La compression de données et son application à la compression JPEG

Sofiane MAMI

14878

Problématique

- Etude et implémentation de différentes méthodes de compression
- Leurs limites et intérêt, théoriques et pratiques
- Implémentation JPEG

Introduction

Minimiser l’espace mémoire occupé tout en conservant une qualité convenable

En ville, information omniprésente → nécessité d’optimiser le stockage des données

Compression d’image : vidéo surveillance

Sommaire

- La compression de données
- Différents algorithmes de compression
- JPEG

Entropie

- X v.a. discrète pouvant prendre n valeurs, $x_1, \cdots, x_n$, avec une probabilité p_i

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

- Analogie v.a.d ←→ source
- Nombre minimal de bits que peut atteindre un fichier compressé sans pertes (théorème du codage de source, Shannon, 1948)

Codage entropique

- Codage sans pertes
- Utilise des statistiques sur la source
- Plus un symbole est fréquent, plus son code est court

Codage entropique

- $C : \Omega \to A$
- Code préfixe : aucun mot n'est le préfixe d'un autre
- Intérêt : décodage par lecture de gauche à droite
- Inégalité de Kraft (1949) :

Pour un code préfixe sur un alphabet de taille D, les longueurs des lettres codées, $l_1, \cdots, l_n$ vérifient :

$$\sum_{i=1}^{n} D^{-l_i} \leq 1$$

Réciproquement, étant donné une suite de longueurs vérifiant cette inégalité, il existe un code préfixe dont les lettres codées ont ces longueurs.

Codage entropique

- Longueur moyenne d'un code : $L(C) = \sum_{x \in \Omega} p(x) \cdot l(x)$

- Théorème de codage de source :

Etant donné une source discrète X et un alphabet de taille D pour le codage, il existe un code C tel que

$$\frac{H(X)}{\log_2(D)} \leq L(C) < \frac{H(X)}{\log_2(D)} + 1$$

- Code optimal : préfixe et minimise cette grandeur

Code de Huffman

- Code optimal au niveau du symbole

- Encodage :
  - Classer les caractères par probabilité d’apparition décroissante
  - Relier à un nœud les 2 de probabilité minimale
  - Ce nœud est de probabilité la somme des 2
  - On itère
  - Arêtes vers la gauche affectées de 0 et celle vers la droite de 1

Code de Huffman

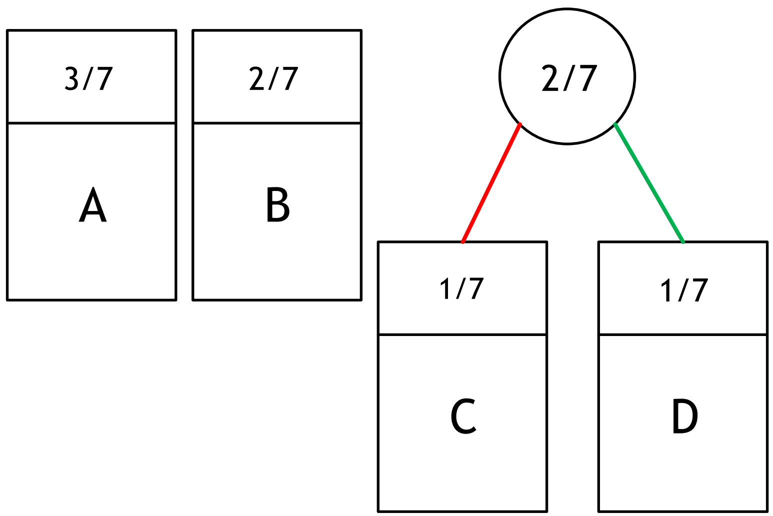
Exemple

Mot : « ABABACD »

3/7	2/7	1/7	1/7
A	B	C	D

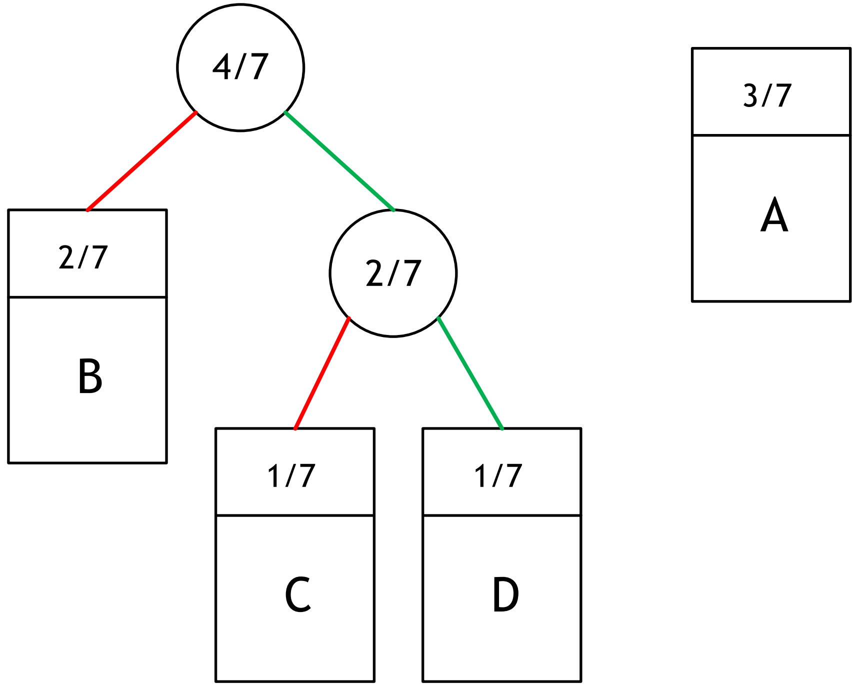
Code de Huffman

Exemple



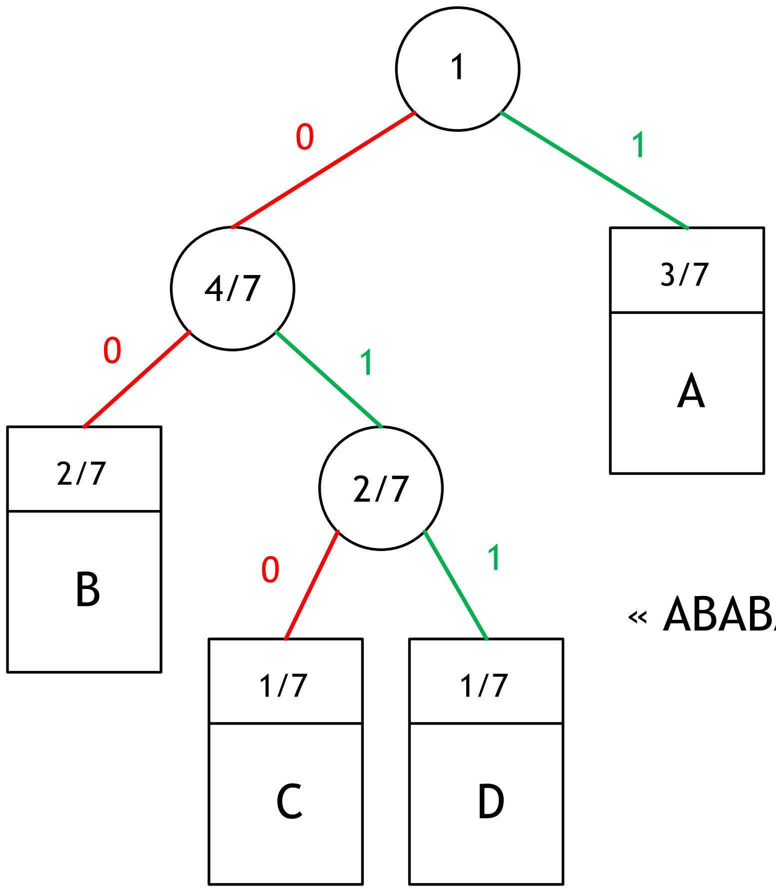
Code de Huffman

Exemple



Code de Huffman

Exemple



A : 1
B : 00
C : 010
D : 011

« ABABACD » = 1|00|1|00|1|010|011

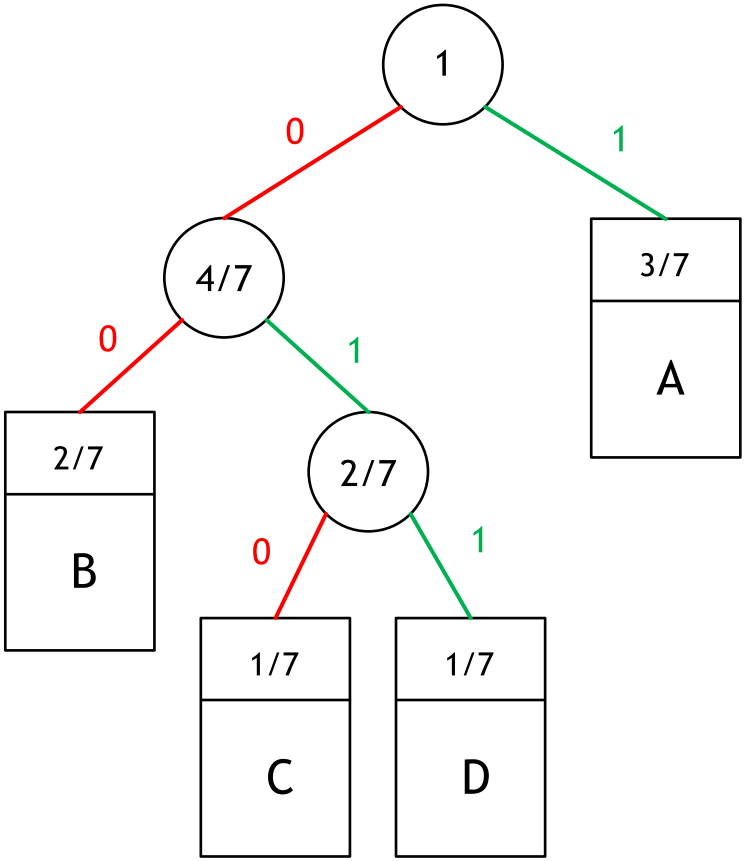
13 bits

Code de Huffman

- Décodage :
  - On connait l’arbre et un mot encodé
  - On lit les chiffres dans l’ordre tout en descendant dans l’arbre
  - Quand on arrive à une feuille, on ajoute cette lettre en bout du mot et on repart de la racine

