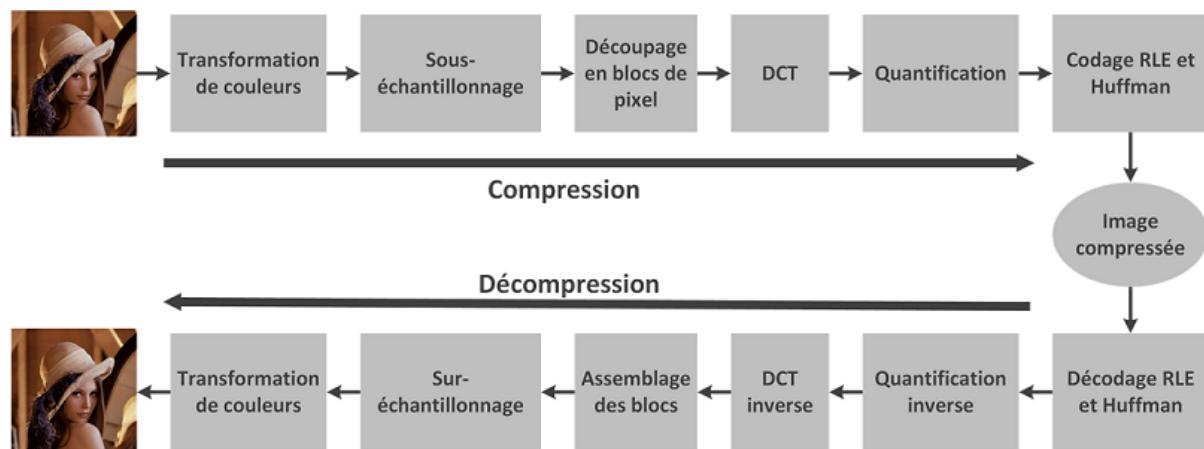


TP - Réalisation d'un codeur/décodeur JPEG simplifié

Durant ce TP nous allons coder un compresseur JPEG puis un décompresseur JPEG, en Python. En réalité JPEG est une norme qui ne concerne que la décompression des images, la compression est donc assez libre, elle doit simplement permettre à une image ainsi compressée d'être décompresser par la norme JPEG.

Les différentes étapes de cette compression / décompression sont présentées sur l'image ci-dessous et nous les détaillerons au fur et à mesure de notre avancement.



COMPRESSION

1) Structure d'une image (non compressée)

L'image à compresser est une matrice de $M \times N$ pixels. Pour une image en niveau de gris, chaque pixel est codé sur un octet (valeurs comprises entre 0 et 255).

Pour une image couleur RGB, chaque pixel est codé sur trois octets, qui représentent les intensités des trois composantes couleur : le rouge, le vert et le bleu.

2) Transformation des couleurs : Conversion en luminance / chrominance

En théorie :

Conversion en luminance/chrominance

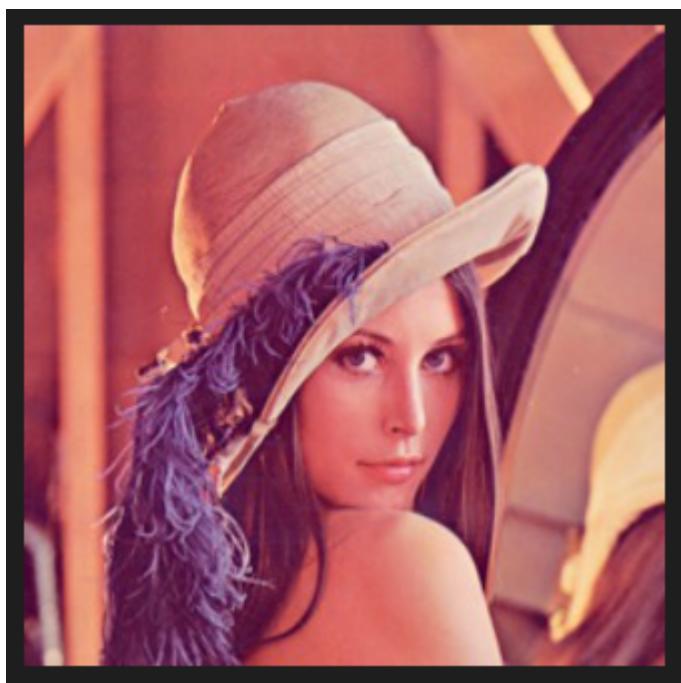
Une autre représentation que le RGB est souvent utilisé : le YCbCr, qui correspond à la luminance Y (intensité du pixel en niveau de gris) et deux chrominances (une rouge Cr et une bleue Cb). L'image est alors constitué par trois tableaux d'octets, associés respectivement aux trois grandeurs Y, Cr, Cb. Les formules permettant de passer d'une représentation à l'autre sont les suivantes :

$$\begin{aligned}Y &= 0.299R + 0.587G + 0.114B \\Cb &= -0.1687R - 0.3313G + 0.5B + 128 \\Cr &= 0.5R - 0.4187G - 0.0813B + 128 \\R &= Y + 1.14020(Cr - 128) \\G &= Y - 0.34414(Cb - 128) - 0.71414(Cr - 128) \\B &= Y + 1.77200(Cb - 128)\end{aligned}$$

Remarque : R, G, B sont compris entre 0 et 255. Ces formules peuvent donner les valeurs YCbCr qui sortent de cet intervalle, et qui devront être saturées si nécessaire afin de les garder comprises entre 0 et 255.

Deux formats différents sont fréquemment utilisés :

- dans le format le plus simple, les trois tableaux Y, Cr, Cb ont les mêmes dimensions, M et N, égales aux dimensions de l'image originale.



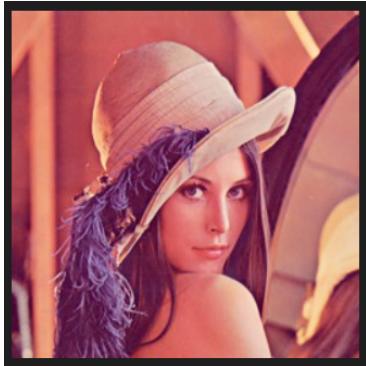
En pratique :

Nous partons de l'image de Lena au format RGB, le RGB est donc un espace de couleurs, il en existe plein d'autres. L'étape de conversion des couleurs consiste donc à convertir notre image dans l'espace RGB en une image dans l'espace de couleur YCrCb (présenté ci-dessus).

Pour faire cela, il existe une fonction Python appartenant à la librairie OpenCV, et qui s'appelle cvtColor. Elle prend en premier argument l'image source dont on

veut modifier l'espace de couleur et en second argument la conversion que l'on souhaite effectuer, ici RGB vers YCrCb.

