Selected files

9 printable files

ycbcr.py
huffman.py
fft.py
complex.py
arithmetic_coding.py
encoder/utils.py
encoder/huffman_jpeg.py
encoder/encoder.py
encoder/comparator.py

ycbcr.py

```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)
    b = mat.dot(a)
    return tuple(b + col)


def ycbcr_to_rgb(t: tuple) -> tuple:
    a = np.asarray(t)
```

```
20      b = alg.inv(mat)
21      c = a - col
22      d = b.dot(c[0])
23      return tuple(d)
24
```

huffman.py

```
1   # Module computing Huffman compression
2
3
4   from collections import Counter, namedtuple
5   from heapq import heapify, heappop, heappush
6
7
8   # Node in a Huffman Tree
9   Node = namedtuple("Node", ["char", "freq"])
10
11  class HuffmanCompressor:
12      """Huffman compression implementation"""
13      def __init__(self):
14          self.encoding_table = {}
15          self.decoding_table = {}
16
17      def build_tables(self, s: str):
18          """create both the encodingn and decoding tables of a given string
19
20      parameters
21      ----------
22          -s : string used to build the tables
23
24      return
25      ------
26          - fill both the encoding and decoding table of the given class instance"""
27
```

```python
        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])


        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])
```

```python
    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

    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
```

fft.py

```python
 1  # fast-fourier transforms
 2
 3  import complex as cpx
 4  from numpy import log2
 5  from cmath import pi, exp, cos
 6  from scipy.fftpack import dct, idct
 7
 8
 9  def FFT(vector:list) -> list:
10      """calculate the fast fourier tranform of a vector
11
12      parameters
13      ----------
14          -vector : list of Complex object
15
16      return
17      ------
18          - 1-D fast fourier transform of the vector"""
19      n = len(vector)
20      assert log2(n).is_integer(), "make sure that the length of the arguement is a power of 2"
21      if n == 1:
22          return vector
23      poly_even, poly_odd = vector[::2] , vector[1::2]
24      res_even, res_odd = FFT(poly_even), FFT(poly_odd)
25      res = [cpx.Complex(0)] * n
26      for j in range(n//2):
27          w_j = cpx.exp_to_literal(-2*pi*j/n)
28          product = w_j * res_odd[j]
29          res[j] = res_even[j] + product
30          res[j + n//2] = res_even[j] - product
31      return res
```

```python
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

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):
```

```python
            res[i] = res[i] / cpx.Complex(n)
        return res

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

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)
```

```
105        factor = - pi / (N * 2)
106        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)]
107        temp = FFT(temp)
108        temp = [val.re for val in temp]
109        result = [None] * N
110        result[ : : 2] = temp[ : (N + 1) // 2]
111        result[-1 - N % 2 : : -2] = temp[(N + 1) // 2 : ]
112        return result
113
114   if __name__ == "__main__":
115        vectorCpx= [cpx.Complex(5), cpx.Complex(2), cpx.Complex(4), cpx.Complex(8)]
116        vector = [5, 2, 4, 8]
117        print("DCT : ", DCT(vectorCpx))
118        print("inverse + DCT : ", IDCT((DCT(vectorCpx))))
119        print("scipy dct :", dct(vector))
120        print("scipy + inverse dct: ", dct(idct(vector)))
121        print("scipy dct (ortho) : ", dct(vector, norm = "ortho"))
122        print("scipy inverse + dct (ortho) : ", idct(dct(vector, norm="ortho"), norm="ortho"))
```

complex.py

```
 1   # Module computing complex numbers
 2   # disclaimer : this class is not made to deal with less than 1e-10 values
 3
 4
 5   from numpy import arctan2, cos, pi, sin, sqrt
 6   from math import isclose
 7   from typing import Union, List
 8
 9
10   class Complex:
11        """Computing complex numbers"""
12        def __init__(self, real=0., imaginary=0.):
13             self.re = real # round(real, 15)
```

```python
            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)
    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)
```

```python
    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)
    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
```

```python
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

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
```

```python
        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


def nth_roots_unity(n:int) -> list:
    """ calculate the n roots of unity

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

