Traitement d'images

I. Les images informatiques

Une image matricielle (ou *carte de points*, *bitmap*) est une matrice de points colorés appelés pixels (= *picture element*).
Chaque case de la matrice contient la couleur du pixel correspondant. On convient de numéroter les colonnes en partant de la gauche, et les lignes en partant du haut.
Cette couleur est représenté par un triplet de nombres entre 0 et 255 (3 octets) : un nombre pour chaque couleur primaire rouge, vert et bleu ; c'est le format RGB (*red, green, blue*). La couleur définitive de chaque pixel est obtenue par addition de ces couleurs primaires.

Exemple :

(0,0,0)	noir	(255,255,255)	blanc
(255,0,0)	rouge	(0,255,0)	vert
(0,0,255)	bleu	(0,255,255)	cyan
(255,0,255)	magenta	(255,255,0)	jaune
(148,129,43)	kaki	(0,86,27)	vert impérial
(248,142,85)	saumon	...	...

Cela fait $256^3 = 16\,777\,216$ couleurs possibles.

Certaines images sont encodées en RGBA ; dans ce cas, un 4e octet est utilisé (*canal alpha*) pour gérer la transparence, de 0 pour un pixel totalement transparent à 255 pour un pixel complètement opaque.

II. Les outils Python

Pour lire, écrire et modifier des images, il existe de nombreux modules en Python (qu'il ne faut pas utiliser en même temps).
Nous utiliserons ici la bibliothèque **pillow**, qui est la version maintenue à jour de la bibliothèque PIL (Python Imaging Library). Vous pouvez éventuellement trouver les fichiers d'installation de **pillow** ici : https://pypi.python.org/pypi/Pillow/, et une documentation (en Anglais) ici :
http://pillow.readthedocs.org/en/3.0.x/handbook/index.html ou là :
https://media.readthedocs.org/pdf/pillow/latest/pillow.pdf .

Nous utiliserons aussi la bibliothèque **numpy** qui contient des outils très partiques et performants pour manipuler des matrices.

Pour simplifier, les images utilisées seront placées dans le même répertoire que votre programme (cela évite d'écrire tout le chemin d'accès).
Les instructions de base sont illustrées dans le petit programme (sans utilité) ci-dessous.

```
1  from PIL import Image
2  import numpy as np
3  # import os
4  # pour connaître le répertoire courant
5
6  # current_dir = os.getcwd()
7  # print(current_dir)
8
9  # lecture du fichier image
10 im = Image.open('Joconde.jpg')
11 print(im.size, im.mode, im.format)
12 im.show()
13 # im.size = largeur, hauteur
14
15 # conversion de l'image en tableau Numpy
16 tab = np.array(im)
17 print(tab.shape)
18 # hauteur, largeur, 3 (ou 4)
19
20 tab2 = tab.copy()
```

```python
hauteur, largeur, n = tab2.shape

# trait blanc horizontal créé par un tableau
blanc = np.zeros( (20, largeur, 3), dtype = 'uint8')
blanc.fill(255)
tab2[hauteur//2-10:hauteur//2+10, :, :] = blanc

# trait noir vertical créé par une image
im2 = Image.new("RGB", (20, hauteur), (255, 255, 255))
jaune = np.array(im2)
tab2[:, largeur//2-10:largeur//2+10, :] = jaune

# conversion du tableau en image
nouv_image = Image.fromarray(tab2)

# on peut extraire des pixels depuis l'image
for x in range(largeur):
    for y in range(hauteur):
        pixel = nouv_image.getpixel( (x,y) )
        if sum(pixel)>450:
            nouv_image.putpixel( (x,y), (255,255,255) )

# on peut aussi directement écrire dans l'image
for x in range(2,min(largeur, hauteur)-2):
    y=int(hauteur/largeur*x)
    for k in range(-2, 3):
        for l in range(-2, 3):
            nouv_image.putpixel( (x+k,y+l), (255,255,0) )
            nouv_image.putpixel( (largeur-x-k-1,y+l), (255,0,0) )

# affichage
nouv_image.show()
# sauvegarde sous un nouveau format
nouv_image.save("Joconde_defiguree.ps")
```

```
(620, 701) RGB JPEG
(701, 620, 3)
```