Notre image de départ est donc :



Et on la convertit en YCrCb et on l'affiche via le code suivant :

```
img = cv2.cvtColor(imgOriginal, cv2.COLOR_RGB2YCrCb)  
show(img)
```



Et on obtient alors :

Remarque : nous avons aussi testé ce code afin de la mettre en noir et blanc ou plutôt en nuances de gris.

```
img_gray = cv2.cvtColor(imgOriginal, cv2.COLOR_RGB2GRAY)  
#  
  
#show img_gray  
show(img_gray)
```



3) Sous-échantillonnage

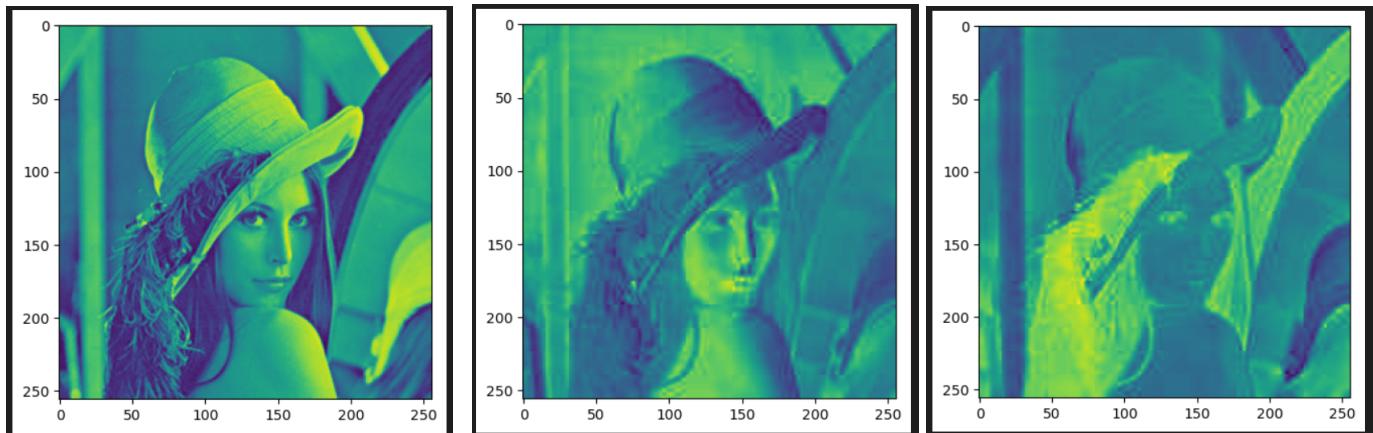
- dans le format le plus simple, les trois tableaux Y, Cr, Cb ont les mêmes dimensions, M et N, égales aux dimensions de l'image originale.
 - un format plus efficace en terme de compression (et donc plus souvent utilisé) consiste à sous-échantillonner les deux signaux de chrominance d'un facteur 2 (ou 4). Plus précisément, avec le facteur 2, le tableau Y reste de taille [M,N] mais les tableaux Cr et Cb sont de dimension [M/2,N/2] (la valeur de chrominance est associée à quatre pixels voisins).

Nous allons donc sous-échantillonner les chrominances d'un facteur 2. Pour cela, il faut tout d'abord isoler les matrices Y, Cr et Cb.

On le fait via le code suivant car notre image a pour format initial (256, 256, 3), Y constituant le premier élément de chaque sous matrice de dimension (1, 3), Cr le second élément et Cb le troisième.

```
def luminance(img):
    return(img[:, :, 0])
def Cr(img):
    return(img[:, :, 1])
def Cb(img):
    return(img[:, :, 2])
```

On obtient respectivement, en affichant avec plt.imshow le résultat de ces fonctions, la luminance Y, et les chrominances Cr et Cb :



Remarque : ces 3 images devraient être en nuance de gris et non en nuance de vert, c'est l'affichage de la librairie matplotlib qui veut ça.

On comprend alors pourquoi on a choisi l'espace de couleur YCrCb. En effet, l'objectif du format JPEG est notamment de compresser l'image, or si on moyenne directement sur un format RGB on va rapidement dégrader l'image, alors que l'œil humain est peu sensible à la chrominance. Ainsi, en ne sous-échantillonnant que la chrominance, la qualité de la photo décompressée s'en verra beaucoup moins affectée.

Nous allons maintenant nous occuper du sous-échantillonnage en lui-même. Une possibilité serait de faire un moyennage, c'est-à-dire remplacer chaque bloc de 4 pixels voisins par un pixel contenant leur valeur moyenne.

Une autre possibilité est de simplement supprimer une ligne sur 2 des matrices initiales Cr et Cb, puis supprimer une colonne sur 2. On obtient ainsi 2 matrices Cr et Cb de dimensions divisées par 2, soit de dimension (128, 128).

Pour supprimer une ligne sur 2 puis une colonne sur 2 dans une matrice on utilise le code suivant :

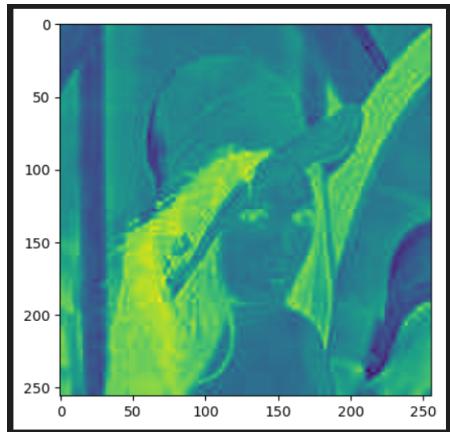
```
# Sous-échantillonnage de la chrominance

def souschantillonage(img):
    # LIGNES
    n = len(img) #256
    m = n/2 #128
    i = 1
    while(len(img)>m):
        img = np.delete(img, i, 0)
        i += 1

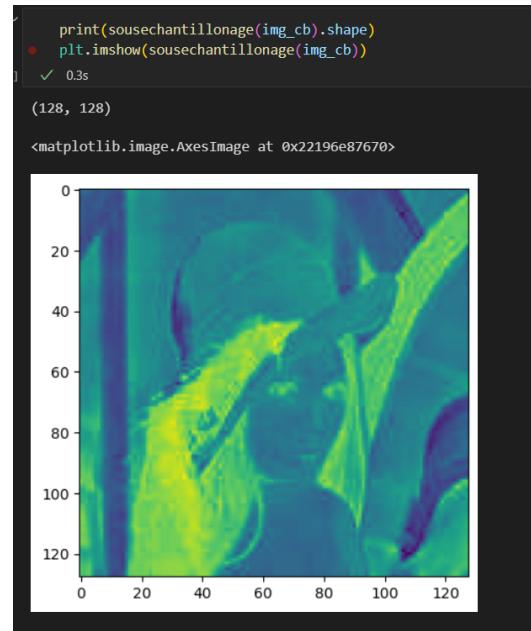
    # COLONNES
    imgt = np.transpose(img)
    i = 1
    while(len(imgt)>m):
        imgt = np.delete(imgt, i, 0)
        i += 1
    img2 = np.transpose(imgt)

    return(img2)
```

Et on obtient les matrices Cr et Cb sous échantillonnées suivantes.



Cb non sous-échantillonné (256*256)



Cb sous-échantillonné (128*128)

En effet, à l'œil nu il est difficile de dire s'il y a une différence entre ces 2 images.

4) Découpage en blocs de pixels

Le codeur JPEG travaille sur des data unit qui sont des blocs de taille 8x8 d'une image. Chaque composante est découpée en carrés de 8x8 pixels. Si les dimensions de la composante ne sont pas des multiples de 8, l'image est complétée par duplication de la dernière ligne (ou colonne) jusqu'à obtenir le multiple de 8 immédiatement supérieur. Chaque carré 8x8 est ensuite traité indépendamment, en décrivant l'image de gauche à droite et de haut en bas.

