TRAVAUX PRATIQUES n° 10
Traitement d'images

I. Formulaire

1) Tableaux

`import numpy as np`	Importe le module `numpy` sous l'alias `np`
`np.array(liste)`	Convertit une liste en un tableau de `numpy` (type `array`)
`np.zeros((n_1,n_2,...,n_p))`	Création d'un tableau de zéros de dimension `p` et de taille `(n_1,n_2,...,n_p)`
`np.ones((n_1,n_2,...,n_p))`	Création d'un tableau de uns de dimension `p` et de taille `(n_1,n_2,...,n_p)`
`np.shape(T)`	Taille du tableau `T`

2) Tableaux-images de dimension 3

`image[i,j]`	Triplet RGB du pixel ligne `i` et colonne `j`
`image[i,j,k]`	Valeur de la couleur `k` (`k = 0, 1, 2`) du pixel ligne `i` et colonne `j`
`image[i]`	Tableau de tous les pixels de la ligne `i`
`image[:,j]`	Tableau de tous les pixels de la colonne `j`
`n, p, _ = np.shape(image)`	Permet de récupérer le nombre de lignes `n` et le nombre de colonnes `p`

3) Lire et enregistrer une image

`import matplotlib.image as img`	Importe le module `image` sous l'alias `img`
`image = img.imread(nom_fichier)`	Création du tableau-image à partir du fichier `nom_fichier`
`img.imsave(nom_fichier, image)`	Enregistre le tableau-image dans un fichier sous le nom `nom_fichier`

4) Afficher une image

`import matplotlib.pyplot as plt`	Importe le module `pyplot` sous l'alias `plt`
`plt.imshow(image)`	Transformation du tableau en image.
`plt.show()`	Affichage de l'image

II. Informations complémentaires pour travailler sur une image

1) Couleurs et code RGB

Voici une liste de quelques couleurs et de leur code RGB pour un fichier `".png"` :

Code	Couleur du pixel	Code	Couleur du pixel
$[0.0, 0.0, 0.0]$	Noir	$[1.0, 1.0, 0.0]$	Jaune
$[1.0, 1.0, 1.0]$	Blanc	$[1.0, 0.0, 1.0]$	Magenta
$[x, x, x]$	Nuance de gris	$[0.0, 1.0, 1.0]$	Cyan
$[1.0, 0.0, 0.0]$	Rouge	$[1.0, 0.5, 0.0]$	Orange
$[0.0, 1.0, 0.0]$	Vert	$[1.0, 0.0, 0.5]$	Rose
$[0.0, 0.0, 1.0]$	Bleu	$[0.35, 0.2, 0.0]$	Marron

2) Afficher un tableau-image

Pour afficher un tableau-image depuis Python, il faut utiliser les deux instructions suivantes :

```
plt.imshow(image)
plt.show()
```

3) Modifier un pixel

Pour modifier la couleur d'un pixel, il suffit de changer son code RGB à l'aide de la syntaxe suivante :

```
image[i,j,0] = nouvelle_quantité_de_rouge
image[i,j,1] = nouvelle_quantité_de_vert
image[i,j,2] = nouvelle_quantité_de_bleu
```

On peut aussi modifier les trois couleurs en même temps :

```
image[i,j] = [quantité_de_rouge, quantité_de_vert, quantité_de_bleu]
```

Pour modifier tous les pixels, il faut donc faire deux boucles imbriquées (une pour les lignes et une pour les colonnes) et effectuer les modifications pixel par pixel :

```
n, p, _ = np.shape(image)
for i in range(n):
    for j in range(p):
        image[i,j] = [quantité_de_rouge, quantité_de_vert, quantité_de_bleu]
```

4) Exemples d'image

Placez les fichiers `".png"` du TP **dans le même dossier** que votre script Python. On utilisera par exemple `"monarch.png"`.

On dispose en début de fichier des commandes suivantes :

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as img
from os import chdir
chdir(r"C:\Users\Prenom\Documents\TP10")

def affiche(image):
    """Fonction qui permet d'afficher l'image associée au tableau image."""
    plt.imshow(image)
    plt.show()
```

Adaptez le chemin dans `chdir` comme au TP 9 (chemin complet du dossier, précédé de `r`).

III. Traitement d'images pixel par pixel

1) Saturation

Pour saturer une image suivant une couleur, par exemple le rouge, on doit transformer les quantités de rouge de tous les pixels en 1.0.

Écrire la fonction suivante et la tester en essayant de comprendre ce qu'elle fait et pourquoi elle le fait :