```python
        return
        ------
            - a list of Complex containing the 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 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


if __name__ == "__main__":
    pass
```

arithmetic_coding.py

```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 = 0.0
    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 = 0.0
    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 = (0.0, 1.0)
    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


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:
```

```python
            for c, v in d.items():
                if appartient(i, c):
                    res.append(v)
                    i = (i - c[0]) / (c[1] - c[0])
                    break

        return res


# Examples

if __name__ == "__main__":
    print(encode("WIKI"))
    print(decode(0.171875, {"W": 0.25, "I": 0.5, "K": 0.25}, 4))
    print(encode("AAABBCCCCC"))
    print(decode(0.010783125000000005, {"A": 0.3, "B": 0.2, "C": 0.5}, 10))
    print(encode([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
    print(
        decode(
            encode([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
            {
                1: 0.1,
                2: 0.1,
                3: 0.1,
                4: 0.1,
                5: 0.1,
                6: 0.1,
                7: 0.1,
                8: 0.1,
                9: 0.1,
                10: 0.1,
            },
            10,
        )
    )
```

encoder/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):
```

```python
        return int2bytes(self.id_, 1) + int2bytes(
            (self.dc_ht_id << 4) + self.ac_ht_id, 1
        )

    def encode_SOF0_info(self):
        return (
            int2bytes(self.id_, 1)
            + int2bytes((self.horizontal << 4) + self.vertical, 1)
            + int2bytes(self.qt_id, 1)
        )

    def __repr__(self):
        return (
            f"{self.id_}: qt-{self.qt_id}, ht-(dc-{self.dc_ht_id}, "
            f"ac-{self.ac_ht_id}), sample-{self.vertical, self.horizontal} "
        )


class BitStreamReader:
    """simulate bitwise read"""

    def __init__(self, bytes_: bytes):
        self.bits = np.unpackbits(np.frombuffer(bytes_, dtype=np.uint8))
        self.index = 0

    def read_bit(self):
        if self.index >= self.bits.size:
            raise EOFError("Ran out of element")
        self.index += 1
        return self.bits[self.index - 1]

    def read_int(self, length):
        result = 0
        for _ in range(length):
            result = result * 2 + self.read_bit()
        return result

    def __repr__(self):
```

```python
            return f"[{self.index}, {self.bits.size}]"


class BitStreamWriter:
    """simulate bitwise write"""

    def __init__(self, length=10000):
        self.index = 0
        self.bits = np.zeros(length, dtype=np.uint8)

    def write_bitstring(self, bitstring):
        length = len(bitstring)
        if length + self.index > self.bits.size * 8:
            arr = np.zeros((length + self.index) // 8 * 2, dtype=np.uint8)
            arr[: self.bits.size] = self.bits
            self.bits = arr
        for bit in bitstring:
            self.bits[self.index // 8] |= int(bit) << (7 - self.index % 8)
            self.index += 1

    def to_bytes(self):
        return self.bits[: math.ceil(self.index / 8)].tobytes()

    def to_hex(self):
        length = math.ceil(self.index / 8) * 8
        for i in range(self.index, length):
            self.bits[i] = 1
        bytes_ = np.packbits(self.bits[:length])
        return " ".join(f"{b:2x}" for b in bytes_)


class BytesWriter(BytesIO):
    def __init__(self, *args, **kwargs):
        super(BytesWriter, self).__init__(*args, **kwargs)

    def add_bytes(self, *args):
        self.write(b"".join(args))
```

```python
def bytes2int(bytes_, byteorder="big"):
    return int.from_bytes(bytes_, byteorder)


def int2bytes(int_: int, length):
    return int_.to_bytes(length, byteorder="big")


def decode_2s_complement(complement, length) -> int:
    if length == 0:
        return 0
    if complement >> (length - 1) == 1:  # sign bit equal to one
        number = complement
    else:  # sign bit equal to zero
        number = 1 - 2**length + complement
    return number


def encode_2s_complement(number) -> str:
    """return the 2's complement representation as string"""
    if number == 0:
        return ""
    if number > 0:
        complement = bin(number)[2:]
    else:
        length = int(np.log2(-number)) + 1
        complement = bin(number - (1 - 2**length))[2:].zfill(length)
    return complement


def load_quantization_table(quality, component):
