Travaux pratiques

informatique

Traitement d'image

Une image numérique peut être représentée par une matrice dans laquelle chaque case représente un pixel (pour *picture element*), unité minimale adressable par le contrôleur vidéo et supposé de couleur uniforme. Le *format* de l'image (png, jpeg, etc) désigne la manière dont cette matrice est représentée en mémoire.

- Pour une image en noir et blanc, chaque pixel ne peut prendre que deux valeurs, ce qui permet de l'encoder sur un seul bit ;
- pour une image en niveau de gris, chaque teinte de gris (en passant du noir au blanc) est encodée sur un octet, ce qui permet de représenter $2^8 = 256$ nuances de gris ;
- pour une image en couleur, chaque pixel est encodé sur trois octets, un octet étant associé à une nuance de couleur primaire. Par exemple, dans le cas du format RGB (*Red*, *Green*, *Blue*), le premier octet dose la quantité de rouge, le second la quantité de vert et le troisième la quantité de bleu pour une synthèse additive ;
- enfin, les pixels d'une image en couleur peuvent être encodés par quatre octets au format RGBA (A pour *alpha*), le quatrième octet codant la transparence du pixel.

1. Représentation d'une image en Python

Durant ce TP nous aurons besoin de trois modules : `numpy` pour la manipulation des matrices, `mathplotlib.pyplot` pour la visualisation des images et enfin `imageio` pour la conversion image/matrice. Commençons donc par les importer :

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

– La fonction `io.imread`(*source*) permet de convertir une image au format courant (png, jpeg, etc) en un tableau `numpy`. Elle prend en argument une chaîne de caractère décrivant un fichier présent sur votre ordinateur ou l'adresse http d'une image présente sur le net.

– La fonction `plt.imshow`(*tab*) permet de visualiser l'image décrite par un tableau.

Le script qui suit doit vous permettre de récupérer sur le net l'image qui nous servira de modèle et de la visualiser.

```
img = io.imread('http://pc-etoile.schola.fr/wp-content/uploads/images/picasso.png')
img = np.array(img)
plt.imshow(img)
```

Ainsi défini `img` est tableau de type `numpy.array`. Observons ses dimensions :

```
In [1]: img.shape
Out[1]: (256, 256, 3)
```

Les deux premiers nombres nous indiquent les dimensions de l'image : 256×256, le troisième la « profondeur » d'un pixel. Ici cette profondeur est égale à 3, ce qui indique qu'un pixel est codé par un triplet RGB.
Observons maintenant la représentation d'un pixel particulier, situé à peut près au centre de l'image :

```
In [2]: img[128, 128]
Out[2]: array([237, 234, 217], dtype=uint8)
```

Le type `uint8` est celui des entiers non signés représentés sur 8 bits (soit un octet), autrement dit un entier positif compris entre 0 et 255. Le pixel en question est donc composé de $\frac{237}{255} \approx 93\%$ de rouge, $\frac{234}{255} \approx 92\%$ de vert et de $\frac{217}{255} \approx 85\%$ de bleu en synthèse additive.

Question 1.

a) Créer puis visualiser une image `img_rouge`, copie de l'image `img` mais dans laquelle les composantes verte et bleue ont été supprimées (ou plus exactement mises à 0). Créer de même les images `img_vert` et `img_bleu`. Attention à ne pas modifier l'image initiale ! (Lire à ce sujet l'encadré qui suit.)

> Une image, comme tout tableau, est un objet *mutable* : il est possible de modifier une de ses composantes sans avoir à recréer le tableau dans son entier. Une conséquence fâcheuse de cette fonctionnalité (et une erreur fréquente) est que l'instruction `im = img` *ne crée pas une copie* de l'image `img`. Pour créer les images qui vous seront demandées vous avez plusieurs possibilités :
>
> – commencer par créer une copie de `img` en écrivant : `im = img.copy()` puis la modifier ;
>
> – créer une image vide de mêmes dimensions avec : `im = np.zeros_like(img)` puis la remplir ;
>
> – créer de toute pièce une image vide en précisant ses dimensions et le type de ses éléments avec : `im = np.zeros((n, p, q), dtype=np.uint8)` puis la remplir.

b) Visualiser les images obtenues en ne conservant que deux des trois composantes de couleur, c'est-à-dire les images `img_vert + img_bleu`, `img_bleu + img_rouge` et `img_rouge + img_vert`.

c) Visualiser enfin L'image `img // 2` dans laquelle chacune des trois composantes de couleur a été atténuée de 50%.

Transformations élémentaires

Nous allons maintenant écrire quelques fonctions de transformation élémentaire d'une image. À chaque fois on vérifiera leur bon fonctionnement en les appliquant à l'image test `img`.

Question 2. Négatif d'une image.

Rédiger une fonction `negatif(im)` qui prend pour argument une image et renvoie son négatif, c'est-à-dire l'image dans laquelle chaque composante de couleur de chaque pixel est remplacée par la valeur complémentaire dans l'intervalle $[\![0, 255]\!]$. On prendra garde à ne pas modifier l'image passée en argument.

Question 3. Symétrie verticale.

Rédiger une fonction `symetrie(im)` qui prend pour argument une image et renvoie une nouvelle image obtenue par symétrie autour d'un axe vertical passant par le milieu de l'image.

Question 4. Rotation.

Rédiger une fonction `rotation(im)` qui prend pour argument une image et renvoie une nouvelle image, obtenue par rotation d'un angle $\pi/2$ autour du centre de l'image.

