# TP 8 : Matrices de pixels et images

Dans ce TP, on expérimente l'acquisition et la manipulation d'images : modification de couleurs, symétries, rotations, mise en niveau de gris, contraste, etc. En fin de TP, on expérimente une méthode de stéganographie, c'est-à-dire de dissimulation d'un message à l'intérieur d'une image.

Nous utiliserons les librairies `numpy` et `matplotlib` dont on rappelle les imports standards :

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

## I. Codage des images, import, affichage

### I.1. Codage

Une image est constituée d'un ensemble de points appelés **pixels** (pixel est une abréviation de PICture ELement). Le pixel représente ainsi le plus petit élément constitutif d'une image numérique. L'ensemble de ces pixels est contenu dans un tableau à deux dimensions constituant l'image.
Une image est donc représentée par un tableau à deux dimensions dont chaque case est un pixel. Pour représenter informatiquement une image, il suffit donc de créer un tableau de pixels dont chaque case contient une valeur. La valeur stockée dans une case est codée sur un certain nombre de bits déterminant la couleur ou l'intensité du pixel, on l'appelle **profondeur de codage** (parfois profondeur de couleur).
**Python** utilise le standard « Couleurs vraies » (True colors) ou « couleurs réelles » : cette représentation permet de représenter une image en définissant chacune des composantes (RGB, pour rouge, vert et bleu). Chaque pixel est représenté par une liste de trois entiers compris entre 0 et 255. Toutes les nuances entières sont possibles entre 0 et 255, créant ainsi une palette de plus de 16 milliions de couleurs. Il est possible d'ajouter une quatrième composante permettant d'ajouter une information de transparence ou de texture.

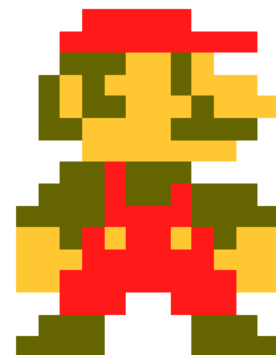
- ► Copier-coller le script suivant.

```
b = [255, 255, 255] # blanc
r = [255, 25, 25] # rouge
j = [255, 200, 50] # jaune
k = [100, 100, 0] # kaki

M = [[b, b, b, r, r, r, r, r, b, b, b, b],
[b, b, r, r, r, r, r, r, r, r, r, b],
[b, b, k, k, k, j, j, k, j, b, b, b],
[b, k, j, k, j, j, j, k, j, j, j, b],
[b, k, j, k, k, j, j, j, k, j, j, j],
[b, k, k, j, j, j, j, k, k, k, k, b],
[b, b, b, j, j, j, j, j, j, j, b, b],
[b, b, k, k, r, k, k, k, b, b, b, b],
[b, k, k, k, r, k, k, r, k, k, k, b],
[k, k, k, k, r, r, r, r, k, k, k, k],
[j, j, k, r, j, r, r, j, r, k, j, j],
[j, j, j, r, r, r, r, r, r, j, j, j],
[j, j, r, r, r, r, r, r, r, r, j, j],
[b, b, r, r, r, b, b, r, r, r, b, b],
[b, k, k, k, b, b, b, b, k, k, k, b],
[k, k, k, k, b, b, b, b, k, k, k, k]]

plt.imshow(M, interpolation = ‘nearest‘)
plt.axis(‘off‘)
plt.show()
```

Allons sauver la princesse !



Ici, nous avons utilisé une liste de listes (de listes) pour représenter un tableau à deux dimensions. On peut également utiliser à profit les tableaux (`array`) du nodule `numpy`. C'est d'ailleurs ce que fait **Python** par défaut. Cela n'est cependant pas au programme.

## I.2. Import

Dans ce TP nous manipulerons une charmante prise de vue de l'Aiguille Creuse, célèbre falaise d'Étretat (immortalisée dans un roman de la série des *Arsène Lupin* par Maurice Leblanc).

- Télécharger cette image au lien suivant :

  http://rduroux.ovh/PCSI/ITC/CH8/Etretat.jpg

- Créer un dossier TP8.
- Enregistrer l'image d'Etretat **dans le dossier TP8**.
- Créer un nouveau fichier .py que vous enregistrerez **dans le dossier TP8**.
- L'import de l'image en **Python** s'effectue grâce à l'instruction suivante.

```
etretat = plt.imread('Etretat.jpg')
```

  La matrice `etretat` contient alors la valeur RGB de chaque pixel de l'image initiale.