    # the below two tables was processed by zigzag encoding
    # in JPEG bit stream, the table is also stored in this order
    if component == "lum":
        q = np.array(
            [
                16,
```

```
151                      11,
152                      12,
153                      14,
154                      12,
155                      10,
156                      16,
157                      14,
158                      13,
159                      14,
160                      18,
161                      17,
162                      16,
163                      19,
164                      24,
165                      40,
166                      26,
167                      24,
168                      22,
169                      22,
170                      24,
171                      49,
172                      35,
173                      37,
174                      29,
175                      40,
176                      58,
177                      51,
178                      61,
179                      60,
180                      57,
181                      51,
182                      56,
183                      55,
184                      64,
185                      72,
186                      92,
187                      78,
188                      64,
```

```
                                    68,
                                    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,
```

```
227                         26,
228                         26,
229                         47,
230                         99,
231                         66,
232                         56,
233                         66,
234                         99,
235                         99,
236                         99,
237                         99,
238                         99,
239                         99,
240                         99,
241                         99,
242                         99,
243                         99,
244                         99,
245                         99,
246                         99,
247                         99,
248                         99,
249                         99,
250                         99,
251                         99,
252                         99,
253                         99,
254                         99,
255                         99,
256                         99,
257                         99,
258                         99,
259                         99,
260                         99,
261                         99,
262                         99,
263                         99,
264                         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)
```

```python
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)),
        DOWN_LEFT: lambda p: move(DOWN, move(LEFT, p)),
    }

    # move the point in different directions
    def move(direction, point):
        return move_func[direction](point)

    # return true if point is inside the block bounds
    def inbounds(p):
        return 0 <= p[0] < rows and 0 <= p[1] < cols

    # start in the top-left cell
    now = (0, 0)

    # True when moving up-right, False when moving down-left
    move_up = True
    trace = []

    for i in range(rows * cols):
        trace.append(now)
        if move_up:
            if inbounds(move(UP_RIGHT, now)):
                now = move(UP_RIGHT, now)
            else:
                move_up = False
                if inbounds(move(RIGHT, now)):
                    now = move(RIGHT, now)
                else:
```

```python
                    now = move(DOWN, now)
            else:
                if inbounds(move(DOWN_LEFT, now)):
                    now = move(DOWN_LEFT, now)
                else:
                    move_up = True
                    if inbounds(move(DOWN, now)):
                        now = move(DOWN, now)
                    else:
                        now = move(RIGHT, now)
    """
    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):
```

```python
        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]
```

```
417   def averageMatrix(
418       arrayMatrix,
419   ):  # given an array of 2D-array, return the average (coef by coef) 2D array
420       avgMatrix = np.zeros_like(arrayMatrix[0])
421       for i in range(avgMatrix.shape[0]):
422           for j in range(avgMatrix.shape[1]):
423               avgMatrix[i, j] = np.average(arrayMatrix[:, i, j])
424       return avgMatrix
425
426
427   if __name__ == "__main__":
428       arrMatrix = np.array([[[1, 2], [3, 4]], [[5, 2], [3, 4]]])
429       print(averageMatrix(arrMatrix))
430
```

