

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

```
1 # Module transforming RGB images into YCbCr
2 import numpy as np
3 import numpy.linalg as alg
4
5 mat = np.array(
6     [[65.481, 128.553, 24.966], [-37.797, -74.203, 112.0], [112.0, -93.786,
7     -18.214]]
8 )
9 col = np.array([[16, 128, 128]])
10
11
12 def rgb_to_ycbcr(rgb: tuple) -> tuple:
13     a = np.asarray(rgb)
14     b = mat.dot(a)
15     return tuple(b + col)
16
17
18 def ycbcr_to_rgb(t: tuple) -> tuple:
19     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
28     freq_table = Counter(s)
29
30     # create a heap of the nodes in the tree
31     heap = []
32     for char, freq in freq_table.items():
33         heap.append(Node(char, freq))
34     heapify(heap)
35
36     # create the Huffman tree
37     while len(heap) > 1:
38         left_node = heappop(heap)
39         right_node = heappop(heap)
40         combined_node = Node(None, left_node.freq + right_node.freq)
41         heappush(heap, combined_node)
42
43     def build_encoding_table(node, code=''):
44         if node.char is not None:
45             # if the node is a leaf, add it to the encoding table
46             self.encoding_table[node.char] = code
47             return
48         # if the node is not a leaf, recursively build the encoding table
49         build_encoding_table(node.left, code + '0')
50         build_encoding_table(node.right, code + '1')
51
52     build_encoding_table(heap[0])
53
54
55     def build_decoding_table(node, code=''):
56         if node.char is not None:
57             # if the node is a leaf, add it to the decoding table
58             self.decoding_table[code] = node.char
59             return
60         # if the node is not a leaf, recursively build the decoding table
61         build_decoding_table(node.left, code + "0")
62         build_decoding_table(node.right, code + "1")
63
64     build_decoding_table(heap[0])
65
66     def compress(self, s: str) -> str:
67         """compress the inputed string

```

```

68
69     parameters
70     -----
71     -s : string to be compressed
72
73     return
74     -----
75     - compressed string"""
76     compressed = ""
77     for char in s:
78         compressed += self.encoding_table[char]
79     return compressed
80
81     def decompress(self, compressed: str) -> str:
82         """decompress the inputed string
83
84     parameters
85     -----
86     -s : string to be compressed
87
88     return
89     -----
90     - decompressed string"""
91     decompressed = ""
92     i = 0
93     while i < len(compressed):
94         for j in range(i+1, len(compressed)+1):
95             if compressed[i:j] in self.decoding_table:
96                 decompressed += self.decoding_table[compressed[i:j]]
97                 i = j
98                 break
99
100     return decompressed
101
102
103

```

fft.py

```

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 argument 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
32
33 def IFFT_aux(vector:list) -> list:
34     """auxiliary function that makes the recursive steps of the IFFT algorithm
35     parameters
36     -----
37     -vector : list of Complex object
38
39     return
40     -----
41     - partial inverse of the 1-D fast fourier transform of the vector (lack
the division by n)"""
42     n = len(vector)
43     assert log2(n).is_integer(), "make sure that the length of the argument is a
power of 2"
44     if n == 1:
45         return vector
46     poly_even, poly_odd = vector[::2] , vector[1::2]
47     res_even, res_odd = IFFT_aux(poly_even), IFFT_aux(poly_odd)
48     res = [cpx.Complex(0)] * n
49     for j in range(n//2):
50         w_j = cpx.exp_to_literal((2 * pi * j) / n)
51         product = w_j * res_odd[j]
52         res[j] = res_even[j] + product
53         res[j + n//2] = res_even[j] - product
54     return res
55
56 def IFFT(vector:list) -> list:
57     """caculate the inverse of the fast fourier tranform of a vector (in order to
have ifft(fft(poly)) == poly)
58
59     parameters
60     -----
61     -vector : list of Complex object
62
63     return
64     -----
65     - inverse of the 1-D fast fourier transform of the vector"""
66     n = len(vector)
67     res = IFFT_aux(vector)
68     for i in range(n):
69         res[i] = res[i] / cpx.Complex(n)
70     return res

