TIPE - LA COMPRESSION DE DONNÉES

Et son application aux images

Arsène MALLET

Candidat - 14873

06/06/2023

Introduction

- **Compression** : maximum d'information avec une taille minimal
- Deux types : **avec** (lossy) ou **sans** perte (lossless).
- Dans le domaine urbain : beaucoup d'information $\rightarrow$ compression $\Rightarrow$ **stocker et traiter efficacement**

Procédés de Compression

- Compression sans perte : **entropique** et **algorithmique**
- Une **réorganisation** des données
- Application de **transformées mathématiques**
- Implémentation de l'**algorithme JPEG**

Entropie et Codage Optimal (1)

Théorie de l'Information de Shannon

Théorie probabiliste **quantifiant l'information** d'un ensemble de messages.

Définition - Entropie

Pour une source X comportant n symboles, un symbole x_i ayant une probabilité $p_i = \mathbb{P}(X = x_i)$ d'apparaître, l'entropie H est définie par :

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

Entropie et Codage Optimal (2)

Définition - **Code de Source**

Un **code de source C** pour une variable aléatoire X de distribution de probabilité p, est une application de Ω (ensemble des symboles sources) vers A^* (où A est l'alphabet du code).

Définition - Code **Uniquement Décodable**

Un code est dit **uniquement décodable** si

$$\forall x, y \in \Omega^+, \; x \neq y \implies C^+(x) \neq C^+(y)$$

Entropie et Codage Optimal (3)

Définition - Code **Préfixe**

Un code est dit préfixe si **aucun mot de code n'est le préfixe d'un autre mot** de code

Rq. : Code **préfixe** $\implies$ **code uniquement décodable**

Un code **non-préfixe**

a	0
b	1
c	01

$ab = 01 = c \rightarrow$ **Non uniquement décodable**

Un code **préfixe**

a	0
b	10
c	11

Chaque code est **unique**

Entropie et Codage Optimal (4)

Inégalité de Kraft

Pour un code défini sur un alphabet de taille D, et un alphabet de source Ω de taille $|\Omega|$, alors il est **préfixé** si et seulement si

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

$l_i =$ longueur des mots du codes

Théorème du Codage de Source - Shannon 1948

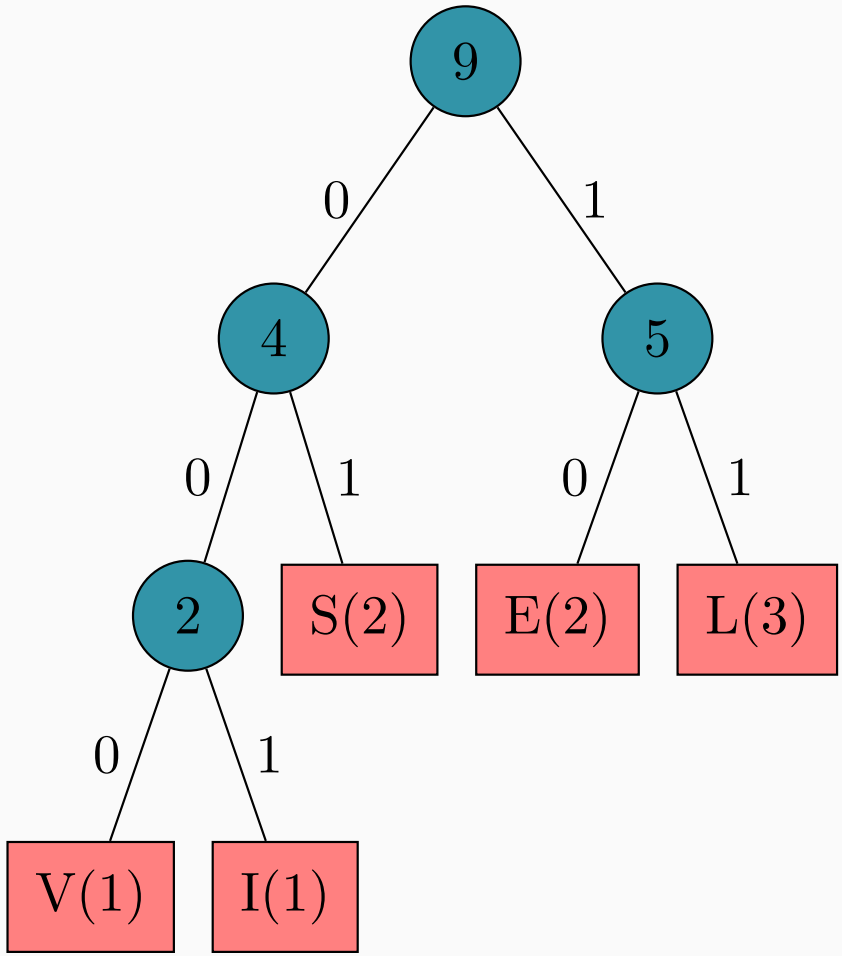
Lorsque l'efficacité de la compression augmente, la longueur moyenne du code tend vers l'entropie H.

Codage De Huffman (1)

- Codage **optimal** au niveau **symbole**, à longueur **variable**
- Impose un **nombre entier de bit** pour un symbole
- Exemple de codage de "LES VILLES" :

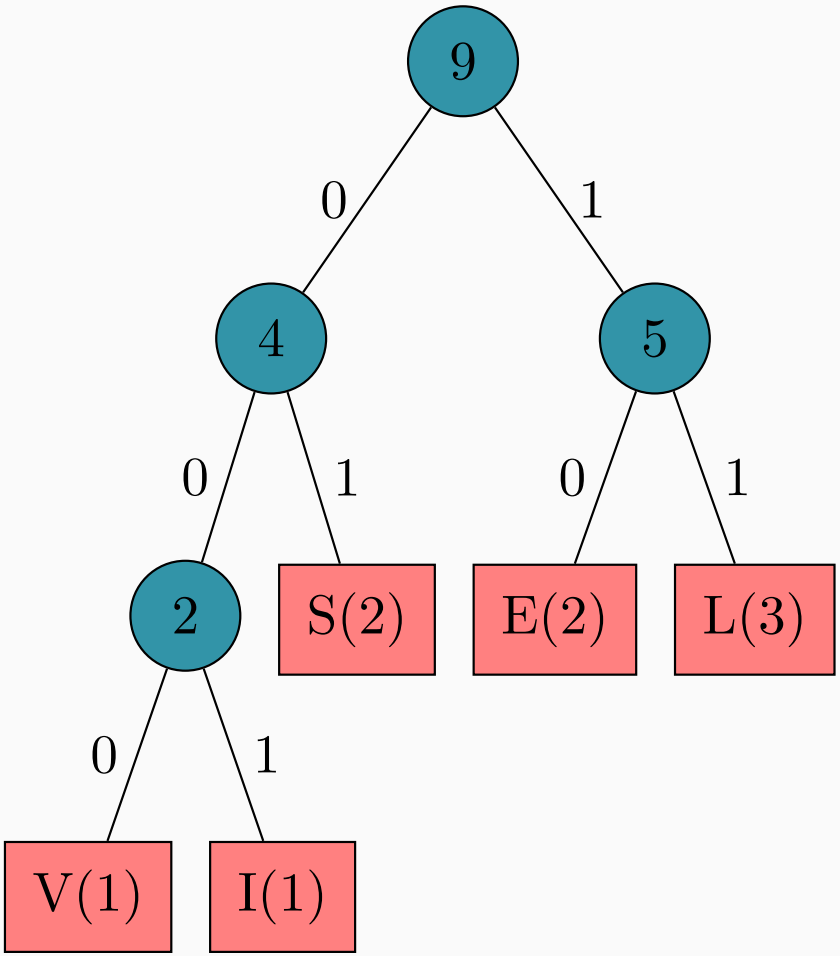
symbole source	fréquence
L	3
E	2
S	2
V	1
I	1

Arbre de Huffman de "LES VILLES"



Codage De Huffman (2)

Arbre de Huffman de "LES VILLES"



- **Table de code** de Huffman

symbole source	code
L	11
E	10
S	01
V	000
I	001

- **Code** de Huffman : 11100100000111111001

Codage Arithmétique (1)