encoder/huffman_jpeg.py

```
 1   import numpy as np
 2
 3   MAX_CLEN = 32  # assumed maximum initial code length
 4
 5
 6   def getFreq(data):
 7       freq = [0] * 257
 8       for elem in data:
 9           freq[elem] += 1
10       freq[256] = 1
11       return freq
12
13
14   def jpegGenerateOptimalTable(freq):
15       bits = [0] * (MAX_CLEN + 1)
16       bitPos = [0] * (MAX_CLEN + 1)
17       codesize = [0] * 257
```

```python
        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
```

```python
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

    for p in range(lastp):
        i = huffval[p]
        ehufco[i] = huffcode[p]
        ehufsi[i] = huffsize[p]

    return ehufsi, ehufco
```

```python
def jpegTransformTable(ehufsi, ehufco):
    table = {}
    for i in range(len(ehufco)):
        if ehufsi[i] != 0:
            endCode = bin(ehufco[i])[2:]
            nbZeros = ehufsi[i] - len(endCode)
            table[i] = "0" * nbZeros + endCode
    return table


def jpegCreateHuffmanTable(arr):
    freq = getFreq(arr)
    bits, huffval = jpegGenerateOptimalTable(freq)
    ehufsi, ehufco = jpegGenerateHuffmanTable(bits, huffval)
    table = jpegTransformTable(ehufsi, ehufco)
    return table


def convert_huffman_table(table):
    """convert huffman table to count and weigh"""
    # table[int] = string
    pairs = sorted(table.items(), key=lambda x: (len(x[1]), x[1]))
    weigh, codes = zip(*pairs)
    weigh = np.array(weigh, dtype=np.uint8)
    # count[i]: there are count[i] codes of length i+1
    count = np.zeros(16, dtype=np.uint8)
    for c in codes:
        count[len(c) - 1] += 1
    return count, weigh


def read_huffman_code(table, stream):
    prefix = ""
    while prefix not in table:
        prefix += str(stream.read_bit())
    return table[prefix]
```

```python
def reverse(table):
    return {v: k for k, v in table.items()}


# 4 recommended huffman tables in JPEG standard
# luminance DC
RM_Y_DC = {
    "00": 0,
    "010": 1,
    "011": 2,
    "100": 3,
    "101": 4,
    "110": 5,
    "1110": 6,
    "11110": 7,
    "111110": 8,
    "1111110": 9,
    "11111110": 10,
    "111111110": 11,
}

# luminance AC
RM_Y_AC = {
    "00": 1,
    "01": 2,
    "100": 3,
    "1010": 0,
    "1011": 4,
    "1100": 17,
    "11010": 5,
    "11011": 18,
    "11100": 33,
    "111010": 49,
    "111011": 65,
    "1111000": 6,
    "1111001": 19,
    "1111010": 81,
    "1111011": 97,
```

```
"11111000": 7,
"11111001": 34,
"11111010": 113,
"111110110": 20,
"111110111": 50,
"111111000": 129,
"111111001": 145,
"111111010": 161,
"1111110110": 8,
"1111110111": 35,
"1111111000": 66,
"1111111001": 177,
"1111111010": 193,
"11111110110": 21,
"11111110111": 82,
"11111111000": 209,
"11111111001": 240,
"111111110100": 36,
"111111110101": 51,
"111111110110": 98,
"111111110111": 114,
"1111111110000000": 130,
"1111111110000010": 9,
"1111111110000011": 10,
"1111111110000100": 22,
"1111111110000101": 23,
"1111111110000110": 24,
"1111111110000111": 25,
"1111111110001000": 26,
"1111111110001001": 37,
"1111111110001010": 38,
"1111111110001011": 39,
"1111111110001100": 40,
"1111111110001101": 41,
"1111111110001110": 42,
"1111111110001111": 52,
"1111111110010000": 53,
"1111111110010001": 54,
```

```
"1111111110010010": 55,
"1111111110010011": 56,
"1111111110010100": 57,
"1111111110010101": 58,
"1111111110010110": 67,
"1111111110010111": 68,
"1111111110011000": 69,
"1111111110011001": 70,
"1111111110011010": 71,
"1111111110011011": 72,
"1111111110011100": 73,
"1111111110011101": 74,
"1111111110011110": 83,
"1111111110011111": 84,
"1111111110100000": 85,
"1111111110100001": 86,
"1111111110100010": 87,
"1111111110100011": 88,
"1111111110100100": 89,
"1111111110100101": 90,
"1111111110100110": 99,
"1111111110100111": 100,
"1111111110101000": 101,
"1111111110101001": 102,
"1111111110101010": 103,
"1111111110101011": 104,
"1111111110101100": 105,
"1111111110101101": 106,
"1111111110101110": 115,
"1111111110101111": 116,
"1111111110110000": 117,
"1111111110110001": 118,
"1111111110110010": 119,
"1111111110110011": 120,
"1111111110110100": 121,
"1111111110110101": 122,
"1111111110110110": 131,
"1111111110110111": 132,
```