```

```

71
72 def DCT(vector:list, orthogonalize:bool=False, norm="forward"):
73     """calculate the one-dimensional type-II discrete cosine tranform of a matrix
74     (MAKHOUL) (using the FFT function previously defined)
75
76     parameters
77     -----
78         - vector: list of Numerical Object
79
80     return
81     -----
82         - discrete cosine tranform of the input"""
83     N = len(vector)
84     temp = vector[ : : 2] + vector[-1 - N % 2 : : -2]
85     temp = FFT(temp)
86     factor = - pi / (N * 2)
87     result = [2 * (val * (cpx.exp_to_literal(i * factor))).re for (i, val) in
88 enumerate(temp)]
89     if orthogonalize:
90         result[0] *= 2 ** (-1 / 2)
91     if norm == "ortho":
92         result[0] *= (N) **(-1 / 2)
93         result[1::] = [(2 / N) ** (1 / 2) * result[i] for i in range(1,
94 len(result))]
95     return result
96
97 def IDCT(vector:list):
98     """calculate the one-dimensional "inverse" type-III discrete cosine tranform
99     of a matrix (MAKHOUL) (using the FFT function previously defined)
100
101     parameters
102     -----
103         - vector: list of Numerical Object
104
105     return
106     -----
107         - type-III discrete cosine tranform of the input"""
108     N = len(vector)
109     factor = - pi / (N * 2)
110     temp = [(cpx.Complex(val) if i > 0 else (cpx.Complex(val) / cpx.Complex(2))) *
111 cpx.exp_to_literal(i * factor) for (i, val) in enumerate(vector)]
112     temp = FFT(temp)
113     temp = [val.re for val in temp]
114     result = [None] * N
115     result[ : : 2] = temp[ : (N + 1) // 2]
116     result[-1 - N % 2 : : -2] = temp[(N + 1) // 2 : ]
117     return result
118
119 if __name__ == "__main__":
120     vectorCpx= [cpx.Complex(5), cpx.Complex(2), cpx.Complex(4), cpx.Complex(8)]
121     vector = [5, 2, 4, 8]
122     print("DCT : ", DCT(vectorCpx))
123     print("inverse + DCT : ", IDCT((DCT(vectorCpx))))
124     print("scipy dct :", dct(vector))
125     print("scipy + inverse dct: ", dct(idct(vector)))
126     print("scipy dct (ortho) : ", dct(vector, norm = "ortho"))
127     print("scipy inverse + dct (ortho) : ", idct(dct(vector, norm="ortho"),
128 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)
14         self.im = imaginary # round(imaginary,15)
15     def __str__(self) -> str:
16         if self.im == 0.:
17             string = f"{self.re}"
18         elif self.re == 0:
19             string = f"i({self.im})"
20         else:
21             string = f"{self.re} + i({self.im})"
22         return string
23     __repr__ = __str__
24     def __eq__(self, other) -> bool:
25         return bool(isclose(self.re, other.re) and isclose(self.im, other.im))
26     def is_null(self):
27         return isclose(self.re, 0) and isclose(self.im, 0)
28     def is_real(self):
29         return isclose(self.im, 0)
30     def is_imaginary(self):
31         return isclose(self.re, 0)
32     def arg(self):
33         """return the argument of the complex number
34         return None if 0"""
35         if self.is_null():
36             arg = None
37         elif isclose(self.re, 0) and self.im > 0:
38             arg = pi / 2
39         elif isclose(self.re, 0) and self.im < 0:
40             arg = - pi / 2
41         else:
42             arg = round(arctan2(self.im, self.re), 15)
43         return arg
44     def module(self):
45         """return the module of the complex number"""
46         return round(sqrt(self.re**2 + self.im**2), 15)
47     def conjugate(self):
48         return Complex(self.re, -self.im)
49     #arithmetic
50     def __add__(self, other):
51         return Complex(self.re + other.re, self.im + other.im)
52     def __sub__(self, other):
53         return Complex(self.re - other.re, self.im - other.im)
54     def __mul__(self, other):
55         real = (self.re * other.re) - (self.im * other.im)
```

```

56         imaginary = (self.re * other.im) + (self.im * other.re)
57         return Complex(real, imaginary)
58     def __truediv__(self, other):
59         if other.is_null():
60             raise ValueError("Error : dividing by 0")
61         elif other.is_real():
62             return Complex(self.re / other.re, self.im / other.re)
63         else:
64             denominator = (other.re ** 2) + (other.im ** 2)
65             real = ((self.re * other.re) + (self.im * other.im)) / denominator
66             imaginary = ((self.im * other.re) - (self.re * other.im)) /
denominator
67             return Complex(real, imaginary)
68
69     Num = Union[int, float]
70
71     def addition(*complexes:Complex) -> Complex: #partially depreciated (can still be
usefull for more iterable arguments)
72         """calculate the sum of complex numbers
73
74         parameters
75         -----
76         - *complexes : iterable type of Complex
77
78         return
79         -----
80         - sum of the complex numbers"""
81
82         res = Complex(0)
83         for number in complexes:
84             res.re += number.re
85             res.im += number.im
86         return res
87
88     def difference(cpx1:Complex, cpx2:Complex = Complex(0)): #fully depreciated
(replaced by __sub__ Complex methods)
89         """calculate the difference of two complex numbers
90
91         parameters
92         -----
93         - cpx1 : Complex number
94         - cpx2 : Complex number to subtract to cpx1 (=Complex(0) by default)
95
96         return
97         -----
98         - difference of the two complex numbers"""
99         res = Complex()
100         res.re = cpx1.re - cpx2.re
101         res.im = cpx1.im - cpx2.im
102         return res
103
104     def product(*complexes:Complex) -> Complex: #partially depreciated (can still be
usefull for more iterable arguments)
105         """calculate the product of complex numbers
106
107         parameters
108         -----
109         - *complexes : iterable type of Complex
110

```

```

111     return
112     -----
113     - product of the complex numbers"""
114     res = Complex(1)
115     for number in complexes:
116         re = res.re * number.re - res.im * number.im
117         im = res.re * number.im + res.im * number.re
118         res.re = re
119         res.im = im
120     return res
121
122 def exp_to_literal(arg:float, module:float = 1.0) -> Complex:
123     """ return the literal expression of a complex number defined by its argument
124     and module
125
126     parameters
127     -----
128         - arg : type(float) (should be between 0 and 2pi)
129         - module : type(float) (must have a positive value)(=1 by default)
130
131     return
132     -----
133         - Complex number associated"""
134     assert(module >= 0), "second-argument(module) must have a positive value"
135     return Complex(module*cos(arg), module*sin(arg))
136
137 def nth_root(n:int, cpx:Complex = Complex(1)) -> Complex:
138     """calculate the nth root of a complex number
139
140     parameters
141     -----
142         - n : type(int)
143         - complex : type(Complex) (=Complex(1) by default) (must not be
144         Complex(0))
145
146     return
147     -----
148         - list of the nth roots"""
149     assert(cpx.re != 0 or cpx.im != 0), "second argument must be a non-zero
150     complex number"
151     module = cpx.module()
152     arg = cpx.arg()
153     if arg is not None:
154         return exp_to_literal((arg/n), module**(1/n))
155     else:
156         return Complex(1) #Not used case but just here to ensure nth_root cannot
157     return None
158
159 def nth_roots_unity(n:int) -> list:
160     """ calculate the n roots of unity
161
162     parameter
163     -----
164         - n : type(int) : must be a positive integer
165
166     return
167     -----
168         - a list of Complex containing the n roots of unity"""

