TP 05 : Traitement d'images sous Python

Objectifs du TP

- Manipuler des tableaux NumPy
- Comprendre un format simple de stockage d'image
- Appliquer des techniques de transformation par masques
- Cacher de l'information dans une image

I Les images informatiques

Format matriciel

Une manière de représenter informatiquement une image, est de l'échantillonner en petits éléments (*picture element= pixel*) supposés de couleur uniforme, suivant un quadrillage.

L'image est alors représentée informatiquement par une matrice de couleurs (*bitmap*), chaque case de la matrice donnant la couleur du pixel correspondant du quadrillage. Les formats les plus connus d'images bitmap sont JPEG, BMP, GIF, TIFF et PNG. Certains formats, comme JPEG ou GIF, sont compressés (la compression JPEG repose sur une variante de la Transformée de Fourier, accompagnée d'un filtre passe-basses fréquences spatiales). Certains gèrent la *transparence* (GIF, PNG), en ajoutant un canal *alpha*.

La *résolution* de l'image dépend bien sûr de la finesse du quadrillage, c'est-à-dire de la taille de la matrice de couleurs. En zoomant suffisamment, on voit toujours apparaître la trame sous la forme d'une mosaïque de carrés de couleur unie (les pixels).

On convient de numéroter les colonnes en partant de la gauche, et les lignes en partant du haut de l'image. Un pixel de la trame est repéré par un couple (i, j) où i et j sont respectivement les numéros de la ligne et de la colonne à l'intersection desquelles se trouve le pixel.

Les couleurs

Selon le format de l'image bitmap, la palette de couleurs est encodée sur un certain nombre de bits, appelé *profondeur* de l'image (exprimée en bpp = bits par pixel).

- Dans le cas d'une image en noir et blanc, chaque couleur (noir ou blanc) est encodée sur 1 bit, par exemple, on peut convenir que le bit 0 encode le noir et le bit 1 encode le blanc.
- Dans le cas des images en niveau de gris, chaque couleur (du noir au blanc en passant par des gris plus ou moins foncés) est encodée sur 1 octet.

  Donc 256 nuances de gris pour ces images ...
- On peut également encoder une image couleur avec 1 octet pour chaque pixel. Chaque valeur entre 0 et 255 représente une couleur de la palette.
- Nous considérerons ici des images couleur au format RGB (Red, Green, Blue). La couleur de chaque pixel est obtenue par synthèse additive (Rouge + Bleu + Vert = Blanc).

  On utilise 1 octet pour la quantité de rouge, 1 octet pour la quantité de vert, et 1 octet pour la quantité de bleu, si bien qu'une couleur est représentée sur 3 octets = 24 bits. En pratique, on donne un triplet de nombres entre 0 et 255.

rouge	vert	noir	blanc	cyan	kaki
(255,0,0)	(0,255,0)	(0,0,0)	(255,255,255)	(0,255,255)	(148,129,43)

On peut aussi donner ces 3 octets sous forme de 6 chiffres hexadécimaux (FF0000 pour le rouge, 94812B pour le kaki, ...)

Les images qui gèrent la transparence sont encodées en RGBA (A pour Alpha). Un quatrième octet est alors utilisé pour encoder la transparence : de 0 pour un pixel totalement transparent à 255 pour un pixel totalement opaque. Il faut alors 32 bits pour chaque pixel.

1. Combien de couleurs sont disponibles en format RGB ? Pourquoi se limiter à un seul octet par couleur primaire ?

II Manipulations

Ouverture d'une image

On commence par importer des modules utiles pour la suite.

```
import os
import numpy as np
import matplotlib.pyplot as plt
from skimage import data
```

— Placer avec l'explorateur Windows vos images de travail (au format png, pas trop grosses) dans le répertoire de travail de Python, que l'on peut visualiser en tapant dans la console `os.getcwd()` (*Get Current Working Directory*).

— Ouvrez votre image par `plt.imread('image.png')`, qui renvoie un tableau NumPy.
Le tableau NumPy d'une image a 2 ou 3 dimensions. Les images de profondeur 256 bpp n'ont que 2 dimensions (chaque pixel est encodé par un nombre entre 0 et 255), tandis que les images RGB et RGBA ont 3 dimensions (chaque pixel est encodé par 3 ou 4 nombres entre 0 et 255).
- la première dimension est la hauteur de l'image (première coordonnée = n° de ligne du pixel i)
- la deuxième dimension est la largeur de l'image (2ème coordonnée = n° de colonne j)
- la troisième dimension éventuelle représente le triplet RGB associé au pixel (i, j)

— Sinon, vous pouvez travailler avec des images générées par `data` : `data.coins()`, `data.lena()`, `data.coffee()`, ... certaines sont en niveaux de gris, d'autres en couleur. La méthode retourne alors directement un tableau NumPy.