    "1111111110111000": 133,
    "1111111110111001": 134,
    "1111111110111010": 135,
    "1111111110111011": 136,
    "1111111110111100": 137,
    "1111111110111101": 138,
    "1111111110111110": 146,
    "1111111110111111": 147,
    "1111111111000000": 148,
    "1111111111000001": 149,
    "1111111111000010": 150,
    "1111111111000011": 151,
    "1111111111000100": 152,
    "1111111111000101": 153,
    "1111111111000110": 154,
    "1111111111000111": 162,
    "1111111111001000": 163,
    "1111111111001001": 164,
    "1111111111001010": 165,
    "1111111111001011": 166,
    "1111111111001100": 167,
    "1111111111001101": 168,
    "1111111111001110": 169,
    "1111111111001111": 170,
    "1111111111010000": 178,
    "1111111111010001": 179,
    "1111111111010010": 180,
    "1111111111010011": 181,
    "1111111111010100": 182,
    "1111111111010101": 183,
    "1111111111010110": 184,
    "1111111111010111": 185,
    "1111111111011000": 186,
    "1111111111011001": 194,
    "1111111111011010": 195,
    "1111111111011011": 196,
    "1111111111011100": 197,
    "1111111111011101": 198,

```python
        "1111111111011110": 199,
        "1111111111011111": 200,
        "1111111111100000": 201,
        "1111111111100001": 202,
        "1111111111100010": 210,
        "1111111111100011": 211,
        "1111111111100100": 212,
        "1111111111100101": 213,
        "1111111111100110": 214,
        "1111111111100111": 215,
        "1111111111101000": 216,
        "1111111111101001": 217,
        "1111111111101010": 218,
        "1111111111101011": 225,
        "1111111111101100": 226,
        "1111111111101101": 227,
        "1111111111101110": 228,
        "1111111111101111": 229,
        "1111111111110000": 230,
        "1111111111110001": 231,
        "1111111111110010": 232,
        "1111111111110011": 233,
        "1111111111110100": 234,
        "1111111111110101": 241,
        "1111111111110110": 242,
        "1111111111110111": 243,
        "1111111111111000": 244,
        "1111111111111001": 245,
        "1111111111111010": 246,
        "1111111111111011": 247,
        "1111111111111100": 248,
        "1111111111111101": 249,
        "1111111111111110": 250,
}

# chroma DC
RM_C_DC = {
    "00": 0,
```

```
        "01": 1,
        "10": 2,
        "110": 3,
        "1110": 4,
        "11110": 5,
        "111110": 6,
        "1111110": 7,
        "11111110": 8,
        "111111110": 9,
        "1111111110": 10,
        "11111111110": 11,
}

# chroma AC
RM_C_AC = {
        "00": 0,
        "01": 1,
        "100": 2,
        "1010": 3,
        "1011": 17,
        "11000": 4,
        "11001": 5,
        "11010": 33,
        "11011": 49,
        "111000": 6,
        "111001": 18,
        "111010": 65,
        "111011": 81,
        "1111000": 7,
        "1111001": 97,
        "1111010": 113,
        "11110110": 19,
        "11110111": 34,
        "11111000": 50,
        "11111001": 129,
        "111110100": 8,
        "111110101": 20,
        "111110110": 66,
```

```
"111110111": 145,
"111111000": 161,
"111111001": 177,
"111111010": 193,
"1111110110": 9,
"1111110111": 35,
"1111111000": 51,
"1111111001": 82,
"1111111010": 240,
"11111110110": 21,
"11111110111": 98,
"11111111000": 114,
"11111111001": 209,
"111111110100": 10,
"111111110101": 22,
"111111110110": 36,
"111111110111": 52,
"11111111100000": 225,
"11111111000010": 37,
"11111111000011": 241,
"111111110001000": 23,
"111111110001001": 24,
"111111110001010": 25,
"111111110001011": 26,
"111111110001100": 38,
"111111110001101": 39,
"111111110001110": 40,
"111111110001111": 41,
"111111110010000": 42,
"111111110010001": 53,