Mais dans notre cas nous n'en aurons pas besoin car notre image initiale est de dimension 256*256, et les sous matrice que nous en avons ressortie jusque là sont Y de dimension 256*256, Cr et Cb toutes deux de dimension 128*128, qui sont des multiples de 8.

```
# define block size
blockSize = 8

# compute number of blocks
number_blocs_long = width/blockSize
number_blocs_larg = height/blockSize
```

Cependant, dans la fonction ci-dessous, nous donnons la possibilité de rajouter des "zéros" (et non une copie de la ligne) dans les cas où l'image à compresser n'aura pas avec des dimensions en multiple de 8. Cela s'appelle du zero padding.

```
def completer_matrice(A):
    nb_lignes, nb_colonnes = A.shape

    nb_lignes_a_ajouter = 8 - (nb_lignes % 8)
    nb_colonnes_a_ajouter = 8 - (nb_colonnes % 8)

    # le cas si y'a besoin de rien faire
    if nb_lignes_a_ajouter == 8 and nb_colonnes_a_ajouter == 8:
        return A

    # sinon
    colonnes_a_ajouter = np.zeros((nb_lignes, nb_colonnes_a_ajouter), dtype=A.dtype)
    A = np.hstack((A, colonnes_a_ajouter))
    lignes_a_ajouter = np.zeros((nb_lignes_a_ajouter, A.shape[1]), dtype=A.dtype)
    A = np.vstack((A, lignes_a_ajouter))
```

Le code ci-dessus permet dans un premier temps de récupérer les dimensions de la matrice fournie en entrée de la fonction, puis de calculer le nombre de lignes et colonnes de zéros qu'il faudra ajouter pour atteindre une matrice dont les dimensions sont des multiples de 8.

Ensuite, il vérifie si zéro padding est nécessaire ou non et si ce n'est pas le cas renvoie la matrice donnée en entrée et non modifiée.

En revanche, si le zéro padding est nécessaire, on crée une matrice de zéros ayant autant de colonnes que nécessaire pour atteindre un multiple de 8. Puis on accolé

cette matrice de zéros à la matrice fournie en entrée. On fait de même pour les lignes, en faisant bien attention à prendre le nouveau nombre de colonnes (après le zéro padding des colonnes).

L'étape suivante va donc consister à "découper" nos matrices Y, Cr et Cb en blocs de 8*8 pixels.

```
def data_unit(A):
    m, n = A.shape
    sous_matrices = []
    for i in range(0, m, 8):
        for j in range(0, n, 8):
            sous_matrices.append(A[i:i+8, j:j+8])
    return sous_matrices
```

La fonction ci-contre crée un tableau composé de toutes les matrices 8x8 en ligne. Autrement dit, cela crée un vecteur des blocs 8*8 de pixels de la matrice initiale.

5) DCT

L'étape suivante consiste à effectuer une DCT (Transformée en cosinus discrète) sur chaque bloc de 8*8. On va ainsi passer de la représentation spatiale de l'image (ou plutôt de chaque bloc 8*8) à sa représentation fréquentielle.

Mais avant d'appliquer la fonction de la DCT il est nécessaire de centrer les valeurs des pixels autour de 0 comme expliqué ci-dessous de manière à avoir une moyenne nulle.

2.1 Centrage

Les échantillons des carrés 8x8 sont des nombres compris entre 0 et 255. La première opération à réaliser est un centrage afin d'obtenir des valeurs comprises entre -128 et 127 (en retirant 128 à chaque valeur). Les valeurs centrées seront notées $f(i,j)$ avec $i,j \in [0, 7]$.

Le code ci-dessous réalise ce centrage de chaque pixel pour chaque bloc de taille 8*8

```
#Centrage
def centrage(A):
    A -= 128
    return A
```

Attention ! Contrairement aux fonctions précédentes, cette fonction ne prend pas en entrée la matrice qui "sort" de la fonction précédente, c'est à dire qu'elle ne prend pas en entrée le vecteur dont les éléments sont les blocs 8*8 mais un unique bloc 8*8.

Il va maintenant s'agir d'appliquer la DCT dont le fonctionnement est explicité ci-dessous.

2.2 Transformée en cosinus

Après avoir réalisé le centrage, il faut calculer la transformée en cosinus discrète (DCT) de chaque carré 8x8. Cette transformation donne une nouvelle matrice 8x8 de coefficients, appelée $F(u, v)$, $u, v \in [0, 7]$. Cette transformée est une variante de la transformée de Fourier. Elle décompose un bloc, considéré comme une fonction numérique à deux variables, en une somme de fonctions cosinus oscillant à des fréquences différentes. Chaque bloc est ainsi décrit en une carte de fréquences et en amplitudes plutôt qu'en pixels et coefficients de couleur. Pour le décodage, on peut retrouver les $f(i, j)$ à partir de la transformée 2D inverse des $F(u, v)$. Les équations de cette transformée ainsi que de la transformée inverse :

$$F(u, v) = C_u C_v \sum_{i=0}^7 \sum_{j=0}^7 f(i, j) \cos\left(\frac{(2i+1)u\pi}{16}\right) \cos\left(\frac{(2j+1)v\pi}{16}\right) \quad (1)$$

$$f(i, j) = \sum_{u=0}^7 \sum_{v=0}^7 C_u C_v F(u, v) \cos\left(\frac{(2i+1)u\pi}{16}\right) \cos\left(\frac{(2j+1)v\pi}{16}\right) \quad (2)$$

avec $C_0 = \frac{1}{\sqrt{8}}$ et $C_u = \frac{1}{2}$ si $u \neq 0$.

L'application de la DCT est une opération théoriquement sans perte d'informations : les coefficients initiaux peuvent être retrouvés en appliquant la DCT inverse au résultat de la DCT.

Ces transformations en deux dimensions sont séparables car elles peuvent être réalisées en appliquant des transformations en une dimension, successivement sur les lignes et les colonnes.

(Remarque : u constitue alors la fréquence horizontale et v la fréquence verticale)

L'application de la DCT n'apporte aucune compression et est sans perte. C'est l'étape suivante qui apportera la compression que l'on recherche.

Pour l'instant, on applique la formule 1), c'est-à-dire la DCT 2D, avec le code ci-dessous à chaque bloc de 8*8.

Pour cela nous avons utilisé la fonction python `dct` de la librairie opencv. Cette fonction peut réaliser différents calculs en fonction des paramètres qu'on lui fournit. Dans notre cas, nous voulons effectuer une DCT 2D sur une matrice réelle. Pour cela, il suffit de fournir une matrice source en premier argument et de ne pas donner de second argument. Ainsi, la matrice renvoyée est réelle et de même dimensions que la matrice d'entrée.

```
# define empty matrices to store Dct
def dct(A):
    imgDct = cv2.dct(np.float32(A))
    return(imgDct)
```

Attention ! Idem, cette fonction est faite pour recevoir en entrée un unique bloc 8*8.

6) Quantification

L'étape suivante consiste à quantifier chaque bloc de 8×8 en sortie de la DCT. Cette étape constitue l'étape de compression principale.

L'idée de la quantification repose sur les limites de l'œil humain, en effet l'œil humain est peu sensible aux hautes fréquences. L'idée est donc d'allouer plus de bits pour les basses fréquences et moins de bits pour les hautes fréquences.