- Codage **optimal** au niveau **bit**, à longueur **variable**
- Principe : codage **par morceaux** et non par symbole (Huffman)
- Exemple de codage de "VILLE" :

symbole source	probabilité	intervalle
V	$1/5$	$[0; 0,2[$
I	$1/5$	$[0,2; 0,4[$
L	$2/5$	$[0,4; 0,8[$
E	$1/5$	$[0,8; 1[$

Ajout d'un symbole s :

1. $BB = BS - BI$
2. $BS \leftarrow BI + BB \times BS_s$
3. $BI \leftarrow BI + BB \times BI_s$

Codage Arithmétique (2)

symbole source	probabilité	intervalle
V	$1/5$	$[0; 0.2[$
I	$1/5$	$[0.2; 0.4[$
L	$2/5$	$[0.4; 0.8[$
E	$1/5$	$[0.8; 1[$

Ajout de $s = \text{V}$:

$BS = BB = 1,\ BI = 0,\ BS_s = 0.2,\ BI_s = 0$

$BS \leftarrow 0 + 1 \times 0.2 = 0.2$

$BI \leftarrow 0 + 1 \times 0 = 0$

Ajout, ensuite, de $s' = \text{I}$:

$BS = BB = 0.2,\ BI = 0,\ BS_{s'} = 0.4,\ BI_{s'} = 0.2$

$BS \leftarrow 0 + 0.2 \times 0.4 = 0.08$

$BI \leftarrow 0 + 0.2 \times 0.2 = 0.04$

...

Codage Arithmétique (3)

symbole source	probabilité	intervalle
V	$1/5$	$[0; 0.2[$
I	$1/5$	$[0.2; 0.4[$
L	$2/5$	$[0.4; 0.8[$
E	$1/5$	$[0.8; 1[$

Code de "VILLE" : $x \in [0.06752; 0.0688]$
Par exemple $x = 0.068$ fonctionne.

Décompression :

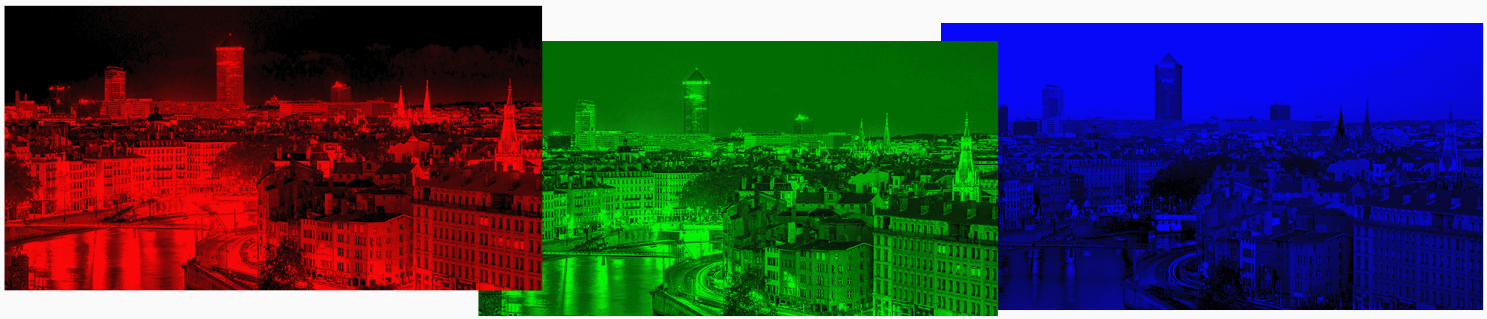
1. $x \in [0; 0.2] \rightarrow$ V
2. $x \leftarrow \frac{x - BI_V}{p_V} = 0.34$
3. $x \in [0.2; 0.4] \rightarrow$ VI
4. $x \leftarrow \frac{x - BI_I}{p_I} = 0.7$

...

5. Mot **décodé** : VILLE

La Représentation d'Image

$$\text{Image} = \begin{pmatrix} (r,g,b)_{1,1} & \dots & (r,g,b)_{1,p} \\ \vdots & \ddots & \vdots \\ (r,g,b)_{n,1} & \dots & (r,g,b)_{n,p} \end{pmatrix}$$



La Représentation YCbCr

Transformation YCbCr

$$\varphi : [\![0, 255]\!]^3 \to [\![0, 255]\!] \times [\![-128, 127]\!]^2$$
$$X = (r, g, b) \longmapsto TX = (Y, Cb, Cr)$$

$$T = 255 \begin{pmatrix} K_r & K_r & K_b \\ -\frac{1}{2} \cdot \frac{K_r}{1-K_b} & -\frac{1}{2} \cdot \frac{K_g}{1-K_b} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \cdot \frac{K_g}{1-K_r} & -\frac{1}{2} \cdot \frac{K_b}{1-K_r} \end{pmatrix} \text{ et } K_r + K_g + K_b = 1$$

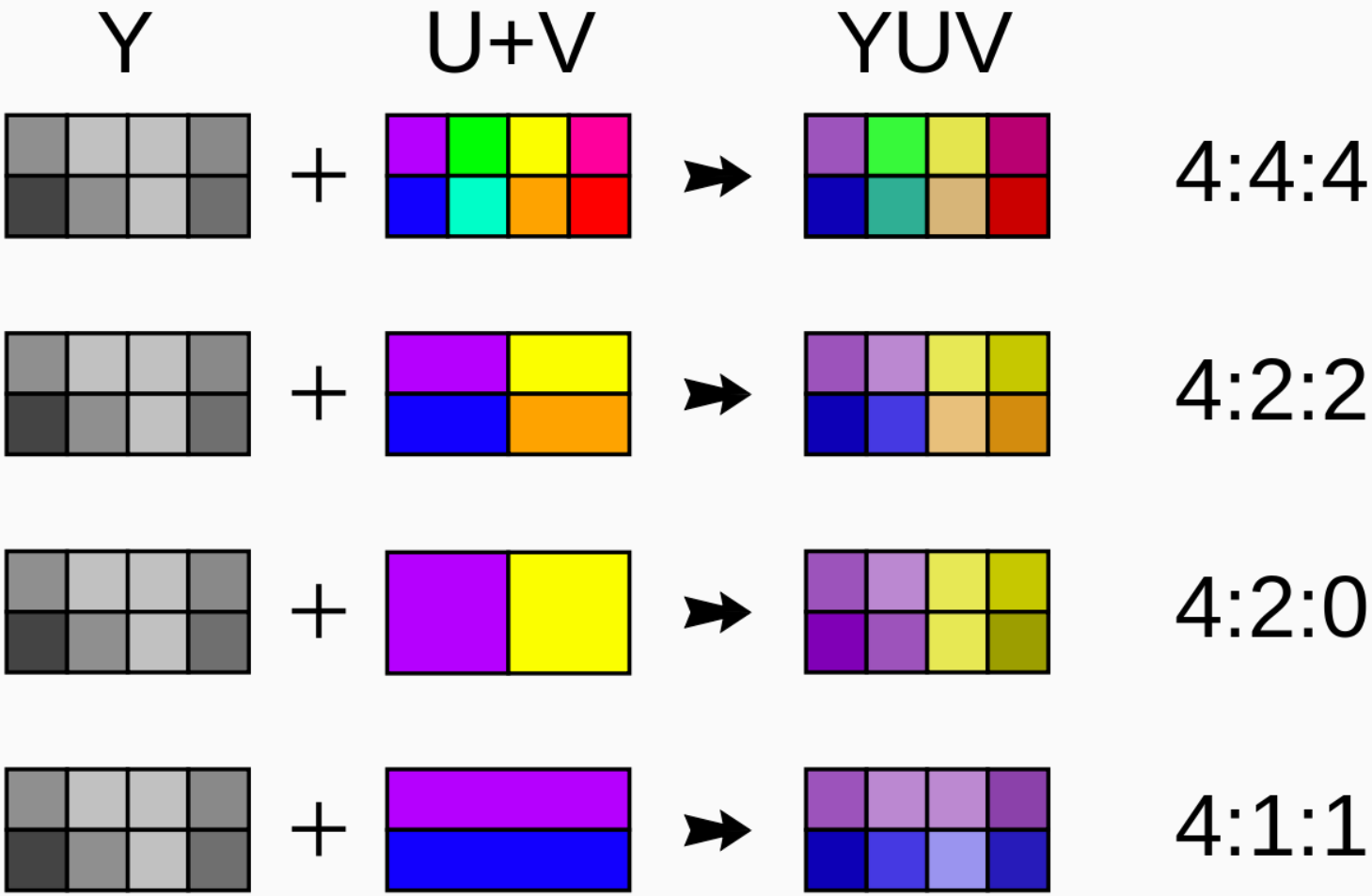
Rq. : En général $K_r = 0.299,\ K_g = 0.587,\ K_b = 0.114$