"111111110010010": 54,
"111111110010011": 55,
"111111110010100": 56,
"111111110010101": 57,
"111111110010110": 58,
"111111110010111": 67,
"111111110011000": 68,
"111111110011001": 69,
```

```
"1111111110011010": 70,
"1111111110011011": 71,
"1111111110011100": 72,
"1111111110011101": 73,
"1111111110011110": 74,
"1111111110011111": 83,
"1111111110100000": 84,
"1111111110100001": 85,
"1111111110100010": 86,
"1111111110100011": 87,
"1111111110100100": 88,
"1111111110100101": 89,
"1111111110100110": 90,
"1111111110100111": 99,
"1111111110101000": 100,
"1111111110101001": 101,
"1111111110101010": 102,
"1111111110101011": 103,
"1111111110101100": 104,
"1111111110101101": 105,
"1111111110101110": 106,
"1111111110101111": 115,
"1111111110110000": 116,
"1111111110110001": 117,
"1111111110110010": 118,
"1111111110110011": 119,
"1111111110110100": 120,
"1111111110110101": 121,
"1111111110110110": 122,
"1111111110110111": 130,
"1111111110111000": 131,
"1111111110111001": 132,
"1111111110111010": 133,
"1111111110111011": 134,
"1111111110111100": 135,
"1111111110111101": 136,
"1111111110111110": 137,
"1111111110111111": 138,
```

```
474        "1111111111000000": 146,
475        "1111111111000001": 147,
476        "1111111111000010": 148,
477        "1111111111000011": 149,
478        "1111111111000100": 150,
479        "1111111111000101": 151,
480        "1111111111000110": 152,
481        "1111111111000111": 153,
482        "1111111111001000": 154,
483        "1111111111001001": 162,
484        "1111111111001010": 163,
485        "1111111111001011": 164,
486        "1111111111001100": 165,
487        "1111111111001101": 166,
488        "1111111111001110": 167,
489        "1111111111001111": 168,
490        "1111111111010000": 169,
491        "1111111111010001": 170,
492        "1111111111010010": 178,
493        "1111111111010011": 179,
494        "1111111111010100": 180,
495        "1111111111010101": 181,
496        "1111111111010110": 182,
497        "1111111111010111": 183,
498        "1111111111011000": 184,
499        "1111111111011001": 185,
500        "1111111111011010": 186,
501        "1111111111011011": 194,
502        "1111111111011100": 195,
503        "1111111111011101": 196,
504        "1111111111011110": 197,
505        "1111111111011111": 198,
506        "1111111111100000": 199,
507        "1111111111100001": 200,
508        "1111111111100010": 201,
509        "1111111111100011": 202,
510        "1111111111100100": 210,
511        "1111111111100101": 211,
```

```
512        "1111111111100110": 212,
513        "1111111111100111": 213,
514        "1111111111101000": 214,
515        "1111111111101001": 215,
516        "1111111111101010": 216,
517        "1111111111101011": 217,
518        "1111111111101100": 218,
519        "1111111111101101": 226,
520        "1111111111101110": 227,
521        "1111111111101111": 228,
522        "1111111111110000": 229,
523        "1111111111110001": 230,
524        "1111111111110010": 231,
525        "1111111111110011": 232,
526        "1111111111110100": 233,
527        "1111111111110101": 234,
528        "1111111111110110": 242,
529        "1111111111110111": 243,
530        "1111111111111000": 244,
531        "1111111111111001": 245,
532        "1111111111111010": 246,
533        "1111111111111011": 247,
534        "1111111111111100": 248,
535        "1111111111111101": 249,
536        "1111111111111110": 250,
537    }
538
539    if __name__ == "__main__":
540        arr = np.array([np.random.randint(-127, 128) for _ in range(64)])
541        table = jpegCreateHuffmanTable(arr)
542        print(table)
543
```