Voici le résultat de ce programme stupide :

Figure 1 – Joconde

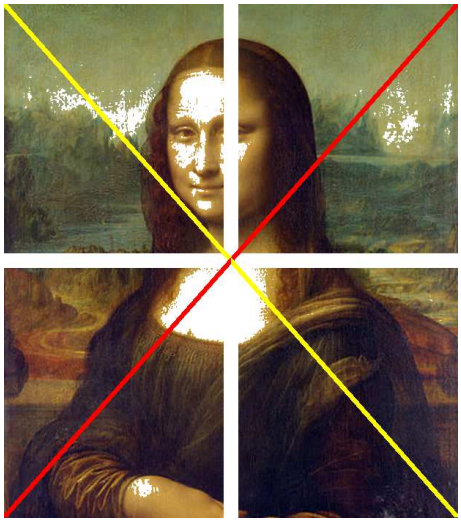

Figure 2 – Joconde défigurée

Remarque : il est important de noter que les coordonnées des pixels dans une image chargée par le module `Image` sont semblables aux coordonnées dans un repère xOy habituel, mais avec l'axe Oy vers le bas, le point $(0,0)$ étant le coin en haut à gauche de l'image.

Par contre, lorsque l'on utilise les tableaux `Numpy`, les coordonnées d'un point respectent les mêmes conventions que celles du calcul matriciel (mais en partant de 0). Ainsi, le pixel de coordonnées (x, y) dans l'image obtenue par `pillow` sera le terme d'indice $[y, x]$ de la matrice correspondante.

De la même manière, les couleurs dans `Numpy` sont représentées par des listes [R,G,B] alors que dans `pillow` ce sont des tuples (R,G,B).

III. Exercices

III.1. Décomposition d'une image en ses composantes

Écrire une fonction `composantes` prenant en argument le nom d'un fichier image sous la forme 'nom.ext' ainsi que l'une des variables 'R', 'G' ou 'B' et qui crée un nouveau fichier de nom 'nom_rouge.ext' ou 'nom_vert.ext' ou 'nom_bleu.ext' et contenant la composante rouge, verte ou bleue de l'image initiale.

Résultat :

Figure 3 – Joconde

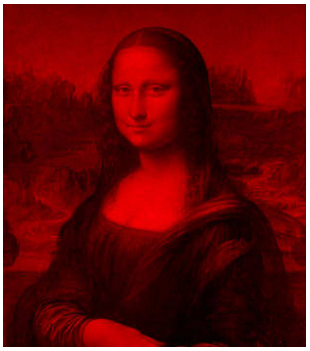

Figure 4 – Joconde rouge

Figure 5 – Joconde verte

Figure 6 – Joconde bleue

III.2. Transformation en niveaux de gris, puis en sépia

1. Une première solution pour transformer une image en noir et blanc consiste à remplacer les 3 couleurs (R,G,B) de chaque pixel par le triplet (L,L,L) où $L = \frac{1}{3}(R + G + B)$. De plus, pour gagner de la place en mémoire, le module `pillow` permet de définir une image monochrome en affectant chaque pixel d'une seule couleur au lieu de 3 (ici ce sera L). Ce sont des images dont le mode est 'L' (comme *luminance*) au lieu de 'RGB' ; elles seront obtenues en sauvant un tableau à deux dimensions par `Image.fromarray(tab, 'L')`.