```
def saturation(image, couleur):
    n, p, _ = np.shape(image)

    for i in range(n):
        for j in range(p):
            if couleur == "R":
                image[i,j,0] = 1.0
            elif couleur == "V":
                image[i,j,1] = 1.0
            elif couleur == "B":
                image[i,j,2] = 1.0

    return image
```

La fonction `saturation` **modifie** le tableau-image (effet de bord) et renvoie le résultat, mais son travail n'est pas d'afficher l'image : toutes les fonctions que vous allez créer devront respecter ce principe. Elles ne doivent pas utiliser la fonction `affiche`.
Pour tester plusieurs traitements sans écraser l'image d'origine, travaillez sur une **copie** :

```
image = img.imread("monarch.png")
image_sat_rouge = saturation(image.copy(), "R")
affiche(image_sat_rouge)
```

On peut aussi recharger l'image avec `img.imread` avant chaque nouveau test.

Exercice 1 :

1) Que fait l'appel `saturation(image, "R")` sur chaque pixel ?
2) Que devient le pixel de code RGB `[0.2, 0.5, 0.8]` après `saturation(..., "V")` ?
3) Tester la fonction sur `"monarch.png"` avec `"R"`, puis `"V"`, puis `"B"` (en pensant à utiliser `image.copy()` ou à recharger l'image).

2) Coloration

Exercice 2 : Écrire une fonction `coloration(image, couleur)` qui prend en arguments un tableau-image et une couleur (`"R"`, `"V"` ou `"B"`) et qui renvoie l'image ayant uniquement la composante couleur demandée.

Par exemple, si `couleur = "R"`, alors le pixel de code RGB `[0.4, 0.3, 0.8]` sera transformé en pixel de code RGB `[0.4, 0.0, 0.0]`.

3) Négatif

Exercice 3 : Écrire une fonction `negatif(image)` qui transforme l'image en négatif, c'est-à-dire que pour chaque pixel et pour chaque couleur, on applique la transformation $x \mapsto 1 - x$.

Par exemple, si le code RGB d'un pixel est `[0.4, 0.3, 0.8]`, le code RGB du même pixel de l'image en négatif sera alors `[0.6, 0.7, 0.2]`.

4) Niveaux de gris

Les pixels gris sont les pixels ayant la même quantité de rouge, vert et bleu. Pour transformer une image en niveaux de gris, il faut donc déterminer, pour chaque pixel, une valeur commune de quantité de chaque couleurs : la luminosité du pixel.

Exercice 4 (*Luminosité*) : Écrire une fonction `luminosite(pixel)` qui prend en argument une liste de 3 nombres et qui renvoie leur moyenne.

```
>>> luminosite([0.4, 0.3, 0.8])
0.5
```

Exercice 5 (*Niveaux de gris*) : Écrire une fonction `niveaux_de_gris(image)` transformant une image en niveaux de gris à l'aide de la fonction `luminosite` précédente.

Par exemple, le pixel `[0.4, 0.3, 0.8]` sera transformé en `[0.5, 0.5, 0.5]` car sa luminosité est de `0.5`.

Exercice 6 (*Noir et blanc*) : Écrire une fonction `noir_et_blanc(image, seuil = 0.5)` transformant une image en noir et blanc en transformant les pixels de luminosité inférieure à `seuil` en noir et les pixels de luminosité supérieure à `seuil` en blanc (pour le cas d'égalité, on choisira l'un ou l'autre).

Par exemple, pour un seuil de `0.4`, le pixel `[0.4, 0.3, 0.8]` sera transformé en `[1.0, 1.0, 1.0]` car sa luminosité est `0.5 > 0.4`.

IV. Traitement d'images par déplacement de pixels

1) Image miroir

Exercice 7 : Compléter la fonction `miroir(image)` permettant de créer une image miroir de la première par symétrie par rapport à un axe vertical centré sur l'image initiale.

```
def miroir(image):
    n, p, _ = np.shape(image)
    image_miroir = np.zeros(...)

    for i in range(n):
        for j in range(p):
            image_miroir[i, j] = ...