encoder/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_jpeg 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]
```

```python
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
```

```python
            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])

    # 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),
    )
```

```python
        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()


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
```

```python
            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)
                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]))
                    )
```

```python
            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()


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/villeLyon.jpg")
    write_jpeg("data/villeLyonLow.jpg", np.array(im), 5, "4:1:1", False)


if __name__ == "__main__":
    main()
```

encoder/comparator.py

```python
import numpy as np
import encoder
import sys, os
from pathlib import Path
```

```python
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
import random as rd

LIM = 2  # number of files to test to


def compare(
    qualities=None,
    dataDirectory=None,
    outputDirectory=None,
    subsamples=None,
    useStdHuffmanTable=None,
    DeleteFilesAfterward=True,
):
    if qualities is None:
        qualities = [np.random.randint(0, 101)]
    if subsamples is None:
        subsamples = ["4:2:2"]
    if dataDirectory is None:
        dataDirectory = "./data/datasetBmp"
    if useStdHuffmanTable is None:
        useStdHuffmanTable = [False]
    stat = np.zeros(
        (LIM * len(qualities) * len(subsamples) * len(useStdHuffmanTable), 6),
        dtype=object,
    )  # dim 0 : quality factor, dim 1 : subsample method, dim 2 : usage of std Hf Tables, dim 3 : size
    before compression, dim 4 : size after compression, dim 5 : time to compress
    i = 0
    i_max = LIM * len(qualities) * len(subsamples) * len(useStdHuffmanTable)
    filesTreated = rd.choices(os.listdir(dataDirectory), k=LIM)
```

```python
        for quality in qualities:
            for subsample in subsamples:
                for hfTables in useStdHuffmanTable:
                    outputDirectory = f"./data/treated/quality{quality}-subsample{subsample}-stdHf{hfTables}"
                    for filename in filesTreated:
                        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()
                            encoder.write_jpeg(
                                f_out, np.array(image), quality, subsample, hfTables
                            )
                            time = t.time() - time
                            newSize = os.stat(f_out).st_size
                            stat[i][0] = quality
                            stat[i][1] = subsample
                            stat[i][2] = hfTables
                            stat[i][3] = previousSize
                            stat[i][4] = newSize
                            stat[i][5] = time
                        i += 1
                        print(f"{i}/{i_max}", end="\r")
                if DeleteFilesAfterward:
                    shutil.rmtree(outputDirectory)
    return stat


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(
```

```python
            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(output, stat):
    if os.path.isfile(output):
        pd.DataFrame(stat).to_csv(output, mode="a", index=False, header=False)
    else:
        pd.DataFrame(stat).to_csv(
            output,
            index=False,
            header=[
                "quality",
                "subsample",
                "stdHuffmanTables",
                "oldSize",
                "newSize",
                "time",
            ],
        )


def csv_to_stat(csvFile):
    stat = pd.read_csv(csvFile)
    return stat


def dataInterpreation(dataFrame):
    df = dataFrame
    qualities = df["quality"].unique()
```

```python
    qualitySize = {}
    qualityTime = {}
    for quality in qualities:
        qualitySize[quality] = int(df[df["quality"] == quality]["newSize"].mean())
        qualityTime[quality] = round(df[df["quality"] == quality]["time"].mean(), 3)
    stdSize = int(df[df["stdHuffmanTables"] == True]["newSize"].mean())
    stdTime = round(df[df["stdHuffmanTables"] == True]["time"].mean(), 3)
    nonStdSize = int(df[df["stdHuffmanTables"] == False]["newSize"].mean())
    nonStdTime = round(df[df["stdHuffmanTables"] == False]["time"].mean(), 3)

    plt.rcParams["figure.figsize"] = [10, 5]

    fig, (ax1, ax3) = plt.subplots(1, 2)
    ax2 = ax1.twinx()

    fig.suptitle("Comparaison des compressions en fonction du facteur de qualité")

    width = 0.25

    initialSize = 786486
    xaxis = list(qualitySize.keys())
    yaxisSize = np.array(list(qualitySize.values()))
    yaxisTime = np.array(list(qualityTime.values()))
    yaxisRatio = (initialSize - yaxisSize) / yaxisTime

    color1 = "tab:red"
    color2 = "tab:blue"
    color3 = "tab:green"

    ax1.bar(
        np.arange(len(qualitySize)) - width,
        yaxisSize,
        width,
        tick_label=xaxis,
        color=color1,
        label="Taille après compression",
    )
    ax2.bar(
```

