chapitre 10

Les Images

Table des matières

I Représentation informatique d'une image

Il existe deux grands types de représentations pour les images.

La **représentation vectorielle**, pour les images qu'on peut représenter à partir d'un nombre fini de figures géométriques élémentaires. Le fichier image contient alors une suite d'instructions de tracé exprimées dans un langage de programmation.

La **représentation bitmap**, avec laquelle nous travaillerons, où l'image est quadrillée selon une matrice de $largeur \times hauteur$ cases, appelées **pixels** (*picture element*). Chaque pixel code la couleur d'un petit carré de l'image à l'aide de 1, 3 ou 4 valeurs suivant le type d'image.

- ⋆ Pour une image en niveaux de gris, la couleur de chaque pixel est codée sur un octet (8 bits), ce qui permet d'obtenir $2^8 = 256$ valeurs possibles (de 0 à 255). Par exemple, la valeur 0 code la couleur noir et la valeur 255 code le blanc.
- ⋆ Pour une image en couleur, le codage le plus utilisé est le codage RGB (RedGreenBlue). Ici, la couleur d'un pixel est codée sur 3 octets (3×8 bits) par un triplet de trois entiers compris entre 0 et 255 : le premier entier code la quantité de rouge, le deuxième la quantité de vert et le troisième la quantité de bleu. La couleur du pixel est obtenue par synthèse additive de ces trois couleurs. Par exemple, le triplet $(0, 0, 0)$ correspond au noir, $(200, 200, 200)$ à un gris clair, $(255, 0, 0)$ à un rouge éclatant, $(255, 255, 0)$ à un jaune vif (rouge + vert), $(255, 127, 0)$ à du orange... Finalement une image en couleur est la superposition de trois images monochromes (cf exercice 10.3)
- ⋆ Pour une image en couleur, on utilise également

Il existe d'autres façons de représenter des images en couleurs, notamment le codage RGBA (RedGreenBlueAlpha). Dans ce cas, la couleur d'un pixel est codée sur 4 octets par quatre entiers compris entre 0 et 255 : les trois premiers entiers codent la quantité de rouge, de vert et de bleu, le quatrième entier code la transparence de la couleur. Ou encore « CYMK » très utile pour l'impression, mais nous ne travaillerons lors de ce TP qu'avec les deux premières évoquées.

Les fichiers images de type bitmap sont adaptés à la représentation de figures aux contours irréguliers, en photographie par exemple. Le format `bmp` permet une simple description pixel par pixel, le poids de l'image est alors $Largeur \times Hauteur \times 3$ *octets*. Les fichiers images peuvent donc être de grande taille, pour cela ils sont souvent compressés avec ou sans perte d'information. Les formats `png` et `jpg` sont des exemples de formats d'images bitmap compressés. Pour le premier la compression est réversible (c'est-à-dire sans perte d'information), pour le second c'est avec perte d'information, mais le taux de compression est plus important.

Exemple 10.1. d'un système de compression : $(1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, \ldots) \longrightarrow (1, 2, 3, 4, 1, 3, \ldots)$.

II Traitement d'images en Python

Nous aurons trois niveaux d'abstraction pour traiter nos images. Le fichier physique sur le disque (`png`, `jpg` par exemple). L'« objet Python » intermédiaire (on ne s'en occupera pas vraiment). Et enfin la matrice des pixels sur laquelle on opèrera.

Pour traiter l'« objet Python intermédiaire » nous utiliserons le module `Image` de la bibliothèque `PIL` (Python Imaging Library). `PIL` ne fait pas partie de la bibliothèque standard de Python : il s'agit d'une bibliothèque supplémentaire (libre) qu'il faut installer. Pour se faire il faut avoir une connexion internet et dans l'interpréteur Python il suffit de taper une des deux lignes suivantes :

```
>>> pip install Pillow
>>> conda install Pillow
```

l'installation se fera automatiquement. La documentation est disponible à partir de la page web http://effbot.org/imagingbook/pil-index.htm.