La quantification consiste à diviser point à point chaque bloc de 8×8 par une matrice de quantification Q également 8×8 . Le bloc ainsi compressé est obtenu par la formule suivante.

$$\hat{F}(u, v) = \text{round} \left[\frac{F(u, v)}{Q(u, v)} \right]$$

Il existe des matrices de quantification Q propre à chaque type d'image issues d'expériences psycho-visuelles. Dans notre cas nous allons avoir besoin de la matrice Q propre à la luminance, qui est la suivante (qui est aussi valable pour les composantes RGB).

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Mais aussi de la matrice Q propre aux chrominances :

Les matrices de quantification sont les éléments centraux de la compression avec pertes parce que c'est la quantification qui permet de régler les pertes, et donc le taux de compression. En toute rigueur, ces matrice devraient être calculées pour chaque image, en tenant compte du taux de compression désiré et des propriétés de l'image et de l'œil. En pratique, ce calcul est complexe, et l'on pourra dans ce projet utiliser les matrices précalculées.

Comme on l'a vu, le but est d'atténuer les hautes fréquences (puisque l'œil humain y est peu sensible). Ces dernières se situent en bas à droite de chacune des matrices Q et on remarque que ces valeurs sont élevées.

D'autre part, on a vu qu'une matrice $F(u, v)$ (soit un bloc 8×8 en sortie de la DCT) contient les valeurs des amplitudes en fonction de la fréquence (C'est logique puisque la DCT permet de passer du spatial au fréquentiel, elle constitue donc le spectre de chaque bloc 8×8 .) qui augmente de la gauche vers la droite et du haut vers le bas. Ainsi, la prise de la partie entière (round) du résultat de la division de $F(u, v)/Q$ va ramener un bon nombre de valeur des bloc 8×8 à 0.

Cela est grandement intéressant car cela va nous permettre d'utiliser un RLC (Run Length Code ou RLE : Run Length Encoding) qui permet une forte compression et consiste à remplacer chaque répétition de symbole par le symbole et son nombre d'occurrences associées. Exemple : BBAAAAACCAAAA \Rightarrow 2B5A2C4A.

En terme de code, la quantification s'effectue ainsi :

```
def quantif_Y(A):
    A=np.round(A/qY)
    return A

def quantif_C(A):
    A=np.round(A/qC)
    return A
```

7) Parcours en zig-zag, RLC et codage de Huffman

D'après l'explication du RLC, on déduit aisément que plus les chaînes de symboles identiques sont longs, plus le code est compressé. L'étape suivante va donc consister à allonger les chaînes de zéros comme expliqué ci-dessous.

Comme dit dans la section précédente, après la quantification, beaucoup de coefficients de la matrice \hat{F} sont nuls et ils localisés en bas à droite de cette matrice. Le nombre de bits moyen de ces coefficients est donc très réduit. Afin d'exploiter cette propriété, le bloc 8x8 sera lu avec un parcours zigzag (Figure 4) afin de construire de longues plages de 0 (afin d'optimiser au mieux l'utilisation d'un RLC sur le symbole 0).

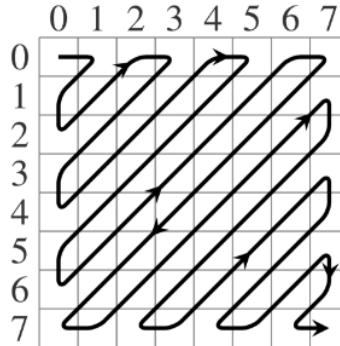


FIGURE 1 – Ré-ordonnancement Zigzag.

Pour coder cela il va falloir plusieurs étapes. Tout d'abord, il faut construire les vecteurs contenant les 64 valeurs de chaque bloc 8x8.

Les matrices lig et col ci-dessous, que l'on nous a fourni, contiennent respectivement les indices des lignes et des colonnes du bloc 8*8, dans l'ordre dans lequel on doit les prendre pour respecter l'ordre du zigzag présenté ci-dessus.

Par exemple, la première valeur à mettre dans le vecteur est la valeur à la position [lig[0], col[0]] = [1, 1] (en python les indices des tableaux commencent à 0) et la cinquième valeur à mettre dans le vecteur est la valeur à la position [lig[4], col[4]] = [2,2].

Il reste un détail à régler, les indices stockés dans lig et col sont numérotés de 1 à 8, donc pour obtenir la bonne valeur en python, il faut retrancher 1 à chaque valeur d'indice retourné par lig[i] et col[i].

Tout cela est effectué via le code suivant.

```
# This will be used for of ZigZag...
col = np.array([1, 2, 1, 1, 2, 3, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5,
               6, 7, 8, 7, 6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8, 8, 7, 6, 5, 4, 3, 4, 5, 6, 7, 8, 8, 7, 6, 5, 6, 7, 8, 8, 7, 8, 7, 8])

lig = np.array([1, 1, 2, 3, 2, 1, 1, 2, 3, 4, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 1, 2,
               3, 4, 5, 6, 7, 8, 8, 7, 6, 5, 4, 3, 2, 3, 4, 5, 6, 7, 8, 8, 7, 6, 5, 4, 5, 6, 7, 8, 8, 7, 6, 7, 8, 8])

def zigzag(A):
    B = np.zeros(64)
    for k in range(64):
        B[k] = A[lig[k]-1,col[k]-1]
    return B
```

Encore une fois, cette fonction prend en entrée un bloc 8*8.

Le codage de Huffman est un algorithme de compression de données sans perte. Le codage de Huffman utilise un code à longueur variable pour représenter un symbole de la source (par exemple un caractère dans un fichier). Le code est déterminé à partir d'une estimation des probabilités d'apparition des symboles de source, un code court étant associé aux symboles de source les plus fréquents.

Ce principe est expliqué par l'extrait de la page wikipédia suivant.

Principe [modifier | modifier le code]

Le principe du codage de Huffman repose sur la création d'une structure d'**arbre** composée de nœuds.

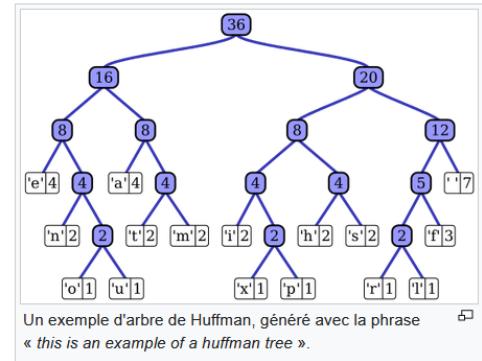
Supposons que la phrase à coder est « *this is an example of a huffman tree* ». On recherche tout d'abord le nombre d'**occurrences** de chaque caractère. Dans l'exemple précédent, la phrase contient 2 fois le caractère *h* et 7 espaces. Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids égal à son nombre d'occurrences.

L'arbre est créé de la manière suivante, on associe chaque fois les deux nœuds de plus faibles poids, pour donner un nouveau nœud dont le poids équivaut à la somme des poids de ses fils. On réitère ce processus jusqu'à n'en avoir plus qu'un seul nœud : la racine. On associe ensuite par exemple le code 0 à chaque embranchement partant vers la gauche et le code 1 vers la droite.