Code de Huffman

Exemple



1001001010011 → « ABABACD »

Code de Huffman

Limites et solutions

- Théorème du codage de source à C un code de Huffman

$$H(X) \leq L(C) < H(X) + 1$$

- Solution : travailler sur des blocs de taille n

$$H(X) \leq L(C) < H(X) + \frac{1}{n}$$

Mais calcul des probabilités plus coûteux

- Codage sur un nombre entier de bits. Si un symbole a une probabilité 0,8 :
  - taille optimale du symbole : $-log_2(0,8) = 0,32$ bit
  - Huffman : au moins 1 bit

Code de Huffman

Utilisation

- Basé uniquement sur la fréquence des symboles d'entrée
- A utiliser après un algorithme mettant en évidence une redondance de certains caractères
- JPEG, MPEG, MP3

Code arithmétique

- Code optimal au niveau du bit
- Utilisé pour JPEG2000 ou JBIG2
- Limite pratique : max 15 symboles
- Codage du message par morceaux

Code arithmétique

- Principe de l'encodage :
  - On connait la probabilité d'apparition de chaque symbole
  - A chaque symbole, on associe un intervalle de [0;1[de largeur sa probabilité
  - BI, BS $\leftarrow$ 0, 1
  - On parcourt le texte
  - A chaque symbole s :
    - BB $\leftarrow$ BS – BI
    - BS $\leftarrow$ BI + BB * (BS(s))
    - BI $\leftarrow$ BI + BB * (BI(s))
  - On choisit un nombre entre BI et BS

Code arithmétique

Exemple

Mot : « ABAC »

Lettre	Probabilité	Intervalle
A	1/2	[0; 0,5[
B	1/4	[0,5; 0,75[
C	1/4	[0,75; 1[
Symbole ajouté	**BI**	**BS**
	0	1
A	0	0,5
B	0,25	0,375
A	0,25	0,3125
C	0,296875	0,3125

Tout nombre compris dans [0,296875; 0,3125[convient

Code arithmétique

- Principe du décodage :
  - On a un nombre n
  - On a la probabilité de chaque caractère ainsi que son intervalle, utilisés pour l'encodage
  - On connait le nombre m de caractère du mot, sinon l'algorithme tournera indéfiniment
  - Mot ← « »
  - On fait m fois :
    - Si n est dans l'intervalle de s
    - Mot ← Mot + 's'
    - n ← (n-BI(s))/p(s)

Code arithmétique

Exemple

n = 0,31, le mot a 4 lettres

Lettre	Probabilité	Intervalle
A	1/2	[0; 0,5[
B	1/4	[0,5; 0,75[
C	1/4	[0,75; 1[

Code compressé	Intervalle	Lettre	Texte récupéré
0,31	[0; 0,5[	A	A
0,62	[0,5; 0,75[	B	AB
0,48	[0; 0,5[	A	ABA
0,96	[0,75; 1[	C	ABAC

Mot codé : « ABAC »

Code arithmétique

Limites et solutions

- Si le fichier contient des symboles ayant une grande probabilité d’apparition alors qu’en général ils sont peu utilisés, ou l’inverse, le fichier sera plus volumineux une fois compressé

- Flottants plus compliqués à manipuler et erreurs d’arrondi possibles
- BI, BS ← 0, 99999

- Inconvénient : on ne peut subdiviser [5555; 5556[

JPEG

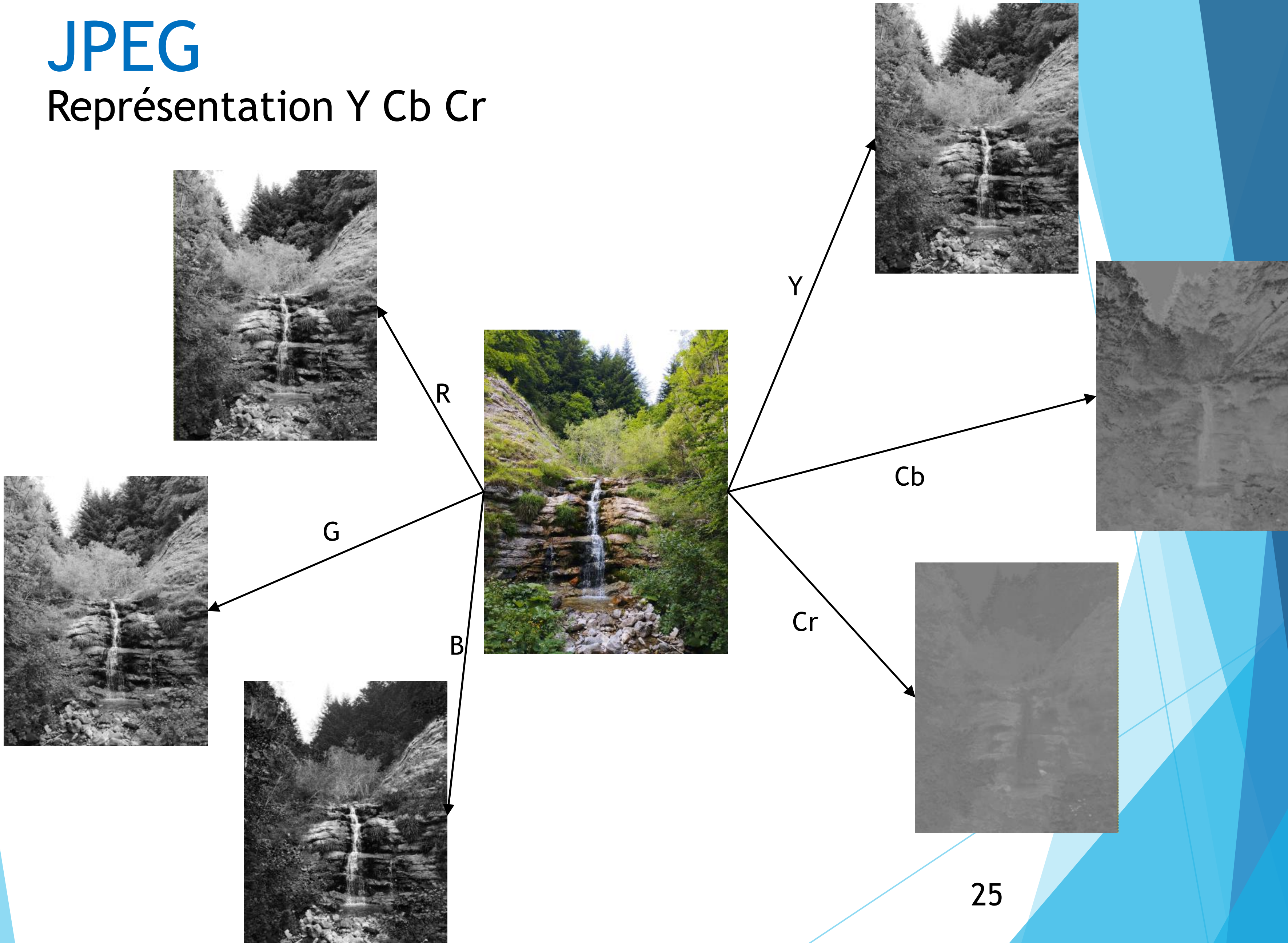
Représentation Y Cb Cr

- Représentation RGB → Représentation Y Cb Cr

$$\begin{pmatrix} y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.0 \\ 112.0 & -93.786 & -18.214 \end{pmatrix} \cdot \begin{pmatrix} r/256 \\ g/256 \\ b/256 \end{pmatrix} + \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix}$$