2. Le résultat précédent n'est pas toujours très satisfaisant. En effet, l'œil est plus sensible à certaines couleurs qu'à d'autres. Le vert (pur), par exemple, paraît plus clair que le bleu (pur). Pour tenir compte de cette sensibilité dans la transformation d'une image couleur en une image en niveaux de gris, on ne prend généralement pas la moyenne arithmétique des intensités de couleurs fondamentales, mais une moyenne pondérée. La formule standard donnant le niveau de gris en fonction des trois composantes est :

$$L = \lfloor 0.299 * R + 0.587 * G + 0.114 * B \rfloor \ .$$

(L s'appelle la *luminance* du pixel).

Voici les résultats obtenus :

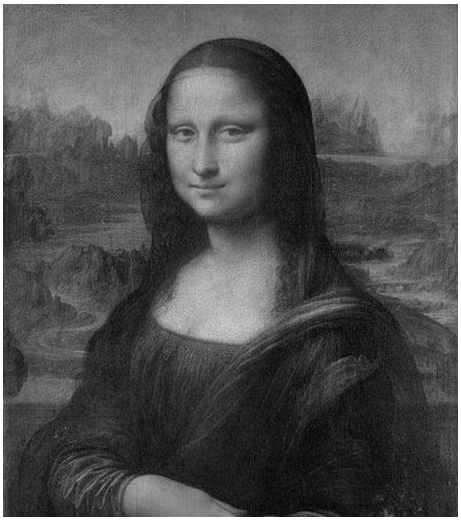

Figure 7 – 1re formule

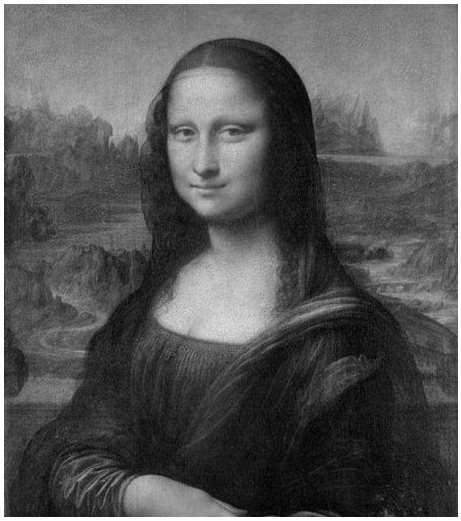

Figure 8 – 2e formule

3. La transformation en sépia est un peu plus complexe.
En photographie, le sépia est une qualité de tirage qui ressemble au noir et blanc, mais avec des variations de brun, et non de gris. La couleur sépia dans le système RGB est S = (94, 38, 18).
Dans la transformation d'une image couleur en une image en nuances de sépia, on tient compte d'un seuil ($0 < \text{seuil} < 255$), qui sépare le sépia assombri et le sépia éclairci. La transformation se fait alors pixel par pixel en deux temps. Pour chaque pixel, on calcule d'abord un niveau de gris qui est la moyenne m des intensités de rouge, vert et bleu. Puis, si le gris obtenu est foncé ($m < \text{seuil}$), on le remplace par une couleur du segment NS, couleur d'autant plus proche du noir N que m est petit. Si le gris est plus clair ($m > \text{seuil}$), il est remplacé par une couleur du segment SB, couleur d'autant plus proche du blanc B que m est grand.

Voici le résultat :