La Représentation YCbCr (2)

Œil humain $\rightarrow$ Y **prédomine**, Cb et Cr moins importants

Sous-Échantillonage

- **Principe** : Cb, Cr moins importants -> **moyenner** ces valeurs sur **plusieurs pixels**
- Exemples de **sous-échantillonnages** :



DCT (transformée en cosinus discrète) (1)

Transformation DCT

$$\psi : \mathbb{R}^N \to \mathbb{R}^N$$

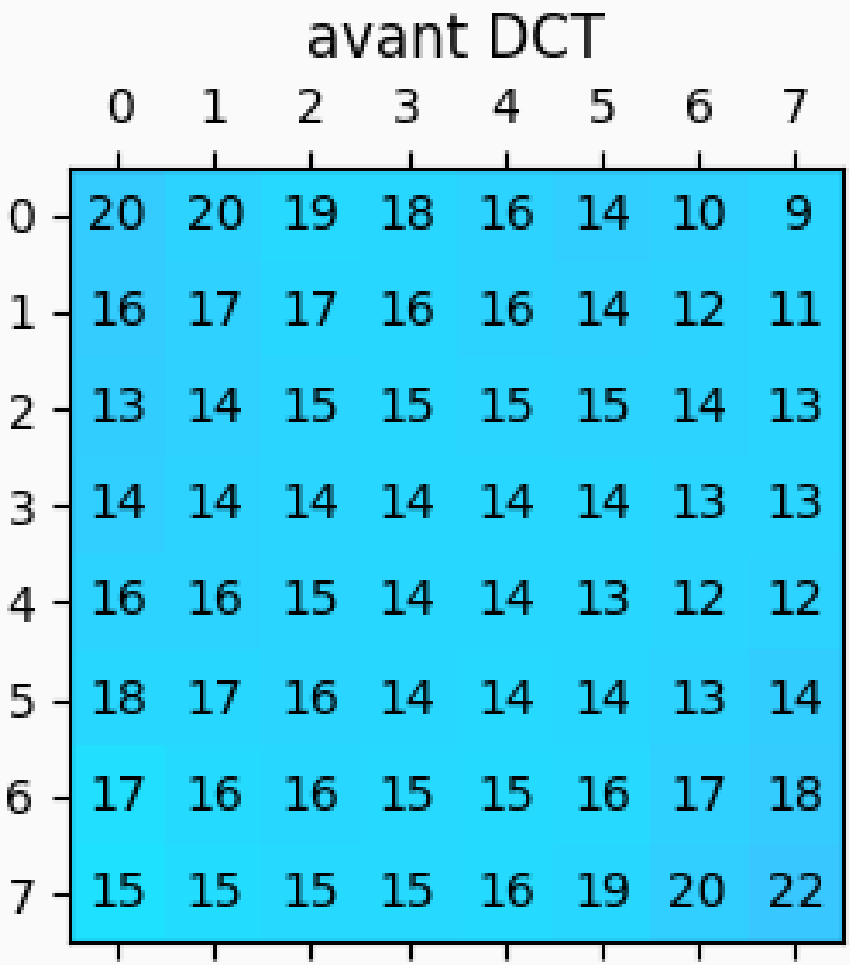
$$(x_0, \ldots, x_{N-1}) \longmapsto \left(\sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}(n + \frac{1}{2})k \right] \right)_{k \in [\![0, N-1]\!]}$$

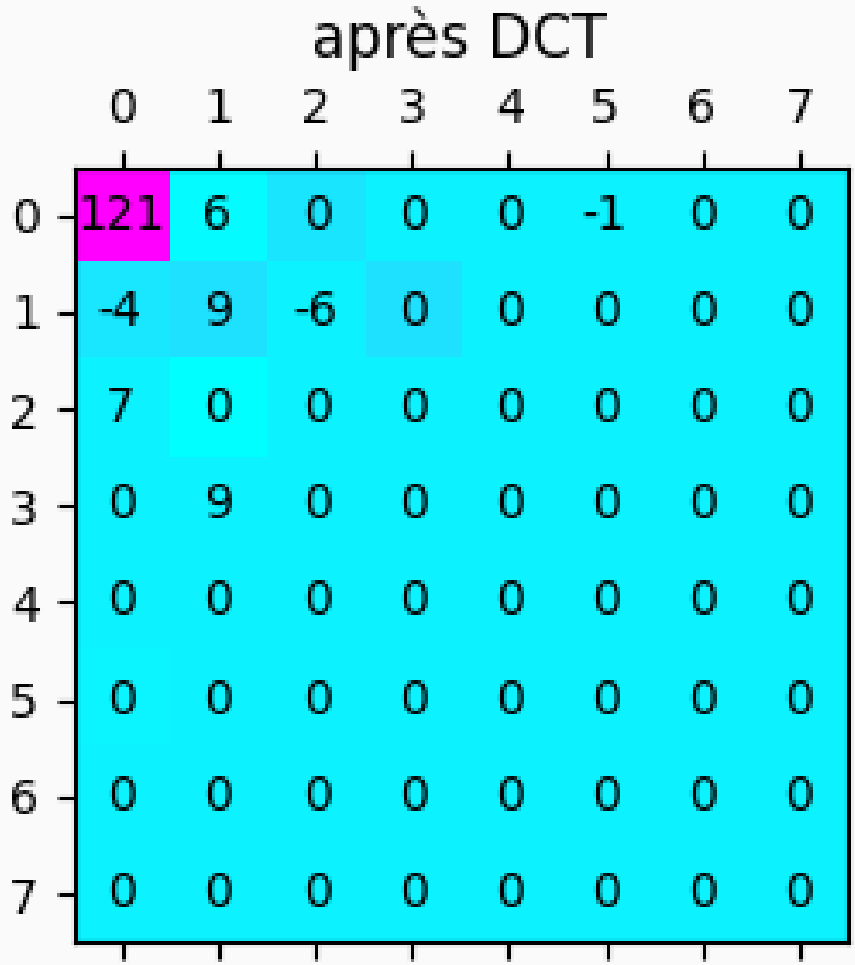
On peut rendre la matrice associée à ψ **orthogonale** en multipliant le terme X_0 par $\frac{1}{\sqrt{N}}$ et toute la matrice par $\sqrt{2/N}$.