Une fois la ligne de code ci-dessus exécutée, l'objet `etretat` est une matrice (type `numpy.ndarray`) telle que `etretat[i,j]` contient un pixel, c'est-à-dire un objet de type `array` à 3 composantes `[R,G,B]` où `R` (resp. `G`, `B`) est un entier compris entre 0 et 255 représentant le niveau de rouge (resp. de vert de bleu).
Par exemple :

× la donnée `etretat[12, 151, 2]` renvoie le niveau de bleu du pixel situé à la 13$^{\text{ème}}$ ligne et à la 152$^{\text{ème}}$ colonne.

× la donné `etretat[98, 135]` renvoie un triplet du type `[r, g, b]` représentant le pixel situé à la 99$^{\text{ème}}$ ligne et à la 136$^{\text{ème}}$ colonne.

Désormais, nous appellerons aussi « image » une telle matrice.

## I.3. Résolution, affichage

Après acquisition de l'image `Etretat.jpg`, la matrice obtenue en **Python** est de taille $512 \times 512$. On dit que l'image `Etretat.jpg` possède une **résolution** de $512 \times 512$ pixels. Il est possible de récupérer la taille d'un tableau `numpy`, sous forme de tuple, en utilisant la méthode `shape` :

```
n, p, q = etretat.shape
```

- Comme mentionné plus haut, pour certaines images, les pixels sont codés sur 4 composantes (les niveaux RGB, plus un facteur d'opacité). Pour ramener la matrice à des pixels sous forme RGB, en ignorant le facteur d'opacité, on pourra utiliser la ligne suivante :

```
etretat = etretat[:, :, :3]
# (extraction des 3 premières composantes de chaque pixel)
```

- Pour réaliser l'affichage de la matrice `etretat`, on pourra utiliser les méthodes suivantes, de la librairie `matplotlib.pyplot` :

```
plt.imshow(etretat)
plt.show()
```

  On n'oubliera pas de refermer la fenêtre graphique après coup (manuellement, ou par `plt.close()`).

# II. Premières modifications

## II.1. Filtres et négatif

Pour effacer d'un pixel les nuances de bleu et de vert, il suffit de remplacer le tableau `[R,G,B]` par `[R,0,0]`.

- ▶ Créer une fonction `rouge` prenant en argument une image `image` et renvoyant sa version avec effacement des nuances de bleu et de vert.
- ▶ Écrire des fonctions analogues `bleu` et `vert` effectuant les mêmes opérations pour, respectivement, le rouge et le vert, et le rouge et le bleu, toujours sur une image `image`.
- ▶ Tester les fonctions `rouge`, `bleu` et `vert` sur la variable `etretat`.

**Figure 1.** Effacement des nuances de couleurs.

On se propose maintenant de créer un *négatif* de l'image d'origine, c'est-à-dire une image dans laquelle les niveaux de rouge / vert / bleu sont inversés. Pour cela, chaque coordonnée de chaque pixel doit être remplacée par son complémentaire à 255. Par exemple :

- × un 255 est remplacé par un 0,
- × un 124 par un 131,
- × un 12 par 243.

- ▶ Créer une fonction `negatif` prenant en argument une image `image` et renvoyant sa version en négatif.
- ▶ Tester la fonction `negatif` sur la variable `etretat`.

L'effet sépia est obtenu en remplaçant chaque pixel `[R,G,B]` par le pixel `[R', G', B']` où :

$$\begin{cases} R' &= \min\left(1,\ \lfloor 0.393 \times R + 0.769 \times G + 0.189 \times B \rfloor\right) \\ G' &= \min\left(1,\ \lfloor 0.349 \times R + 0.686 \times G + 0.168 \times B \rfloor\right) \\ B' &= \min\left(1,\ \lfloor 0.272 \times R + 0.534 \times G + 0.131 \times B \rfloor\right) \end{cases}$$

- ▶ Écrire une fonction `sepia` qui prend en paramètre d'entrée une liste `L` de la forme `[R, G, B]` et renvoie la liste `[R', G', B']`.
- ▶ Écrire une fonction `filtre_sepia` qui prend en entrée une matrice de pixels `image` et renvoie la matrice correspondante après application du filtre sépia à chacun des pixels.
- ▶ Tester la fonction `filtre_sepia` sur la variable `etretat`.