Pour obtenir le **code binaire** de chaque caractère, on remonte l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code un 0 ou un 1 selon la branche suivie. La phrase « *this is an example of a huffman tree* » se code alors sur 135 bits au lieu de 288 bits (si le codage initial des caractères tient sur 8 bits). Il est nécessaire de partir de la racine pour obtenir les codes binaires car sinon lors de la décompression, partir des feuilles peut entraîner une confusion lors du décodage.

Pour coder « *Wikipedia* », nous obtenons donc en binaire : 101 11 011 11 100 010 001 11 000, soit 24 bits au lieu de 63 (9 caractères x 7 bits par caractère) en utilisant les codes **ASCII** (7 bits).

Différentes méthodes de construction de l'arbre [modifier | modifier le code]



Les différentes fonctions permettant de construire l'arbre de Huffman associé à chaque vecteur et ensuite encoder ces vecteurs sont déjà codées dans le notebook que l'on nous a fourni.

Plus précisément, on fait tout d'abord appel à la fonction “construct_huffman_table” qui retourne un dictionnaire des symboles, c'est-à-dire un tableau contenant les symboles et leur code binaire associé, déduit de la construction de l'arbre de Huffman. C'est pourquoi, cette fonction compte tout d'abord le nombre d'occurrence de chaque symbole dans la liste qu'on donne en entrée grâce à la fonction “count_symbols” (dont on parlera après), puis elle fait appel à la fonction “compute_huffman_tree” qui à partir des occurrences des symboles en déduit l'arbre de Huffman et enfin la fonction “bintree_to_table” déduit de l'arbre de Huffman le dictionnaire des symboles.

Ensuite, on appelle la fonction “encode_huffman” qui prend en entrée le dictionnaire précédemment créé ainsi que la liste que l'on souhaite coder et retourne cette liste encodée via la méthode d'Huffman.

La seule chose dans tout cela qu'il nous restait à coder est la fonction `count_symbols` qui permet de compter le nombre d'occurrence de chaque symbole dans un vecteur donné. Cette fonction est utilisée dans les autres fonctions déjà codées. En effet, elle sert à établir le poids initial des feuilles de l'arbre, chaque symbole étant associé à une feuille.

```
def count_symbols(data:list) -> Dict[Any, int]:
    freq = {}
    for symbole in data:
        if symbole in freq:
            freq[symbole] += 1
        else:
            freq[symbole] = 1
    return freq
```

Cette fonction fait appel à un nouveau type d'outil Python : un dictionnaire. Son principe de fonctionnement : chaque élément qu'il contient est associé à une valeur. On ne peut pas faire appel à la valeur directement mais seulement à l'objet qui est donc

appelé une clef. Les éléments d'un dictionnaire n'ont pas d'indice, c'est l'appel du dictionnaire en une clef donnée qui renvoie la valeur associée à cette clef. Dans notre cas, pour chaque symbole dans notre liste de départ, on regarde si ce symbole existe déjà dans le dictionnaire (est déjà une clef de ce dictionnaire). Si c'est le cas, on ajoute +1 à la valeur associée à ce symbole (la valeur associée à chaque symbole constitue donc leur occurrence dans la liste de départ). Si le symbole n'est pas dans le dictionnaire, alors on l'ajoute.

Nous avons effectué le test afin de vérifier la fonction `count_symbols` qui renvoie bien le résultat attendu.

```
#testing
freq = count_symbols([0, 0, 0, 0, 1, 1, 1, 2, 2, 3])
print(freq)
#output: {0: 4, 1: 3, 2: 2, 3: 1}

[4] ✓ 0.0s
· {0: 4, 1: 3, 2: 2, 3: 1}
```

CODE GENERAL POUR ENCODER

On utilise successivement les fonctions précédemment codées et citées. A noter que les fonctions qui s'appliquent directement à des blocs de 8*8 sont placées dans une boucle variant dans le vecteur des blocs 8*8 correspondants. En effet, Y n'étant pas sous-échantillonnée contrairement aux matrices de chrominance, Y n'est pas de même dimension que Cr et Cb. Leur nombre de bloc 8*8 est donc différent, il est donc nécessaire de créer 2 boucles for différentes, une pour la matrice traitant Y et une autre pour les matrices traitant les chrominances.

Notre fonction décode renvoie donc en sortie 6 listes. 2 listes pour chacune des 3 composantes de l'image d'origine.

Pour chacune des composantes, il y a une liste qui contient les chaînes de caractères d'encodage d'Huffman de chacun des blocs 8*8. Et une seconde liste qui contient, aux indices correspondants à la liste des encodages d'Huffman, les tables suivant lesquelles chaque bloc 8*8 a été encodé. Ces 2 listes nous seront nécessaires pour le décodage.

```
## RUN_ENCODE

#Ce code sert à totalement encoder une image. Il est composé des fonctions précédentes

def encode(imgOriginal):

    # 1. Passage en luminance, chrominance.
    imgYCrCb = cv2.cvtColor(imgOriginal, cv2.COLOR_RGB2YCrCb)
    img_Y = luminance(imgYCrCb)
    img_cr = cr(imgYCrCb)
    img_cb = cb(imgYCrCb)

    # 2. sous echantillonage

    img_cr = sousechantillonage(img_cr)
    img_cb = sousechantillonage(img_cb)

    # 3. Zero padding
    img_Y = completer_matrice(img_Y)
    img_cr = completer_matrice(img_cr)
    img_cb = completer_matrice(img_cb)

    # 4. Mise en data units
    datas_units_Y = data_unit_function(img_Y)
    datas_units_cr = data_unit_function(img_cr)
    datas_units_cb = data_unit_function(img_cb)
```

```
# Maintenant nous allons créer une boucle pour travailler sur chaque data units
tailleY = len(datas_units_Y)
tailleC = len(datas_units_cr)

list_Yencode = []
list_Ytable = []
list_crencode = []
list_crttable = []
list_cbencode = []
list_cbttable = []

for i in range(tailleY):

    # 5. Centrage
    datas_units_Y[i] = centrage(datas_units_Y[i])

    # 6. DCT
    datas_units_Y[i] = dct(datas_units_Y[i])

    # 7. Quantification
    datas_units_Y[i] = quantif_Y(datas_units_Y[i])

    print("quantifié", datas_units_Y[i])

    # 8. zig zag
    datas_units_Y[i] = zigzag(datas_units_Y[i])

    #9 RLE
    rle_Y = rlc_encode(datas_units_Y[i])

    print("rle", rle_Y)

    # 10. Huffman
    table_huffman_Y = construct_huffman_table(rle_Y)

    huffman_Y = encode_huffman(rle_Y, table_huffman_Y)

    print("huff", huffman_Y)

    list_Yencode.append(huffman_Y)
    list_Ytable.append(table_huffman_Y)
```

```
for j in range(tailleC):

    # 5. Centrage
    datas_units_cr[j] = centrage(datas_units_cr[j])
    datas_units_cb[j] = centrage(datas_units_cb[j])

    # 6. DCT
    datas_units_cr[j] = dct(datas_units_cr[j])
    datas_units_cb[j] = dct(datas_units_cb[j])

    # 7. Quantification
    datas_units_cr[j] = quantif_C(datas_units_cr[j])
    datas_units_cb[j] = quantif_C(datas_units_cb[j])

    # 8. zig zag
    datas_units_cr[j] = zigzag(datas_units_cr[j])
    datas_units_cb[j] = zigzag(datas_units_cb[j])

    #9 RLE
    rle_Cr = rlc_encode(datas_units_cr[j])
    rle_Cb = rlc_encode(datas_units_cb[j])

    # 10. Huffman

    table_huffman_cr = construct_huffman_table(rle_Cr)
    table_huffman_cb = construct_huffman_table(rle_Cb)

    huffman_cr = encode_huffman(rle_Cr, table_huffman_cr)
    huffman_cb = encode_huffman(rle_Cb, table_huffman_cb)

    list_crencode.append(huffman_cr)
    list_crttable.append(table_huffman_cr)
    list_cbencode.append(huffman_cb)
    list_cbttable.append(table_huffman_cb)

return(list_Yencode, list_Ytable, list_crencode, list_crttable, list_cbencode, list_cbttable)
```

Nous allons maintenant faire différents tests afin d'attester du bon fonctionnement de notre code.