2D-DCT $\to \psi$ sur chaque lignes puis chaque colonne

DCT (2)

- **"Continuité"** des images → peu de variations des **hautes fréquences**
- **Compactage** de l'énergie vers les **basses fréquences**





Quantification

- Seule étape **avec perte** de la compression **JPEG**
- **Réduction** des coefficients
- **Différence** entre Y, Cb et Cr
- Fonction de quantification :

$$\begin{aligned} \varepsilon \colon \mathcal{M}_{8,8}(\mathbb{Z})^2 &\to \mathcal{M}_{8,8}(\mathbb{Z}) \\ Q, B &\longmapsto \lfloor B/Q \rfloor \end{aligned}$$

- Souvent Q dépend d'un **facteur de qualité** q

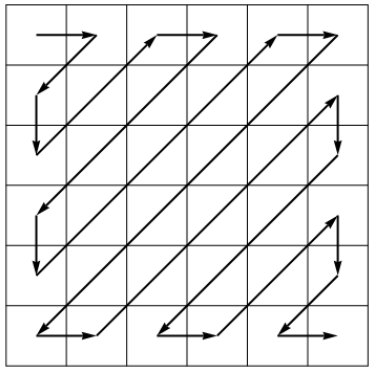
Codage par plages (Run-Length Encoding)

- Tire son avantage des **répétitions** de symboles
- Exemple :

$$\underbrace{AAAA}_{4\times A}\underbrace{B}_{1\times B}\underbrace{CCC}_{3\times C}\underbrace{BBBBBBB}_{7\times B} \xrightarrow[RLE]{} A4B1C3B7$$

  15 caractères $\rightarrow$ 8 caractères

- Pour faire apparaître ces répétitions dans les **images**, lecture de la matrice en **zigzag** :



Exemple de Compression

Image Original

Taille : 10,3 Mo

Exemple de Compression

Image Compressée ($q = 100$, $4:4:4$) (presque **lossless**)

Taille : 2,2 Mo

Ratio : $\eta = \frac{T_{init}}{T_{compres}} = 4.68$

Exemple de Compression

Image Compressée ($q = 50$, $4:2:2$)

Taille : 519 Ko

Ratio : $\eta = \frac{T_{init}}{T_{compres}} = 19.8$

Exemple de Compression

Image Compressée ($q = 5$, $4:1:1$)

Taille : 77 Ko

Ratio : $\eta = \frac{T_{init}}{T_{compres}} = 134$

Comparaison (1)

Comparaison (2)

utils.py

```python
import numpy as np
import math
import cv2
from io import BytesIO


# DCT block size
BH, BW = 8, 8


class MARKER:
    SOI = b'\xff\xd8'
    APP0 = b'\xff\xe0'
    APPn = (b'\xff\xe1', b'\xff\xef')  # n=1~15
    DQT = b'\xff\xdb'
    SOF0 = b'\xff\xc0'
    DHT = b'\xff\xc4'
    DRI = b'\xff\xdd'
    SOS = b'\xff\xda'
    EOI = b'\xff\xd9'


class ComponentInfo:
    def __init__(self, id_, horizontal, vertical, qt_id, dc_ht_id, ac_ht_id):
        self.id_ = id_
        self.horizontal = horizontal
        self.vertical = vertical
        self.qt_id = qt_id
        self.dc_ht_id = dc_ht_id
        self.ac_ht_id = ac_ht_id

    @classmethod
    def default(cls):
        return cls.__init__(*[0 for _ in range(6)])

    def encode_SOS_info(self):
```

```
37          return int2bytes(self.id_, 1) + \
38                 int2bytes((self.dc_ht_id << 4) +
self.ac_ht_id, 1)
39
40      def encode_SOF0_info(self):
41          return int2bytes(self.id_, 1) + \
42                 int2bytes((self.horizontal << 4) +
self.vertical, 1) + \
43                 int2bytes(self.qt_id, 1)
44
45      def __repr__(self):
46          return f'{self.id_}: qt-{self.qt_id}, ht-(dc-
{self.dc_ht_id}, ' \
47                 f'ac-{self.ac_ht_id}), sample-
{self.vertical, self.horizontal} '
48
49
50  class BitStreamReader:
51      """simulate bitwise read"""
52      def __init__(self, bytes_: bytes):
53          self.bits =
np.unpackbits(np.frombuffer(bytes_, dtype=np.uint8))
54          self.index = 0
55
56      def read_bit(self):
57          if self.index >= self.bits.size:
58              raise EOFError('Ran out of element')
59          self.index += 1
60          return self.bits[self.index - 1]
61
62      def read_int(self, length):
63          result = 0
64          for _ in range(length):
65              result = result * 2 + self.read_bit()
66          return result
67
68      def __repr__(self):
69          return f'[{self.index}, {self.bits.size}]'
70
71
72  class BitStreamWriter:
73      """simulate bitwise write"""
```

```
74      def __init__(self, length=10000):
75          self.index = 0
76          self.bits = np.zeros(length, dtype=np.uint8)
77
78      def write_bitstring(self, bitstring):
79          length = len(bitstring)
80          if length + self.index > self.bits.size * 8:
81              arr = np.zeros((length + self.index) // 8
    * 2, dtype=np.uint8)
82              arr[:self.bits.size] = self.bits
83              self.bits = arr
84          for bit in bitstring:
85              self.bits[self.index // 8] |= int(bit) <<
    (7 - self.index % 8)
86              self.index += 1
87
88      def to_bytes(self):
89          return self.bits[:math.ceil(self.index /
    8)].tobytes()
90
91      def to_hex(self):
92          length = math.ceil(self.index / 8) * 8
93          for i in range(self.index, length):
94              self.bits[i] = 1
95          bytes_ = np.packbits(self.bits[:length])
96          return ' '.join(f'{b:2x}' for b in bytes_)
97
98
99  class BytesWriter(BytesIO):
100
101     def __init__(self, *args, **kwargs):
102         super(BytesWriter, self).__init__(*args,
    **kwargs)
103
104     def add_bytes(self, *args):
105         self.write(b''.join(args))
106
107
108 def bytes2int(bytes_, byteorder='big'):
109     return int.from_bytes(bytes_, byteorder)
110
111
```

```
112 def int2bytes(int_: int, length):
113     return int_.to_bytes(length, byteorder='big')
114 
115 
116 def decode_2s_complement(complement, length) -> int:
117     if length == 0:
118         return 0
119     if complement >> (length - 1) == 1:  # sign bit
    equal to one
120         number = complement
121     else:  # sign bit equal to zero
122         number = 1 - 2**length + complement
123     return number
124 
125 
126 def encode_2s_complement(number) -> str:
127     """return the 2's complement representation as
    string"""
128     if number == 0:
129         return ''
130     if number > 0:
131         complement = bin(number)[2:]
132     else:
133         length = int(np.log2(-number)) + 1
134         complement = bin(number - (1 - 2**length))
    [2:].zfill(length)
135     return complement
136 
137 
138 def load_quantization_table(quality, component):
139     # the below two tables was processed by zigzag
    encoding
140     # in JPEG bit stream, the table is also stored in
    this order
141     if component == 'lum':
142         q = np.array([
143             16,  11,  12,  14,  12,  10,  16,  14,
144             13,  14,  18,  17,  16,  19,  24,  40,
145             26,  24,  22,  22,  24,  49,  35,  37,
146             29,  40,  58,  51,  61,  60,  57,  51,
147             56,  55,  64,  72,  92,  78,  64,  68,
148             87,  69,  55,  56,  80, 109,  81,  87,
```

```
            95,  98, 103, 104, 103,  62,  77, 113,
            121, 112, 100, 120,  92, 101, 103, 99],
dtype=np.int32)
    elif component == 'chr':
        q = np.array([
            17, 18, 18, 24, 21, 24, 47, 26,
            26, 47, 99, 66, 56, 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, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99],
dtype=np.int32)
    else:
        raise ValueError((
            f"component should be either 'lum' or
'chr', "
            f"but '{component}' was found."))
    if 0 < quality < 50:
        q = np.minimum(np.floor(50/quality * q +
0.5), 255)
    elif 50 <= quality <= 100:
        q = np.maximum(np.floor((2 - quality/50) * q
+ 0.5), 1)
    else:
        raise ValueError("quality should belong to
(0, 100].")
    return q.astype(np.int32)


def zigzag_points(rows, cols):
    # constants for directions
    UP, DOWN, RIGHT, LEFT, UP_RIGHT, DOWN_LEFT =
range(6)

    move_func = {
        UP: lambda p: (p[0] - 1, p[1]),
        DOWN: lambda p: (p[0] + 1, p[1]),
        LEFT: lambda p: (p[0], p[1] - 1),
        RIGHT: lambda p: (p[0], p[1] + 1),
        UP_RIGHT: lambda p: move(UP, move(RIGHT, p)),
```

```
184             DOWN_LEFT: lambda p: move(DOWN, move(LEFT,
p))
185         }
186
187         # move the point in different directions
188         def move(direction, point):
189             return move_func[direction](point)
190
191         # return true if point is inside the block bounds
192         def inbounds(p):
193             return 0 <= p[0] < rows and 0 <= p[1] < cols
194
195         # start in the top-left cell
196         now = (0, 0)
197
198         # True when moving up-right, False when moving
down-left
199         move_up = True
200         trace = []
201
202         for i in range(rows * cols):
203             trace.append(now)
204             if move_up:
205                 if inbounds(move(UP_RIGHT, now)):
206                     now = move(UP_RIGHT, now)
207                 else:
208                     move_up = False
209                     if inbounds(move(RIGHT, now)):
210                         now = move(RIGHT, now)
211                     else:
212                         now = move(DOWN, now)
213             else:
214                 if inbounds(move(DOWN_LEFT, now)):
215                     now = move(DOWN_LEFT, now)
216                 else:
217                     move_up = True
218                     if inbounds(move(DOWN, now)):
219                         now = move(DOWN, now)
220                     else:
221                         now = move(RIGHT, now)
222         """
```

```
    for rows = cols = 8, the actual 1-D index:
        0, 1, 8, 16, 9, 2, 3, 10, 17, 24, 32, 25, 18,
11, 4, 5,
        12, 19, 26, 33, 40, 48, 41, 34, 27, 20, 13,
6, 7, 14, 21, 28,
        35, 42, 49, 56, 57, 50, 43, 36, 29, 22, 15,
23, 30, 37, 44, 51,
        58, 59, 52, 45, 38, 31, 39, 46, 53, 60, 61,
54, 47, 55, 62, 63
    """
    return trace


def RGB2YCbCr(im):
    im = im.astype(np.float32)
    im = cv2.cvtColor(im, cv2.COLOR_RGB2YCrCb)
    """
    RGB [0, 255]
    opencv uses the following equations to conduct
color conversion in float32
        Y = 0.299 * R + 0.587 * G + 0.114 * B
        Cb = (B - Y) * 0.564 + 0.5
        Cr = (R - Y) * 0.713 + 0.5
    Y [0, 255], Cb, Cr [-128, 127]
    """
    # convert YCrCb to YCbCr
    Y, Cr, Cb = np.split(im, 3, axis=-1)
    im = np.concatenate([Y, Cb, Cr], axis=-1)
    return im


def YCbCr2RGB(im):
    im = im.astype(np.float32)
    Y, Cb, Cr = np.split(im, 3, axis=-1)
    im = np.concatenate([Y, Cr, Cb], axis=-1)
    im = cv2.cvtColor(im, cv2.COLOR_YCrCb2RGB)
    """
    Y [0, 255], Cb, Cr [-128, 127]
    conversion equation (float32):
        B = (Cb - 0.5) / 0.564 + Y
        R = (Cr - 0.5) / 0.713 + Y
        G = (Y - 0.299 * R - 0.114 * B) / 0.587
```

```
    RGB [0, 255]
    """
    return im


def bits_required(n):
    n = abs(n)
    result = 0
    while n > 0:
        n >>= 1
        result += 1
    return result


def divide_blocks(im, mh, mw):
    h, w = im.shape[:2]
    return im.reshape(h//mh, mh, w//mw, mw).swapaxes(1, 2).reshape(-1, mh, mw)


def restore_image(block, nh, nw):
    bh, bw = block.shape[1:]
    return block.reshape(nh, nw, bh, bw).swapaxes(1, 2).reshape(nh*bh, nw*bw)


def flatten(lst):
    return [item for sublist in lst for item in sublist]

def averageMatrix(arrayMatrix): # given an array of 2D-array, return the average (coef by coef) 2D array
    avgMatrix = np.zeros_like(arrayMatrix[0])
    for i in range(avgMatrix.shape[0]):
        for j in range(avgMatrix.shape[1]):
            avgMatrix[i, j] = np.average(arrayMatrix[:, i, j])
    return avgMatrix

def main():
    pass