— Déterminez la dimension de votre image avec la méthode `shape` de NumPy, ainsi que le type des éléments du tableau NumPy par la méthode `dtype`.
On manipulera dans ce TP deux types d'éléments : `np.float32` (flottant) et `np.uint8` (entier non signé codé sur 8 bits).

— Affichez l'image par `plt.imshow(image)`. Si vous travaillez dans une console Ipython (disponible dans Spyder), l'affichage se fait "inline", dans la console (pratique pour ce TP), sinon il faut afficher dans une fenêtre graphique externe par le traditionnel `plt.show()`.

Création d'images simples

Pour modifier une image, il suffit de modifier le tableau associé, ou de créer un nouveau tableau de même dimension que l'on remplit par la suite.

On rappelle quelques commandes Numpy de création de tableaux :

```
# création d'un tableau à 10 lignes et 20 colonnes rempli de zéros
# éléments de type entier non signé sur 8 bits
# par défaut, éléments du type float64
A = np.zeros( (10,20) , dtype = np.uint8)    # argument = tuple

B = np.copy(A)  # copie

 # création d'un tableau Numpy à partir d'une liste de listes
C = np.array([ [1,2] , [3,4] , [5,6] ], dtype = ..)

# pour enlever les axes de plot
plt.axis('off')
```

Les données insérées dans un tel tableau sont automatiquement transtypées dans le type déclaré du tableau.

2. Dessiner le drapeau français. On impose un rapport largeur/hauteur de 3 :2.
   Examinez l'effet de l'option

   `plt.imshow(image, interpolation = 'nearest')`

   dans l'affichage du drapeau avec un petit nombre de pixels.
3. Maintenant, le drapeau japonais !
4. et le Brésil pour finir ☺

Manipulations géométriques

5. Écrire une fonction `quadrant(image, i, j)` qui renvoie l'un des quatre sous-quadrants de l'image : $(i, j) = (1,2)$ correspondra au quart supérieur droit.
6. Écrire une fonction `symetrie(image)` qui renvoie l'image symétrisée par rapport au plan médiateur vertical.
7. Écrire une fonction `rotation(image)` qui renvoie l'image tournée d'un quart de tour dans le sens direct.

Manipulations sur les couleurs

8. Écrire une fonction `composante(image, k)` qui retire la composante rouge ($k = 0$), verte ($k = 1$) ou bleue ($k = 2$) de l'image.
9. Écrire une fonction `primaire(image, k)` qui renvoie uniquement la composante choisie de l'image.
10. Conversion en niveaux de gris
    On définit la *luminance* d'un pixel (R, G, B), sensation visuelle associée à la luminosité, par :

$$L = R \times \frac{299}{1000} + G \times \frac{587}{1000} + \frac{114}{1000}$$

    Commenter cette expression. Écrire une fonction `gris(image)` qui convertit une image couleur en image sur 256 niveaux de gris.
    Affichez l'image ... grosse déception ?
    → par défaut, `matplotlib` utilise une palette de couleurs (ou *colormap*) `'jet'` qui fabrique des fausses couleurs à partir de données monochromes. Il suffit pour visualiser correctement le tableau d'utiliser :

```
plt.imshow(image, cmap = 'gray')
```

Il existe d'autres palettes prédéfinies : "spectral", "hot", ...

Créer une version en niveaux de gris de votre image couleur à utiliser par la suite.