Pour travailler avec les matrices nous utiliserons la bibliothèque `numpy`. C'est un outils très puissant déjà installé avec Pyzo et dont vous aurez l'occasion de vous servir notamment en algèbre linéaire l'an prochain.

Remarque : On peut faire du traitement d'image avec d'autres modules de python `matplotlib.pyplot`, `imageio` par exemple.

Avant de commencer ce TP, prenez soin de créer un répertoire dans lequel vous allez travailler. Vous y copierez depuis le dossier de partage : le fichier `.py` dans lequel une partie des codes est déjà tapé, les images que nous utiliserons.

Pensez aussi que vous allez travailler avec des fichiers, et comme dans le TP précédent lors de la première exécution vous prendrez soin de faire `Exécuter / Démarrer le script`.

Dans ce qui suit nous allons décrire les principales commandes dont nous aurons besoin pour la suite.

```
from PIL import Image
import numpy as np

im = Image.open("nom_fichier.extension") # "objet python" intermédiaire.
im.show()  # affiche l'image dans une fenêtre graphique
pixels = np.array(im) # matrice des pixels
hauteur, largeur = pixels.shape[:2] # dimensions de la matrice
```

Remarques :

- ⋆ `pixels` est un objet de type `array` (tableau). Chaque élément qui le compose est un entier (non signé) compris entre 0 et 255.
- ⋆ Si on travaille avec une image en couleur `pixels.shape` renverra un triplet (hauteur, largeur, 3), si on travaille avec une image en niveau de gris `pixels.shape` renvoie directement le couple (hauteur, largeur). En imposant de ne renvoyer que les deux première composantes avec la commande `pixels.shape[:2]` on n'a pas à se soucier si on travaille en niveau de gris ou en couleur.

Un des buts de ce TP est de faire du traitement d'image, c'est à dire modifier la luminosité, le contraste, appliquer des filtres ... On va donc être amené à modifier la valeur des pixels, voici donc comment accéder à ces valeurs et comment les modifier. (on supposera que la matrice des pixels se nomme `pixels`)

```
couleur=pixels[i,j]
# On récupère la couleur du pixel situé sur la i-ième ligne et la j-ième colonne.
```

Si l'image est en couleur on obtiendra un triplet (en fait un `array` avec 3 éléments) correspondant à la quantité de rouge de vert et de bleu. Si l'image est en niveau de gris on aura un seul entier.

Pour ne pas perdre les données de la matrice initiale, on ne travaillera jamais sur celle-là, on va en créer une nouvelle ayant les mêmes caractéristiques (hauteur, largeur, nombre de couleurs) que la première.

```
pixels_res = np.zeros_like(pixels) # Une matrice qu'avec des zéros
pixels_res = np.copy(pixels) # Une copie en profondeur
```

Ensuite pour modifier les pixels on peut par exemple procéder ainsi

```
pixels[i,j]=[255,255,255]
# Le pixel de coordonnées (i,j) est colorié en blanc (image en couleur).
pixels[i,j]=255
# Le pixel de coordonnées (i,j) est colorié en blanc (image en niveau de gris).
```

Une fois toutes les manipulations faites il faut transformer la matrice en une image et l'enregistrer sur le disque.

```
img_res = Image.fromarray(pixels_res)
img_res.save("nouveau_fichier.extension")
# enregistre la nouvelle image dans le fichier nouveau_fichier.extension
```

Exemple 10.2. Voici l'exemple d'une fonction qui permet de créer l'image miroir d'une autre image (pour cela, il « suffit » d'inverser l'ordre des colonnes de la matrice des pixels de l'image de départ) :

```
def miroir(fichier):
    """prend en entrée un fichier image et retourne l'image de l'image
     obtenue par une réflexion par rapport à l'axe vertical droit
    """
    im=Image.open(fichier)
    pixels = np.array(im)
    hauteur, largeur = pixels.shape[:2]
    pixels_res = np.zeros_like(pixels)
    for i in range(hauteur):
        for j in range(largeur):
            pixels_res[i, j,:] = pixels[i, largeur - 1 - j,:]
    img_res = Image.fromarray(pixels_res)
    img_res.save("miroir"+fichier)
    img_res.show()
```

Après avoir exécuter la fonction, on l'appelle de la manière suivante.

```
>>> miroir("valleluna.jpg")
```

On obtient alors quelque chose qui ressemble à ce qui suit.