JPEG

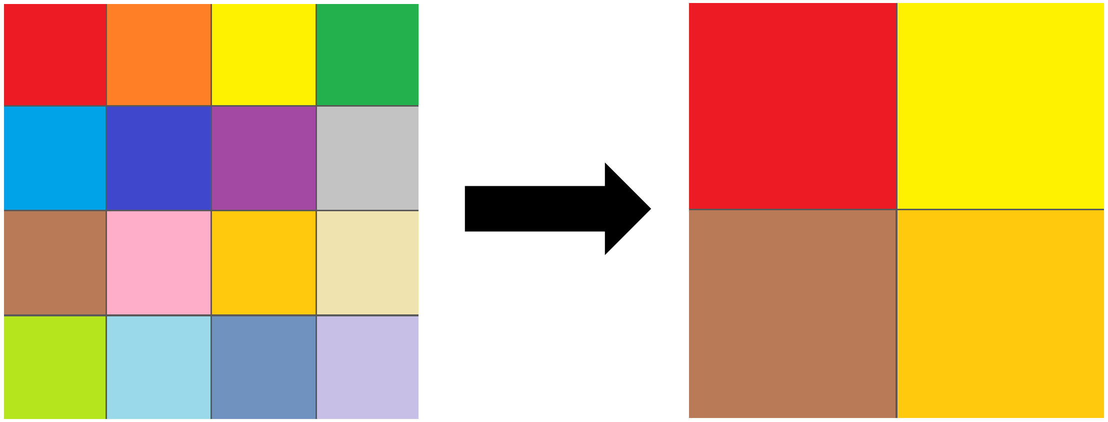
Représentation Y Cb Cr



JPEG

Sous-échantillonnage de la chrominance 4:2:0

Chaque bloc de 4 pixels est colorié de la couleur du pixel en haut à gauche



JPEG

Sous-échantillonnage de la chrominance 4:2:0

	Y	Cb	Cr
Pourcentage de pixels conservés	100 %	25%	25%
Fraction de l'image de départ	1/3	1/3 x 1/4 = 1/12	1/3 x 1/4 = 1/12

1/3 + 1/12 + 1/12 = 1/2

- Intérêt :
  - Œil humain peu sensible à la chrominance donc petites approximations ne dérangent pas
  - Image finale occupe 2x moins de place que image initiale

JPEG

Découper image en blocs 8x8

- Bloc 8x8 de luminance (Y) correspond à bloc 8x8 de l'image de départ
- Bloc 8x8 de chrominance (Cb, Cr) correspond à bloc 16x16 de l'image de départ à cause du sous-échantillonnage

Centrer les coefficients sur 0

- Chaque coefficient ← -128 pour être centré sur 0

JPEG

Transformée en cosinus discrète (DCT)

- Changer la représentation des valeurs des pixels

- Si on a n points du plan $(\frac{\pi}{2n} \cdot 1, y_0), \cdots, (\frac{\pi}{2n} \cdot (2n-1), y_{n-1})$,

  il existe $X_0, \cdots, X_{n-1} \in \mathbb{R}$ tels que si $f : x \to \sum_{i=0}^{n-1} X_i \cos(ix)$,

  $\forall k \in [\![0, n-1]\!], f(\frac{\pi}{2n} \cdot (2k+1)) = y_k$

JPEG

Transformée en cosinus discrète (DCT)

$$[y_0; \cdots ; y_{n-1}] \xrightarrow{\text{DCT}} [X_0; \cdots ; X_{n-1}]$$

$$\begin{cases} X_0 = y_0 \cos(0 \cdot \frac{\pi}{2n} \cdot 1) + \cdots + y_{n-1} \cos(0 \cdot \frac{\pi}{2n} \cdot (2n-1)) \\ \vdots \\ X_{n-1} = y_0 \cos((n-1) \cdot \frac{\pi}{2n} \cdot 1) + \cdots + y_{n-1} \cos((n-1) \cdot \frac{\pi}{2n} \cdot (2n-1)) \end{cases}$$

JPEG

Transformée en cosinus discrète (DCT)

$$n = 8$$

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \cos(\frac{\pi}{16}) & \cos(\frac{3\pi}{16}) & \cdots & \cos(\frac{15\pi}{16}) \\ \cos(\frac{2\pi}{16}) & \cos(\frac{6\pi}{16}) & \cdots & \cos(\frac{30\pi}{16}) \\ \cos(\frac{3\pi}{16}) & \cos(\frac{9\pi}{16}) & \cdots & \cos(\frac{45\pi}{16}) \\ \cos(\frac{4\pi}{16}) & \cos(\frac{12\pi}{16}) & \cdots & \cos(\frac{60\pi}{16}) \\ \cos(\frac{5\pi}{16}) & \cos(\frac{15\pi}{16}) & \cdots & \cos(\frac{75\pi}{16}) \\ \cos(\frac{6\pi}{16}) & \cos(\frac{18\pi}{16}) & \cdots & \cos(\frac{90\pi}{16}) \\ \cos(\frac{7\pi}{16}) & \cos(\frac{21\pi}{16}) & \cdots & \cos(\frac{105\pi}{16}) \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix}$$

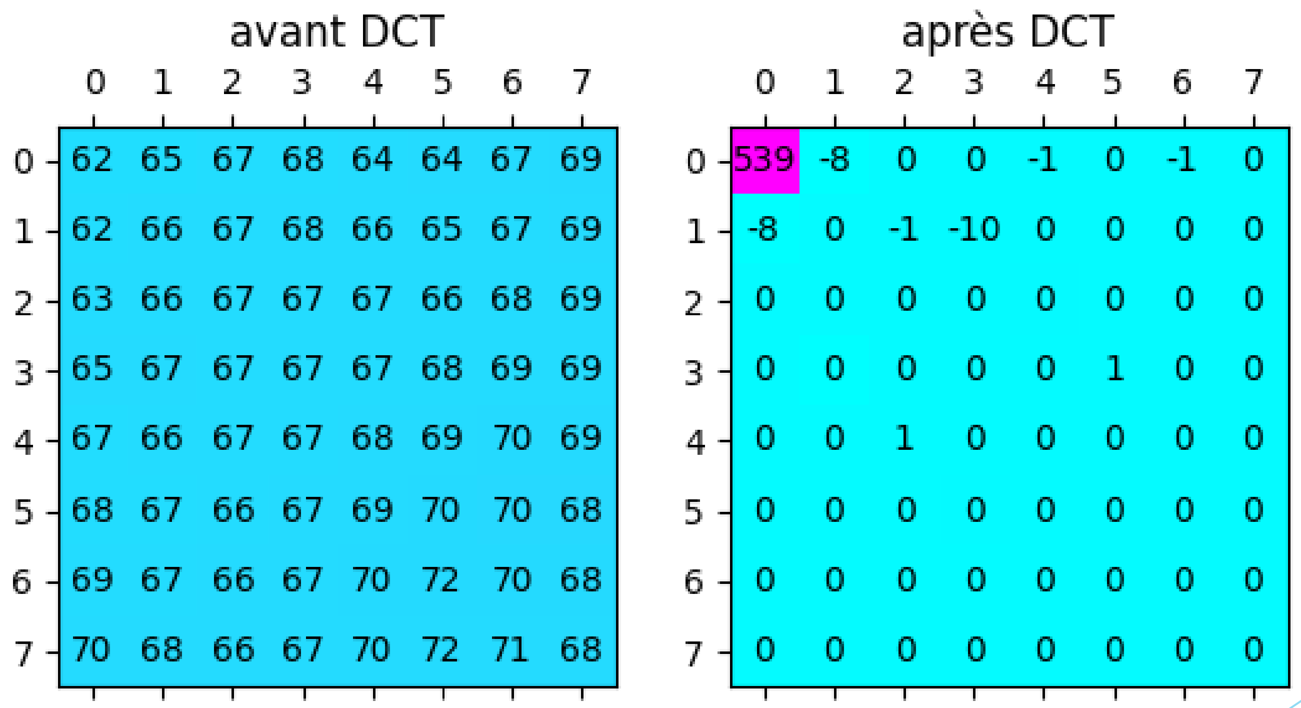
JPEG

Transformée en cosinus discrète (DCT)

- On applique DCT aux lignes puis aux colonnes de chaque bloc 8x8
- Regroupe l'énergie dans basses fréquences donc la majeure partie d'information est en haut à gauche dans la matrice

JPEG

Transformée en cosinus discrète (DCT)



https://raw.githubusercontent.com/arsnm/tipe_mp/main/code/encoder/data/energyCompaction.png

JPEG

Quantification

- Perte d'information importante
- Permet de gagner le plus de place
- Coefficients des matrices Y, Cb et Cr divisés par des scalaires donnés et arrondis à l'entier le plus proche

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

Table de luminance

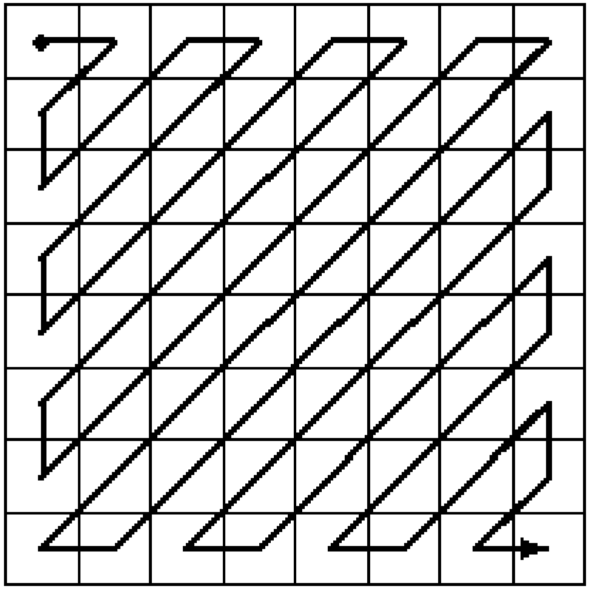
$$\begin{pmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{pmatrix}$$