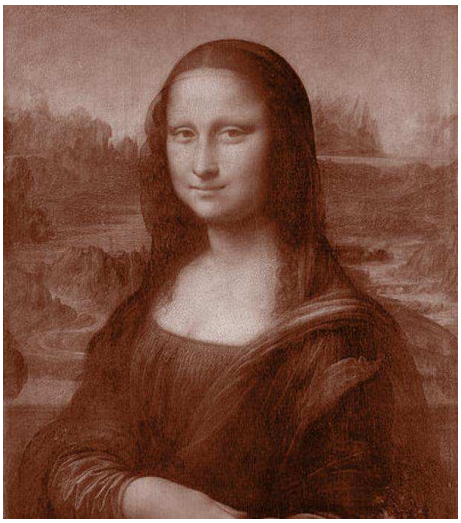

Figure 9 – seuil=40

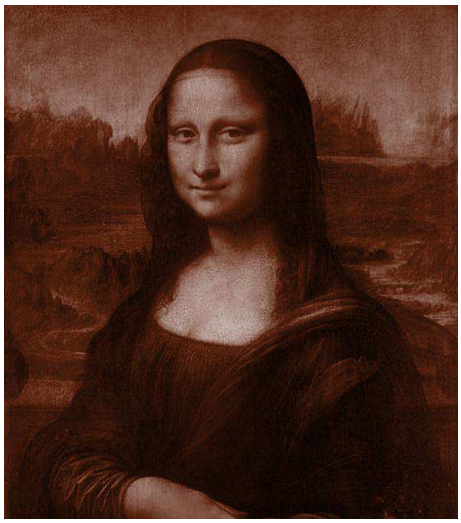

Figure 10 – seuil=100

III.3. La méthode point

La méthode `point` permet d'effectuer rapidement une opérations sur tous les pixels d'un fichier image en une seule instruction (donc pas besoin de faire des boucles).
La fonction doit être une application de $[\![0\,;255]\!]$ dans $[\![0\,;255]\!]$. Elle peut être définie par un bloc `def` ou simplement par une instruction `lambda`.

Exemple :

```
1  from PIL import Image
2  import numpy as np
3  import math
4
```

```
5  im = Image.open('Chambre.jpg')
6
7  def f(x):
8      return 0.5 + 0.5*math.sin((x-0.5)*math.pi)
9
10 out1 = im.point(lambda i: int(255*f(i/255)))
11 out2 = im.point(lambda i: int(255*f(f(i/255))))
12 im.show()
13 out1.show()
14 out2.show()
```

Le résultat est le suivant :

Figure 11 – Chambre de Van Gogh

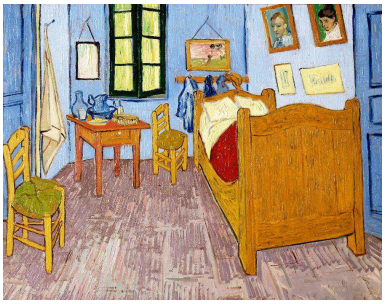

Figure 12 – f appliquée

Figure 13 – $f \circ f$ appliquée

Cette opération a donc augmenté le contraste. Tracez la courbe représentative de f pour comprendre pourquoi. Essayez également avec la fonction $f : x \mapsto 3x^2 - 2x^3$.

Exercice : Utilisez cette méthode pour produire à partir d'une image :

- l'image en négatif ;
- une image plus lumineuse : on pourrait se contenter d'ajouter une valeur fixe à tous les pixels, c'est-à-dire considérer une fonction de la forme $x \mapsto min(255, x+h)$, mais cette méthode est trop brutale et entraîne des pertes d'information. Il faut donc trouver une fonction plus « lisse » (et bijective !). On pourra utilise la fonction $x \in [0\,;1] \mapsto x^{\frac{1}{\gamma}}$ où $\gamma \in \mathbb{R}_+$: c'est la *correction gamma*.
- une image moins lumineuse ;
- une image moins contrastée.

III.4. Superposition d'images

Écrire un programme qui superpose deux images. Si les images ne sont pas de meme taille, on les redimensionnera avec la commande `Image.resize (width, height)`. L'image obtenue sera donnée par :

$$im3 = \alpha * im1 + (1-\alpha) * im2$$

où α est un coefficient entre 0 et 1.

Résultat :

Figure 14 – Potiron

Figure 15 – Python

Figure 16 – Superposition $\alpha = 0.6$

III.5. Pixellisation

L'image est divisée en rectangles de taille donnée. Chaque rectangle est ensuite rempli avec la couleur moyenne de la zone.

