TP3 MP1 TRAITEMENT D'IMAGE

Nous réviserons dans ce TP les notions suivantes :
— chargement d'une image
— modification, enregistrement d'une image,
— tableaux numpy

1 Constitution d'une image

Une image numérique peut être représentée par une matrice dans laquelle chaque case représente un pixel ("picture element") d'une couleur uniforme que l'on peut coder sur un certain nombre de bits. Il existe plusieurs types de *formats*, à savoir différents types de repésenter numériquement la matrice (.png, .jpeg, etc.)

La valeur des pixels dépend de la nature de l'image :

— Pour une image en noir et blanc, chaque pixel ne peut prendre que 2 valeurs 0 ou 1, ce qui permet de l'encoder sur seulement 1 bit.

— Pour une image en niveaux de gris, chaque teinte de gris est codée sur un octet, ce qui permet de coder 256 niveaux de gris.

— Pour une image en couleurs, chaque pixel est codé sur 3 octets, chaque octet représentant une nuance de couleur. Dans le format RGB (Red, Green, Blue) le premier octet sert à donner la nuance de rouge, le deuxième la nuance de vert et le troisième la nuance de bleu. Cela produit une synthèse additive correspondant aux couleurs perçues par les 3 types de cellules correspondantes au fond de notre oeil.

— Il se peut que chaque pixel soit codé sur 4 octet, avec le quatrième octet celui qui donne la transparence de l'image.

Pour convertir une image en couleurs en niveaux de gris, un pixel doit être remplacé par un pixel à une seule valeur, appelé "luminance". Cette quantité traduit la sensation visuelle de la luminosité et dépend de manière inégale des 3 couleurs primaires RGB. La formule généralement recommandée pour calculer cette luminance est la suivante :

$$luminance = 0,2126 \times R + 0,7152 \times G + 0,0722 \times B$$

2 Gestion des images

Nous importerons pour travailler sur les images les modules suivants :