Conversion en niveau de gris

Pour convertir une image couleur en niveau de gris, un pixel représenté par ses composantes (r, g, b) doit être remplacé par un pixel à une seule composante $y \in [\![0, 255]\!]$ appelée *luminance*. Cette quantité, qui traduit la sensation visuelle de luminosité, dépend de manière inégale des trois composantes RGB. La formule communément recommandée pour cette conversion est la suivante :

$$y = 0,2126\, r + 0,7152\, g + 0,0722\, b \qquad (y \text{ étant arrondi à l'entier le plus proche})$$

Ces valeurs, déterminées empiriquement, sont liées au fait que pour un œil humain le vert parait plus lumineux que le rouge, lui-même plus lumineux que le bleu.

Question 5. Rédiger une fonction `niveaudegris(im)` qui prend an argument une image couleur et renvoie la matrice de luminance de l'image. Pour visualiser correctement l'image produite, lire l'encadré suivant.

> Lorsque les pixels d'une image n'ont qu'une composante, la fonction `plt.imshow` utilise une échelle de couleur pour la représenter, utilisant par défaut le spectre de l'arc-en-ciel (voir figure 1). Pour visualiser l'image en niveau de gris il faut utiliser une échelle qui va du noir au blanc, ce que réalise l'instruction `plt.imshow(im, cmap='gray')`.

Figure 1 – L'échelle de couleur utilisée par défaut, et l'échelle des niveaux de gris.

2. Traitement d'image

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 :

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

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.

Remarque. 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.

Question 6. Rédiger une fonction `convolution(m, c` qui prend en arguments deux matrices M (associée à une image) et C et renvoie la matrice $M \otimes C$ (ne pas oublier que les pixels d'une image en couleur possèdent trois composantes et que chacune de ces composantes doit être traitée).

Attention. Les éléments de $M \otimes C$ doivent être de type `np.uint8`, c'est-à-dire des entiers de l'intervalle $[\![0, 255]\!]$. Or suivant les valeurs de C il se peut que les valeurs de m^*_{ij} ne soient pas comprises dans cet intervalle. Il conviendra donc, lorsque $m^*_{ij} < 0$ ou $m^*_{ij} > 255$, de remplacer ces valeurs calculées par respectivement 0 et 255 et, **seulement après**, introduire la valeur modifiée dans la nouvelle matrice.

Exemples de masques

Lissage Pour obtenir un effet de lissage on utilise le masque C_1 : chaque pixel est remplacé par la moyenne de lui-même et de ses huit voisins ; pour cette raison on parle de filtre *moyenneur*.

Augmentation du contraste Au contraire, pour augmenter le contraste on utilise le masque C_2.

Repoussage Enfin, le masque C_3 donne un effet de relief à l'image, appelé *repoussage*.

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

Figure 2 – Trois filtres classiques en traitement d'images.

Question 7. Rédiger trois fonctions `lissage`, `contraste` et `repoussage` qui prennent toutes trois en argument une image et renvoient une nouvelle image obtenue en appliquant respectivement un effet de lissage, d'augmentation de contraste et de repoussage. Observer le résultat de ces trois filtres sur l'image test.

3. Stéganographie d'une image

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 :

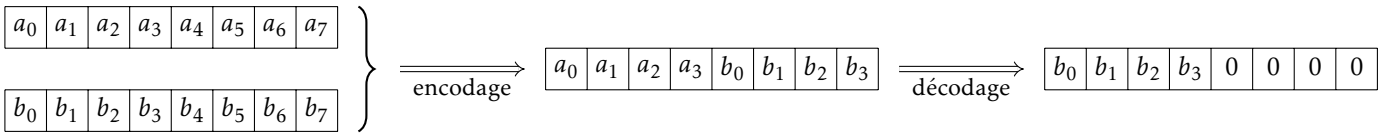

FIGURE 3 – Camouflage d'une image b au sein d'une image a.

La perte de qualité qui résulte de la perte des bits de poids faible pour chacune des deux images (l'image cachée b et l'image masquante a) reste quasiment indécelable à l'œil.

Question 8. Encodage.

a) Rédiger une fonction `encodage(a, b)` qui prend pour arguments deux images de mêmes tailles a et b et qui renvoie une image dans laquelle b a été camouflée au sein de a suivant le procédé décrit ci-dessus. On pourra s'aider de l'encadré ci-dessous.

Décalage de bits. Les opérateurs binaires `>>` et `<<` permettent de réaliser facilement les opérations de décalage des bits d'un entier codé sur un nombre fixe de bits : l'opération `n >> 4` décale de 4 bits vers la droite les bits de n ; l'opération `n << 4` décale de 4 bits vers la gauche les bits de n.

`n :`	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
`n >> 4 :`	0	0	0	0	a_0	a_1	a_2	a_3
`n << 4 :`	a_4	a_5	a_6	a_7	0	0	0	0

b) Utiliser cette fonction pour cacher l'image

```
img2 = io.imread('http://pc-etoile.schola.fr/wp-content/uploads/images/matisse.png')
```

au sein de l'image test.

Question 9. Décodage.

a) Rédiger enfin une fonction `decodage(m)` qui prend en argument une image qui a été encodée suivant le procédé ci-dessus et qui renvoie l'image cachée au sein de celle-ci.

b) Vérifier que le décodage de l'image que vous avez encodé à la question précédente donne bien l'image que vous y aviez caché.