Photo initiale

Photo après transformation

Au cours du TP la plupart des fonctions que vous allez devoir écrire auront une structure quasi identique à la fonction donnée dans l'exemple.

III Traitement d'une image pixel par pixel

3.1 Passage en niveau de gris

Exercice 10.1

On souhaite convertir une image couleur en niveaux de gris. Le plus « naturel » est de prendre la moyenne des trois valeurs R, G et B (qu'on appelle *luminance*), mais ce n'est pas toujours ce qui donne le meilleur résultat. 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, on ne prend généralement pas la moyenne des composantes, mais une moyenne pondérée. Classiquement dans les logiciels de retouches la pondération des trois valeurs R, G et B est faite de la façon suivante `coeff_R=0,299`, `coeff_G=0,587`, `coeff_B=0,114`

On va donc écrire une fonction `gris("nom_fichier", coeff)` où `coeff` sera une liste composée de 3 éléments, qui renvoie une version en niveau de gris de l'image initiale, calculée en prenant la moyenne coefficientée des composantes R, G et B.

Remarques :

- `numpy` permet des calculs pratique sur les matrices par exemple :
  - la multiplication terme à terme de deux tableaux de même taille : `[1,3,8]*[2,4,1]` renvoie `[2,12,8]`
  - la somme des éléments d'un tableau se fait avec `np.sum`
- Les valeurs que vous devez obtenir doivent être des entiers entre 0 et 255. Vous prendrez donc bien soin de faire `int` pour obtenir un entier, et vous vous interdirez dans un premier temps de mettre des coefficients négatifs.

Vous aurez à écrire quelque chose de la forme : (les `....` sont à compléter)

```
pixels_res[i, j] = .... pixels[i,j,:] .....
```

Exemple 10.3.

```
>>> gris("valleluna.jpg", [30, 59, 11])
```

`gris("valleluna.jpg", [30, 59, 11)]`

`gris("valleluna.jpg", [1, 1, 1])`

3.2 Négatif / Monochrome

Exercice 10.2 *Négatif*

Écrire une fonction `negatif(imsource)` qui applique la transformation $x \mapsto 255 - x$ à chacun des canaux de l'image.

Remarque : Si `matrice` est un tableau `numpy` (de type `array`) alors `matrice + 15` ajoute 15 à chaque élément de la matrice.

Exercice 10.3 *Monochrome*

Écrire une fonction `monochrome(imsource)` qui, étant donnée une image, renvoie les trois images monochromes, une rouge, une verte, une bleue, obtenue en annulant deux des trois composantes. Par exemple pour le rouge on aura un résultat de la forme `[r,0,0]`. Vous enregistrerez chacune des trois images sous des noms différents.

3.3 Luminosité, contraste

Le type de données dans les matrices obtenues à partir des images sources sont des entiers entre 0 et 255, et tous les calculs fait avec les matrices sont fait modulo 255.

Dans les prochains exercices, nous allons modifier les valeurs des pixels et potentiellement obtenir des valeurs inférieure à 0, ou supérieure à 255.

Si lors des calculs on a une valeur au delà par exemple si on obtient 270, dans le pixel c'est 14 qui apparaitra. Or pour nous la valeur la plus proche dans la plage $[\![0, 255]\!]$ c'est 255, il faudra donc forcer la valeur à 255. De même si on on obtient quelque chose de négatif c'est sa valeur modulo 255 qui est donnée par exemple si on obtient -24 c'est 232 qui est stocké. Or nous la valeur la plus proche qui nous intéresse est 0 il faut donc forcer la valeur à 0.