Pour visualiser quelques étapes de la transformation, on peut observer le premier bloc 8*8 (en haut à gauche) de la photo de Lena, prélevé ci-dessous.

```
[[161 162 163 160 158 157 157 154]
 [161 161 160 157 155 155 156 154]
 [157 157 158 157 155 154 154 151]
 [157 156 156 156 156 155 155 153]
 [157 156 157 157 155 156 153 157]
 [155 155 157 157 159 158 153 158]
 [155 155 157 158 158 156 155 162]
 [158 157 159 159 158 159 162 167]]
```

Ensuite le même premier bloc après application de la DCT. On remarque alors qu'à part le premier élément en haut à gauche de ce bloc (ce qui équivaut à la valeur moyenne du bloc en fréquentielle) qui est toujours très élevé (233), toutes les autres valeurs ont été très atténuerées comme on l'attendait.

[[2.32625000e+02	4.83569622e+00	-5.52803159e-01	-3.38284445e+00
	1.37500000e+00	-3.21082860e-01	2.25846386e+00	8.25467110e-02
[-	1.79456317e+00	1.29400330e+01	-1.81268024e+00	2.63287902e+00
	-5.13380289e+00	2.64955378e+00	-2.45819163e+00	1.04695380e+00
[1.09888926e+01	-1.65374207e+00	2.64904857e+00	-2.63291788e+00
	-6.81302845e-01	5.64124465e-01	-6.27601981e-01	7.79549658e-01
[8.20913732e-01	1.22348177e+00	-2.49081683e+00	3.63158323e-02
	8.53749573e-01	-2.42449713e+00	1.87594974e+00	-6.24016881e-01
[4.12500000e+00	-1.05003607e+00	1.64481056e+00	-6.13154732e-02
	-6.25000000e-01	1.34853375e+00	-2.75405794e-01	-1.12340614e-01
[-2.10594749e+00	-1.68876246e-01	-2.82311153e+00	3.12939316e-01
	6.22044444e-01	-2.23500222e-01	-7.58428454e-01	7.66375244e-01
[7.24913895e-01	-4.45397615e-01	-2.37760210e+00	6.81260049e-01
	-6.64888322e-01	7.63154030e-02	3.50951523e-01	3.37088257e-01
[-1.16538775e+00	7.84390986e-01	-2.39350915e-01	-1.69244659e+00
	4.64476287e-01	4.2577220e-01	4.69750792e-01	-7.52848446e-01

On observe ensuite ce même bloc après l'étape de quantification. On remarque alors que la quasi totalité des valeurs sont maintenant nulles.

```

[[15.  0. -0. -0.  0. -0.  0.  0.]  

 [-0.  1. -0.  0. -0.  0. -0.  0.]  

 [ 1. -0.  0. -0. -0.  0. -0.  0.]  

 [ 0.  0. -0.  0.  0. -0.  0. -0.]  

 [ 0. -0.  0. -0. -0.  0. -0. -0.]  

 [-0. -0. -0.  0.  0. -0. -0.  0.]  

 [ 0. -0. -0.  0. -0.  0.  0.  0.]  

 [-0.  0. -0. -0.  0.  0.  0. -0.]]
```

Sur un autre bloc, on peut ensuite observer le bloc quantifié, puis linéarisé suivant un zigzag (on observe bien une très grande chaîne de zéro), puis son codage RLE.

```
[[15.  1. -0.  0. -0. -0.  0.  0.]
 [ 1.  0. -0.  0.  0. -0.  0.  0.]
 [ 0.  0. -0.  0.  0.  0. -0. -0.]
 [ 0. -0.  0.  0.  0.  0.  0.  0.]
 [ 0. -0. -0.  0. -0. -0.  0. -0.]
 [ 0.  0.  0. -0. -0.  0.  0.  0.]
 [ 0.  0.  0.  0. -0. -0.  0.  0.]
 [ 0.  0.  0.  0. -0.  0.  0.  0.]]
[15.  1.  1.  0.  0. -0.  0. -0.  0.  0.  0. -0. -0.  0. -0. -0.  0.  0.
 0. -0. 0.  0.  0. -0.  0.  0. -0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
 0.  0. -0. -0.  0.  0. -0.  0. -0.  0. -0. -0.  0.  0.  0. -0.  0.  0.
 -0.  0. -0. -0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Nous voulons maintenant calculer le taux de compression qu'a permis d'obtenir notre code. C'est ce que fait le code ci-dessous.

```
# number of bits in original image

a, b, c, d, e, f = encode(imgOriginal)

H, W, C = imgOriginal.shape
number_init_bits = H*W*C
print("Nombre de bits dans l'image d'origine :", number_init_bits)

s_bits = 0
for i in range(len(a)):
    s_bits += len(a[i])
for i in range(len(c)):
    s_bits += len(c[i])
for i in range(len(e)):
    s_bits += len(e[i])

print("Nombre de bits dans l'image compressée :", s_bits)

taux_compression = number_init_bits/s_bits

print("Taux de compression :", taux_compression)
```

```
Nombre de bits dans l'image d'origine : 196608
Nombre de bits dans l'image compressée : 142967
Taux de compression : 1.375198472374744
```

On constate que le taux de compression n'est pas énorme.

DÉCOMPRESSION

A présent, nous voulons pouvoir décoder une image encodée comme précédemment. Pour cela, il va globalement s'agir de refaire les mêmes étapes que précédemment mais dans l'autre sens, comme le montre l'image en tête de notre compte-rendu.

La première étape va donc consister à décoder le code de Huffman grâce à la fonction qui nous est fournie dans le notebook. Cette fonction prend en argument un vecteur encodé (ici ce vecteur correspond à l'encodage d'un bloc 8*8) ainsi que sa table de Huffman que nous avons créée lors de la compression. Il faut donc récupérer ces 2 éléments, pour chacun des blocs 8*8 de chacune des 3 "dimensions" de notre image (Y, Cr et Cb), en sortie de l'encodage.

La fonction de décodage de Huffman est la suivante et est fournie dans le notebook.