```



```

166 roots = [Complex(1) for i in range(n)]
167 for k in range(0,n):
168     roots[k] = exp_to_literal((2*k*pi/n), 1.0)
169 return roots
170
171 def inverse_nth_roots_unity(n:int) -> list:
172     """ calculate the inversed n roots of unity
173
174     parameter
175     -----
176     - n : type(int) : must be a positive integer
177
178     return
179     -----
180     - a list of Complex containing the inversed n roots of unity"""
181     roots = [Complex(1) for i in range(n)]
182     for k in range(0,n):
183         roots[k] = exp_to_literal((-2*k*pi/n), 1.0)
184     return roots
185
186 def make_complex(values:List[Num]) -> List[Complex]:
187     res = []
188     for value in values:
189         res.extend([Complex(value)])
190     return res
191
192
193 if __name__ == "__main__":
194     pass
195

```

arithmetic_coding.py

```

1  def proba(data):
2      """
3      Créer le dictionnaire de probabilités d'apparition des différents caractères
4      """
5      assert len(data) != 0
6      d = {}
7      for x in data:
8          d[x] = d.get(x, 0) + (1 / len(data))
9      return d
10
11
12  def create_int(data):
13      """
14      Créer le dictionnaire des intervalles des différents caractères connaissant
15      les données
16      """
17      p = proba(data)
18      d = {}
19      n = 0.0
20      for c, v in p.items():
21          d[c] = (n, n + v)
22          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
```

```
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:
```

```
        for c, v in d.items():
```

```
            if appartient(i, c):
```

```
                res.append(v)
```

```
                i = (i - c[0]) / (c[1] - c[0])
```

```
                break
```

```
    return res
```

```

80
81
82 # Examples
83
84 if __name__ == "__main__":
85     print(encode("WIKI"))
86     print(decode(0.171875, {"W": 0.25, "I": 0.5, "K": 0.25}, 4))
87     print(encode("AAABBCCCCC"))
88     print(decode(0.010783125000000005, {"A": 0.3, "B": 0.2, "C": 0.5}, 10))
89     print(encode([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
90     print(
91         decode(
92             encode([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
93             {
94                 1: 0.1,
95                 2: 0.1,
96                 3: 0.1,
97                 4: 0.1,
98                 5: 0.1,
99                 6: 0.1,
100                7: 0.1,
101                8: 0.1,
102                9: 0.1,
103                10: 0.1,
104            },
105            10,
106        )
107    )
108

```

encoder/utils.py

```

1 import numpy as np
2 import math
3 import cv2
4 from io import BytesIO
5
6
7 # DCT block size
8 BH, BW = 8, 8
9
10
11 class MARKER:
12     SOI = b"\xff\xd8"
13     APP0 = b"\xff\xe0"
14     APPn = (b"\xff\xe1", b"\xff\xef") # n=1~15
15     DQT = b"\xffxdb"
16     SOF0 = b"\xff\xc0"
17     DHT = b"\xff\xc4"
18     DRI = b"\xff\xdd"
19     SOS = b"\xffxda"
20     EOI = b"\xff\xd9"
21
22

```

```

23 class ComponentInfo:
24     def __init__(self, id_, horizontal, vertical, qt_id, dc_ht_id, ac_ht_id):
25         self.id_ = id_
26         self.horizontal = horizontal
27         self.vertical = vertical
28         self.qt_id = qt_id
29         self.dc_ht_id = dc_ht_id
30         self.ac_ht_id = ac_ht_id
31
32     @classmethod
33     def default(cls):
34         return cls.__init__(*[0 for _ in range(6)])
35
36     def encode_SOS_info(self):
37         return int2bytes(self.id_, 1) + int2bytes(
38             (self.dc_ht_id << 4) + self.ac_ht_id, 1
39         )
40
41     def encode_SOF0_info(self):
42         return (
43             int2bytes(self.id_, 1)
44             + int2bytes((self.horizontal << 4) + self.vertical, 1)
45             + int2bytes(self.qt_id, 1)
46         )
47
48     def __repr__(self):
49         return (
50             f"{self.id_}: qt-{self.qt_id}, ht-(dc-{self.dc_ht_id}, "
51             f"ac-{self.ac_ht_id}), sample-{self.vertical, self.horizontal} "
52         )
53
54
55 class BitStreamReader:
56     """simulate bitwise read"""
57
58     def __init__(self, bytes_: bytes):
59         self.bits = np.unpackbits(np.frombuffer(bytes_, dtype=np.uint8))
60         self.index = 0
61
62     def read_bit(self):
63         if self.index >= self.bits.size:
64             raise EOFError("Ran out of element")
65         self.index += 1
66         return self.bits[self.index - 1]
67
68     def read_int(self, length):
69         result = 0
70         for _ in range(length):
71             result = result * 2 + self.read_bit()
72         return result
73
74     def __repr__(self):
75         return f"[{self.index}, {self.bits.size}]"
76
77
78 class BitStreamWriter:
79     """simulate bitwise write"""
80

```

```

81 def __init__(self, length=10000):
82     self.index = 0
83     self.bits = np.zeros(length, dtype=np.uint8)
84
85 def write_bitstring(self, bitstring):
86     length = len(bitstring)
87     if length + self.index > self.bits.size * 8:
88         arr = np.zeros((length + self.index) // 8 * 2, dtype=np.uint8)
89         arr[: self.bits.size] = self.bits
90         self.bits = arr
91     for bit in bitstring:
92         self.bits[self.index // 8] |= int(bit) << (7 - self.index % 8)
93         self.index += 1
94
95 def to_bytes(self):
96     return self.bits[: math.ceil(self.index / 8)].tobytes()
97
98 def to_hex(self):
99     length = math.ceil(self.index / 8) * 8
100    for i in range(self.index, length):
101        self.bits[i] = 1
102    bytes_ = np.packbits(self.bits[:length])
103    return " ".join(f"{b:2x}" for b in bytes_)
104
105
106 class BytesWriter(BytesIO):
107     def __init__(self, *args, **kwargs):
108         super(BytesWriter, self).__init__(*args, **kwargs)
109
110     def add_bytes(self, *args):
111         self.write(b"".join(args))
112
113
114 def bytes2int(bytes_, byteorder="big"):
115     return int.from_bytes(bytes_, byteorder)
116
117
118 def int2bytes(int_: int, length):
119     return int_.to_bytes(length, byteorder="big")
120
121
122 def decode_2s_complement(complement, length) -> int:
123     if length == 0:
124         return 0
125     if complement >> (length - 1) == 1: # sign bit equal to one
126         number = complement
127     else: # sign bit equal to zero
128         number = 1 - 2**length + complement
129     return number
130
131
132 def encode_2s_complement(number) -> str:
133     """return the 2's complement representation as string"""
134     if number == 0:
135         return ""
136     if number > 0:
137         complement = bin(number)[2:]
138     else:

```

```

139         length = int(np.log2(-number)) + 1
140         complement = bin(number - (1 - 2**length))[2:].zfill(length)
141     return complement
142
143
144 def load_quantization_table(quality, component):
145     # the below two tables was processed by zigzag encoding
146     # in JPEG bit stream, the table is also stored in this order
147     if component == "lum":
148         q = np.array(
149             [
150                 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,
189                 68,
190                 87,
191                 69,
192                 55,
193                 56,
194                 80,
195                 109,
196                 81,

```

```
197         87,
198         95,
199         98,
200         103,
201         104,
202         103,
203         62,
204         77,
205         113,
206         121,
207         112,
208         100,
209         120,
210         92,
211         101,
212         103,
213         99,
214     ],
215     dtype=np.int32,
216 )
217 elif component == "chr":
218     q = np.array(
219         [
220             17,
221             18,
222             18,
223             24,
224             21,
225             24,
226             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,
265         99,
266         99,
267         99,
268         99,
269         99,
270         99,
271         99,
272         99,
273         99,
274         99,
275         99,
276         99,
277         99,
278         99,
279         99,
280         99,
281         99,
282         99,
283         99,
284     ],
285     dtype=np.int32,
286 )
287 else:
288     raise ValueError(
289         (
290             f"component should be either 'lum' or 'chr', "
291             f"but '{component}' was found."
292         )
293     )
294 if 0 < quality < 50:
295     q = np.minimum(np.floor(50 / quality * q + 0.5), 255)
296 elif 50 <= quality <= 100:
297     q = np.maximum(np.floor((2 - quality / 50) * q + 0.5), 1)
298 else:
299     raise ValueError("quality should belong to (0, 100].")
300 return q.astype(np.int32)
301
302
303 def zigzag_points(rows, cols):
304     # constants for directions
305     UP, DOWN, RIGHT, LEFT, UP_RIGHT, DOWN_LEFT = range(6)
306
307     move_func = {
308         UP: lambda p: (p[0] - 1, p[1]),
309         DOWN: lambda p: (p[0] + 1, p[1]),
310         LEFT: lambda p: (p[0], p[1] - 1),
311         RIGHT: lambda p: (p[0], p[1] + 1),
312         UP_RIGHT: lambda p: move(UP, move(RIGHT, p)),

```



```

313     DOWN_LEFT: lambda p: move(DOWN, move(LEFT, p)),
314 }
315
316 # move the point in different directions
317 def move(direction, point):
318     return move_func[direction](point)
319
320 # return true if point is inside the block bounds
321 def inbounds(p):
322     return 0 <= p[0] < rows and 0 <= p[1] < cols
323
324 # start in the top-left cell
325 now = (0, 0)
326
327 # True when moving up-right, False when moving down-left
328 move_up = True
329 trace = []
330
331 for i in range(rows * cols):
332     trace.append(now)
333     if move_up:
334         if inbounds(move(UP_RIGHT, now)):
335             now = move(UP_RIGHT, now)
336         else:
337             move_up = False
338             if inbounds(move(RIGHT, now)):
339                 now = move(RIGHT, now)
340             else:
341                 now = move(DOWN, now)
342     else:
343         if inbounds(move(DOWN_LEFT, now)):
344             now = move(DOWN_LEFT, now)
345         else:
346             move_up = True
347             if inbounds(move(DOWN, now)):
348                 now = move(DOWN, now)
349             else:
350                 now = move(RIGHT, now)
351
352 """
353 for rows = cols = 8, the actual 1-D index:
354     0, 1, 8, 16, 9, 2, 3, 10, 17, 24, 32, 25, 18, 11, 4, 5,
355     12, 19, 26, 33, 40, 48, 41, 34, 27, 20, 13, 6, 7, 14, 21, 28,
356     35, 42, 49, 56, 57, 50, 43, 36, 29, 22, 15, 23, 30, 37, 44, 51,
357     58, 59, 52, 45, 38, 31, 39, 46, 53, 60, 61, 54, 47, 55, 62, 63
358 """
359 return trace
360
361 def RGB2YCbCr(im):
362     im = im.astype(np.float32)
363     im = cv2.cvtColor(im, cv2.COLOR_RGB2YCrCb)
364     """
365     RGB [0, 255]
366     opencv uses the following equations to conduct color conversion in float32
367         Y = 0.299 * R + 0.587 * G + 0.114 * B
368         Cb = (B - Y) * 0.564 + 0.5
369         Cr = (R - Y) * 0.713 + 0.5
370     Y [0, 255], Cb, Cr [-128, 127]

```

```

371 """
372 # convert YCrCb to YCbCr
373 Y, Cr, Cb = np.split(im, 3, axis=-1)
374 im = np.concatenate([Y, Cb, Cr], axis=-1)
375 return im
376
377
378 def YCbCr2RGB(im):
379     im = im.astype(np.float32)
380     Y, Cb, Cr = np.split(im, 3, axis=-1)
381     im = np.concatenate([Y, Cr, Cb], axis=-1)
382     im = cv2.cvtColor(im, cv2.COLOR_YCrCb2RGB)
383     """
384     Y [0, 255], Cb, Cr [-128, 127]
385     conversion equation (float32):
386         B = (Cb - 0.5) / 0.564 + Y
387         R = (Cr - 0.5) / 0.713 + Y
388         G = (Y - 0.299 * R - 0.114 * B) / 0.587
389     RGB [0, 255]
390     """
391     return im
392
393
394 def bits_required(n):
395     n = abs(n)
396     result = 0
397     while n > 0:
398         n >>= 1
399         result += 1
400     return result
401
402
403 def divide_blocks(im, mh, mw):
404     h, w = im.shape[:2]
405     return im.reshape(h // mh, mh, w // mw, mw).swapaxes(1, 2).reshape(-1, mh, mw)
406
407
408 def restore_image(block, nh, nw):
409     bh, bw = block.shape[1:]
410     return block.reshape(nh, nw, bh, bw).swapaxes(1, 2).reshape(nh * bh, nw * bw)
411
412
413 def flatten(lst):
414     return [item for sublist in lst for item in sublist]
415
416
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]]])