Voilà ce que vous devez obtenir :

Figure 17 – Pixels 6x6

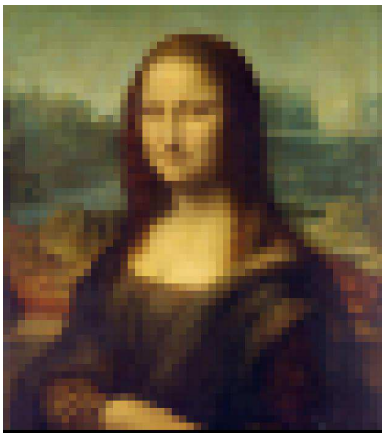

Figure 18 – Pixels 12x12

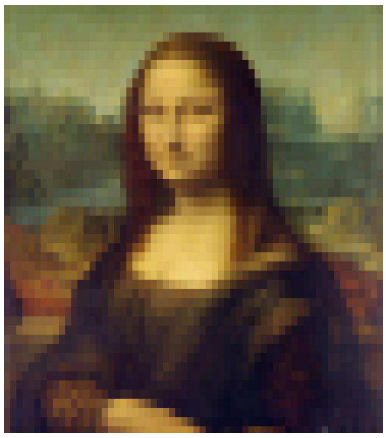

Figure 19 – Pixels 15x10

III.6. Utilisation d'un masque

On dispose d'un masque de forme quelconque, de préférence une image en deux couleurs. Ecrire un programme qui, à partir d'une image et d'un masque donne une image masquée, comme ci-dessous :

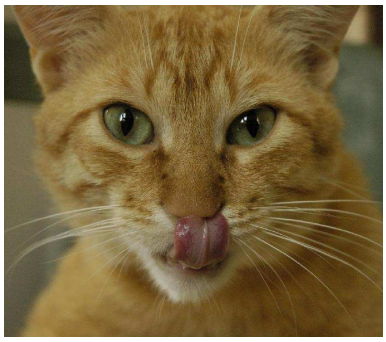

Figure 20 – Figure source

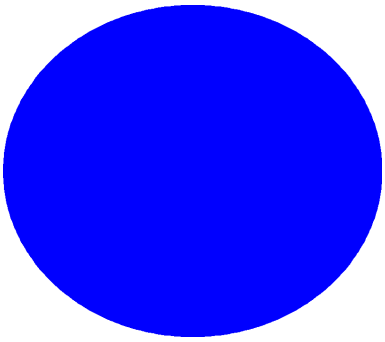

Figure 21 – Masque

Figure 22 – Figure masquée

III.7. Transformations géométriques

1. Écrire une fonction `quadrant(image, i, j)` qui renvoie l'un des quatre quadrants de l'image ($(i, j) = (1, 2)$ correspondra au quart supérieur droit).

Utilisez ces fonctions pour produire les images suivantes :

Figure 23 – Figure initiale

Figure 24 – Figure transformée

2. a) Écrire une fonction `symétrie(image, type)` qui renvoie le symétrique de l'image par rapport à l'axe médian vertical si `type='V'` ou l'axe médian horizontal si `type = 'H'`.

b) Écrire une fonction `rotation(image, angle, centre)` qui renvoie l'image obtenue après une rotation de centre le point `centre` et d'angle `theta`.

Ces deux exercices sont un cas particulier du suivant. Étant donnée la matrice $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ d'une application linéaire u et un point Ω de coordonnées (x_0, y_0) (origine du repère), il s'agit de transformer l'image par l'application qui au point M de coordonnées (x, y) associe le point M' de coordonnées (x', y') tel que :

$$\overrightarrow{\Omega M'} = u(\overrightarrow{\Omega M}) \quad \text{soit} \quad \begin{pmatrix} x' - x_0 \\ y' - y_0 \end{pmatrix} = M \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix}.$$

Il s'agit de la composée d'une application linéaire et d'une translation. Les points de l'image qui, après transformation, sortiraient du cadre seront simplement supprimés, et les points qui n'ont pas d'antécédent dans la nouvelle image seront d'une couleur de fond prédéfinie.