```

```
297
298 if __name__ == '__main__':
299     main()
300     arrMatrix = np.array([[[1, 2],
301                            [3, 4]],
302                           [[5, 2],
303                            [3, 4]]])
304     print(averageMatrix(arrMatrix))
```

huffman.py

```python
import numpy as np

MAX_CLEN = 32 # assumed maximum initial code length

def getFreq(data):
    freq = [0] * 257
    for elem in data:
        freq[elem] += 1
    freq[256] = 1
    return freq

def jpegGenerateOptimalTable(freq):
    bits = [0] * (MAX_CLEN + 1)
    bitPos = [0] * (MAX_CLEN + 1)
    codesize = [0] * 257
    nzIndex = [0] * 257

    others = [-1] * 257

    numNzSymbols = 0
    for i in range(257):
        if freq[i]:
            nzIndex[numNzSymbols] = i
            freq[numNzSymbols] = freq[i]
            numNzSymbols += 1

    huffval = [0] * (numNzSymbols - 1)

    while True:
        c1 = -1
        c2 = -1
        v = 1000000000
        v2 = 1000000000
        for i in range(numNzSymbols):
            if freq[i] <= v2:
                if freq[i] <= v:
                    c2 = c1
```

```
                    v2 = v
                    v = freq[i]
                    c1 = i
                else:
                    v2 = freq[i]
                    c2 = i

        if (c2 < 0):
            break

        freq[c1] += freq[c2]
        freq[c2] = 1000000001

        codesize[c1] += 1
        while others[c1] >= 0:
            c1 = others[c1]
            codesize[c1] += 1

        others[c1] = c2

        codesize[c2] += 1
        while others[c2] >= 0:
            c2 = others[c2]
            codesize[c2] += 1

    for i in range(numNzSymbols):
        bits[codesize[i]] += 1

    p = 0
    for i in range(1, MAX_CLEN + 1):
        bitPos[i] = p
        p += bits[i]

    for i in range(MAX_CLEN, 16, -1):
        while bits[i] > 0:
            j = i - 2
            while bits[j] == 0:
                j -= 1
            bits[i] -= 2
            bits[i - 1] += 1
            bits[j + 1] += 2
```

```
            bits[j] -= 1

    i = MAX_CLEN
    while bits[i] == 0:
        i -= 1
    bits[i] -= 1

    for i in range(numNzSymbols - 1):
        huffval[bitPos[codesize[i]]] = nzIndex[i]
        bitPos[codesize[i]] += 1

    return bits, huffval

def jpegGenerateHuffmanTable(bits, huffval):
    huffsize = [0] * 257
    huffcode = [0] * 257

    p = 0
    for l in range(1, 17):
        i = bits[l]
        while i:
            huffsize[p] = l
            p += 1
            i -= 1

    huffsize[p] = 0
    lastp = p

    code = 0
    si = huffsize[0]
    p = 0
    while huffsize[p]:
        while huffsize[p] == si:
            huffcode[p] = code
            code += 1
            p += 1
        code <<= 1
        si += 1

    ehufco = [0] * 257
    ehufsi = [0] * 257
```

```
120
121     for p in range(lastp):
122         i = huffval[p]
123         ehufco[i] = huffcode[p]
124         ehufsi[i] = huffsize[p]
125
126     return ehufsi, ehufco
127
128 def jpegTransformTable(ehufsi, ehufco):
129     table = {}
130     for i in range(len(ehufco)):
131         if ehufsi[i] != 0:
132             endCode = bin(ehufco[i])[2:]
133             nbZeros = ehufsi[i] - len(endCode)
134             table[i] = '0' * nbZeros + endCode
135     return table
136
137 def jpegCreateHuffmanTable(arr):
138     freq = getFreq(arr)
139     bits, huffval = jpegGenerateOptimalTable(freq)
140     ehufsi, ehufco = jpegGenerateHuffmanTable(bits,
huffval)
141     table = jpegTransformTable(ehufsi, ehufco)
142     return table
143
144 def convert_huffman_table(table):
145     """convert huffman table to count and weigh"""
146     # table[int] = string
147     pairs = sorted(table.items(), key=lambda x:
(len(x[1]), x[1]))
148     weigh, codes = zip(*pairs)
149     weigh = np.array(weigh, dtype=np.uint8)
150     # count[i]: there are count[i] codes of length
i+1
151     count = np.zeros(16, dtype=np.uint8)
152     for c in codes:
153         count[len(c)-1] += 1
154     return count, weigh
155
156
157 def read_huffman_code(table, stream):
158     prefix = ''
```

```
159     while prefix not in table:
160         prefix += str(stream.read_bit())
161     return table[prefix]
162 
163 
164 def reverse(table):
165     return {v: k for k, v in table.items()}
166 
167 # 4 recommended huffman tables in JPEG standard
168 # luminance DC
169 RM_Y_DC = {'00': 0, '010': 1, '011': 2, '100': 3,
    '101': 4, '110': 5,
170             '1110': 6, '11110': 7, '111110': 8,
    '1111110': 9, '11111110': 10,
171             '111111110': 11}
172 
173 # luminance AC
174 RM_Y_AC = {'00': 1, '01': 2, '100': 3, '1010': 0,
    '1011': 4, '1100': 17,
175             '11010': 5, '11011': 18, '11100': 33,
    '111010': 49, '111011': 65,
176             '1111000': 6, '1111001': 19, '1111010':
    81, '1111011': 97,
177             '11111000': 7, '11111001': 34, '11111010':
    113, '111110110': 20,
178             '111110111': 50, '111111000': 129,
    '111111001': 145,
179             '111111010': 161, '1111110110': 8,
    '1111110111': 35,
180             '1111111000': 66, '1111111001': 177,
    '1111111010': 193,
181             '11111110110': 21, '11111110111': 82,
    '11111111000': 209,
182             '11111111001': 240, '111111110100': 36,
    '111111110101': 51,
183             '111111110110': 98, '111111110111': 114,
    '111111111000000': 130,
184             '1111111110000010': 9, '1111111110000011':
    10,
185             '1111111110000100': 22,
    '1111111110000101': 23,
186             '1111111110000110': 24,
    '1111111110000111': 25,
187             '1111111110001000': 26,
    '1111111110001001': 37,
```

```
188             '1111111110001010': 38,
    '1111111110001011': 39,
189             '1111111110001100': 40,
    '1111111110001101': 41,
190             '1111111110001110': 42,
    '1111111110001111': 52,
191             '1111111110010000': 53,
    '1111111110010001': 54,
192             '1111111110010010': 55,
    '1111111110010011': 56,
193             '1111111110010100': 57,
    '1111111110010101': 58,
194             '1111111110010110': 67,
    '1111111110010111': 68,
195             '1111111110011000': 69,
    '1111111110011001': 70,
196             '1111111110011010': 71,
    '1111111110011011': 72,
197             '1111111110011100': 73,
    '1111111110011101': 74,
198             '1111111110011110': 83,
    '1111111110011111': 84,
199             '1111111110100000': 85,
    '1111111110100001': 86,
200             '1111111110100010': 87,
    '1111111110100011': 88,
201             '1111111110100100': 89,
    '1111111110100101': 90,
202             '1111111110100110': 99,
    '1111111110100111': 100,
203             '1111111110101000': 101,
    '1111111110101001': 102,
204             '1111111110101010': 103,
    '1111111110101011': 104,
205             '1111111110101100': 105,
    '1111111110101101': 106,
206             '1111111110101110': 115,
    '1111111110101111': 116,
207             '1111111110110000': 117,
    '1111111110110001': 118,
208             '1111111110110010': 119,
    '1111111110110011': 120,
209             '1111111110110100': 121,
    '1111111110110101': 122,
210             '1111111110110110': 131,
    '1111111110110111': 132,
```

```
211             '1111111110111000': 133,
'1111111110111001': 134,
212             '1111111110111010': 135,
'1111111110111011': 136,
213             '1111111110111100': 137,
'1111111110111101': 138,
214             '1111111110111110': 146,
'1111111110111111': 147,