```
def decode_huffman(encoded:str, table: dict) -> np.ndarray:  
    ...  
    Decode a list of values using huffman code dictionary  
    ...  
    data = []  
    i = 0  
    while i < len(encoded):  
        for k, v in table.items():  
            if encoded[i:].startswith(v): # prefix match  
                data.append(k)  
                i += len(v)  
                break  
    return np.array(data)
```

Résultat : [15. 1. 1. 257. 61.]

Ensuite, il faut passer du code de Huffman décodé au code RLE décodé grâce à la fonction suivante.

```
def rlc_decode(rle):  
    A = np.zeros(64, dtype=np.int16)  
    i = 0 #notre compteur pour pas sortir de rle  
    j = 0 #notre second compteur pour notre A sortie  
    while i < len(rle):  
        if rle[i] == 257:  
            i += 1  
            count = rle[i]  
            i += 1  
            while count > 0:  
                A[j] = 0  
                j += 1  
                count -= 1  
        else:  
            A[j] = rle[i]  
            j += 1  
            i += 1  
    return A
```

Résultat :

```
[ 15.  1.  1.  257.  61.]
[15  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
```

Nous avons eu un problème lors de l'exécution des fonctions encode et decode, cela venait du fait qu'on avait créé une seconde fonction quantif_Y dans le décodage et portant le même nom que la fonction au rôle inverse dans l'encodage. Cela faisait que si on exécutait la fonction encode après la nouvelle définition de cette fonction quantif_Y alors ce n'était plus le bon code qui était exécuté à cet endroit.

La sortie précédente est donc un vecteur contenant les 64 valeurs du bloc 8*8 que l'on va reconstruire, pour cela, on inverse la fonction zigzag de l'encodage.

Inverse de zigzag :

```
def inverse_zigzag(B):
    A = np.zeros((8,8))
    for k in range(64):
        A[lig2[k]-1, col2[k]-1] = B[k]
    return A
```

A noter que les matrices lig2 et col2 sont identiques aux matrices lig et col servant à l'encodage.

Résultat :

```
[15  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[[15.  1.  0.  0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
```

On procède ensuite à l'étape de dé-quantification. Cette étape révèle bien une perte d'information, puisque durant l'encodage on utilise la fonction round pour prendre la partie entière de la division, ce qui n'a pas d'équivalent lors du décodage.

```
def Dequantif_Y(A):
    return np.multiply(A, qY)

def Dequantif_C(A):
    return np.multiply(A, qC)
```

Résultat :

```
[[15.  1.  0.  0.  0.  0.  0.  0.]  
 [ 1.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [[240.  11.  0.  0.  0.  0.  0.  0.]  
 [ 12.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
```

On calcule ensuite l'inverse de la DCT grâce à la fonction idct de la librairie OpenCV (cv2).

```
def inverse_dct(A):  
     imgDct = cv2.idct(np.float32(A))  
     return(imgDct)
```

Résultat :

```
[[240.  11.  0.  0.  0.  0.  0.  0.]  
 [ 12.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.  0.  0.  0.]  
 [[33.98774  33.697388 33.16089  32.459923 31.701199 31.000229 30.46373  
  30.173378]  
 [33.670994 33.380642 32.844143 32.143177 31.38445  30.683483 30.146984  
 29.856632]  
 [33.085724 32.795372 32.258873 31.557905 30.799181 30.098213 29.561714  
 29.271362]  
 [32.32103  32.030678 31.494179 30.793211 30.034487 29.333519 28.79702  
 28.506668]  
 [31.493332 31.20298  30.666481 29.965513 29.206789 28.505821 27.969322  
 27.67897 ]  
 [30.728638 30.438286 29.901787 29.200819 28.442095 27.741127 27.204628  
 26.914276]  
 [30.143368 29.853016 29.316517 28.61555  27.856825 27.155857 26.619358  
 26.329006]  
 [29.826622 29.53627  28.999771 28.298801 27.540077 26.839111 26.302612  
 26.01226 ]]
```

Puis on centre et on utilise round pour avoir des valeurs entières :

```
def decentrage(A):
    A = A + 128
    return(np.round(A))
```

Résultat :

```
[[162. 162. 161. 160. 160. 159. 158. 158.]
 [162. 161. 161. 160. 159. 159. 158. 158.]
 [161. 161. 160. 160. 159. 158. 158. 157.]
 [160. 160. 159. 159. 158. 157. 157. 157.]
 [159. 159. 159. 158. 157. 157. 156. 156.]
 [159. 158. 158. 157. 156. 156. 155. 155.]
 [158. 158. 157. 157. 156. 155. 155. 154.]
 [158. 158. 157. 156. 156. 155. 154. 154.]]
```

A partir des bloc 8*8 ainsi décodés, il nous reste à les rassembler correctement afin de ré-obtenir des matrices 256*256 pour Y et 128*128 pour Cr et Cb.

```
def inverse_data_unit_function(mtx):
    num_blocks = int(np.sqrt(len(mtx)))
    block_size = 8
    A = np.zeros((num_blocks*block_size, num_blocks*block_size))
    k = 0
    for i in range(num_blocks):
        for j in range(num_blocks):
            A[i*block_size:(i+1)*block_size, j*block_size:(j+1)*block_size] = mtx[k]
            k += 1
    return A
```

Puis, dans le cas des matrices de chrominance, on sur-échantillonne afin de retrouver des matrices de dimension 256*256 afin de pouvoir reconstruire une image 256*256. Pour cela, on choisit de doubler chaque ligne et chaque colonne grâce à la fonction repeat de la librairie numpy.

```
def inverse_sous_echantillonage(A):
    repeated_cols = np.repeat(A, 2, axis=1)
    repeated_ligs = np.repeat(repeated_cols, 2, axis=0)
    return(repeated_ligs)
```

CODE GENERAL POUR DÉCODER

```
## RUN_DECODE

#Ce code sert à totalement décoder une image. Il est composé des fonctions précédentes

def decode(list_Yencode, list_Ytable, list_crencode, list_crttable, list_cbencode, list_cbttable):

    sous_matrices_Y = []
    sous_matrices_cr = []
    sous_matrices_cb = []

    for i in range(len(list_Yencode)):

        # 1. Huffman decode
        decoded_Y = decode_huffman(list_Yencode[i], list_Ytable[i])

        # 2. RLC decode
        decoded_Y = rlc_decode(decoded_Y)

        # 3. Inverse zig zag
        decoded_mat_Y = inverse_zigzag(decoded_Y)

        # 4. Quantification
        decoded_mat_Y = Dequantif_Y(decoded_mat_Y)

        # 5. Inverse dct
        decoded_mat_Y = inverse_dct(decoded_mat_Y)

        # 6. Décentrage
        decoded_mat_Y = decentrage(decoded_mat_Y)
```

```
# 7. Ajouter à la liste avant de tout reconstruire
sous_matrices_Y.append(decoded_mat_Y)

img_Y_reconstruit = inverse_data_unit_function(sous_matrices_Y)

for i in range(len(list_crencode)):

    # 1. Huffman decode
    decoded_cr = decode_huffman(list_crencode[i], list_crttable[i])
    decoded_cb = decode_huffman(list_cbencode[i], list_cbttable[i])