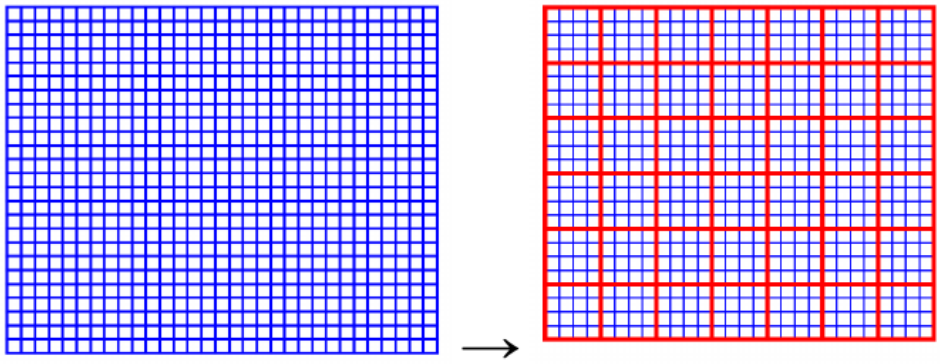
11. Quelle est la complexité des opérations de traitement d'image que nous avons exécuté ?

III Transformations plus avancées

Pixellisation

Pour réduire la résolution d'une image trop volumineuse, on peut la dégrader en la pixellisant : l'opération consiste à regrouper des pixels voisins et à les remplacer par un seul pixel moyen.

On regroupe ainsi des carrés de p lignes par p colonnes. La moyenne est donc réalisée sur p^2 pixels.



12. Écrire une fonction `pixellisation(image, p)` prenant en argument un tableau NumPy et renvoyant une image de dimension plus faible pixellisée.
    On affichera le résultat avec l'option `interpolation = 'nearest'`.
    On pourra utiliser la fonction `np.mean()` et des *slicings* pour calculer simplement des moyennes.

Lissage

On parle aussi de floutage, ou de filtre moyenneur.

13. Remplacer chaque pixel par la moyenne de lui-même et de ses 8 voisins (attention aux bords !).
    Appliquer plusieurs fois le processus, afin d'accentuer l'effet, peu perceptible au départ.

On a en fait appliqué le *masque* $\frac{1}{9}\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$ à l'image.

Détection de contours

On part de l'idée que les contours dans une image correspondent à de brusques changements de luminosité.

Pour étudier les changements de luminance, on pourrait remplacer chaque pixel $I_{i,j}$ par

$$(I_{i+1,j} - I_{i,j})^2 + (I_{i,j+1} - I_{i,j})^2$$

On applique ainsi séparément les masques $\begin{pmatrix}0&0&0\\0&-1&1\\0&0&0\end{pmatrix}$ et $\begin{pmatrix}0&1&0\\0&-1&0\\0&0&0\end{pmatrix}$, puis on calcule la norme 2 du résultat.

Ceci est l'analogue discret de la norme N du vecteur gradient :

$$\left(\frac{\partial f}{\partial x}(x,y)\right)^2 + \left(\frac{\partial f}{\partial y}(x,y)\right)^2$$

En effet, $\dfrac{\partial f}{\partial x}(x,y) = \dfrac{f(x+h,y) - f(x,y)}{h} + o(1)$

14. Écrire une fonction `contour(image, seuil)` basée sur ce principe, qui prend comme argument une image convertie en 256 niveaux de gris et renvoie une image noir et blanc (2 couleurs) de même dimension, avec accentuation de contours. `seuil` est un seuil compris entre 0 et 1 à ajuster manuellement.

On peut obtenir une meilleure approximation du gradient par la méthode des *différences finies centrées* (déjà utilisée en Sup pour calculer numériquement une dérivée de fonction). En effet,

$$\begin{cases} f(x+h,y) = f(x,y) + h\dfrac{\partial f}{\partial x}(x,y) + \dfrac{h^2}{2}\dfrac{\partial^2 f}{\partial x^2}(x,y) + o(h^2) \\ f(x-h,y) = f(x,y) - h\dfrac{\partial f}{\partial x}(x,y) + \dfrac{h^2}{2}\dfrac{\partial^2 f}{\partial x^2}(x,y) + o(h^2) \end{cases}$$

d'où

$$\boxed{\frac{\partial f}{\partial x}(x,y) = \frac{f(x+h,y) - f(x-h,y)}{2h} + o(h)}$$

15. Transposer dans le domaine discret la détermination de la quantité N par cette méthode. Quels masques applique-t-on au tableau ?
    Écrire une fonction `contour_dfc(image, seuil)` qui met en œuvre cette technique. Constate-t-on une amélioration de la détection de contours ?
16. En fait, il est encore meilleur de faire une moyenne autour du point en même temps : c'est le filtre de Sodel, qui utilise les masques $\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$ et $\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$.
    Écrire une fonction `contour_sobel(image, seuil)`. Comparer aux détections précédentes, notamment sur la sensibilité du choix du seuil.