215             '1111111111000000': 148,
'1111111111000001': 149,
216             '1111111111000010': 150,
'1111111111000011': 151,
217             '1111111111000100': 152,
'1111111111000101': 153,
218             '1111111111000110': 154,
'1111111111000111': 162,
219             '1111111111001000': 163,
'1111111111001001': 164,
220             '1111111111001010': 165,
'1111111111001011': 166,
221             '1111111111001100': 167,
'1111111111001101': 168,
222             '1111111111001110': 169,
'1111111111001111': 170,
223             '1111111111010000': 178,
'1111111111010001': 179,
224             '1111111111010010': 180,
'1111111111010011': 181,
225             '1111111111010100': 182,
'1111111111010101': 183,
226             '1111111111010110': 184,
'1111111111010111': 185,
227             '1111111111011000': 186,
'1111111111011001': 194,
228             '1111111111011010': 195,
'1111111111011011': 196,
229             '1111111111011100': 197,
'1111111111011101': 198,
230             '1111111111011110': 199,
'1111111111011111': 200,
231             '1111111111100000': 201,
'1111111111100001': 202,
232             '1111111111100010': 210,
'1111111111100011': 211,
233             '1111111111100100': 212,
'1111111111100101': 213,
```

```
234             '1111111111100110': 214,
    '1111111111100111': 215,
235             '1111111111101000': 216,
    '1111111111101001': 217,
236             '1111111111101010': 218,
    '1111111111101011': 225,
237             '1111111111101100': 226,
    '1111111111101101': 227,
238             '1111111111101110': 228,
    '1111111111101111': 229,
239             '1111111111110000': 230,
    '1111111111110001': 231,
240             '1111111111110010': 232,
    '1111111111110011': 233,
241             '1111111111110100': 234,
    '1111111111110101': 241,
242             '1111111111110110': 242,
    '1111111111110111': 243,
243             '1111111111111000': 244,
    '1111111111111001': 245,
244             '1111111111111010': 246,
    '1111111111111011': 247,
245             '1111111111111100': 248,
    '1111111111111101': 249,
246             '1111111111111110': 250}
247
248 # chroma DC
249 RM_C_DC = {'00': 0, '01': 1, '10': 2, '110': 3,
    '1110': 4, '11110': 5,
250             '111110': 6, '1111110': 7, '11111110': 8,
    '111111110': 9,
251             '1111111110': 10, '11111111110': 11}
252
253 # chroma AC
254 RM_C_AC = {'00': 0, '01': 1, '100': 2, '1010': 3,
    '1011': 17, '11000': 4,
255             '11001': 5, '11010': 33, '11011': 49,
    '111000': 6, '111001': 18,
256             '111010': 65, '111011': 81, '1111000': 7,
    '1111001': 97,
257             '1111010': 113, '11110110': 19,
    '11110111': 34, '11111000': 50,
258             '11111001': 129, '111110100': 8,
    '111110101': 20, '111110110': 66,
259             '111110111': 145, '111111000': 161,
    '111111001': 177,
```

```
260             '111111010': 193, '1111110110': 9,
'1111110111': 35,
261             '1111111000': 51, '1111111001': 82,
'1111111010': 240,
262             '11111110110': 21, '11111110111': 98,
'11111111000': 114,
263             '11111111001': 209, '111111110100': 10,
'111111110101': 22,
264             '111111110110': 36, '111111110111': 52,
'11111111100000': 225,
265             '111111111000010': 37, '111111111000011':
241,
266             '1111111110001000': 23,
'1111111110001001': 24,
267             '1111111110001010': 25,
'1111111110001011': 26,
268             '1111111110001100': 38,
'1111111110001101': 39,
269             '1111111110001110': 40,
'1111111110001111': 41,
270             '1111111110010000': 42,
'1111111110010001': 53,
271             '1111111110010010': 54,
'1111111110010011': 55,
272             '1111111110010100': 56,
'1111111110010101': 57,
273             '1111111110010110': 58,
'1111111110010111': 67,
274             '1111111110011000': 68,
'1111111110011001': 69,
275             '1111111110011010': 70,
'1111111110011011': 71,
276             '1111111110011100': 72,
'1111111110011101': 73,
277             '1111111110011110': 74,
'1111111110011111': 83,
278             '1111111110100000': 84,
'1111111110100001': 85,
279             '1111111110100010': 86,
'1111111110100011': 87,
280             '1111111110100100': 88,
'1111111110100101': 89,
281             '1111111110100110': 90,
'1111111110100111': 99,
282             '1111111110101000': 100,
'1111111110101001': 101,
```

```
283             '1111111110101010': 102,
'1111111110101011': 103,
284             '1111111110101100': 104,
'1111111110101101': 105,
285             '1111111110101110': 106,
'1111111110101111': 115,
286             '1111111110110000': 116,
'1111111110110001': 117,
287             '1111111110110010': 118,
'1111111110110011': 119,
288             '1111111110110100': 120,
'1111111110110101': 121,
289             '1111111110110110': 122,
'1111111110110111': 130,
290             '1111111110111000': 131,
'1111111110111001': 132,
291             '1111111110111010': 133,
'1111111110111011': 134,
292             '1111111110111100': 135,
'1111111110111101': 136,
293             '1111111110111110': 137,
'1111111110111111': 138,
294             '1111111111000000': 146,
'1111111111000001': 147,
295             '1111111111000010': 148,
'1111111111000011': 149,
296             '1111111111000100': 150,
'1111111111000101': 151,
297             '1111111111000110': 152,
'1111111111000111': 153,
298             '1111111111001000': 154,
'1111111111001001': 162,
299             '1111111111001010': 163,
'1111111111001011': 164,
300             '1111111111001100': 165,
'1111111111001101': 166,
301             '1111111111001110': 167,
'1111111111001111': 168,
302             '1111111111010000': 169,
'1111111111010001': 170,
303             '1111111111010010': 178,
'1111111111010011': 179,
304             '1111111111010100': 180,
'1111111111010101': 181,
305             '1111111111010110': 182,
'1111111111010111': 183,
```

```
306             '1111111111011000': 184,
    '1111111111011001': 185,
307             '1111111111011010': 186,
    '1111111111011011': 194,
308             '1111111111011100': 195,
    '1111111111011101': 196,
309             '1111111111011110': 197,
    '1111111111011111': 198,
310             '1111111111100000': 199,
    '1111111111100001': 200,
311             '1111111111100010': 201,
    '1111111111100011': 202,
312             '1111111111100100': 210,
    '1111111111100101': 211,
313             '1111111111100110': 212,
    '1111111111100111': 213,
314             '1111111111101000': 214,
    '1111111111101001': 215,
315             '1111111111101010': 216,
    '1111111111101011': 217,
316             '1111111111101100': 218,
    '1111111111101101': 226,
317             '1111111111101110': 227,
    '1111111111101111': 228,
318             '1111111111110000': 229,
    '1111111111110001': 230,
319             '1111111111110010': 231,
    '1111111111110011': 232,
320             '1111111111110100': 233,
    '1111111111110101': 234,
321             '1111111111110110': 242,
    '1111111111110111': 243,
322             '1111111111111000': 244,
    '1111111111111001': 245,
323             '1111111111111010': 246,
    '1111111111111011': 247,
324             '1111111111111100': 248,
    '1111111111111101': 249,
325             '1111111111111110': 250}
326
327 if __name__ == "__main__":
328     arr = np.array([np.random.randint(-127, 128) for
    _ in range(64)])
329     table = jpegCreateHuffmanTable(arr)
330     print(table)
```