**Figure 2.** Négatif et Filtre sépia

## II.2. Opérations géométriques

On se propose ici d'effectuer des symétries et des rotations de l'image étudiée. Les rotations se limiteront à des angles de 90, 180 et 270 degrés.
On commence par programmer la création d'une « image vide », une image de taille fixée en paramètre et comportant uniquement des pixels `[0,0,0]`. Le résultat est donc une image noire.

- Programmer une fonction qui fabrique une image noire d'une taille donnée. Programmer une fonction `image_noire` prenant en argument deux entiers `n` et `p` et renvoyant une image de taille $n \times p$ dont tous les pixels sont `[0,0,0]` (type int).
  On veillera bien à ce que le type `int` soit respecté. Par exemple, l'instruction `np.zeros((n,p,3))` fait ce travail, mais renvoie des pixels de type flottant, c'est-à-dire `[0.,0.,0.]`, mais ce n'est pas le résultat voulu.

  **Remarque**
  On pourra fabriquer une telle image à partir d'une *liste de listes de listes*. Par exemple, la commande suivante crée une matrice de taille $2 \times 2$ dont chaque élément est une liste de longueur 3 contenant des entiers.

```
1  A = np.array([ [ [1,2,3],[4,5,6] ] , [ [7,8,9],[10,11,12] ] ])
```

On pourra s'appuyer sur la fonction programmée dans l'exercice précédent pour implémenter les modifications géométriques d'images, ci-dessous.

- **Symétrie verticale**
  Créer une fonction `symetrie_verticale` prenant en argument une image `image` et renvoyant sa version après symétrie par rapport à l'axe vertical passant en son centre.
- **Rotation de 90 degrés dans le sens horaire**
  Créer une fonction `rotation_90` prenant en argument une image `image` et renvoyant sa version après rotation de 90 degrés dans le sens trigonométrique.
- **Autres rotations**
  Programmer des fonctions `rotation_180` et `rotation_270` effectuant une rotation de, respectivement, 180 et 270 degrés (toujours dans le sens trigonométrique).

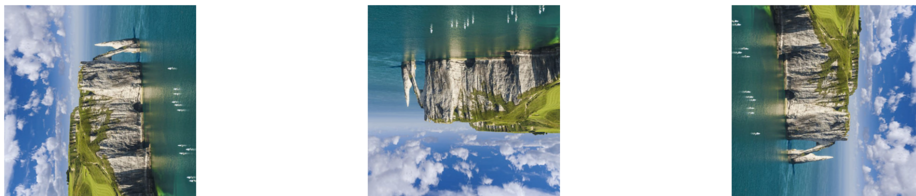


**Figure 3.** Rotations de l'image (de gauche à droite : 90, 180, 270 degrés).

# III. Niveaux de gris, augmentation de contraste

Pour convertir une image en **niveaux de gris** (ce qu'on appelle couramment le « noir et blanc »), il faut affecter à chaque pixel le même niveau pour les paramètres `R,G,B` appelé **luminance**.
Il existe plusieurs modes de calcul de la luminance, nous utiliserons celui-ci :

$$\text{luminance} \;=\; \lfloor 0.2126 \times R + 0.7152 \times G + 0.0722 \times B \rfloor$$

(on notera que la somme des trois facteurs fait 1).

**Remarque**
On n'est pas obligé de convertir la luminance de flottant en entier : **Python** s'en charge seul au moment de l'affichage de l'image par la fonction `plt.imshow`. Néanmoins, en effectuant cette conversion, l'image reste une matrice d'entiers, et non une matrice de flottants (pour cela, un transtypage `luminance = int(luminance)` a le même effet qu'une partie entière, pour les flottants positifs).

- **Niveaux de gris**
  Écrire une fonction `niveaux_de_gris` prenant en argument une image `image` et renvoyant sa version en niveaux de gris.
  On utilisera la formule écrite ci-dessus pour la luminance.