Agrandissement

Si vous avez le temps, en fin de TP, vous pouvez essayer d'agrandir une image.. bien sûr ce serait mieux d'interpoler un peu les valeurs pour éviter des effets d'escalier !

IV Stéganographie

Principe : une image ou un texte cachée dans une autre image

Alice veut faire parvenir un message secret à Bob, sans qu'Oscar ne puisse le lire. Elle peut pour cela utiliser un procédé cryptographique connu de tout le monde, dont la sécurité repose sur une clé secrète.

Avec l'approche stéganographique, le secret réside dans le procédé lui-même. Ainsi, si la cryptographie est l'art du secret, la stéganographie est l'art de la dissimulation. Autrement dit, on va dissimuler des données dans d'autres données. Ici, dans des images, de façon imperceptible pour l'œil.

Sur le site de la classe, (ou sur une clé USB, à la demande), vous pouvez trouver une image `mystere1.png`. On a utilisé le principe suivant :

Les 4 bits de poids faible (peu importants) de l'image masquante ont été remplacés par les 4 bits de poids fort (suffisants pour voir l'image cachée) du pixel correspondant dans l'image cachée. Autrement dit, on consent à perdre un peu d'information sur l'image masquante pour cacher de l'information importante permettant de reconstituer une autre image.

Par exemple, si la composante verte du pixel ligne 345, colonne 1021 du fichier `chapeau.png` est

$$10100100_2 = 164_{10}$$

et que la composante verte du pixel ligne 345, colonne 1021 du fichier `lapin.png` est

$$01111101_2 = 125_{10}$$

alors si on veut cacher un lapin dans le chapeau, la composante verte du pixel (345, 1021) du fichier `magie.png` est

$$10100111_2 = 167_{10}$$

La personne qui connaît le procédé pourra retrouver le lapin ...

Cherchez l'intrus !

On donne une fonction de conversion de tableaux d'images de type `np.float32` vers des tableaux de type `np.uint8`. En effet, certaines images ont été converties en flottant par simple sauvegarde ou lecture ...

```
def convert(image):
    """convertit image float32 en uint8"""
    def conv(x):
        return np.uint8(x*255)
    conv_v = np.vectorize(conv)
    return conv_v(image)
```

Assurez-vous de bien manipuler un tableau de type `np.uint8` !

17. Comment obtenir simplement l'écriture décimale des k bits de poids faible de l'entier x représenté par un octet ? et des ℓ bits de poids fort ?
18. Écrire une fonction `appauvrir(image, n)` qui conserve $n \leq 8$ bits (en partant du plus fort) des 8 bits associés à chaque couleur d'une image.
    À partir de quand voit-on la différence à l'œil nu ?
19. Écrire une fonction `devoiler(image)` qui permet de voir ce qui se cache dans la jungle de `mystere.png`...
20. De même, on peut cacher un texte dans une image : on peut encoder les caractères sur un octet grâce au code ASCII ... En pratique, cela peut être utilisé pour tatouer numériquement une image (*watermark*) pour des copyrights.
    Un texte est caché dans la première ligne de l'image `mystere2.png`, écrit de gauche à droite. Chaque caractère est caché pour moitié dans la composante rouge d'un pixel, et pour moitié dans la composante verte. Par exemple, si on veut cacher un `a` de code ASCII

    $$97_{10} = 01100001_2$$

    dans le pixel RGB

    $$(154_{10}, 76_{10}, 200_{10}) = (10011010_2, 01001100_2, 11001000_2)$$

    alors on enverra

    $$(10010110_2, 01000001_2, 11001000_2) = (150_{10}, 65_{10}, 200_{10})$$

    Retrouver le texte caché dans `mystere2.png`. La fonction `chr(n)` permet de renvoyer le caractère ASCII associé à l'entier n.