    return image_miroir
```

2) Rotation

Exercice 8 : En s'inspirant de la fonction `miroir`, écrire une fonction `rotation(image)` qui renvoie l'image tournée d'un quart de tour dans le sens des aiguilles d'une montre.

Indication : si l'image initiale a `n` *lignes et* `p` *colonnes, l'image tournée a* `p` *lignes et* `n` *colonnes. On créera donc un tableau* `np.zeros((p, n, 3))`.

V. Traitement d'images par convolution

On appelle filtre de convolution, une matrice de taille 3×3.

Considérons par exemple le filtre de convolution suivant :

$$M = \begin{pmatrix} 1 & -3 & 0 \\ 1 & 1 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

Pour le tableau-image suivant (on n'écrit que la composante de rouge pour simplifier) :

0.5	0.1	0.2	0.9	0.9
0.0	0.5	0.3	0.1	1.0
0.4	0.3	0.2	0.5	0.6
0.6	0.1	0.8	0.9	0.6
0.5	0.1	0.2	0.1	0.7

on va, pour chaque pixel (excepté ceux du bord), regarder les pixels voisins

0.5	0.1	0.2	0.9	0.9
0.0	0.5	0.3	0.1	1.0
0.4	0.3	0.2	0.5	0.6
0.6	0.1	0.8	0.9	0.6
0.5	0.1	0.2	0.1	0.7

et effectuer le calcul :

$$1 \times 0.5 - 3 \times 0.1 + 0 \times 0.2 + 1 \times 0.0 + 1 \times 0.5 + 1 \times 0.3 + 0 \times 0.4 - 1 \times 0.3 + 0 \times 0.2 = 0.7$$

Le pixel 0.5 sera alors remplacé par 0.7 dans la nouvelle image. On procède de même pour tous les autres pixels (excepté ceux du bord) et toutes les autres couleurs.

Dans le cas où le résultat du calcul est inférieur à 0.0, on mettra la valeur 0.0 et dans le cas où le résultat du calcul est supérieur à 1.0, on mettra la valeur 1.0.

Ainsi, pour l'exemple précédent, on obtiendra le tableau suivant :

0.5	0.1	0.2	0.9	0.9
0.0	0.7	0.2	0.0	1.0
0.4	0.0	0.0	0.4	0.6
0.6	0.9	1.0	0.9	0.6
0.5	0.1	0.2	0.1	0.7

Exercice 9 : Compléter la fonction suivante programmant un filtre de convolution :

```
def convolution(image, filtre):
    n, p, _ = np.shape(image)
    image_filtre = np.zeros((n, p, 3))
    for i in range(..., ...):
        for j in range(..., ...):
            for k in range(3):
                v = np.sum(image[...:..., ...:..., k] * filtre)
                v = min(v, ...)
                v = max(v, ...)
                image_filtre[i, j, k] = v
    return image_filtre
```

Exercice 10 (*Floutage*) : Écrire une fonction `floutage(image)` permettant de flouter une image grâce au filtre de convolution :

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

Exercice 11 (*Détection de contours*) : Écrire une fonction `contours(image)` permettant de renforcer les contours d'une image grâce au filtre de convolution :

$$M = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

VI. Pour aller plus loin : création d'une image, simulations

Exercice 12 : Essayer de deviner à l'avance ce que fait ce programme et le tester ensuite pour $n = 210$ et $p = 420$:

```
def mystere(n,p):
    image = np.zeros((n,p,3))
    rouge = [1,0,0]
    blanc = [1,1,1]
    bleu = [0,0,1]

    for i in range(n):
        for j in range(p):
            if j < p//3:
                image[i,j] = bleu
            elif j < 2*p//3:
                image[i,j] = blanc
            else:
                image[i,j] = rouge

    plt.imshow(image)
    plt.show()
```

Exercice 13 : Un marcheur se déplace aléatoirement sur un quadrillage de la façon suivante : pour chaque nouveau pas, le marcheur lance un dé à 4 faces et, selon le numéro obtenu, se déplace d'un cran vers la gauche, vers la droite, vers le haut ou vers le bas.

Créer un programme prenant en arguments trois entiers n, p et N et simulant le parcours de N pas du marcheur sur une grille $n \times p$.

On créera une image blanche de taille $n \times p$, puis on dessinera un pixel noir sur chaque nouveau pas du marcheur.

Exercice 14 : Étant donnée une image, il peut être intéressant de compter le nombre de pixels ayant une couleur donnée. Les pixels pouvant ne pas avoir exactement la même couleur (au niveau du code RGB), on peut utiliser la distance euclidienne pour déterminer si deux pixels ont des couleurs plus ou

moins proches c'est-à-dire qu'en considérant un pixel comme un vecteur de l'espace, on dira que deux pixels $\overrightarrow{u}(R, G, B)$ et $\overrightarrow{v}(R', G', B')$ ont à peu près la même couleur lorsque :

$$\|\overrightarrow{u} - \overrightarrow{v}\| = \sqrt{(R - R')^2 + (G - G')^2 + (B - B')^2} < \varepsilon$$

où $\varepsilon > 0$ est une marge d'erreur fixée.

1) Écrire une fonction `distance(pixel1, pixel2)` qui, étant donnés deux pixels (tableaux ou liste de trois éléments) renvoie la distance entre ces deux pixels.
2) Écrire une fonction `meme_couleur(pixel1, pixel2, epsilon)` qui renvoie `True` si cette distance est inférieure à `epsilon` et `False` sinon.
3) Écrire une fonction `nb_pixels(image, couleur, epsilon)` qui renvoie le nombre de pixels du tableau `image` dont la couleur est proche à `epsilon` près de celle du pixel `couleur`.

   Par exemple, pour compter le nombre de pixels jaunes dans une image, on pourra appeler la fonction `nb_pixels` de la manière suivante : `nb_pixels(image, [1.0, 1.0, 0.0], 0.1)`.