Table de chrominance

JPEG

Codage

- Zigzag



Utilité :
Cases en haut à gauche non nulles, permet d'avoir de grandes séquences de 0

https://fr.wikipedia.org/wiki/JPEG#/media/Fichier:CompressionJPEGImage13.png

JPEG

Compression

- RLE (codage par longueur de plages) des 0 :

Ex : [90;40;0;0;5;2;0;1;0;0;0;0;0;0;0;0;0] → [90;40;0(x2);5;2;0;1;0(x9)]

- Huffman ou arithmétique

JPEG

Décompression

- Décodage de la compression
- Décodage RLE
- Zigzag à l'envers
- Inverse de la quantification

JPEG

Transformée en cosinus discrète inverse (IDCT)

$$[y_0; \cdots ; y_{n-1}] \xleftarrow{\text{IDCT}} [X_0; \cdots ; X_{n-1}]$$

$$\begin{cases} y_0 = \dfrac{X_0}{2} + \cdots + X_{n-1} \cos((n-1) \cdot \dfrac{\pi}{2n} \cdot 1) \\ \vdots \\ y_{n-1} = \dfrac{X_0}{2} + \cdots + X_{n-1} \cos((n-1) \cdot \dfrac{\pi}{2n} \cdot (2n-1)) \end{cases}$$

JPEG

Transformée en cosinus discrète inverse (IDCT)

$$n = 8$$

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1/2 & \cos(\frac{\pi}{16}) & \cdots & \cos(\frac{7\pi}{16}) \\ 1/2 & \cos(\frac{3\pi}{16}) & \cdots & \cos(\frac{21\pi}{16}) \\ 1/2 & \cos(\frac{5\pi}{16}) & \cdots & \cos(\frac{35\pi}{16}) \\ 1/2 & \cos(\frac{7\pi}{16}) & \cdots & \cos(\frac{49\pi}{16}) \\ 1/2 & \cos(\frac{9\pi}{16}) & \cdots & \cos(\frac{63\pi}{16}) \\ 1/2 & \cos(\frac{11\pi}{16}) & \cdots & \cos(\frac{77\pi}{16}) \\ 1/2 & \cos(\frac{13\pi}{16}) & \cdots & \cos(\frac{91\pi}{16}) \\ 1/2 & \cos(\frac{15\pi}{16}) & \cdots & \cos(\frac{105\pi}{16}) \end{pmatrix} \cdot \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix}$$

JPEG

Décompression

- Décentrer les coefficients de 0
- Réassemblage des blocs 8x8
- ~~Sous-échantillonnage inverse~~
- Y, Cb, Cr → RGB

Effectué

- Etude théorique de la compression de données
- Etude théorique de ces différentes méthodes de compression
- Implémentation en Python de la compression JPEG des fonctions :
  - RGB et YCbCr
  - Centrage coefficients sur 0
  - DCT et IDCT
  - Quantification
  - Zigzag
  - Code arithmétique
  - Encodeur code de Huffman (en Ocaml)

Annexes

Propriétés de l'entropie

Soit p une probabilité sur un ensemble fini Ω. On définit l'entropie de p :

$$\begin{aligned} H(p) &= -\sum_{\omega\in\Omega} p(\omega)\ \ln(p(\omega)) \\ &= -\sum_{\omega\in\Omega} \Phi(p(\omega)) \text{ où } \Phi : x \to x\ln(x) \end{aligned}$$

Φ est prolongeable par continuité en 0 avec $\Phi(0) = 0$

Propriétés de l'entropie

$0 \leq H(p) \leq \ln(|\Omega|) = H(U)$ où U est la probabilité uniforme sur Ω

- $\forall \omega \in \Omega,\ 0 \leq p(\omega) \leq 1$ donc $p(\omega)\ln(p(\omega)) < 0$ donc $H(p) \geq 0$
- $H(U) = -\sum_{\omega \in \Omega} \frac{1}{|\Omega|} \ln\left(\frac{1}{|\Omega|}\right) = \frac{\ln(|\Omega|)}{|\Omega|} \sum_{\omega \in \Omega} 1 = \ln(|\Omega|)$
- Φ est C^1 sur $]0,1]$, $\Phi' : x \to \ln(x) + 1$ est croissante, donc Φ' est convexe sur $[0,1]$ car sur $]0,1]$ et continue en 0
- $\Phi\left(\frac{1}{|\Omega|}\right) = -\frac{\ln(|\Omega|)}{|\Omega|} = \Phi\left(\frac{1}{|\Omega|}\sum_{\omega \in \Omega} p(\omega)\right) \leq \frac{1}{|\Omega|}\sum_{\omega \in \Omega} \Phi(p(\omega)) = -\frac{H(p)}{|\Omega|}$ donc $H(p) \leq \ln(|\Omega|)$

Propriétés de l'entropie

Si X et Y sont 2 v.a réelles indépendantes sur (Ω, P), $H(X,Y) = H(X) + H(Y)$

$$
\begin{aligned}
H(X,Y) &= -\sum_{(x,y)\in X(\Omega)\times Y(\Omega)} p\left((X,Y)=(x,y)\right)\ \ln\left(p\left((X,Y)=(x,y)\right)\right) \\
&= -\sum_{(x,y)\in X(\Omega)\times Y(\Omega)} p(X=x)p(Y=y)\ \left(\ln\left(p(X=x)\right)+\ln\left(p(Y=y)\right)\right)
\end{aligned}
$$

Mais

$$
\sum_{(x,y)\in X(\Omega)\times Y(\Omega)} p(X=x)p(Y=y)\ \ln\left(p(X=x)\right) = \left(\sum_{y\in Y(\Omega)} p(Y=y)\right)\left(\sum_{x\in X(\Omega)} p(X=x)\ln\left(p(X=x)\right)\right) = -H(X)
$$

D'où le résultat

Démonstration inégalité de Kraft

Pour un code préfixe sur un alphabet de taille D, les longueurs des lettres codées, $l_1, \cdots, l_n$ vérifient :

$$\sum_{i=1}^{n} D^{-l_i} \leq 1$$

Démonstration inégalité de Kraft

Soit $m = \max_{i \in [\![1,n]\!]}(l_i)$

Si C est préfixe, on peut placer de manière unique les lettres sur un arbre D-aire de profondeur m, chaque lettre étant une feuille.

On complète l'arbre en un arbre complet de profondeur m

Soit $i \in [\![1,n]\!]$, il y a D^{m-l_i} feuilles à la profondeur m issues des lettres codées de longueur l_i

Comme le code est préfixe, les sous-arbres issus des lettres sont disjoints. Ainsi, le nombre de feuilles à la profondeur m issues de lettres codées est $\displaystyle\sum_{i=1}^{n} D^{m-l_i}$.

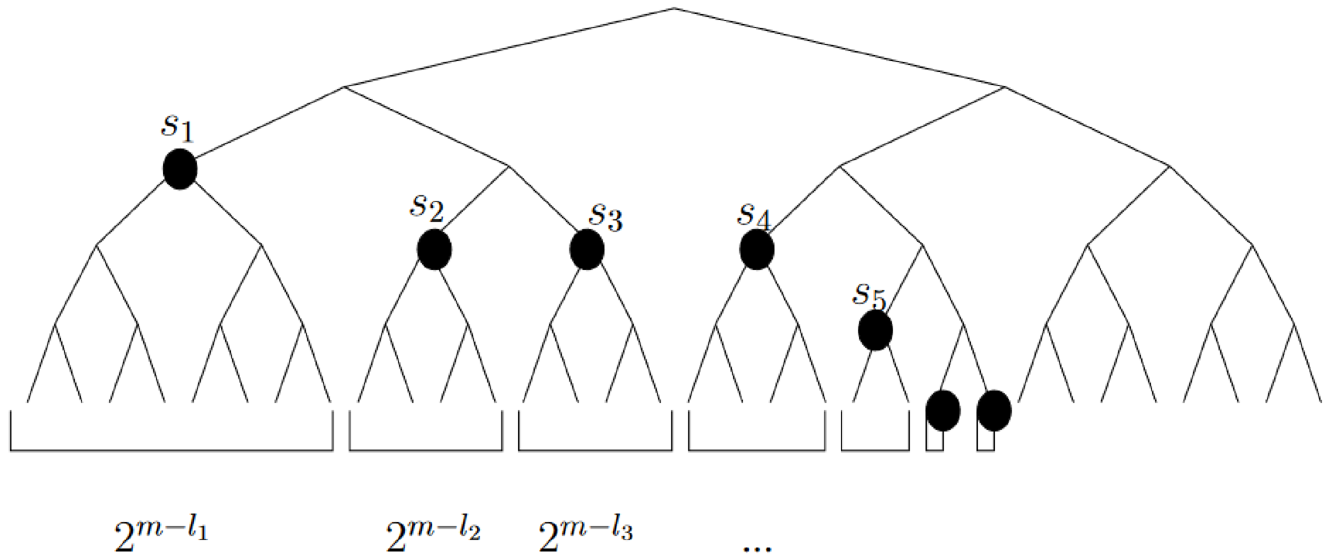
Ce nombre de feuilles est inférieur ou égal au nombre total de feuilles à la profondeur m, soit D^m. Ainsi, $\displaystyle\sum_{i=1}^{n} D^{m-l_i} \leq D^m$ donc $\displaystyle\sum_{i=1}^{n} D^{-l_i} \leq 1$

Démonstration inégalité de Kraft

Réciproquement, étant donné une suite de longueurs vérifiant cette inégalité, il existe un code préfixe dont les lettres codées ont ces longueurs.