Le gros problème dans les transformations géométriques d'image est que, le plus souvent, l'image d'un pixel à coordonnées entières ne donne pas des coordonnées entières ; il y aura donc dans l'image destination des pixels qui ne seront pas atteints, ce qui va faire apparaître des « trous » dans l'image. Une solution consiste alors combler ces trous par interpolation, mais il est également possible d'utiliser la transformation inverse, en calculant l'antécédent de tout pixel et en cherchant dans l'image source le point le plus près, ou, mieux, en faisant une interpolation bilinéaire à l'aide des quatre pixels autour (mais cette méthode est assez chronophage).

Voici le résultat d'une rotation de 50^o :

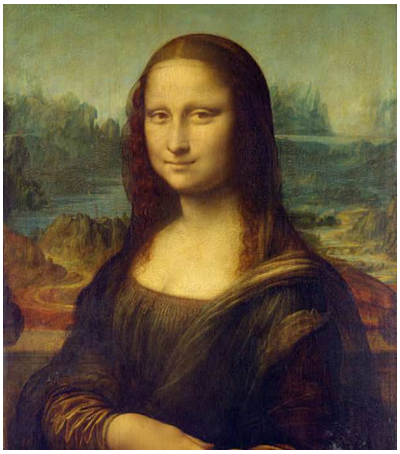

Figure 25 – Joconde

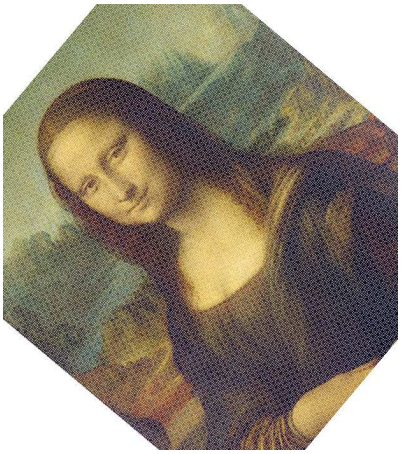

Figure 26 – Rotation directe

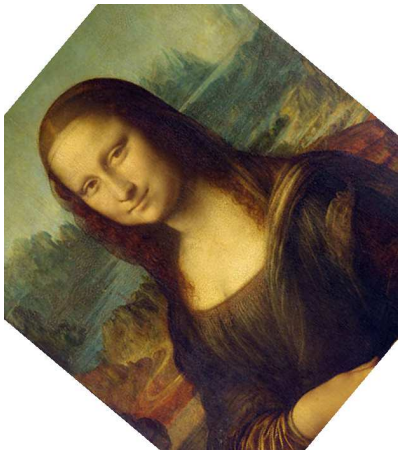

Figure 27 – Utilisation de f^{-1}

III.8. Convolution

La convolution d'image consiste à modifier la valeur d'un pixel en fonction des valeurs des pixels voisins.
La convolution est définie par une matrice N_{ij} appelée *noyau* ou *filtre*, de taille $(2p+1) \times (2p+1)$, où p est un entier.
Elle est définie par les formules :

$$nouv\_pixel(i,j) = \sum_{k=0}^{2p} \sum_{\ell=0}^{2p} N(k,\ell) * pixel(i+k-p, j+\ell-p).$$

1. Écrire une fonction `convolution(image,N)` qui renvoie l'image résultant de l'opération de convolution.

On fera attention aux bords (le choix le plus simple étant de ne pas les modifier).

2. Essayez les noyaux $N = \frac{1}{9} I_3$ et $N = \frac{1}{25} I_5$; ce filtre, dit filtre *moyenneur*, est un filtre passe-bas : il lisse l'image (en donnant un effet de flou), réduit les bruits et les détails.