A FAIRE

I Les images informatiques

Les couleurs

1. $2^{24} = 16\ 777\ 216$ (presque 17 millions de couleurs, plus que l'œil ne peut discerner)

II Manipulations

Ouverture d'une image

```
img = plt.imread('got.png')
print( img.shape, img.dtype )
```

Création d'images simples

```
# Un drapeau français..

blanc = [255, 255, 255]
bleu = [0, 0, 255]
rouge = [255, 0 , 0]

L = 30  # observer l'effet de la réduction
hauteur = 20
France = [bleu]*L + [blanc]*L + [rouge]*L
France = [France]*hauteur
France = np.array(France, dtype= np.uint8)
plt.axis('off')
plt.imshow(France, interpolation='nearest')

#%% Un drapeau japonais

largeur, hauteur = 60, 40
Japon = 255 * np.ones((hauteur, largeur, 3),dtype=np.uint8)
# drapeau blanc, on dessine le disque rouge

x = largeur // 2
y = hauteur // 2
r = int(y*3/5)

for i in range(hauteur):
    for j in range(largeur):
        if (j-x)**2 + (i-y)**2 < r**2:
            Japon[i,j] = rouge

plt.axis('off')
plt.imshow(Japon, interpolation='nearest')
```

Manipulations géométriques

2.
```python
def quadrant(img, i, j):
    # i, j = 1 ou 2
    x_dim, y_dim = img.shape[1], img.shape[0]
    plt.imshow( img[(i-1)*y_dim//2:i*y_dim//2, (j-1)*x_dim//2:j*x_dim//2])
```

3. La fonction effectue une symétrie verticale ou horizontale.

```python
def symetrie(img, direction):
    # renvoie nouvelle image sans détruire précédente
    x_dim, y_dim = img.shape[1], img.shape[0]
    new_img = np.copy(img)
    if direction == 0:  # horizontal
       for j in range(x_dim):
           for i in range(y_dim):
               new_img[i,j] = img[i, x_dim-j-1]
    elif direction == 1:  # vertical
        for i in range(y_dim):
           for j in range(x_dim):
               new_img[y_dim-1-i,j] = img[i, j]
    return new_img
```

4.
```python
def rotation(img):
    """rotation d'un quart de tour anti-horaire"""
    x_dim, y_dim = img.shape[1], img.shape[0]
    new_img = np.zeros((x_dim, y_dim, 3), dtype=np.float32)
    for i in range(y_dim):
        for j in range(x_dim):
            new_img[x_dim-1-j,i] = img[i, j]
    return new_img
```

Manipulations sur les couleurs

```python
#%% Affichage sans composante k
def composante(img,k):
    x_dim, y_dim = img.shape[1], img.shape[0]
    new_img = np.copy(img)
    for i in range(y_dim):
        for j in range(x_dim):
            new_img[i,j,k] = 0

    plt.imshow(new_img)
    return new_img

# Affichage une seule couleur
def primaire(img,k):
    x_dim, y_dim = img.shape[1], img.shape[0]
    new_img = np.copy(img)
    for i in range(y_dim):
        for j in range(x_dim):
```

```
            for l in range(3):
                if l != k:
                    new_img[i,j,l] = 0

    plt.imshow(new_img)
    return new_img

#%% RGB vers gris, luminance
def luminance(L):
    R, G, B = L
    return R*299/1000 + G*587/1000 + B*114/1000

def gris(img):
    x_dim, y_dim = img.shape[1], img.shape[0]
    new_img = np.zeros((y_dim, x_dim), dtype=np.uint8)

    for i in range(y_dim):
        for j in range(x_dim):
            R, G, B = img[i,j,:]
            L = luminance((R, G, B))
            new_img[i,j] = L * 255

    plt.imshow(new_img, cmap = 'gray')
    # sinon colormap "jet" (fausses couleurs)
    return new_img
```

Les fonctions sont de complexité linéaire par rapport au nombre de pixels (heureusement...)

III Transformations plus avancées

Pixellisation

```
def pixellisation(img,p):
    x_dim, y_dim = img.shape[1], img.shape[0]
    new_xdim, new_ydim = x_dim // p, y_dim // p
    # on détruit les pixels qui dépassent
    new_img = np.zeros((new_ydim, new_xdim,3), dtype=np.float32)
    for i in range(new_ydim):
        for j in range(new_xdim):
            new_img[i,j] = [np.mean( img[i*p:(i+1)*p, j*p:(j+1)*p, k] ) for k in
range(3) ]
    plt.imshow(new_img, interpolation='nearest')
    return new_img
```