```python
        np.arange(len(qualityTime)),
        yaxisTime,
        width,
        tick_label=xaxis,
        color=color2,
        label="Temps de compression",
    )
    ax3.bar(
        np.arange(len(qualitySize)),
        yaxisRatio,
        width,
        tick_label=xaxis,
        color=color3,
        label="Octets gagnés par seconde",
    )

    ax1.legend(loc="upper left")
    ax2.legend(loc="upper left", bbox_to_anchor=(0, 0.9))
    ax3.legend(loc="upper right")

    ax3.yaxis.tick_right()

    ax1.set_xlabel("Facteur de qualité")
    ax1.set_ylabel("Taille (en octets)", color=color1)
    ax2.set_ylabel("Temps (en secondes)", color=color2)
    ax3.set_xlabel("Facteur de qualité")
    ax3.set_ylabel("Taille gagné par unité de temps (octets.Hz)", color=color3)
    ax3.yaxis.set_label_position("right")

    plt.savefig("./data/treated/compressionComparaison", transparent=True)

    plt.rcParams["figure.figsize"] = [7, 5]
    plt.clf()

    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax2 = ax1.twinx()
```

```python
        fig.suptitle(
            "Comparaison des compressions en fonction des tables de Huffman utilisées"
        )

        yaxisSize = [stdSize, nonStdSize]
        yaxisTime = [stdTime, nonStdTime]

        labels = ["Tables Standards", "Tables Optimales"]
        ax1.bar(
            np.arange(2) - width / 2,
            yaxisSize,
            width,
            tick_label=labels,
            color=color1,
            label="Taille après compression",
        )
        ax2.bar(
            np.arange(2) + width / 2,
            yaxisTime,
            width,
            tick_label=labels,
            color=color2,
            label="Temps de compression",
        )

        ax1.legend(loc="upper center", bbox_to_anchor=(0.45, 1))
        ax2.legend(loc="upper center", bbox_to_anchor=(0.45, 0.9))

        ax1.set_ylabel("Taille (en octets)", color=color1)
        ax2.set_ylabel("Temps (en secondes)", color=color2)

        plt.savefig("./data/treated/compressionComparaison2", transparent=True)


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

    imgYCrCB = cv2.cvtColor(
```

```python
            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)
        )
        # fig.suptitle('Matrice de la luminance de "villeLyon.jpg"')

        ax1.matshow(arr1, cmap="cool", vmin=valueMin, vmax=valueMax)
        ax1.set_title("avant DCT")

        ax2.matshow(arr2, cmap="cool", vmin=valueMin, vmax=valueMax)
        ax2.set_title("après DCT")

        for i in range(arr1.shape[0]):
            for j in range(arr1.shape[1]):
                cNormal = int(arr1[i, j])
                cDct = int(arr2[i, j])
                ax1.text(i, j, str(cNormal), va="center", ha="center")
                ax2.text(i, j, str(cDct), va="center", ha="center")
        plt.savefig("./data/energyCompaction.png", transparent=True)
```

```python
def rgbToYCbCr_channel_bis():
    img = cv2.imread("./data/villeLyon.jpg")  # Read input image in BGR format

    imgYCrCB = cv2.cvtColor(
        img, cv2.COLOR_BGR2YCrCb
    )  # Convert RGB to YCrCb (Cb applies V, and Cr applies U).

    Y, Cr, Cb = cv2.split(imgYCrCB)

    # Fill Y and Cb with 128 (Y level is middle gray, and Cb is "neutralized").
    onlyCr = imgYCrCB.copy()
    onlyCr[:, :, 0] = 128
    onlyCr[:, :, 2] = 128
    onlyCr_as_bgr = cv2.cvtColor(
        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)
```

```python
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]])
    # print(dct(dct(test, axis=0, norm="ortho"), axis=1, norm='ortho'))

    # stat = compare(qualities = list(range(1, 101, 10)), subsamples=['4:4:4', '4:2:0', '4:1:1', '4:2:2'],
    useStdHuffmanTable=[True, False], DeleteFilesAfterward=True)
    # write_stat_csv("./data/treated/stat.csv", stat)
    stat = csv_to_stat("./data/treated/stat.csv")
    dataInterpreation(stat)
```