Démonstration inégalité de Kraft

On suppose les l_i classés par ordre croissant.



Cours UFR IMAG : https://polaris.imag.fr/jean-marc.vincent/index.html/ALGO5/Codage-Shannon-Huffman.pdf

Démonstration du théorème de codage de source

Etant donné une source discrète X et un alphabet de taille D pour le codage, il existe un code C tel que

$$\frac{H(X)}{\log_2(D)} \leq L(C) < \frac{H(X)}{\log_2(D)} + 1$$

Démonstration du théorème de codage de source

$$H(X)-L(C)\log_2(D) = -\sum_{x\in\Omega} p(x)\log_2(p(x)) - \sum_{x\in\Omega} p(x)l(x)\log_2(D) = \sum_{x\in\Omega} p(x)\log_2\left(\frac{D^{-l(x)}}{p(x)}\right)$$

Par concavité de ln, $\ln(x) \leq x-1$, donc $\log_2(x) \leq \dfrac{x-1}{\ln(2)} = (x-1)\log_2(e)$

donc $H(X)-L(C)\log_2(D) \leq \log_2(e)\sum_{x\in\Omega} p(x)\left(\dfrac{D^{-l(x)}}{p(x)} - 1\right) = \log_2(e)\left(\sum_{x\in\Omega} D^{-l(x)} - 1\right)$

Donc si C est préfixe, $H(X) - L(C)\log_2(D) \leq 0$.

Donc $\dfrac{H(X)}{\log_2(D)} \leq L(C)$ pour tout code préfixe

Démonstration du théorème de codage de source

Si $x \in \Omega$, on choisit $l(x)$ tel que $D^{-l(x)} \leq p(x) < D^{-l(x)+1}$,
i.e. $l(x) = \lceil \log_D(p(x)) \rceil$.
Alors, $\sum_{x \in \Omega} D^{-l(x)} \leq 1$
D'après l'inégalité de Kraft, il existe un code préfixe C avec ces longueurs.
De plus,
$\log_2(p(x)) < (-l(x)+1)\log_2(D)$
donc $l(x) < 1 - \dfrac{log(p(x))}{\log_2(D)}$
donc $L(C) = \sum_{x \in \Omega} p(x)l(x) < \dfrac{H(X)}{\log_2(D)} + 1$

RGB → YUV et YUV →RGB

```python
#Module transforming RGB images into YCbCr
import numpy as np
import numpy.linalg as alg

mat = np.array([[65.481, 128.553, 24.966],
                [-37.797, -74.203, 112.0],
                [112.0, -93.786, -18.214]])

col = np.array([[16, 128, 128]])


def rgb_to_ycbcr(rgb:tuple) -> tuple:
    a = np.asarray(rgb)/256
    b = mat.dot(a)
    return tuple(b + col)

def ycbcr_to_rgb(t:tuple) -> tuple:
    a = np.asarray(t)
    b = alg.inv(mat)
    c = a - col
    d = b.dot(c[0])
    return tuple(256 * d)
```

Découpage en blocs 8x8 (fait par un camarade)

```python
def matrix_to_block(matrix, k:int=0, fill:int=0):
    """given a n * p matrix, return a list of k*k blocks(fill is the filler value =0 by default)

    parameters
    ----------
        -matrix : list * list
        -k: int
        -fill: Any

    return
    ------
        - list of k*k blocks"""
    n = len(matrix)
    try:
        p = len(matrix[0])
    except IndexError:
        print("matrix must be non-empty")
    list_blocks = []
    for r in range(0, n, k):
        for c in range(0, p, k):
            block = [[] for _ in range(k)]
            for i in range(k):
                try:
                    row = matrix[r + i][c:c + k]
                    while len(row) != k:
                        row.append(fill)
                except:
                    row = [fill] * k
                block[i] = row
            list_blocks.append(block)
    return list_blocks
```

Normalisation des coefficients

```python
def normalize(matrice):
    for i in range(len(m)):
        for j in range(len(m[0])):
            matrice[i][j] -= 128
    return matrice
```

DCT et IDCT

```python
import numpy as np
from math import cos, pi

mat_dct = np.array([[1., 1., 1., 1., 1., 1., 1., 1.],
                    [cos(pi/16), cos(3*pi/16), cos(5*pi/16), cos(7*pi/16), cos(9*pi/16), cos(11*pi/16), cos(13*pi/16), cos(15*pi/16)],
                    [cos(2*pi/16), cos(6*pi/16), cos(10*pi/16), cos(14*pi/16), cos(18*pi/16), cos(22*pi/16), cos(26*pi/16), cos(30*pi/16)],
                    [cos(3*pi/16), cos(9*pi/16), cos(15*pi/16), cos(21*pi/16), cos(27*pi/16), cos(33*pi/16), cos(39*pi/16), cos(45*pi/16)],
                    [cos(4*pi/16), cos(12*pi/16), cos(20*pi/16), cos(28*pi/16), cos(36*pi/16), cos(44*pi/16), cos(52*pi/16), cos(60*pi/16)],
                    [cos(5*pi/16), cos(15*pi/16), cos(25*pi/16), cos(35*pi/16), cos(45*pi/16), cos(55*pi/16), cos(65*pi/16), cos(75*pi/16)],
                    [cos(6*pi/16), cos(18*pi/16), cos(30*pi/16), cos(42*pi/16), cos(54*pi/16), cos(66*pi/16), cos(78*pi/16), cos(90*pi/16)],
                    [cos(7*pi/16), cos(21*pi/16), cos(35*pi/16), cos(49*pi/16), cos(63*pi/16), cos(77*pi/16), cos(91*pi/16), cos(105*pi/16)]])
```

DCT et IDCT

```
A = mat_dct.copy()
mat_idct = A.T
for i in range(8):
    mat_idct[i, 0] = 0.5

def dct(l):
    m = np.array([l])
    x = mat_dct.dot(m.T)
    x = np.transpose(x)
    return x.tolist()[0]

def idct(l):
    m = np.array([l])
    y = mat_idct.dot(m.T)
    y = y.T
    return y.tolist()[0]
```

DCT utilisant FFT (fait avec un camarade)

Classe complexes

```
# Module computing complex numbers
# disclaimer : this class is not made to deal with less than 1e-10 values


from numpy import arctan2, cos, pi, sin, sqrt
from math import isclose
from typing import Union, List


You, 7 months ago | 2 authors (Arsn M and others)
class Complex:
    """Computing complex numbers"""
    def __init__(self, real=0., imaginary=0.):
            self.re = real # round(real, 15)
            self.im = imaginary # round(imaginary,15)
    def __str__(self) -> str:
        if self.im == 0.:
            string = f"{self.re}"
        elif self.re == 0:
            string = f"i({self.im})"
        else:
            string = f"{self.re} + i({self.im})"
        return string
    __repr__ = __str__
    def __eq__(self, other) -> bool:
        return bool(isclose(self.re, other.re) and isclose(self.im, other.im))
    def is_null(self):
        return isclose(self.re, 0) and isclose(self.im, 0)
    def is_real(self):
        return isclose(self.im, 0)
    def is_imaginary(self):
        return isclose(self.re, 0)
```

DCT utilisant FFT (fait avec un camarade)

Classe complexes

```
def arg(self):
    """return the argument of the complex number
    return None if 0"""
    if self.is_null():
        arg = None
    elif isclose(self.re, 0) and self.im > 0:
        arg = pi / 2
    elif isclose(self.re, 0) and self.im < 0:
        arg = - pi / 2
    else:
        arg = round(arctan2(self.im, self.re), 15)
    return arg
def module(self):
    """return the module of the complex number"""
    return round(sqrt(self.re**2 + self.im**2), 15)
def conjuagate(self):
    return (Complex(self.re, -self.im))
#arithmetic
def __add__(self, other):
    return Complex(self.re + other.re, self.im + other.im)
def __sub__(self, other):
    return Complex(self.re - other.re, self.im - other.im)
def __mul__(self, other):
    real = (self.re * other.re) - (self.im * other.im)
    imaginary = (self.re * other.im) + (self.im * other.re)
    return Complex(real, imaginary)
```

DCT utilisant FFT (fait avec un camarade)

Classe complexes

```python
    def __truediv__(self, other):
        if other.is_null():
            raise ValueError("Error : dividing by 0")
        elif other.is_real():
            return Complex(self.re / other.re, self.im / other.re)
        else:
            denominator = (other.re ** 2) + (other.im ** 2)
            real = ((self.re * other.re) + (self.im * other.im)) / denominator
            imaginary = ((self.im * other.re) - (self.re * other.im)) / denominator
            return Complex(real, imaginary)

Num = Union[int, float]

def addition(*complexes:Complex) -> Complex: #partially depreciated (can still be usefull for more iterable arguments)
    """calculate the sum of complex numbers

    parameters
    ----------
        - *complexes : iterable type of Complex

    return
    ------
        - sum of the complex numbers"""

    res = Complex(0)
    for number in complexes:
        res.re += number.re
        res.im += number.im
    return res
```

DCT utilisant FFT (fait avec un camarade)