```

```
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
18     nzIndex = [0] * 257
19
20     others = [-1] * 257
21
22     numNzSymbols = 0
23     for i in range(257):
24         if freq[i]:
25             nzIndex[numNzSymbols] = i
26             freq[numNzSymbols] = freq[i]
27             numNzSymbols += 1
28
29     huffval = [0] * (numNzSymbols - 1)
30
31     while True:
32         c1 = -1
33         c2 = -1
34         v = 1000000000
35         v2 = 1000000000
36         for i in range(numNzSymbols):
37             if freq[i] <= v2:
38                 if freq[i] <= v:
39                     c2 = c1
40                     v2 = v
41                     v = freq[i]
42                     c1 = i
43                 else:
44                     v2 = freq[i]
45                     c2 = i
46
47         if c2 < 0:
48             break
49
```

```

50     freq[c1] += freq[c2]
51     freq[c2] = 1000000001
52
53     codesize[c1] += 1
54     while others[c1] >= 0:
55         c1 = others[c1]
56         codesize[c1] += 1
57
58     others[c1] = c2
59
60     codesize[c2] += 1
61     while others[c2] >= 0:
62         c2 = others[c2]
63         codesize[c2] += 1
64
65     for i in range(numNzSymbols):
66         bits[codesize[i]] += 1
67
68     p = 0
69     for i in range(1, MAX_CLEN + 1):
70         bitPos[i] = p
71         p += bits[i]
72
73     for i in range(MAX_CLEN, 16, -1):
74         while bits[i] > 0:
75             j = i - 2
76             while bits[j] == 0:
77                 j -= 1
78             bits[i] -= 2
79             bits[i - 1] += 1
80             bits[j + 1] += 2
81             bits[j] -= 1
82
83     i = MAX_CLEN
84     while bits[i] == 0:
85         i -= 1
86     bits[i] -= 1
87
88     for i in range(numNzSymbols - 1):
89         huffval[bitPos[codesize[i]]] = nzIndex[i]
90         bitPos[codesize[i]] += 1
91
92     return bits, huffval
93
94
95 def jpegGenerateHuffmanTable(bits, huffval):
96     huffsize = [0] * 257
97     huffcode = [0] * 257
98
99     p = 0
100    for l in range(1, 17):
101        i = bits[l]
102        while i:
103            huffsize[p] = l
104            p += 1
105            i -= 1
106
107    huffsize[p] = 0

```

```

108     lastp = p
109
110     code = 0
111     si = huffsize[0]
112     p = 0
113     while huffsize[p]:
114         while huffsize[p] == si:
115             huffcode[p] = code
116             code += 1
117             p += 1
118             code <= 1
119             si += 1
120
121     ehufco = [0] * 257
122     ehufsi = [0] * 257
123
124     for p in range(lastp):
125         i = huffval[p]
126         ehufco[i] = huffcode[p]
127         ehufsi[i] = huffsize[p]
128
129     return ehufsi, ehufco
130
131
132 def jpegTransformTable(ehufsi, ehufco):
133     table = {}
134     for i in range(len(ehufco)):
135         if ehufsi[i] != 0:
136             endCode = bin(ehufco[i])[2:]
137             nbZeros = ehufsi[i] - len(endCode)
138             table[i] = "0" * nbZeros + endCode
139     return table
140
141
142 def jpegCreateHuffmanTable(arr):
143     freq = getFreq(arr)
144     bits, huffval = jpegGenerateOptimalTable(freq)
145     ehufsi, ehufco = jpegGenerateHuffmanTable(bits, huffval)
146     table = jpegTransformTable(ehufsi, ehufco)
147     return table
148
149
150 def convert_huffman_table(table):
151     """convert huffman table to count and weigh"""
152     # table[int] = string
153     pairs = sorted(table.items(), key=lambda x: (len(x[1]), x[1]))
154     weigh, codes = zip(*pairs)
155     weigh = np.array(weigh, dtype=np.uint8)
156     # count[i]: there are count[i] codes of length i+1
157     count = np.zeros(16, dtype=np.uint8)
158     for c in codes:
159         count[len(c) - 1] += 1
160     return count, weigh
161
162
163 def read_huffman_code(table, stream):
164     prefix = ""
165     while prefix not in table:

```

```

166     prefix += str(stream.read_bit())
167     return table[prefix]
168
169
170 def reverse(table):
171     return {v: k for k, v in table.items()}
172
173
174 # 4 recommended huffman tables in JPEG standard
175 # luminance DC
176 RM_Y_DC = {
177     "00": 0,
178     "010": 1,
179     "011": 2,
180     "100": 3,
181     "101": 4,
182     "110": 5,
183     "1110": 6,
184     "11110": 7,
185     "111110": 8,
186     "1111110": 9,
187     "11111110": 10,
188     "111111110": 11,
189 }
190
191 # luminance AC
192 RM_Y_AC = {
193     "00": 1,
194     "01": 2,
195     "100": 3,
196     "1010": 0,
197     "1011": 4,
198     "1100": 17,
199     "11010": 5,
200     "11011": 18,
201     "11100": 33,
202     "111010": 49,
203     "111011": 65,
204     "1111000": 6,
205     "1111001": 19,
206     "1111010": 81,
207     "1111011": 97,
208     "11111000": 7,
209     "11111001": 34,
210     "11111010": 113,
211     "111110110": 20,
212     "111110111": 50,
213     "111111000": 129,
214     "111111001": 145,
215     "111111010": 161,
216     "1111110110": 8,
217     "1111110111": 35,
218     "1111111000": 66,
219     "1111111001": 177,
220     "1111111010": 193,
221     "11111110110": 21,
222     "11111110111": 82,
223     "11111111000": 209,