Exercice 10.4 *max / min*

1. Écrire une fonction `maxmin(x)` qui retourne un entier qui sera x si $x \in [\![0, 255]\!]$, 0 si $x \leqslant 0$ et 255 si $x \geqslant 255$. Pour cela, on pourra utiliser les fonctions `min(a,b)` et `max(a,b)` qui renvoient respectivement le minimum et le maximum des deux nombres `a` et `b`.
2. Écrire une fonction `maxminmat(matrice)` qui parcourt l'ensemble des éléments d'une matrice et qui renvoie `maxmin(element)`. Attention on distinguera suivant que la matrice a deux dimensions (image en niveau de gris), ou trois dimensions (image en couleur)

Exercice 10.5 *Luminosité 1*

Une première façon assez simple pour éclaircir une image est d'ajouter à chaque pixel (sur chaque canal si l'image est en mode RGB) un certain nombre d'unités. Et bien sûr s'il s'agit d'assombrir l'image il faudra enlever un certain nombre d'unités à chaque pixel.

Écrire une fonction `luminosite1(imsource,unite)` qui « ajoute » le nombre d'unité `unite` sur chaque composante du pixel, où `unite` sera un entier relatif appartenant à $[\![-255, 255]\!]$. On fera attention à ce que la valeur obtenue pour chaque pixel ne dépasse pas 255, et soit positive.

Remarque : À propos des lignes suivantes sur votre fichier `.py`

```
pixels = pixels.astype(int)
```

permet de ne plus travailler modulo 255 et ainsi d'obtenir des valeurs supérieures à 255 et inférieures à 0.

```
pixels_res=pixels_res.astype(np.uint8)
```

permet de revenir modulo 255 pour pouvoir ensuite transformer la matrice en une image.

Exercice 10.6 *Luminosité / Contraste*

1. *Luminosité*
   Une autre façon de procéder, un peu moins archaïque, est d'appliquer une fonction dite *filtre* à chaque pixel. Par exemple pour éclaircir l'image la fonction à appliquer aura les propriétés suivantes :

   $$\star\ f([0,1]) = [0,1] \quad \text{et} \quad f(0) = 0 \quad \text{et} \quad f(1) = 1 \qquad \star \forall x \in [0,1],\ \text{on a } f(x) \geqslant x$$

   a. Quelles sont les caractéristiques d'une fonction filtre qui assombris l'image ?

   b. Parmi les fonctions usuelles lesquelles pourrait-on utiliser pour éclaircir ou assombrir une image ?

2. *Contraste*

Pour augmenter le contraste, une idée est d'éclaircir les pixels les plus clairs et d'assombrir les pixels les plus foncés. On peut donc procéder comme précédemment en appliquant une fonction filtre ayant certaines propriétés. Par exemple

$$\star f([0,1]) = [0,1] \quad \text{et} \quad f(0) = 0,\ f(1) = 1 \quad \text{et} \quad f(0,5) = 0,5$$

$$\star \quad \forall x \in]0,0,5[,\ f(x) < x \quad \text{et} \quad \forall x \in]0,5,1[,\ f(x) > x$$

En étudiant les fonctions suivantes sur $[0,1]$, montrer qu'elles ont toutes les propriétés demandées.

$$f(x) = 3x^2 - 2x^3 \qquad g(x) = x^3(6x^2 - 15x + 10)$$

3. Une des propriétés qui vous a été donnée sur les fonctions est qu'elle sont définies sur $[0,1]$. Mais les pixels des images sont un (ou trois) nombre(s) entier(s) compris entre 0 et 255. Il faut donc modifier la fonction filtre proposée pour qu'elle agisse bien sur des entiers et renvoie des entiers.