Classe complexes

```
def difference(cpx1:Complex, cpx2:Complex = Complex(0)): #fully depreciated (replaced by __sub__ Complex methods)
    """calculate the difference of two complex numbers

    parameters
    ----------
        - cpx1 : Complex number
        - cpx2 : Complex number to subtract to cpx1 (=Complex(0) by default)

    return
    ------
        -   difference of the two complex numbers"""
    res = Complex()
    res.re = cpx1.re - cpx2.re
    res.im = cpx1.im - cpx2.im
    return res

def product(*complexes:Complex) -> Complex: #partially depreciated (can still be usefull for more iterable arguments)
    """calculate the product of complex numbers

    parameters
    ----------
        - *complexes : iterable type of Complex

    return
    ------
        - product of the complex numbers"""
    res = Complex(1)
    for number in complexes:
        re = res.re * number.re - res.im * number.im
        im = res.re * number.im + res.im * number.re
        res.re = re
        res.im = im
    return res
```

DCT utilisant FFT (fait avec un camarade)

Classe complexes

```python
def exp_to_literal(arg:float, module:float = 1.0) -> Complex:
    """ return the literal expression of a complex number defined by its argument and module

    parameters
    ----------
        - arg : type(float) (should be between 0 and 2pi)
        - module : type(float) (must have a positive value)(=1 by default)

    return
    ------
        - Complex number associated"""
    assert(module >= 0), "second-argument(module) must have a positive value"
    return Complex(module*cos(arg), module*sin(arg))

def nth_root(n:int, cpx:Complex = Complex(1)) -> Complex:
    """calculate the nth root of a complex number

    parameters
    ----------
        - n : type(int)
        - complex : type(Complex) (=Complex(1) by default) (must not be Complex(0))

    return
    ------
        - list of the nth roots"""
    assert(cpx.re != 0 or cpx.im != 0), "second argument must be a non-zero complex number"
    module = cpx.module()
    arg = cpx.arg()
    if arg is not None:
        return exp_to_literal((arg/n), module**(1/n))
    else:
        return Complex(1) #Not used case but just here to ensure nth_root cannot return None
```

DCT utilisant FFT (fait avec un camarade)

Classe complexes

```
def inverse_nth_roots_unity(n:int) -> list:
    """ calculate the inversed n roots of unity

    parameter
    ---------
    - n : type(int) : must be a positive integer

    return
    ------
    - a list of Complex containing the inversed n roots of unity"""
    roots = [Complex(1) for i in range(n)]
    for k in range(0,n):
        roots[k] = exp_to_literal((-2*k*pi/n), 1.0)
    return roots

def make_complex(values:List[Num]) -> List[Complex]:
    res = []
    for value in values:
        res.extend([Complex(value)])
    return res
```

DCT utilisant FFT (fait par un camarade)

FFT

```
def FFT(vector:list) -> list:
    """calculate the fast fourier tranform of a vector

    parameters
    ----------
        -vector : list of Complex object

    return
    ------
        - 1-D fast fourier transform of the vector"""
    n = len(vector)
    assert log2(n).is_integer(), "make sure that the length of the arguement is a power of 2"
    if n == 1:
        return vector
    poly_even, poly_odd = vector[::2] , vector[1::2]
    res_even, res_odd = FFT(poly_even), FFT(poly_odd)
    res = [cpx.Complex(0)] * n
    for j in range(n//2):
        w_j = cpx.exp_to_literal(-2*pi*j/n)
        product = w_j * res_odd[j]
        res[j] = res_even[j] + product
        res[j + n//2] = res_even[j] - product
    return res
```

DCT utilisant FFT (fait par un camarade)

IFFT

```
def IFFT_aux(vector:list) -> list:
    """auxiliary function that makes the recursive steps of the IFFT algorithm
    parameters
    ----------
        -vector : list of Complex object

    return
    ------
        - partial inverse of the 1-D fast fourier transform of the vector (lack the division by n)"""
    n = len(vector)
    assert log2(n).is_integer(), "make sure that the length of the arguement is a power of 2"
    if n == 1:
        return vector
    poly_even, poly_odd = vector[::2] , vector[1::2]
    res_even, res_odd = IFFT_aux(poly_even), IFFT_aux(poly_odd)
    res = [cpx.Complex(0)] * n
    for j in range(n//2):
        w_j = cpx.exp_to_literal((2 * pi * j) / n)
        product = w_j * res_odd[j]
        res[j] = res_even[j] + product
        res[j + n//2] = res_even[j] - product
    return res
```

DCT utilisant FFT (fait par un camarade)

IFFT

```python
def IFFT(vector:list) -> list:
    """caclulate the inverse of the fast fourier tranform of a vector (in order to have ifft(fft(poly)) == poly)

    parameters
    ----------
        -vector : list of Complex object

    return
    ------
        - inverse of the 1-D fast fourier transform of the vector"""
    n = len(vector)
    res = IFFT_aux(vector)
    for i in range(n):
        res[i] = res[i] / cpx.Complex(n)
    return res
```

DCT utilisant FFT (fait par un camarade)

DCT

```python
def DCT(vector:list, orthogonalize:bool =False, norm="forward"):
    """calculate the one-dimensional type-II discrete cosine tranform of a matrix (MAKHOUL) (using the FFT function previously defined)

    parameters
    ----------
        - vector: list of Numerical Object

    return
    ------
        - discrete cosine tranform of the input"""
    N = len(vector)
    temp = vector[ : : 2] + vector[-1 - N % 2 : : -2]
    temp = FFT(temp)
    factor = - pi / (N * 2)
    result = [2 * (val * (cpx.exp_to_literal(i * factor))).re for (i, val) in enumerate(temp)]
    if orthogonalize:
        result[0] *= 2 ** (-1 / 2)
    if norm == "ortho":
        result[0] *= (N) **(-1 / 2)
        result[1::] = [(2 / N) ** (1 / 2) * result[i] for i in range(1, len(result))]
    return result
```

DCT utilisant FFT (fait par un camarade)

IDCT

```python
def IDCT(vector:list):
    """calculate the one-dimensional "inverse" type-III discrete cosine tranform of a matrix (MAKHOUL) (using the FFT function previously defined)

    parameters
    --------
        - vector: list of Numerical Object

    return
    ------
        - type-III discrete cosine tranform of the input"""
    N = len(vector)
    factor = - pi / (N * 2)
    temp = [(cpx.Complex(val) if i > 0 else (cpx.Complex(val) / cpx.Complex(2))) * cpx.exp_to_literal(i * factor) for (i, val) in enumerate(vector)]
    temp = FFT(temp)
    temp = [val.re for val in temp]
    result = [None] * N
    result[ : : 2] = temp[ : (N + 1) // 2]
    result[-1 - N % 2 : : -2] = temp[(N + 1) // 2 : ]
    return result
```

Quantification

```python
import numpy as np

def load_quantization_table(component):
    if component == 'lum':
        q = np.array([[16, 11, 10, 16, 24, 40, 51, 61],
                      [12, 12, 14, 19, 26, 48, 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]])


    elif component == 'chrom':
        q = np.array([[17, 18, 24, 47, 99, 99, 99, 99],
                      [18, 21, 26, 66, 99, 99, 99, 99],
                      [24, 26, 56, 99, 99, 99, 99, 99],
                      [47, 66, 99, 99, 99, 99, 99, 99],
                      [99, 99, 99, 99, 99, 99, 99, 99],
                      [99, 99, 99, 99, 99, 99, 99, 99],
                      [99, 99, 99, 99, 99, 99, 99, 99],
                      [99, 99, 99, 99, 99, 99, 99, 99]])
    else:
        raise ValueError((
            "component should be either 'lum' or 'chrom', "
            "but '{comp}' was found").format(comp=component))

    return q
```

```python
def quantize(block, component):
    output = np.zeros((8,8))
    quantiz_matrix = load_quantization_table(component)
    for i in range(8):
        for j in range(8):
            output[i, j] = block[i, j] // quantiz_matrix[i, j]
    return output
```

Zigzag

```python
def zigzag(matrix):
    """function that reads the coefficients of a square matrix in zigzag

    parameters
    ----------
        - matrix : (array of array)-like (size must be a square)

    return
    ------
        - list of coefficients read in zigzag"""
    n = len(matrix)
    if n != len(matrix[0]):
        raise ValueError("given paramater must be a square matrix of at least size 1x1")
    solution = [[] for i in range(n+n-1)]
    for i in range(n):
        for j in range(n):
            sum=i+j
            if(sum%2 ==0):
                #add at beginning
                solution[sum].insert(0,matrix[i][j])
            else:
                #add at end of the list
                solution[sum].append(matrix[i][j])

    return [item for sublist in solution for item in sublist]
```

```python
def zigzag_inverse(matrix):
    """return the reversed list returned by `zigzag(matrix)`"""
    return zigzag(matrix)[::-1]
```