On peut également modifier le contraste de l'image. Pour cela, on applique une fonction affine à la luminance, utilisant un facteur $a > 0$ pour régler le contraste. La luminance étant un flottant compris entre 0 et 255, on pose $x = \dfrac{\texttt{luminance}}{256}$ et on applique la fonction suivante :

$$f : x \mapsto a \times (x - 0.5) + 0.5$$

On remplace $f(x)$ par 1 (resp. par 0) si le résultat dépasse 1 (resp. passe sous 0).
On récupère alors une luminance modifiée par luminance $= \lfloor 255 \times f(x) \rfloor$.

- **Niveaux de gris avec contraste**
  Écrire une fonction `niveaux_de_gris_contraste` prenant en arguments une image `image`, un facteur `facteur` sous la forme d'un flottant strictement positif, et renvoyant l'image en niveaux de gris avec prise en compte du facteur de contraste (le flottant `facteur` représente le réel $a$ dans la formule définissant $f$ ci-dessus).

**Figure 4.** Photos en niveaux de gris (à droite, avec un facteur de contraste $a = 2$).

# IV. Photomaton !

On se propose désormais d'« éclater » l'image d'origine en 4 petites vignettes. Les règles sont les suivantes :

- la résolution de l'image renvoyée est la même que celle de l'image d'origine ($512 \times 512$),
- les pixels de l'image d'origine sont regroupés par blocs de taille $2 \times 2$,
- les pixels de chaque bloc sont envoyés dans chacune des 4 sous-images.

Par exemple, prenant le bloc supérieur gauche constitué des pixels d'indices $(0,0)$, $(0,1)$, $(1,0)$ et $(1,1)$, chacun des pixels est envoyé dans le coin supérieur gauche de « sa » sous-image. Plus précisément, numérotons les sous-images dans le sens de la lecture, de 1 à 4. Le pixel d'indice $(0,0)$ fera le pixel supérieur gauche de la sous-image 1, le pixel $(0,1)$ fera le pixel supérieur gauche de la sous-image 2, le pixel $(1,0)$ fera le pixel supérieur gauche de la sous-image 3, et on devine pour le pixel $(1,1)$.
Aucun pixel n'est perdu, chaque pixel est juste déplacé. On n'obtient pas quatre copies conformes de l'image d'origine, mais l'illusion d'optique est presque parfaite.

- **Photomaton**
  Écrire une fonction `photomaton` prenant en argument une image `image` et renvoyant une image correspondant à l'opération de « photomaton » décrite ci-dessus.
  Il faudra que la fonction s'adapte à une image présentant une ou deux dimensions impaires, laissant la dernière ligne ou colonne, le cas échéant, inchangée.

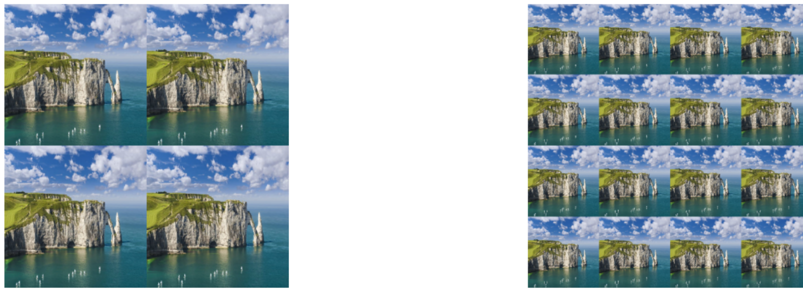


**Figure 4.** Photomaton après plusieurs itérations (une à gauche, deux à droite).

# V. Un peu de stéganographie

## V.1. Procédure

La **stéganographie graphique** est une méthode de dissimulation consistant à cacher une image ou un message à l'intérieur d'une autre image. C'est ceci qu'on propose d'expérimenter dans cette dernière partie du TP.
La méthode de stéganographie que nous proposons de développer ici consiste à utiliser les **bits de poids faible** d'une image. Nous avons jusqu'ici manipulé des pixels codés au format `[R,G,B]` où `R,G,B` sont des entiers compris entre 0 et 255. Ainsi, chacun de ces entiers est codé sur un octet (8 bits, car $2^8 = 256$).