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import imageio as io
```

Les fonctions utiles pour la gestion des fichiers sont :

— mpimg.imread("image.png") ou io.imread("image.png") : elle permet de convertir une image en tableau de pixel.

— np.array(tableau de pixels) : elle permet de convertir le tableau obtenu avec la fonction imread en tableau numpy, ce qui permet notamment d'accélérer les calculs demandés. Les éléments de ce tableau sont de type np.uint8, c'est à dire entiers non signés codés sur 8 bits.

— io.imwrite("nom de l'image", tableau numpy) : elle permet de créer un fichier image à partir d'un tableau de pixels ou un tableau numpy.

— plt.imshow(tableau) : elle permet d'afficher l'image correspondant à un tableau de pixels.

— image.shape : cette méthode permet de renvoyer un triplet de valeurs (nombre de lignes , nombre de colonnes , nombre d'octets sur lesquels est codé chaque pixel)

— np.zeros((n,p,s),dtype=np.uint8) : elle permet de créer une matrice de 0 de dimension (n,p) avec chaque pixel codé sur s bits.

— np.zeros_like(tableau) : elle permet de créer un tableau de zéros de même dimension que son tableau argument

3 Fonctions élémentaires

1. Rédiger une fonction **drapeau** permettant de dessiner le drapeau français, de dimensions 600x900 pixels.
2. Rédiger une fonction **symétrie** qui prend en argument une image et retourne une nouvelle image, obtenue par symétrie autour d'un axe vertical passant par le milieu de l'image.
3. Rédiger une fonction **rotation** qui prend en argument une image et retourne une nouvelle image, obtenue par rotation d'un angle $\frac{\pi}{2}$ autour du centre de l'image.
4. Rédiger une fonction **negatif** qui prend en argument une image et retourne son négatif, c'est-à-dire l'image dans laquelle chaque composante de couleur de chaque pixel est remplacée par sa valeur complémentaire dans l'intervalle $[0, 255]$.
5. Rédiger une fonction **niveaudegris** qui prend an argument une image couleur et retourne une nouvelle image convertie en niveau de gris.

4 Traitements avancés

La pluspart des filtres de traitement d'images utilisent une convolution par un masque pour réaliser une modification. Ces masques sont des matrices de petites tailles, en général 3×3 (taille que nous allons considérer par la suite) ou 5×5 :

$$\mathrm{C} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

Le filtre associé à ce masque transforme la matrice $M = (m_{ij})$, associée à une certaine image, en la matrice $M \otimes C = m*_{ij}$ par une opération appelée *produit de convolution* et définie par la relation :

$$\forall i, j : m*_{ij} = c_{11}m_{i-1,j-1} + c_{12}m_{i-1,j} + c_{13}m_{i-1,j+1}$$

$$+c_{21}m_{i,j-1} + c_{22}m_{i,j} + c_{23}m_{i,j+1} + c_{31}m_{i+1,j-1} + c_{32}m_{i+1,j} + c_{33}m_{i+1,j+1}$$

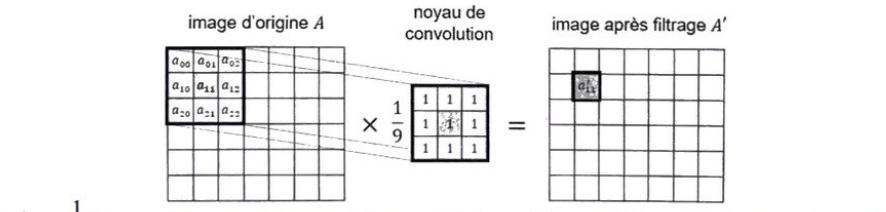
Dans le cas d'une image en couleur, on peut choisir d'appliquer le même masque à chacune des trois composantes RGB de l'image ou leur appliquer des masques différenciés, en fonction de l'effet désiré. Dans la suite de ce sujet, et dans un but simplificateur, nous appliquerons le même masque à chacune des trois composantes de couleur.

Lorsque le pixel initial est sur un bord, une partie de la convolution porte en dehors des limites de l'image. Là encore, nous conviendrons pour simplifier de laisser inchangés ces pixels.

Par exemple, le lissage d'une image vise à moyenner les valeurs des 9 éléments de chaque sous-matrice 3×3, ce qui correspond à la matrice de convolutuion suivante :

$$C_1 = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

selon le schéma suivant :



$$a'_{11} = \frac{1}{9}(\underbrace{1 \times a_{00} + 1 \times a_{01} + 1 \times a_{02}}_{1^e \text{ ligne du noyau}} + \underbrace{1 \times a_{10} + 1 \times a_{11} + 1 \times a_{12}}_{2^e \text{ ligne}} + \underbrace{1 \times a_{20} + 1 \times a_{21} + 1 \times a_{22}}_{3^e \text{ ligne}})$$

D'autres masques donnés ci-dessous C_2 et C_3 correspondent à une augmentation de constraste ou un repoussage (effet de relief) :

$$C_1 = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad C_2 = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} \qquad C_3 = \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

1. Rédiger une fonction **convolution** qui prend en arguments deux matrices M (associée à une image) et C (associée à la transformation souhaitée) et retourne la matrice $M \otimes C$

   Les éléments de la matrice C seront a priori de type float tandis que ceux des matrices M et $M \otimes C$ seront de type np.uint8. Il conviendra donc, lorsque $m_{ij} < 0$ ou $m_{ij} > 255$ de remplacer les valeurs calculées par respectivement 0 et 255.

   Ne pas oublier non plus que les pixels d'une image en couleur possèdent trois composantes et que chacune de ces composantes doit être traitée.
2. Rédiger trois fonctions **lissage**, **contraste** et **repoussage** qui prennent toutes trois en argument une image et retournent une nouvelle image obtenue en appliquant respectivement un effet de lissage, d'augmentation de contraste et de repoussage, et observer le résultat sur l'image test.

5 Détection de contours

Afin de détecter les contours d'une image, on va s'intéresser à l'importance des variations des valeurs des pixels de proche en proche. Le meilleur outil pour cela est l'opérateur gradient dont on pourra calculer la norme :

$$\left(\frac{\partial f}{\partial x}(x,y)\right)^2 + \left(\frac{\partial f}{\partial y}(x,y)\right)^2$$

On pourra pour cela calculer la grandeur discrétisée de ce gradient en remplaçant chaque pixel m_{ij} de la matrice M par la grandeur :

$$(m_{i+1,j} - m_{i-1,j})^2 + (m_{i,j+1} - m_{i,j-1})^2$$

ce qui correspondrait à appliquer séparément les masques

$$\begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \text{ et } \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

puis de calculer leur norme.

Cependant, l'expérience montre que cette formule reste un peu trop dépendante au bruit d'une image, et qu'on obtient de meilleurs résultats en moyennant autour du point, c'est-à-dire en utilisant les masques

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \quad \text{(filtre de Sobel)}$$

1. Rédiger une fonction **gradient** prenant en argument une image et retournant la matrice des gradients.
2. Rédiger une fonction **contour** prenant en argument une image et retournant une image en noir et blanc représentant les contours de l'image.

6 Stéganographie

La stéganographie est l'art de la dissimulation. Nous allons dans cette dernière partie étudier un procédé permettant de cacher une image au sein d'une autre image.

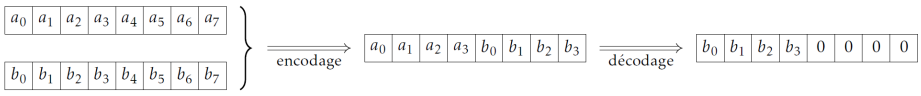
Dans une image, chaque composante de couleur de chaque pixel est représentée par un entier codé sur huit bits :

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7

Les quatre premiers bits, dits de poids forts, ont plus d'importance (plus de poids) que les quatre bits suivants, dits de poids faible, et l'expérience montre qu'on ne change guère l'image en ne gardant que les quatre bits de poids forts :

a_0	a_1	a_2	a_3	0	0	0	0

Dès lors, les quatre bits de poids faibles ainsi libérés vont pouvoir nous servir à cacher les quatre bits de poids forts d'une seconde image :



Afin d'effectuer ces manipulations, on pourra utiliser les commandes x»n ou x«n permettant de décaler les bits du nombre x de n rangs.

1. Afficher les images modifiées d'une première image après avoir retiré successivement les bits de poids faible. A partir de combien de bits retirés obtient-on une nette dégradation de l'image ?
2. Choisissez une image à dissimuler dans une autre et utiliser la démarche présentée ci-dessus pour pouvoir la dissimuler.
3. Tentez de trouver qui peut se cacher derrière le fichier malherbesurprise.