fonction `echange` qui prend en argument une image et retourne la même image où les couleurs ont été échangées selon la séquence $R \to G \to B \to R$: autrement dit, la quantité de rouge de l'ancienne image devient la quantité de vert de la nouvelle image, etc.

Exercice 9. Écrire une fonction `cadre` qui prend en argument une image et un entier `p` et retourne une image de même taille mais avec un cadre noir d'épaisseur `p` pixels sur les bords (les anciens pixels à ces endroits sont donc devenus de couleur noire).

Exercice 10 (Une surprise vous attend !)**.** Écrire une fonction `retournement` qui prend en argument une image et retourne l'image après une rotation de 180°. Tester sur l'image « joconde_inversee.bmp ». *Indication : si un pixel se trouve à la position (i, j), où doit-il se trouver lorsqu'on retourne l'image ?*

Exercice 11 ($\star$)**.** Écrire une fonction `melange` qui prend en argument deux images T_1 et T_2 ainsi qu'un ratio $r \in [0,1]$ et retournera une image T_3 selon les critères suivants :

- Si on note $L_1 \times H_1$ et $L_2 \times H_2$ les tailles respectives de T_1 et T_2, l'image T_3 aura pour taille $\max(L_1, L_2) \times \max(H_1, H_2)$.
- Étant donné une image T, on note $c_{ijk}(T)$ la valeur de la couleur numéro k (0 : Rouge, 1 : Vert, 2 : Bleu) du pixel à la position (i, j). Les couleurs de l'image T_3 sont alors déterminées selon la relation

$$c_{ijk}(T_3) = r c_{ijk}(T_1) + (1-r) c_{ijk}(T_2)$$

  Cependant, si $c_{ijk}(T_1)$ et/ou $c_{ijk}(T_2)$ n'est pas définie car on sort des dimensions de T_1 et/ou T_2 alors on supposera que ces quantités non définies valent 0 : autrement dit on fera un fondu avec la couleur noir.

6 Pour s'amuser...

Un autre traitment d'image : la 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. L'image restera de la même taille, mais les points proches de

(x_0, y_0) seront plus « gros » et ceux éloignés seront plus « petits ». L'idée est d'utiliser une fonction de déformation

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

Le pixel de coordonnées (x,y) de `B` sera identique à celui de coordonnées $f(x,y)$ de `A`. On ne rentrera pas dans les détails de la fonction f.

Exercice 12. Ouvrir le fichier TP7_dilatation.py, et le compiler. Vous pouvez changer (x_0, y_0) et/ou le grossissement (variable `gross`).