Sur le modèle ci-dessous

```
1 def filtre_puissance(x,p):
2     return int(255*(x/255)**p)
```

créer deux fonctions `filtre_f(x,p)` et `filtre_g(x,p)` qui font ce qu'on attend. La variable `p` n'interviendra pas dans la fonction, mais nous en avons besoin pour simplifier la suite.

4. Écrire une fonction `lum_contraste(imsource,fct,param)` qui fera ce qu'on attend d'elle.

On pourra faire appel à la fonction de la manière suivante :

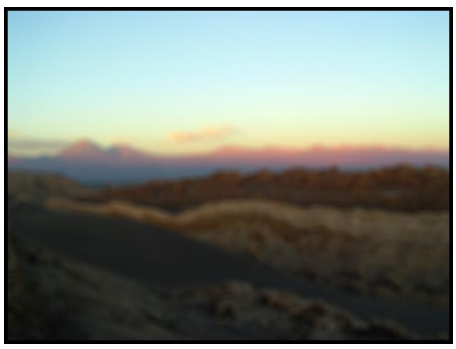
```
>>> lum_contraste("valleluna.jpg", filtre_puissance, 0.5)
```

Pour les fonctions `filtre_f` et `filtre_g` on pourra mettre une valeur quelconque pour le paramètre lors de l'appel de la fonction.

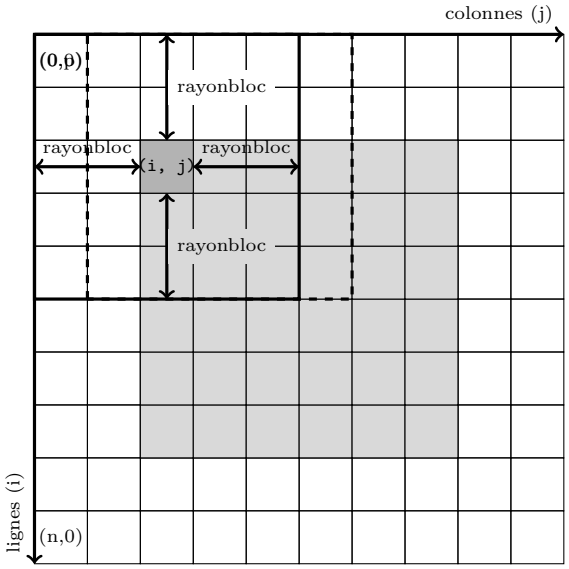
IV Traitement d'une image pixel par bloc de pixels

4.1 Floutage, filtres par convolution

Pour flouter une image la technique la plus simple consiste à remplacer la valeur de chaque pixel par une valeur moyenne calculée sur une petite fenêtre (3×3 typiquement ou 5×5) centrée autour de ce pixel.



Floutage



Si on veut, on peut décider de na pas donner le même poids aux pixels qui entourent le pixel sur lequel on travaille. Autrement dit on peut faire une moyenne coefficientée. Ce calcul est basée sur une matrice qui varie suivant l'effet que l'on veut obtenir. ON dit qu'on fait une convolution.

Prenons par exemple la matrice

$$M = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix},$$

pour la composante rouge, on va remplacer la valeur r d'un pixel par la moyenne coefficientée des valeurs r pour les 8 pixels alentour et le pixel lui même, affectées des coefficients donnés par la matrice. Et on fera pareil ensuite pour la composante verte et la composante bleue.

Le floutage évoqué précédemment correspond alors à une matrice de taille $3\times 3,\ 5\times 5\ldots$ composée uniquement de 1.