Code arithmétique

```python
def proba(data):
    '''
    Créer le dictionnaire de probabilités d'apparition des différents caractères
    '''
    assert len(data) != 0
    d = {}
    for x in data:
        d[x] = d.get(x, 0) + (1/len(data))
    return d

def create_int(data):
    '''
    Créer le dictionnaire des intervalles des différents caractères connaissant les données
    '''
    p = proba(data)
    d = {}
    n = 0.
    for c, v in p.items():
        d[c] = (n, n+v)
        n += v
    return d

def create_int2(p):
    '''
    Créer le dictionnaire des intervalles des différents caractères connaissant les probas des différents caractères
    '''
    d = {}
    n = 0.
    for c, v in p.items():
        d[c] = (n, n+v)
        n += v
    return d
```

```python
def encode(data):
    '''
    effectue l'encodage des données
    '''
    int = create_int(data)
    value = (0., 1.)
    for x in data:
        d = value[1] - value[0]
        sup = value[0] + d * int[x][1]
        inf = value[0] + d * int[x][0]
        value = (inf, sup)
    return (value[0] + value[1])/2
```

```python
def appartient(x, int):
    '''
    teste l'appartenance de x à un intervalle fermé à gauche et ouvert à droite
    '''
    assert len(int) == 2
    return x>= int[0] and x< int[1]


def inverse(dic):
    '''
    renvoie le dictionnaire où les clés et valeurs sont inversées
    '''
    d = {}
    for c, v in dic.items():
        d[v] = c
    return d

def decode(n, p, nbr_carac):
    d = inverse(create_int2(p))
    res = []
    i = n
    while len(res) < nbr_carac:
        for c, v in d.items():
            if appartient(i, c):
                res.append(v)
                i = (i - c[0])/(c[1] - c[0])
                break
    return res
```

Encodeur code de Huffman

Type arbre

```
type abr = V | F of int | N of abr*abr
```

Encodeur code de Huffman

File de priorité min

```
type 'a heap = {a : 'a array; mutable n : int}

'a heap -> int -> int -> unit
let swap h i j =
  let u = h.a.(i) in
  h.a.(i) <- h.a.(j);
  h.a.(j) <- u;;

'a heap -> int -> unit
let rec up h i =
  let p = (i - 1)/2 in
  if i <> 0 && h.a.(p) > h.a.(i)
  then (swap h i p; up h p);;


'a heap -> int -> unit
let rec down h i =
  let m = ref i in
  if 2*i + 1 < h.n && h.a.(2*i + 1) < h.a.(!m)
  then m := 2*i + 1;
  if 2*i + 2 < h.n && h.a.(2*i + 2) < h.a.(!m)
  then m := 2*i + 2;
  if !m <> i
  then (swap h i !m; down h !m);;
```

Encodeur code de Huffman

File de priorité min

```
'a heap -> 'a -> unit
let add h e =
  h.a.(h.n) <- e;
  up h h.n;
  h.n <- h.n + 1;;

'a heap -> 'a
let rec extract_min h =
  swap h 0 (h.n - 1);
  h.n <- h.n - 1;
  down h 0;
  h.a.(h.n);;

('a * 'b) heap -> int -> 'a * 'b -> unit
let update h i v =
  let p = h.a.(i) in
  h.a.(i) <- v;
  if fst v < fst p then up h i else down h i;;
```

Encodeur code de Huffman

Nombre d’occurrences

```
int array -> int array
let occ s =
  let n = Array.length s in
  let a = Array.make 26 0 in
  for i = 0 to n-1 do
    a.(s.(i)) <- a.(s.(i)) + 1
  done;
  a;;
```

Encodeur code de Huffman

Construction de l'arbre

```
int array -> abr
let constr_arb s =
  let o = occ s in
  let f = {a = Array.make 26 (0, V); n = 0} in
  for k = 0 to 25 do
    if o.(k) <> 0 then add f (o.(k), F(k))
  done;
  while f.n > 1 do
    let o1, a1 = extract_min f in
    let o2, a2 = extract_min f in
    let new_o = o1 + o2 in
    let new_a = N(a1, a2) in
    add f (new_o, new_a)
  done;
  snd f.a.(0);;
```

Encodeur code de Huffman

Code de chaque caractère

```
int array -> string array
let code s =
  let a = constr_arb s in
  let c = Array.make 26 ("") in
  let rec aux a i s =
    match a with
    | V -> ""
    | F(x) -> if x = i then s else ""
    | N(a1, a2) -> if aux a1 i (s^"0") = "" then aux a2 i (s^"1") else aux a1 i (s^"0") in
  for i = 0 to 25 do
    c.(i) <- aux a i ""
  done;
  c;;
```

Encodeur code de Huffman

Encodage

```
int array -> string
let codage s =
  let c = code s in
  let n = Array.length s in
  let res = ref "" in
  let mot = s in
  for i = 0 to n-1 do
    res := !res ^ c.(mot.(i))
  done;
  !res;;
```

Code de Huffman (fait par un camarade)

```
from collections import Counter, namedtuple
from heapq import heapify, heappop, heappush


# Node in a Huffman Tree
Node = namedtuple("Node", ["char", "freq"])

arsnm, 5 months ago | 1 author (arsnm)
class HuffmanCompressor:
    """Huffman compression implementation"""
    def __init__(self):
        self.encoding_table = {}
        self.decoding_table = {}
```

Code de Huffman (fait par un camarade)

```python
def build_tables(self, s: str):
    """create both the encodingn and decoding tables of a given string

parameters
----------
    -s : string used to build the tables

return
------
    - fill both the encoding and decoding table of the given class instance"""

    freq_table = Counter(s)

    # create a heap of the nodes in the tree
    heap = []
    for char, freq in freq_table.items():
        heap.append(Node(char, freq))
    heapify(heap)

    # create the Huffman tree
    while len(heap) > 1:
        left_node = heappop(heap)
        right_node = heappop(heap)
        combined_node = Node(None, left_node.freq + right_node.freq)
        heappush(heap, combined_node)

    def build_encoding_table(node, code=''):
        if node.char is not None:
            # if the node is a leaf, add it to the encoding table
            self.encoding_table[node.char] = code
            return
        # if the node is not a leaf, recursively build the encoding table
        build_encoding_table(node.left, code + '0')
        build_encoding_table(node.right, code + '1')

    build_encoding_table(heap[0])
```

Code de Huffman (fait par un camarade)

```
    def build_decoding_table(node, code=''):
        if node.char is not None:
            # if the node is a leaf, add it to the decoding table
            self.decoding_table[code] = node.char
            return
        # if the node is not a leaf, recursively build the decoding table
        build_decoding_table(node.left, code + "0")
        build_decoding_table(node.right, code + "1")

    build_decoding_table(heap[0])

def compress(self, s: str) -> str:
    """compress the inputed string

parameters
----------
    -s : string to be compressed

return
------
    - compressed string"""
    compressed = ""
    for char in s:
        compressed += self.encoding_table[char]
    return compressed
```

Code de Huffman (fait par un camarade)

```
def decompress(self, compressed: str) -> str:
    """decompress the inputed string

parameters
----------
    -s : string to be compressed

return
------
    - decompressed string"""
    decompressed = ""
    i = 0
    while i < len(compressed):
        for j in range(i+1, len(compressed)+1):
            if compressed[i:j] in self.decoding_table:
                decompressed += self.decoding_table[compressed[i:j]]
                i = j
                break

    return decompressed
```

Encodeur JPEG (fait par un camarade)

```
from math import ceil
import cv2
import numpy as np
from PIL import Image
from pathlib import Path

from utils import *
from huffman import *


def padding(im, mh, mw):
    """
    pad use boundary pixels so that its height and width are
    the multiple of the height and width of MCUs, respectively
    """
    h, w, d = im.shape
    if h % mh == 0 and w % mw == 0:
        return im
    hh, ww = ceil(h / mh) * mh, ceil(w / mw) * mw
    im_ex = np.zeros_like(im, shape=(hh, ww, d))
    im_ex[:h, :w] = im
    im_ex[:, w:] = im_ex[:, w - 1 : w]
    im_ex[h:, :] = im_ex[h - 1 : h, :]
    return im_ex


mcu_sizes = {
    "4:2:0": (BH * 2, BW * 2),
    "4:1:1": (BH * 2, BW * 2),
    "4:2:2": (BH, BW * 2),
    "4:4:4": (BH, BW),
}
```

Encodeur JPEG (fait par un camarade)

```
def scan_blocks(mcu, mh, mw):
    """
    scan MCU to blocks for DPCM, for 4:2:0, the scan order is as follows:
    --------- | ---------
    | 0 | 1 | | | 4 | 5 |
    --------- | ---------
    | 2 | 3 | | | 6 | 7 |
    --------- | ---------
    """
    blocks = (
        mcu.reshape(-1, mh // BH, BH, mw // BW, BW).swapaxes(2, 3).reshape(-1, BH, BW)
    )
    return blocks


def DCT(blocks):
    dct = np.zeros_like(blocks)
    for i in range(blocks.shape[0]):
        dct[i] = cv2.dct(blocks[i])
    return dct


def zigzag_encode(dct):
    trace = zigzag_points(BH, BW)
    zz = np.zeros_like(dct).reshape(-1, BH * BW)
    for i, p in enumerate(trace):
        zz[:, i] = dct[:, p[0], p[1]]
    return zz


def quantization(dct, table):
    ret = dct / table[None]
    return np.round(ret).astype(np.int32)
```

Encodeur JPEG (fait par un camarade)

```
def DPCM(dct):
    """
    encode the DC differences
    """
    dc_pred = dct.copy()
    dc_pred[1:, 0] = dct[1:, 0] - dct[:-1, 0]
    return dc_pred


def run_length_encode(arr):
    # determine where the sequence is ending prematurely
    last_nonzero = -1
    for i, elem in enumerate(arr):
        if elem != 0:
            last_nonzero = i
    rss, values = [], []
    run_length = 0
    for i, elem in enumerate(arr):
        if i > last_nonzero:
            rss.append(0)
            values.append(0)
            break
        elif elem == 0 and run_length < 15:
            run_length += 1
        else:
            size = bits_required(elem)
            rss.append((run_length << 4) + size)
            values.append(elem)
            run_length = 0
    return rss, values
```