- **Dissimulation**
  On part d'une image dans laquelle on arrondit chaque entier `R`, `G` et `B` au multiple de 16 immédiatement inférieur. Ainsi, chacun d'entre eux s'écrirait en base 2 sous la forme `bbbb0000` (avec `bbbb` on conserve les bits de poids fort). On utilise alors les 4 derniers bits (bits de poids faible), annulés, pour y cacher un message (ou une autre image : par exemple, les bits de poids fort d'une autre image).
- **Révélation**
  Dans le sens inverse, l'idée est de récupérer ces bits de poids faible d'une image « cryptée », et de reconstruire une autre image avec ces bits en les passant en poids fort.

Par exemple, on part du code RGB suivant `[134,240,12]`. On remarque que :

- l'écriture en base de 2 de 134 est 10000110 car :

$$134 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

  Si on ne conserve que ses bits de poids forts, on obtient, en base 2, le nombre 10000000. Il s'agit du nombre 128 en base décimal.
  Si on ne conserve que ses bits de poids faibles, on obtient, en base 2, le nombre 0110. Il s'agit du nombre 6 en base décimale.

- l'écriture en base de 2 de 240 est 11110000 car :

$$240 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

  Si on ne conserve que ses bits de poids forts, on obtient, en base 2, le nombre 11110000. Il s'agit du nombre 240 en base décimal.
  Si on ne conserve que ses bits de poids faibles, on obtient, en base 2, le nombre 0000. Il s'agit du nombre 0 en base décimale.

- l'écriture en base de 2 de 12 est 00001100 car :

$$12 = 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

  Si on ne conserve que ses bits de poids forts, on obtient, en base 2, le nombre 00000000. Il s'agit du nombre 0 en base décimal.
  Si on ne conserve que ses bits de poids faibles, on obtient, en base 2, le nombre 1100. Il s'agit du nombre 12 en base décimale.

La conservation des bits de poids fort fournit donc le pixel `[128,240,0]`, et la récupération des bits de poids faible fournit le pixel `[6,0,12]`.
Si on multiplie ces bits de poids faible par 16, on crée de nouveaux bits de poids fort : `[96,0,192]`.
En procédant ainsi pixel par pixel, on reconstitue une nouvelle image. Nous appelons cette méthode la « mise en poids fort des bits de poids faible ».

**Stéganographie**

On se propose de coder les méthodes de révélation et de dissimulation décrites ci-dessus. On pourra tester les fonctions en dissimulant une rotation de l'image `etretat` dans `etretat` elle-même, puis en essayant de la révéler.

- Écrire une fonction `stegano_revelation` qui prend en argument une image `image` et qui renvoie l'image obtenue par « mise en poids fort des bits de poids faible ».
- Écrire une fonction `stegano_dissimulation` qui prend en argument deux images `image1` et `image2`, et qui renvoie l'image obtenue en dissimulant l'image `image2` à l'intérieur de l'image `image1` en utilisant les bits de poids faible, conformément à la méthode de dissimulation expliquée plus haut.

### V.2. C'est l'heure de l'enquête

**Qui a lu *l'Aiguille Creuse* ?**
Les images des questions ci-dessous sont au format `*.png`. Lors de leur acquisition par la fonction `plt.imread()`, les pixels comportent 4 composantes : R, G, B et un facteur d'opacité. Nous n'avons pas besoin de ce facteur d'opacité, aussi il est possible de tronquer la 4$^{\text{ème}}$ composante par l'instruction suivante :

```
imageImportee = imageImportee[:, :, :3]
```

Par ailleurs, les niveaux de R, G, B sont indiqués par des flottants compris entre 0 et 1. Il faut les transformer en entiers appartenant à $[\![0, 255]\!]$, ce qui est possible par la commande suivante :

```
imageImportee = (255*imageImportee).astype(np.uint8)
```

La multiplication par 255 est effectuée composante par composante, et la méthode `astype()` s'occupe du transtypage. Le type `np.uint8` est un type entier propre à la librairie `numpy`.
Il est désormais temps de jouer avec les images.

- Télécharger l'image `EtretatSteg.png` au lien suivant :

  http://rduroux.ovh/PCSI/ITC/CH8/EtretatSteg.png

- Dans le fichier `EtretatSteg.png` se cache une image. Laquelle ?