encoder.py

```python
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),
}


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)


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

```
76      return dc_pred
77
78
79  def run_length_encode(arr):
80      # determine where the sequence is ending
    prematurely
81      last_nonzero = -1
82      for i, elem in enumerate(arr):
83          if elem != 0:
84              last_nonzero = i
85      rss, values = [], []
86      run_length = 0
87      for i, elem in enumerate(arr):
88          if i > last_nonzero:
89              rss.append(0)
90              values.append(0)
91              break
92          elif elem == 0 and run_length < 15:
93              run_length += 1
94          else:
95              size = bits_required(elem)
96              rss.append((run_length << 4) + size)
97              values.append(elem)
98              run_length = 0
99      return rss, values
100
101
102 def encode_header(qts, hts, cop_infos, height,
    width):
103     writer = BytesWriter()
104     add_bytes = writer.add_bytes
105     add_bytes(
106         MARKER.SOI,
107         MARKER.APP0,
108         b"\x00\x10",  # length = 16
109         b"JFIF\x00",  # identifier = JFIF0
110         b"\x01\x01",  # version
111         b"\x00",  # unit
112         b"\x00\x01",  # x density
113         b"\x00\x01",  # y density
114         b"\x00\x00",  # thumbnail data
```

```
115     )
116     for id_, qt in enumerate(qts):
117         add_bytes(
118             MARKER.DQT,
119             b"\x00C",  # length = 67
120             # precision (8 bits), table id, = 0, id_
121             int2bytes(id_, 1),
122             qt.astype(np.uint8).tobytes(),
123         )
124     cop_num = len(cop_infos)
125     add_bytes(
126         MARKER.SOF0,
127         int2bytes(8 + 3 * cop_num, 2),  # length
128         int2bytes(8, 1),  # 8 bit precision
129         int2bytes(height, 2),
130         int2bytes(width, 2),
131         int2bytes(cop_num, 1),
132     )
133     add_bytes(*[info.encode_SOF0_info() for info in
cop_infos])
134
135     # type << 4 + id, (type 0: DC, 1 : AC)
136     type_ids = [b"\x00", b"\x10", b"\x01", b"\x11"]
137     for type_id, ht in zip(type_ids, hts):
138         count, weigh = convert_huffman_table(ht)
139         ht_bytes = count.tobytes() + weigh.tobytes()
140         add_bytes(
141             MARKER.DHT,
142             int2bytes(len(ht_bytes) + 3, 2),  #
length
143             type_id,
144             ht_bytes,
145         )
146
147     add_bytes(
148         MARKER.SOS,
149         int2bytes(6 + cop_num * 2, 2),  # length
150         int2bytes(cop_num, 1),
151     )
152     add_bytes(*[info.encode_SOS_info() for info in
cop_infos])
153     add_bytes(b"\x00\x3f\x00")
```

```
154     return writer
155
156
157 def encode_mcu(mcu, hts):
158     bit_stream = BitStreamWriter()
159     for cur in mcu:
160         for dct, (dc_ht, ac_ht) in zip(cur, hts):
161             dc_code = encode_2s_complement(dct[0])
162             container = [dc_ht[len(dc_code)],
dc_code]
163             rss, values = run_length_encode(dct[1:])
164             for rs, v in zip(rss, values):
165                 container.append(ac_ht[rs])
166
container.append(encode_2s_complement(v))
167             bitstring = "".join(container)
168             bit_stream.write_bitstring(bitstring)
169     return bit_stream.to_bytes()
170
171
172 def encode_jpeg(im, quality=95, subsample="4:2:0",
use_rm_ht=True):
173     im = np.expand_dims(im, axis=-1) if im.ndim == 2
else im
174     height, width, depth = im.shape
175
176     mh, mw = mcu_sizes[subsample] if depth == 3 else
(BH, BW)
177     im = padding(im, mh, mw)
178     im = RGB2YCbCr(im) if depth == 3 else im
179
180     # DC level shift for luminance,
181     # the shift of chroma was completed by color
conversion
182     Y_im = im[:, :, 0] - 128
183     # divide image into MCUs
184     mcu = divide_blocks(Y_im, mh, mw)
185     # MCU to blocks, for luminance there are more
than one blocks in each MCU
186     Y = scan_blocks(mcu, mh, mw)
187     Y_dct = DCT(Y)
188     # 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)

    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)
```

```
222         else:
223             ch_ = np.concatenate([Cb_p, Cr_p],
    axis=0)
224             C_dc_ht =
    jpegCreateHuffmanTable(np.vectorize(bits_required)
    (ch_[:, 0]))
225             C_ac_ht = jpegCreateHuffmanTable(
226                 flatten(run_length_encode(ch_[i, 1:])
    [0] for i in range(ch_.shape[0]))
227             )
228         qts.append(qt_c), hts.extend([C_dc_ht,
    C_ac_ht])
229         cop_infos.extend(
230             [ComponentInfo(2, 1, 1, 1, 1, 1),
    ComponentInfo(3, 1, 1, 1, 1, 1)]
231         )
232         mcu_hts.extend((C_dc_ht, C_ac_ht) for _ in
    range(2))
233         mcu_ = np.concatenate([mcu_, Cb_p[:, None],
    Cr_p[:, None]], axis=1)
234
235     writer = encode_header(qts, hts, cop_infos,
    height, width)
236     bytes_ = encode_mcu(mcu_, mcu_hts)
237     writer.write(bytes_.replace(b"\xff",
    b"\xff\x00"))
238     writer.write(MARKER.EOI)
239     return writer.getvalue()
240
241
242 def write_jpeg(filename, im, quality=95,
    subsample="4:2:0", use_rm_ht=True):
243     bytes_ = encode_jpeg(im, quality, subsample,
    use_rm_ht)
244     Path(filename).write_bytes(bytes_)
245
246
247 def main():
248     im = Image.open("./data/villeLyon.jpg")
249     write_jpeg("data/villeLyonLow.jpg", np.array(im),
    5, "4:1:1", False)
250
251
252 if __name__ == "__main__":
```