Exercice 10.7

1. Écrire une fonction `moyenne_bloc(pixels, i, j, matrice)` où `pixels` est la matrice des pixels de l'image, `(i,j)` les coordonnées de l'endroit de la matrice où on fait la modification et `matrice` est la matrice filtre qu'on va utiliser.
   - ⋆ Il vous faudra distinguer suivant que l'image est en couleur ou en niveau de gris pour opérer sur chacune des composantes.
   - ⋆ Comme vu un peu plus haut `numpy` permet des calculs rapides. Par exemple si M et N sont deux matrices de même taille `M*N` fait le produit terme à terme des deux matrices.
   - ⋆ Pour faire la somme des éléments d'une matrice vous pourrez utiliser `np.sum`
   - ⋆ Si la somme des coefficients de la matrice filtre est non nulle, on divise le résultat de la convolution par cette somme (moyenne coefficientée) ceci permet de conserver la même luminance globale. Si la somme des coefficients vaut 0, on divise par 1.
2. Écrire une fonction `applique_filtre(imsource, matrice)` qui permet d'appliquer un des filtres proposés dans le fichier `.py` à une image.
   - ⋆ On se servira bien évidemment de la fonction de la question précédente.
   - ⋆ Pour que le bloc ne sorte pas des limites de l'image source, on ne prendra comme centre des blocs que les pixels à une distance `rayonbloc` des bords (zone grisée sur la figure)

Remarque : Si on effectue la différence entre l'image initiale et l'image floutée on a tendance à accentuer les détails. Par exemple avec la matrice M donnée plus haut on obtient un flou dit gaussien, donc avec la matrice

$$A=\begin{pmatrix}0&0&0\\0&20&0\\0&0&0\end{pmatrix}-M=\begin{pmatrix}-1&-2&-1\\-2&16&-2\\-1&-2&-1\end{pmatrix}$$

on obtient une image « plus nette ».

4.2 Pixellisation

Exercice 10.8

Pour pixelliser une image, l'image est divisée en rectangles de la taille spécifiée (par exemple 10×10 pixels). Chaque rectangle est ensuite rempli avec la couleur moyenne de la zone. Si le côté du bloc n'est pas un diviseur la hauteur et/ou de la largeur de l'image, certaines bandes sur les bord de l'image ne seront pas traitées.

Écrire une fonction `pixellisation(imsource, taille_bloc)` qui, étant donnés une image, et le côté d'un bloc, affiche l'image pixellisée.

On pourra utiliser la fonction `moyenne_bloc` *donnée plus haut, ou* `np.mean(matrice)` *qui donne la moyenne des éléments d'une matrice.*

V D'autres exercices

5.1 Seuillage et détection de contour

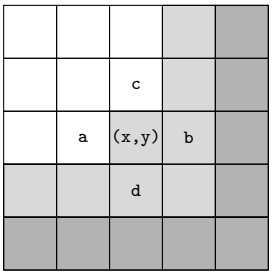
Nous allons travailler ici sur des images en niveau de gris. Dans vos programme si l'image donnée en entrée n'est pas en niveau de gris on prendra soin de d'abord la transformer en niveau de gris.

Exercice 10.9 *Seuillage*

Un filtre de seuillage consiste à mettre à 0 (noir) tous les pixels ayant une valeur inférieure à un certain seuil et à 255 les autres. Écrire une fonction `seuil(imsource,s)` qui fait ce qu'on attend d'elle.

Exercice 10.10 *Détection de contour*

Considérons à nouveau dans cet exercice, une image en niveaux de gris. Les bords des objets correspondent à des zones où les valeurs des pixels changent rapidement. C'est le cas par exemple lorsque l'on passe d'un objet clair (avec des valeurs grandes) à un arrière plan sombre (avec des valeurs petites). Afin de savoir si un pixel en (x,y) est le long d'un bord d'un objet, on prend en compte les valeurs des pixels de ces quatre voisins a, b, c et d (deux horizontalement et deux verticalement).



On calcule une valeur ℓ suivant la formule : $\ell = \sqrt{(a-b)^2 + (c-d)^2}$

On peut remarquer que si $\ell = 0$, alors on a $a = b$ et $c = d$: le pixel n'est pas sur un bord. Au contraire, si ℓ est grand, ceci signifie que les pixels voisins ont des valeurs très différentes : le pixel central est donc probablement sur le bord d'un objet.