Encodeur JPEG (fait par un camarade)

```
def DPCM(dct):
    """
    encode the DC differences
    """
    dc_pred = dct.copy()
    dc_pred[1:, 0] = dct[1:, 0] - dct[:-1, 0]
    return dc_pred


def run_length_encode(arr):
    # determine where the sequence is ending prematurely
    last_nonzero = -1
    for i, elem in enumerate(arr):
        if elem != 0:
            last_nonzero = i
    rss, values = [], []
    run_length = 0
    for i, elem in enumerate(arr):
        if i > last_nonzero:
            rss.append(0)
            values.append(0)
            break
        elif elem == 0 and run_length < 15:
            run_length += 1
        else:
            size = bits_required(elem)
            rss.append((run_length << 4) + size)
            values.append(elem)
            run_length = 0
    return rss, values
```

Encodeur JPEG (fait par un camarade)

```
def encode_header(qts, hts, cop_infos, height, width):
    writer = BytesWriter()
    add_bytes = writer.add_bytes
    add_bytes(
        MARKER.SOI,
        MARKER.APP0,
        b"\x00\x10",  # length = 16
        b"JFIF\x00",  # identifier = JFIF0
        b"\x01\x01",  # version
        b"\x00",  # unit
        b"\x00\x01",  # x density
        b"\x00\x01",  # y density
        b"\x00\x00",  # thumbnail data
    )
    for id_, qt in enumerate(qts):
        add_bytes(
            MARKER.DQT,
            b"\x00C",  # length = 67
            # precision (8 bits), table id, = 0, id_
            int2bytes(id_, 1),
            qt.astype(np.uint8).tobytes(),
        )
    cop_num = len(cop_infos)
    add_bytes(
        MARKER.SOF0,
        int2bytes(8 + 3 * cop_num, 2),  # length
        int2bytes(8, 1),  # 8 bit precision
        int2bytes(height, 2),
        int2bytes(width, 2),
        int2bytes(cop_num, 1),
    )
    add_bytes(*[info.encode_SOF0_info() for info in cop_infos])
```

Encodeur JPEG (fait par un camarade)

```
    # type << 4 + id, (type 0: DC, 1 : AC)
    type_ids = [b"\x00", b"\x10", b"\x01", b"\x11"]
    for type_id, ht in zip(type_ids, hts):
        count, weigh = convert_huffman_table(ht)
        ht_bytes = count.tobytes() + weigh.tobytes()
        add_bytes(
            MARKER.DHT,
            int2bytes(len(ht_bytes) + 3, 2),  # length
            type_id,
            ht_bytes,
        )

    add_bytes(
        MARKER.SOS,
        int2bytes(6 + cop_num * 2, 2),  # length
        int2bytes(cop_num, 1),
    )
    add_bytes(*[info.encode_SOS_info() for info in cop_infos])
    add_bytes(b"\x00\x3f\x00")
    return writer


def encode_mcu(mcu, hts):
    bit_stream = BitStreamWriter()
    for cur in mcu:
        for dct, (dc_ht, ac_ht) in zip(cur, hts):
            dc_code = encode_2s_complement(dct[0])
            container = [dc_ht[len(dc_code)], dc_code]
            rss, values = run_length_encode(dct[1:])
            for rs, v in zip(rss, values):
                container.append(ac_ht[rs])
                container.append(encode_2s_complement(v))
            bitstring = "".join(container)
            bit_stream.write_bitstring(bitstring)
    return bit_stream.to_bytes()
```

Encodeur JPEG (fait par un camarade)

```
def encode_jpeg(im, quality=95, subsample="4:2:0", use_rm_ht=True):
    im = np.expand_dims(im, axis=-1) if im.ndim == 2 else im
    height, width, depth = im.shape

    mh, mw = mcu_sizes[subsample] if depth == 3 else (BH, BW)
    im = padding(im, mh, mw)
    im = RGB2YCbCr(im) if depth == 3 else im

    # DC level shift for luminance,
    # the shift of chroma was completed by color conversion
    Y_im = im[:, :, 0] - 128
    # divide image into MCUs
    mcu = divide_blocks(Y_im, mh, mw)
    # MCU to blocks, for luminance there are more than one blocks in each MCU
    Y = scan_blocks(mcu, mh, mw)
    Y_dct = DCT(Y)
    # the quantization table was already processed by zigzag scan,
    # so we apply zigzag encoding to DCT block first
    Y_z = zigzag_encode(Y_dct)
    qt_y = load_quantization_table(quality, "lum")
    Y_q = quantization(Y_z, qt_y)
    Y_p = DPCM(Y_q)
    # whether to use recommended huffman table
    if use_rm_ht:
        Y_dc_ht, Y_ac_ht = reverse(RM_Y_DC), reverse(RM_Y_AC)
    else:
        Y_dc_ht = jpegCreateHuffmanTable(np.vectorize(bits_required)(Y_p[:, 0]))
        Y_ac_ht = jpegCreateHuffmanTable(
            flatten(run_length_encode(Y_p[i, 1:])[0] for i in range(Y_p.shape[0]))
        )
    qts, hts = [qt_y], [Y_dc_ht, Y_ac_ht]
    cop_infos = [ComponentInfo(1, mw // BW, mh // BH, 0, 0, 0)]
    # the number of Y DCT blocks in an MCU
    num = (mw // BW) * (mh // BH)
    mcu_hts = [(Y_dc_ht, Y_ac_ht) for _ in range(num)]
    # assign DCT blocks to MCUs
    mcu_ = Y_p.reshape(-1, num, BH * BW)
```

Encodeur JPEG (fait par un camarade)

```
if depth == 3:
    # chroma subsample
    ch = im[:: mh // BH, :: mw // BW, 1:]
    Cb = divide_blocks(ch[:, :, 0], BH, BW)
    Cr = divide_blocks(ch[:, :, 1], BH, BW)
    Cb_dct, Cr_dct = DCT(Cb), DCT(Cr)
    Cb_z, Cr_z = zigzag_encode(Cb_dct), zigzag_encode(Cr_dct)
    qt_c = load_quantization_table(quality, "chr")
    Cb_q, Cr_q = quantization(Cb_z, qt_c), quantization(Cr_z, qt_c)
    Cb_p, Cr_p = DPCM(Cb_q), DPCM(Cr_q)
    if use_rm_ht:
        C_dc_ht, C_ac_ht = reverse(RM_C_DC), reverse(RM_C_AC)
    else:
        ch_ = np.concatenate([Cb_p, Cr_p], axis=0)
        C_dc_ht = jpegCreateHuffmanTable(np.vectorize(bits_required)(ch_[:, 0]))
        C_ac_ht = jpegCreateHuffmanTable(
            flatten(run_length_encode(ch_[i, 1:])[0] for i in range(ch_.shape[0]))
        )
    qts.append(qt_c), hts.extend([C_dc_ht, C_ac_ht])
    cop_infos.extend(
        [ComponentInfo(2, 1, 1, 1, 1, 1), ComponentInfo(3, 1, 1, 1, 1, 1)]
    )
    mcu_hts.extend((C_dc_ht, C_ac_ht) for _ in range(2))
    mcu_ = np.concatenate([mcu_, Cb_p[:, None], Cr_p[:, None]], axis=1)

writer = encode_header(qts, hts, cop_infos, height, width)
bytes_ = encode_mcu(mcu_, mcu_hts)
writer.write(bytes_.replace(b"\xff", b"\xff\x00"))
writer.write(MARKER.EOI)
return writer.getvalue()
```

Encodeur JPEG (fait par un camarade)

```
def write_jpeg(filename, im, quality=95, subsample="4:2:0", use_rm_ht=True):
    bytes_ = encode_jpeg(im, quality, subsample, use_rm_ht)
    Path(filename).write_bytes(bytes_)


def main():
    im = Image.open("./data/decoded/image.ppm")
    write_jpeg("data/image.jpg", np.array(im), 5, "4:2:2", False)
    im.save("data/color-pillow.jpg", subsampling="4:2:2", quality=95)
```

Exemple d’images compressées

Image initiale : 3,4 Mo

300 Ko (11 fois moins de place)

Exemple d'images compressées

Image initiale : 3,4 Mo

20 Ko (170 fois moins de place)

Exemple d’images compressées

Image initiale : 10,3 Mo

2200 Ko = 2,2 Mo (4,6 fois moins de place)

Image initiale : https://histoiregeoenvideos.wordpress.com/2021/01/27/terminale-le-rayonnement-de-la-france/

Exemple d’images compressées

Image initiale : 10,3 Mo

500 Ko (20,6 fois moins de place)

Image initiale : https://histoiregeoenvideos.wordpress.com/2021/01/27/terminale-le-rayonnement-de-la-france/

Exemple d’images compressées

Image initiale : 10,3 Mo

75 Ko (137 fois moins de place)

Image initiale : https://histoiregeoenvideos.wordpress.com/2021/01/27/terminale-le-rayonnement-de-la-france/