```

224 "11111111001": 240,
225 "111111110100": 36,
226 "111111110101": 51,
227 "111111110110": 98,
228 "111111110111": 114,
229 "11111111000000": 130,
230 "111111110000010": 9,
231 "111111110000011": 10,
232 "111111110000100": 22,
233 "111111110000101": 23,
234 "111111110000110": 24,
235 "111111110000111": 25,
236 "111111110001000": 26,
237 "111111110001001": 37,
238 "111111110001010": 38,
239 "111111110001011": 39,
240 "111111110001100": 40,
241 "111111110001101": 41,
242 "111111110001110": 42,
243 "111111110001111": 52,
244 "111111110010000": 53,
245 "111111110010001": 54,
246 "111111110010010": 55,
247 "111111110010011": 56,
248 "111111110010100": 57,
249 "111111110010101": 58,
250 "111111110010110": 67,
251 "111111110010111": 68,
252 "111111110011000": 69,
253 "111111110011001": 70,
254 "111111110011010": 71,
255 "111111110011011": 72,
256 "111111110011100": 73,
257 "111111110011101": 74,
258 "111111110011110": 83,
259 "111111110011111": 84,
260 "111111110100000": 85,
261 "111111110100001": 86,
262 "111111110100010": 87,
263 "111111110100011": 88,
264 "111111110100100": 89,
265 "111111110100101": 90,
266 "111111110100110": 99,
267 "111111110100111": 100,
268 "111111110101000": 101,
269 "111111110101001": 102,
270 "111111110101010": 103,
271 "111111110101011": 104,
272 "111111110101100": 105,
273 "111111110101101": 106,
274 "111111110101110": 115,
275 "111111110101111": 116,
276 "111111110110000": 117,
277 "111111110110001": 118,
278 "111111110110010": 119,
279 "111111110110011": 120,
280 "111111110110100": 121,
281 "111111110110101": 122,

282 "1111111110110110": 131,
283 "1111111110110111": 132,
284 "1111111110111000": 133,
285 "1111111110111001": 134,
286 "1111111110111010": 135,
287 "1111111110111011": 136,
288 "1111111110111100": 137,
289 "1111111110111101": 138,
290 "1111111110111110": 146,
291 "1111111110111111": 147,
292 "1111111111000000": 148,
293 "1111111111000001": 149,
294 "1111111111000010": 150,
295 "1111111111000011": 151,
296 "1111111111000100": 152,
297 "1111111111000101": 153,
298 "1111111111000110": 154,
299 "1111111111000111": 162,
300 "1111111111001000": 163,
301 "1111111111001001": 164,
302 "1111111111001010": 165,
303 "1111111111001011": 166,
304 "1111111111001100": 167,
305 "1111111111001101": 168,
306 "1111111111001110": 169,
307 "1111111111001111": 170,
308 "1111111111010000": 178,
309 "1111111111010001": 179,
310 "1111111111010010": 180,
311 "1111111111010011": 181,
312 "1111111111010100": 182,
313 "1111111111010101": 183,
314 "1111111111010110": 184,
315 "1111111111010111": 185,
316 "1111111111011000": 186,
317 "1111111111011001": 194,
318 "1111111111011010": 195,
319 "1111111111011011": 196,
320 "1111111111011100": 197,
321 "1111111111011101": 198,
322 "1111111111011110": 199,
323 "1111111111011111": 200,
324 "1111111111100000": 201,
325 "1111111111100001": 202,
326 "1111111111100010": 210,
327 "1111111111100011": 211,
328 "1111111111100100": 212,
329 "1111111111100101": 213,
330 "1111111111100110": 214,
331 "1111111111100111": 215,
332 "1111111111101000": 216,
333 "1111111111101001": 217,
334 "1111111111101010": 218,
335 "1111111111101011": 225,
336 "1111111111101100": 226,
337 "1111111111101101": 227,
338 "1111111111101110": 228,
339 "1111111111101111": 229,


```
340     "11111111111110000": 230,
341     "11111111111110001": 231,
342     "11111111111110010": 232,
343     "11111111111110011": 233,
344     "11111111111110100": 234,
345     "11111111111110101": 241,
346     "11111111111110110": 242,
347     "11111111111110111": 243,
348     "11111111111110000": 244,
349     "11111111111110001": 245,
350     "11111111111110100": 246,
351     "11111111111110110": 247,
352     "11111111111110000": 248,
353     "11111111111110100": 249,
354     "11111111111110110": 250,
355 }
356
357 # chroma DC
358 RM_C_DC = {
359     "00": 0,
360     "01": 1,
361     "10": 2,
362     "110": 3,
363     "1110": 4,
364     "11110": 5,
365     "111110": 6,
366     "1111110": 7,
367     "11111110": 8,
368     "111111110": 9,
369     "1111111110": 10,
370     "11111111110": 11,
371 }
372
373 # chroma AC
374 RM_C_AC = {
375     "00": 0,
376     "01": 1,
377     "100": 2,
378     "1010": 3,
379     "1011": 17,
380     "11000": 4,
381     "11001": 5,
382     "11010": 33,
383     "11011": 49,
384     "111000": 6,
385     "111001": 18,
386     "111010": 65,
387     "111011": 81,
388     "1111000": 7,
389     "1111001": 97,
390     "1111010": 113,
391     "11110110": 19,
392     "11110111": 34,
393     "11111000": 50,
394     "11111001": 129,
395     "111110100": 8,
396     "111110101": 20,
397     "111110110": 66,
```