Cependant, un des meilleurs moyens pour réduire le bruit est d'utiliser un filtre *médian* ; ce type de filtrage ne peut pas être obtenu par convolution ; il consiste à remplacer la valeur d'un pixel par l'élément médian des 9 ou 25 pixels autour.
Programmer ce filtrage (utiliser la fonction Numpy `np.medain(liste)`), et comparer aux images précédentes.

Figure 28 – Original bruité

Figure 29 – Filtre moyenne 3×3

Figure 30 – Filtre médian 3×3

3. Il existe de nombreux autres filtres, permettant de répondre à différents besoins, en particulier la détection des contours.

Pour détecter les contours dans une image, on la transformera d'abord en niveaux de gris : cela ne permet de ne tenir compte que de la luminance de l'image, et de plus le traitement sera plus rapide. Pour cela, on utilisera soit le programme déjà écrit (voir page 3), soit la méthode PIL : `im.convert('L')`.

Le filtre de Sobel $G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$ permet de détecter les contours dans le sens horizontal (essayez de comprendre pourquoi) et $G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$ permet de détecter ceux dans le sens vertical.

Pour détecter les contours, il faut donc faire une « moyenne » des deux ; on calculera donc, pour chaque pixel, le résultat S_x de la convolution par G_x (attention : il y a ici des nombres négatifs, qui ne sont pas gérés par le type `uint8` !), puis le résultat S_y de celle par G_y, et enfin on affectera le pixel de la couleur $\sqrt{S_x^2 + S_y^2}$ (attention à ne pas dépasser 255 !). De plus, pour améliorer la rapidité du traitement, on n'utilisera pas la procédure de convolution précédente (car elle fait à chaque fois 3 multiplications par 0 inutiles), mais on écrira une procédure spéciale.

Voici le résultat :

Figure 31 – Lena

Figure 32 – Contours

III.9. Stéganographie

La *stéganographie* consiste à cacher une image (ou un message) dans une autre. Le principe est le suivant.

Dans une image, à chaque pixel est associé un triplet de couleurs R,G,B, chacune étant représentée par un octet = 8 bits.

L'idée de la stéganographie est que si l'on modifie très légèrement ces couleurs, cela sera invisible à l'œil nu.

Voyez-ci dessous :

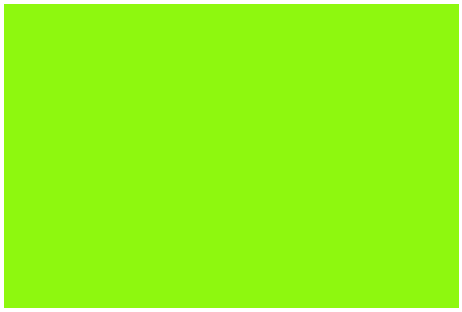

Figure 33 – R,G,B = 142,248,15

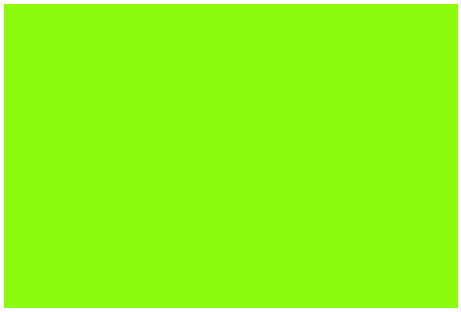

Figure 34 – R,G,B = 140,250,12

Pour cacher un pixel A dans un pixel B, on remplacera les bits de poids faibles (ceux les plus à droite dans l'écriture en base 2) dans les couleurs de A par les bits de poids forts (ceux situés les plus à gauche) des couleurs de B.
On utilise en général 3 ou 4 bits.