Lissage

```
def lissage(img):
    x_dim, y_dim = img.shape[1], img.shape[0]
    new_img = np.copy(img)
```

```
    # on conserve les bords
    new_img[0,:] = img[0,:,:3] # première ligne
    new_img[-1,:] = img[-1,:,:3] # dernière ligne
    new_img[:,0] = img[:,0,:3] # première colonne
    new_img[:,-1] = img[:, -1,:3]  # dernière colonne

    for i in range(1, y_dim-1):
        for j in range(1, x_dim-1):
            new_img[i,j] = [ np.mean( img[i-1:i+2, j-1:j+2,k]) for k in
range(3)]
    plt.imshow(new_img)
    return new_img
```

Détection de contours

```
def contour(img, seuil):
    """img est une image en niveaux de gris, p_seuil est un pourcentage de
seuil"""
    y_dim, x_dim = np.shape(img)
    G = np.zeros((y_dim, x_dim))

    for i in range(y_dim-1):
        for j in range(x_dim-1):
            G[i,j] = (img[i+1,j]-img[i,j])**2 + (img[i,j+1]-img[i,j])**2
    G_max = np.amax(G)
    v_seuil = seuil * G_max

    contour = np.zeros((y_dim, x_dim))
    for i in range(y_dim-1):
        for j in range(x_dim-1):
            if G[i,j] > v_seuil:
                contour[i,j] = 255
    plt.imshow(contour, cmap = 'gray')
    return contour
```

Pour la méthode de différence finie centrée, on doit alors calculer

$$(I_{i+1,j} - I_{i-1,j})^2 + (I_{i,j+1} - I_{i,j-1})^2$$

c'est-à-dire appliquer séparément les masques $\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$ et $\begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, puis calculer la norme 2.

```
def contour_dfc(img, seuil):
    y_dim, x_dim = np.shape(img)
    G = np.zeros((y_dim, x_dim))

    for i in range(1,y_dim-1):
        for j in range(1,x_dim-1):
```

```
            G[i,j] = (img[i+1,j]-img[i-1,j])**2 + (img[i,j+1]-img[i,j-1])**2
    G_max = np.amax(G)
    v_seuil = seuil * G_max

    contour = np.zeros((y_dim, x_dim))
    for i in range(1,y_dim-1):
        for j in range(1,x_dim-1):
            if G[i,j] > v_seuil:
                contour[i,j] = 255
    plt.imshow(contour, cmap = 'gray')
    return contour

def contour_sobel(img, seuil):
    y_dim, x_dim = np.shape(img)
    G = np.zeros((y_dim, x_dim))

    for i in range(1,y_dim-1):
        for j in range(1,x_dim-1):
            S1 = -img[i-1, j+1] + img[i+1,j+1] + 2*(img[i+1,j]-img[i-1,j]) +
img[i-1,j+1]-img[i-1,j-1]
            S2 = img[i-1,j-1]-img[i+1,j-1] + 2*(img[i-1,j]-img[i+1,j]) +
img[i-1,j+1]-img[i+1,j+1]
            G[i,j] = S1**2 + S2**2
    G_max = np.amax(G)
    v_seuil = seuil * G_max

    contour = np.zeros((y_dim, x_dim))
    for i in range(1,y_dim-1):
        for j in range(1,x_dim-1):
            if G[i,j] > v_seuil:
                contour[i,j] = 255
    plt.imshow(contour, cmap = 'gray')
    return contour
```

IV Stéganographie

Cherchez l'intrus !

Pour obtenir l'écriture décimale des k bits de poids faible de l'entier x représenté par un octet, il suffit de considérer le reste dans la division euclidienne de x par 2^k. Pour les ℓ bits de poids fort, le quotient dans la division euclidienne de x par $2^{8-\ell}$.

On peut aussi faire des opérations directement sur les bits :

```
x = 187
print(x, bin(x))
y = x>>(8-5)  # 5 bits de poids fort
print(y, bin(y))
z = (x>>2)<<2 # 2 bits de poids faible mis à zéros
t = x-z
print(t, bin(t))
```

```python
def appauvrir(img,n):
    y_dim, x_dim, N = np.shape(img)
    degrade = np.uint8( 2**(8-n) * (img // 2**(8-n)))
    plt.imshow(degrade)

def devoiler(img):
    cache = 16*(img % 16)
    plt.imshow(cache)

def devoiler2(img):
    y_dim, x_dim, N = np.shape(img)

    texte = ""
    for j in range(x_dim):
        texte += chr(16*(img[0,j,0]%16) + img[0,j,1]%16)

    return texte
```