```
253     main()
```

comparator.py

```python
import numpy as np
import encoder
import sys, os
from pathlib import Path
from PIL import Image
import cv2
import pandas as pd
import time as t
import shutil
import utils
import matplotlib.pyplot as plt
from encoder import DCT, padding
from scipy.fftpack import dct

LIM = 25  # number of files to test to


def compare(
    quality=None,
    dataDirectory=None,
    outputDirectory=None,
    subsample=None,
    useStdHuffmanTable=None,
    DeleteFilesAfterward=True,
):
    if quality is None:
        quality = np.random.randint(0, 101)
    if subsample is None:
        subsample = "4:2:2"
    if dataDirectory is None:
        dataDirectory = "./data/datasetBmp"
    if useStdHuffmanTable is None:
        useStdHuffmanTable = False
    outputDirectory =
f"./data/treated/quality{quality}-
subsample{subsample}-stdHf{useStdHuffmanTable}"
    i = 0
    stat = np.zeros(
```

```
        (LIM, 3), dtype=object
    )  # first parameter is size before compression, second after, and third the time to achieve compression
    for filename in os.listdir(dataDirectory):
        if i >= LIM:
            break
        f = os.path.join(dataDirectory, filename)
        if not os.path.exists(outputDirectory):
            os.makedirs(outputDirectory)
        f_out = os.path.join(outputDirectory, filename + ".jpg")
        if os.path.isfile(f):
            previousSize = os.stat(f).st_size
            image = Image.open(f)
            time = t.time_ns()
            encoder.write_jpeg(
                f_out, np.array(image), quality, subsample, useStdHuffmanTable
            )
            time = t.time_ns() - time
            newSize = os.stat(f_out).st_size
            stat[i][0] = previousSize
            stat[i][1] = newSize
            stat[i][2] = time
        i += 1
    if DeleteFilesAfterward:
        shutil.rmtree(outputDirectory)
    write_stat("./data/treated/stat.txt", stat, quality, subsample, useStdHuffmanTable)


def write_stat(statFile, stat, quality, subsample, standHuffTables):
    with open(statFile, "a+") as f:
        f.write("\n" * 2)
        f.write("New sample \n")
        f.write(f"Size of sample : {LIM} images \n")
        f.write(
            f"Parameters of compression : (quality) {quality}, (subsample) {subsample}, (usage of standard HuffTables) {'Yes' if standHuffTables else 'No'} \n"
```

```
        )
        avgPreviousSize = np.average(stat[:, 0])
        avgNewSize = np.average(stat[:, 1])
        f.write(
            f"Average size of image before compression : {avgPreviousSize} bytes \n"
        )
        f.write(f"Average size of images after compression : {avgNewSize} bytes \n")
        f.write(f"Ratio is {avgPreviousSize / avgNewSize:.2f}")


def write_stat_csv(outputDirectory, stat, quality, subsample, standHuffTables):
    pass

def energyCompaction(imgPath):
    img = cv2.imread(imgPath)

    imgYCrCB = cv2.cvtColor(
        img, cv2.COLOR_RGB2YCrCb
    )  # Convert RGB to YCrCb (Cb applies V, and Cr applies U).

    Y, Cr, Cb = cv2.split(padding(imgYCrCB, 8, 8))
    Y = Y.astype('int') - 128
    blocks_Y = utils.divide_blocks(Y, 8, 8)
    dctBlocks_Y = np.zeros_like(blocks_Y)
    for i in range(len(blocks_Y)):
        dctBlocks_Y[i] = dct(dct(blocks_Y[i], axis=0, norm="ortho"), axis=1, norm="ortho")
    avg_Y = utils.averageMatrix(blocks_Y)
    avgDct_Y = utils.averageMatrix(dctBlocks_Y)

    x = np.random.randint(blocks_Y.shape[0])
    arr1 = blocks_Y[x]
    arr2 = dctBlocks_Y[x]

    fig, (ax1, ax2 )= plt.subplots(1, 2)

    valueMax, valueMin = max(np.max(arr1), np.max(arr2)), min(np.min(arr1), np.min(arr2))
```

```
107     # fig.suptitle('Matrice de la luminance de
"villeLyon.jpg"')
108
109     ax1.matshow(arr1, cmap="cool", vmin=valueMin,
vmax=valueMax)
110     ax1.set_title('avant DCT')
111
112     ax2.matshow(arr2, cmap="cool", vmin=valueMin,
vmax=valueMax)
113     ax2.set_title('après DCT')
114
115
116     for i in range(arr1.shape[0]):
117         for j in range(arr1.shape[1]):
118             cNormal= int(arr1[i, j])
119             cDct = int(arr2[i, j])
120             ax1.text(i, j, str(cNormal), va='center',
ha='center')
121             ax2.text(i, j, str(cDct), va='center',
ha='center')
122     plt.savefig('./data/energyCompaction.png',
transparent=True)
123
124
125 def rgbToYCbCr_channel_bis():
126     img = cv2.imread("./data/villeLyon.jpg")  # Read
input image in BGR format
127
128     imgYCrCB = cv2.cvtColor(
129         img, cv2.COLOR_BGR2YCrCb
130     )  # Convert RGB to YCrCb (Cb applies V, and Cr
applies U).
131
132     Y, Cr, Cb = cv2.split(imgYCrCB)
133
134     # Fill Y and Cb with 128 (Y level is middle gray,
and Cb is "neutralized").
135     onlyCr = imgYCrCB.copy()
136     onlyCr[:, :, 0] = 128
137     onlyCr[:, :, 2] = 128
138     onlyCr_as_bgr = cv2.cvtColor(
139         onlyCr, cv2.COLOR_YCrCb2BGR
```

```
    )  # Convert to BGR - used for display as false color

    # Fill Y and Cr with 128 (Y level is middle gray, and Cr is "neutralized").
    onlyCb = imgYCrCB.copy()
    onlyCb[:, :, 0] = 128
    onlyCb[:, :, 1] = 128
    onlyCb_as_bgr = cv2.cvtColor(
        onlyCb, cv2.COLOR_YCrCb2BGR
    )  # Convert to BGR - used for display as false color

    cv2.imshow("img", img)
    cv2.imshow("Y", Y)
    cv2.imshow("onlyCb_as_bgr", onlyCb_as_bgr)
    cv2.imshow("onlyCr_as_bgr", onlyCr_as_bgr)
    cv2.waitKey()
    cv2.destroyAllWindows()

    cv2.imwrite("./data/treated/villeLyon_Y.jpg", Y)
    cv2.imwrite("./data/treated/villeLyon_Cb.jpg", onlyCb_as_bgr)
    cv2.imwrite("./data/treated/villeLyon_Cr.jpg", onlyCr_as_bgr)


if __name__ == "__main__":
    # compare()
    # rgbToYCbCr_channel_bis()
    energyCompaction("./data/villeLyon.jpg")
    test = np.array([[93, 90, 83, 68, 61, 61, 46, 21],
                     [102, 92, 95, 77, 65, 60, 49, 32],
                     [69, 55, 47, 57, 65, 60, 72, 65],
                     [55, 55, 40, 42, 23, 1, 11, 38],
                     [55, 57, 47, 53, 35, 59, -2, 26],
                     [64, 41, 42, 55, 60, 57, 25, -8],
                     [77, 87, 58, -2, -5, 14, -10, -35],
                     [38, 14, 33, 33, -21, -23, -43, -34]])
```

```
174     print(dct(dct(test, axis=0, norm="ortho"),
axis=1, norm='ortho'))
```