    # 2. RLC decode
    decoded_cr = rlc_decode(decoded_cr)
    decoded_cb = rlc_decode(decoded_cb)

    # 3. Inverse zig zag
    decoded_mat_cr = inverse_zigzag(decoded_cr)
    decoded_mat_cb = inverse_zigzag(decoded_cb)

    # 4. Quantification
    decoded_mat_cr = Dequantif_C(decoded_mat_cr)
    decoded_mat_cb = Dequantif_C(decoded_mat_cb)

    # 5. Inverse dct
    decoded_mat_cr = inverse_dct(decoded_mat_cr)
    decoded_mat_cb = inverse_dct(decoded_mat_cb)

    # 6. Décentrage
    decoded_mat_cr = decentrage(decoded_mat_cr)
    decoded_mat_cb = decentrage(decoded_mat_cb)
```

```
# 7. Ajouter à la liste avant de tout reconstruire
sous_matrices_cr.append(decoded_mat_cr)
sous_matrices_cb.append(decoded_mat_cb)

img_Cr_reconstruit = inverse_data_unit_function(sous_matrices_cr)
img_Cb_reconstruit = inverse_data_unit_function(sous_matrices_cb)
print(img_Cb_reconstruit.shape)

img_Cr_reconstruit2 = inverse_sous_echantillonage(img_Cr_reconstruit)
img_Cb_reconstruit2 = inverse_sous_echantillonage(img_Cb_reconstruit)

print(img_Cb_reconstruit2.shape)
return(img_Y_reconstruit, img_Cr_reconstruit2, img_Cb_reconstruit2)
```

En sortie de cette fonction, on a donc obtenu les 3 matrices, composantes Y, Cr et Cb de l'image à reconstruire. Il faut maintenant les rassembler afin de recréer l'image YCrCb, puis à la re-passé au format RGB.

```
def construit_y_cr_cb(Y, Cr, Cb):
    image_reconstruite = np.zeros(Y.shape + (3,), dtype=Y.dtype)

    for i, j in np.ndindex(Y.shape):
        image_reconstruite[i, j] = (Y[i, j], Cr[i, j], Cb[i, j])
    return(image_reconstruite)
```

Puis il ne manque plus qu'à afficher !

```
a, b, c, d, e, f = encode(imgOriginal)
ad, bd, cd = decode(a, b, c, d, e, f)

imgf = construit_y_cr_cb(ad, bd, cd)
imgf = imgf.astype(np.uint8)
imgf2 = cv2.cvtColor(imgf, cv2.COLOR_YCrCb2RGB)
plt.imshow(imgf2)
```

A noter que les formats YCrCb et RGB étant par définition codés sur 8 bits, il est nécessaire de convertir les valeurs au format 8 bits avant de pouvoir effectuer une conversion de YCrCb vers RGB. (Le format 16 bits a été nécessaire pendant le

décodage afin de ne pas perdre d'information lorsque des valeurs négatives apparaissaient.)

La ligne “`imgf = imgf.astype(np.uint8)`” réalise cela en ramenant les valeurs négatives à 0 et les valeurs supérieures à 255, à 255.

Image décompressée avec une compression/décompression sans sous-échantillonnage :

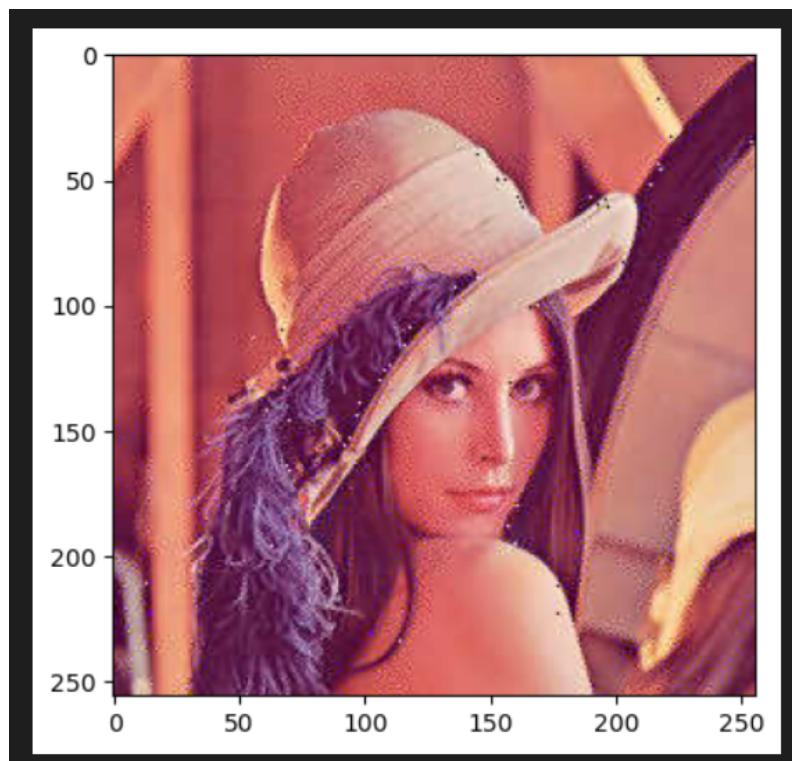
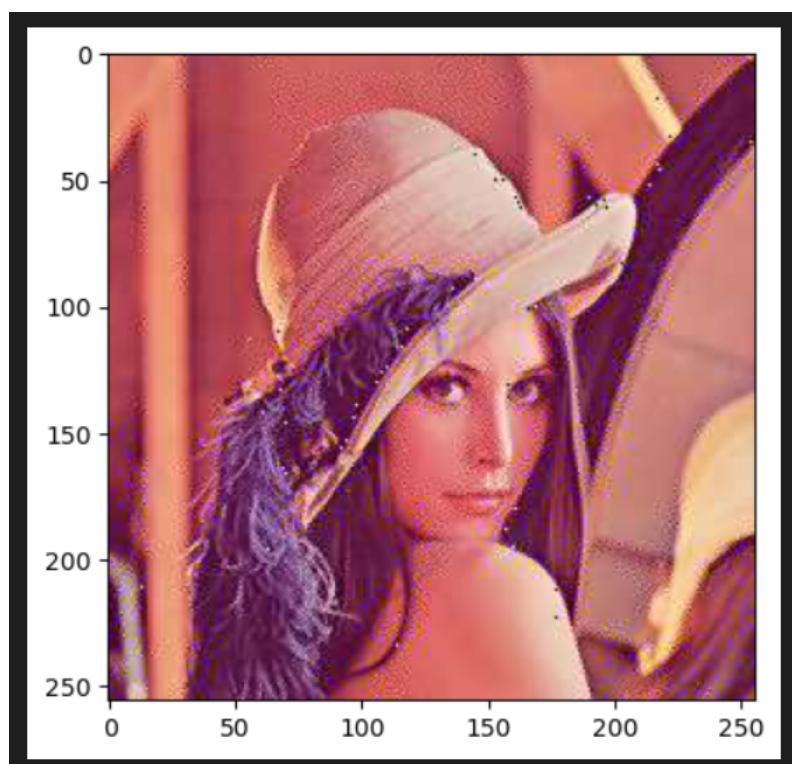


Image compressée puis décompressée avec sous-échantillonnage :



On remarque que l'image décompressée est très ressemblante à l'image compressée. On voit seulement quelques artefacts et des contours moins clairs. C'est d'autant plus le cas, lorsque l'on compare, pour l'image qui a subi un sous-échantillonnage mais qui permet un taux de compression un petit peu plus élevé.