Exemple (en utilisant 3 bits) :
Le pixel A a pour couleurs

$$R_A = 255 = \overline{11111\ 111}\ ,\ G_A = 50 = \overline{00110\ 010}\ \text{ et }\ B_A = 21 = \overline{00010\ 101}$$

et le pixel B :

$$R_B = 124 = \overline{011\ 11100}\ ,\ G_B = 200 = \overline{110\ 01000}\ \text{ et }\ B_B = 57 = \overline{001\ 11001}\,.$$

Pour cacher B dans A, on remplacera le pixel A par le pixel A' tel que :

$$R_{A'} = \overline{\underbrace{11111}_{A}\underbrace{011}_{B}} = 251\ ,\ G_{A'} = \overline{\underbrace{00110}_{A}\underbrace{110}_{B}} = 54\ \text{ et }\ B_{A'} = \overline{\underbrace{00010}_{A}\underbrace{001}_{B}} = 17\,.$$

Pour retrouver le pixel caché, on fait l'opération « inverse » (ce n'est pas vraiment bijectif !) : on extrait les bits de poids faible du pixel reçu et on complète les bits manquants par des 0. On obtiendra donc le pixel B' tel que :

$$R_{B'} = \overline{011\ 00000} = 96\ ,\ G_{B'} = \overline{110\ 00000} = 192\ \text{ et }\ B_{B'} = \overline{001\ 00000} = 32$$

Voici les résultats :

Figure 35 – Pixel A

Figure 36 – Pixel B

Figure 37 – Pixel A'

Figure 38 – Pixel B'

Méthode
Pour obtenir les k bits de poids faible d'un entier x codé sur 8 bits, il suffit de considérer le reste dans la division euclidienne de x par 2^k. Pour obtenir les ℓ bits de poids fort, il faut considérer le quotient dans la division euclidienne de x par $2^{8-\ell}$.

Python dispose aussi d'opérateurs permettant de travailler directement sur l'écriture en base 2 d'un entier. Ce sont :

- l'opérateur '&' : réalise l'opération logique 'et' bit par bit ;
- l'opérateur '|' : réalise l'opération logique 'ou' bit par bit ;
- l'opérateur '^' : réalise l'opération logique 'ou exclusif' bit par bit ;
- l'opérateur '>>' : décale les bits vers la droite (correspond à une division par 2) ;
- l'opérateur '<<' : décale les bits vers la gauche (correspond à une multiplication par 2).

Exercice 1
Écrire deux procédures : l'une permettant de cacher une image dans une autre (on supposera que les deux images sont de la même taille), l'autre permettant de retrouver l'image cachée. On passera en paramètre le nombre de bits utilisé par la méthode (3 ou 4 en général).

Voici les résultats obtenus (on a caché Lena dans un paysage).
Images initiales :

Figure 39 – Lena

Figure 40 – Paysage

Lena cachée dans le paysage :

Figure 41 – k=2

Figure 42 – k=3

Figure 43 – k=4

Figure 44 – k=5

Lena dévoilée :

Figure 45 – k=2

Figure 46 – k=3

Figure 47 – k=4

Figure 48 – k=5

Exercice 2
On veut maintenant cacher un texte dans une image : Chaque lettre du texte est encodée sur un octet (code ASCII) : les chiffres de 0 à 9 ont pour code 48 à 57, les lettres de A à Z ont pour code 65 à 90, celles de a à z ont pour code 97 à 122 etc... Les fonctions Python `chr` et `ord` permettent de passer de l'un à l'autre.
La méthode consiste à cacher chaque lettre du message dans 3 pixels consécutifs, en remplaçant le bit de poids faible de chacune des couleurs des trois pixels (sauf la couleur bleue du dernier pixel) par le bit correspondant de la lettre à cacher.

Écrire une procédure permettant de cacher un texte dans une image et une autre permettant de découvrir le texte caché.

⋆ ⋆ ⋆ ⋆
⋆ ⋆ ⋆
⋆ ⋆
⋆