On transforme ensuite l'image en affichant du blanc lorsque ℓ est faible et du noir lorsque ℓ est élevé ; il y a donc une valeur de seuil à déterminer ...

Écrire une fonction `contour(imsource,seuil)` qui fait ce qu'on attend d'elle. On testera cette fonction avec diverses valeurs de seuil.

On pourra ne pas tenir comptes des pixels au bord de l'image.

5.2 Transformations géométriques

Exercice 10.11 *Quart de tour direct*

On fait subir à l'image initiale une rotation d'un quart de tour direct.

1. Si on a une image de dimension $n \times p$ quelle sera la dimension de l'image obtenue après la transformation ?
2. Quelles étaient les coordonnées initiales du pixel situé en $(0, 0)$ sur l'image finale ?
3. De manière générale, quelles étaient les coordonnées initiales du pixel situé en (i, j) sur l'image finale ?
4. Écrire une fonction `quart_tour_direct(imsource)` qui fait ce que son nom indique.

Exercice 10.12 *Réduction*

Pour réduire une image source de dimension `(L, H)` d'un coefficient `coef` qui divise L et H, il suffit de créer une image de dimension `(L // coef, H // coef)` puis pour chaque pixel `pixel_res[i, j]` de cette image reçoit la valeur du pixel `pixels[i * coef, j * coef]` de l'image source.

1. Compléter le code de la fonction `reduction(imsource, coef)`.
2. Comment modifier cette fonction pour un agrandissement ? Le résultat sera-t-il satisfaisant ?

5.3 Stéganographie

Exercice 10.13 *Stéganographie*

La stéganographie est l'art de la dissimulation : son objet est de faire passer inaperçu un message dans un autre message. Nous allons mettre en œuvre ici un procédé qui permet de cacher des informations dans une image, et plus particulièrement une autre image (mais on peut aussi y dissimuler un texte par exemple).

Voici l'idée directrice : On a vu que chaque pixel d'une image en niveaux de gris peut-être codé par un entier entre 0 et 255. Pour plus de clarté, imaginons, momentanément que les pixels sont codés par des entiers entre 0 et 99. On dispose de deux images de même taille, l'une « anodine » et l'autre « secrète ». On va combiner les pixels des deux images situés à la même place de la façon suivante :

- ⋆ Le pixel de l'image anodine est $a_1a_0 = 10a_1 + a_0$, celui de l'image secrète est : $s_1s_0 = 10s_1 + s_0$
- ⋆ On garde la dizaine du pixel de chacune des deux images : a_1 et s_1
- ⋆ On crée un pixel $a_1s_1 = 10a_1 + s_1$: ainsi à l'affichage, ce pixel est proche du pixel initial : l'image visible est très proche de l'image anodine du départ mais contient tout de même une information sur l'image secrète.
- ⋆ Pour révéler, l'image secrète : on prend chaque pixel, on supprime la dizaine, on obtient ainsi l'entier s_1, on crée le pixel pour l'image révélée : $10s_1$, qui est donc proche du pixel de l'image secrète.

Adaptons donc cette méthode dans notre cas où les pixels sont en fait codés avec des entiers 8 bits. Un pixel est codé par un entier $n = \sum_{i=0}^{7} b_i 2^i$. La partie de poids fort à garder (l'équivalent de la dizaine) est $m = \sum_{i=4}^{7} b_i 2^i = (n//2^4)$.

On construit le pixel de l'image codée par $p = 2^4 \times m_a + m_s$ (a pour anodin, s pour secret). On décode l'image en extrayant m_s de p en faisant : $m_s = p\%2^4$.

1. Écrire une fonction qui extrait une image secrète d'une image codée : la tester sur l'image `mystere.png`.
2. Écrire une fonction qui cache une image secrète dans une image anodine. La tester sur les deux images de même dimension et vérifier avec votre fonction précédente.
3. Faire ce même travail pour des images en couleurs.