398 "111110111": 145,
399 "111111000": 161,
400 "111111001": 177,
401 "111111010": 193,
402 "1111110110": 9,
403 "1111110111": 35,
404 "1111111000": 51,
405 "1111111001": 82,
406 "1111111010": 240,
407 "11111110110": 21,
408 "11111110111": 98,
409 "11111111000": 114,
410 "11111111001": 209,
411 "111111110100": 10,
412 "111111110101": 22,
413 "111111110110": 36,
414 "111111110111": 52,
415 "11111111100000": 225,
416 "111111111000010": 37,
417 "111111111000011": 241,
418 "1111111110001000": 23,
419 "1111111110001001": 24,
420 "1111111110001010": 25,
421 "1111111110001011": 26,
422 "1111111110001100": 38,
423 "1111111110001101": 39,
424 "1111111110001110": 40,
425 "1111111110001111": 41,
426 "1111111110010000": 42,
427 "1111111110010001": 53,
428 "1111111110010010": 54,
429 "1111111110010011": 55,
430 "1111111110010100": 56,
431 "1111111110010101": 57,
432 "1111111110010110": 58,
433 "1111111110010111": 67,
434 "1111111110011000": 68,
435 "1111111110011001": 69,
436 "1111111110011010": 70,
437 "1111111110011011": 71,
438 "1111111110011100": 72,
439 "1111111110011101": 73,
440 "1111111110011110": 74,
441 "1111111110011111": 83,
442 "1111111110100000": 84,
443 "1111111110100001": 85,
444 "1111111110100010": 86,
445 "1111111110100011": 87,
446 "1111111110100100": 88,
447 "1111111110100101": 89,
448 "1111111110100110": 90,
449 "1111111110100111": 99,
450 "1111111110101000": 100,
451 "1111111110101001": 101,
452 "1111111110101010": 102,
453 "1111111110101011": 103,
454 "1111111110101100": 104,
455 "1111111110101101": 105,

456 "1111111110101110": 106,
457 "1111111110101111": 115,
458 "1111111110110000": 116,
459 "1111111110110001": 117,
460 "1111111110110010": 118,
461 "1111111110110011": 119,
462 "1111111110110100": 120,
463 "1111111110110101": 121,
464 "1111111110110110": 122,
465 "1111111110110111": 130,
466 "1111111110111000": 131,
467 "1111111110111001": 132,
468 "1111111110111010": 133,
469 "1111111110111011": 134,
470 "1111111110111100": 135,
471 "1111111110111101": 136,
472 "1111111110111110": 137,
473 "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     "1111111111100000": 229,
523     "1111111111100001": 230,
524     "1111111111100010": 231,
525     "1111111111100011": 232,
526     "1111111111101000": 233,
527     "1111111111101001": 234,
528     "1111111111101010": 242,
529     "1111111111101011": 243,
530     "1111111111100000": 244,
531     "1111111111100001": 245,
532     "1111111111101010": 246,
533     "1111111111101011": 247,
534     "1111111111101000": 248,
535     "1111111111101001": 249,
536     "1111111111101010": 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

```

1  from math import ceil
2  import cv2
3  import numpy as np
4  from PIL import Image
5  from pathlib import Path
6
7  from utils import *
8  from huffman_jpeg import *
9
10
11 def padding(im, mh, mw):
12     """
13     pad use boundary pixels so that its height and width are
14     the multiple of the height and width of MCUs, respectively
15     """
16     h, w, d = im.shape
17     if h % mh == 0 and w % mw == 0:
18         return im
19     hh, ww = ceil(h / mh) * mh, ceil(w / mw) * mw
20     im_ex = np.zeros_like(im, shape=(hh, ww, d))
21     im_ex[:h, :w] = im

```

```

22     im_ex[:, w:] = im_ex[:, w - 1 : w]
23     im_ex[h:, :] = im_ex[h - 1 : h, :]
24     return im_ex
25
26
27 mcu_sizes = {
28     "4:2:0": (BH * 2, BW * 2),
29     "4:1:1": (BH * 2, BW * 2),
30     "4:2:2": (BH, BW * 2),
31     "4:4:4": (BH, BW),
32 }
33
34
35 def scan_blocks(mcu, mh, mw):
36     """
37     scan MCU to blocks for DPCM, for 4:2:0, the scan order is as follows:
38     ----- | -----
39     | 0 | 1 | | | 4 | 5 |
40     ----- | -----
41     | 2 | 3 | | | 6 | 7 |
42     ----- | -----
43     """
44     blocks = (
45         mcu.reshape(-1, mh // BH, BH, mw // BW, BW).swapaxes(2, 3).reshape(-1, BH,
46         BW)
47     )
48     return blocks
49
50 def DCT(blocks):
51     dct = np.zeros_like(blocks)
52     for i in range(blocks.shape[0]):
53         dct[i] = cv2.dct(blocks[i])
54     return dct
55
56
57 def zigzag_encode(dct):
58     trace = zigzag_points(BH, BW)
59     zz = np.zeros_like(dct).reshape(-1, BH * BW)
60     for i, p in enumerate(trace):
61         zz[:, i] = dct[:, p[0], p[1]]
62     return zz
63
64
65 def quantization(dct, table):
66     ret = dct / table[None]
67     return np.round(ret).astype(np.int32)
68
69
70 def DPCM(dct):
71     """
72     encode the DC differences
73     """
74     dc_pred = dct.copy()
75     dc_pred[1:, 0] = dct[1:, 0] - dct[:-1, 0]
76     return dc_pred
77
78
79 def run_length_encode(arr):

```

determine where the sequence is ending prematurely

```
80 last_nonzero = -1
81
82 for i, elem in enumerate(arr):
83     if elem != 0:
84         last_nonzero = i
85
86 rss, values = [], []
87 run_length = 0
88 for i, elem in enumerate(arr):
89     if i > last_nonzero:
90         rss.append(0)
91         values.append(0)
92         break
93     elif elem == 0 and run_length < 15:
94         run_length += 1
95     else:
96         size = bits_required(elem)
97         rss.append((run_length << 4) + size)
98         values.append(elem)
99         run_length = 0
100 return rss, values
```

```
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,
189     # so we apply zigzag encoding to DCT block first
190     Y_z = zigzag_encode(Y_dct)
191     qt_y = load_quantization_table(quality, "lum")
192     Y_q = quantization(Y_z, qt_y)
193     Y_p = DPCM(Y_q)
194     # whether to use recommended huffman table
195     if use_rm_ht:

```

```

196 Y_dc_ht, Y_ac_ht = reverse(RM_Y_DC), reverse(RM_Y_AC)
197 else:
198     Y_dc_ht = jpegCreateHuffmanTable(np.vectorize(bits_required)(Y_p[:, 0]))
199     Y_ac_ht = jpegCreateHuffmanTable(
200         flatten(run_length_encode(Y_p[i, 1:])[0] for i in range(Y_p.shape[0]))
201     )
202 qts, hts = [qt_y], [Y_dc_ht, Y_ac_ht]
203 cop_infos = [ComponentInfo(1, mw // BW, mh // BH, 0, 0, 0)]
204 # the number of Y DCT blocks in an MCU
205 num = (mw // BW) * (mh // BH)
206 mcu_hts = [(Y_dc_ht, Y_ac_ht) for _ in range(num)]
207 # assign DCT blocks to MCUs
208 mcu_ = Y_p.reshape(-1, num, BH * BW)
209
210 if depth == 3:
211     # chroma subsample
212     ch = im[:, :, mh // BH, :, mw // BW, 1:]
213     Cb = divide_blocks(ch[:, :, 0], BH, BW)
214     Cr = divide_blocks(ch[:, :, 1], BH, BW)
215     Cb_dct, Cr_dct = DCT(Cb), DCT(Cr)
216     Cb_z, Cr_z = zigzag_encode(Cb_dct), zigzag_encode(Cr_dct)
217     qt_c = load_quantization_table(quality, "chr")
218     Cb_q, Cr_q = quantization(Cb_z, qt_c), quantization(Cr_z, qt_c)
219     Cb_p, Cr_p = DPCM(Cb_q), DPCM(Cr_q)
220     if use_rm_ht:
221         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__":

```


encoder/comparator.py

```

1  import numpy as np
2  import encoder
3  import sys, os
4  from pathlib import Path
5  from PIL import Image
6  import cv2
7  import pandas as pd
8  import time as t
9  import shutil
10 import utils
11 import matplotlib.pyplot as plt
12 from encoder import DCT, padding
13 from scipy.fftpack import dct
14 import random as rd
15
16 LIM = 2 # number of files to test to
17
18
19 def compare(
20     qualities=None,
21     dataDirectory=None,
22     outputDirectory=None,
23     subsamples=None,
24     useStdHuffmanTable=None,
25     DeleteFilesAfterward=True,
26 ):
27     if qualities is None:
28         qualities = [np.random.randint(0, 101)]
29     if subsamples is None:
30         subsamples = ["4:2:2"]
31     if dataDirectory is None:
32         dataDirectory = "./data/datasetBmp"
33     if useStdHuffmanTable is None:
34         useStdHuffmanTable = [False]
35     stat = np.zeros(
36         (LIM * len(qualities) * len(subsamples) * len(useStdHuffmanTable), 6),
37         dtype=object,
38     ) # 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
39     i = 0
40     i_max = LIM * len(qualities) * len(subsamples) * len(useStdHuffmanTable)
41     filesTreated = rd.choices(os.listdir(dataDirectory), k=LIM)
42     for quality in qualities:
43         for subsample in subsamples:
44             for hfTables in useStdHuffmanTable:
45                 outputDirectory = f"./data/treated/quality{quality}-
subsample{subsample}-stdHf{hfTables}"
46                 for filename in filesTreated:
47                     f = os.path.join(dataDirectory, filename)

```

```

48         if not os.path.exists(outputDirectory):
49             os.makedirs(outputDirectory)
50         f_out = os.path.join(outputDirectory, filename + ".jpg")
51         if os.path.isfile(f):
52             previousSize = os.stat(f).st_size
53             image = Image.open(f)
54             time = t.time()
55             encoder.write_jpeg(
56                 f_out, np.array(image), quality, subsample, hfTables
57             )
58             time = t.time() - time
59             newSize = os.stat(f_out).st_size
60             stat[i][0] = quality
61             stat[i][1] = subsample
62             stat[i][2] = hfTables
63             stat[i][3] = previousSize
64             stat[i][4] = newSize
65             stat[i][5] = time
66         i += 1
67         print(f"{i}/{i_max}", end="\r")
68         if DeleteFilesAfterward:
69             shutil.rmtree(outputDirectory)
70     return stat
71
72
73 def write_stat(statFile, stat, quality, subsample, standHuffTables):
74     with open(statFile, "a+") as f:
75         f.write("\n" * 2)
76         f.write("New sample \n")
77         f.write(f"Size of sample : {LIM} images \n")
78         f.write(
79             f"Parameters of compression : (quality) {quality}, (subsample)
{subsample}, (usage of standard HuffTables) {'Yes' if standHuffTables else 'No'}
\n"
80         )
81         avgPreviousSize = np.average(stat[:, 0])
82         avgNewSize = np.average(stat[:, 1])
83         f.write(
84             f"Average size of image before compression : {avgPreviousSize} bytes
\n"
85         )
86         f.write(f"Average size of images after compression : {avgNewSize} bytes
\n")
87         f.write(f"Ratio is {avgPreviousSize / avgNewSize:.2f}")
88
89
90 def write_stat_csv(output, stat):
91     if os.path.isfile(output):
92         pd.DataFrame(stat).to_csv(output, mode="a", index=False, header=False)
93     else:
94         pd.DataFrame(stat).to_csv(
95             output,
96             index=False,
97             header=[
98                 "quality",
99                 "subsample",
100                 "stdHuffmanTables",
101                 "oldSize",
102                 "newSize",

```

```

103         "time",
104     ],
105 )
106
107
108 def csv_to_stat(csvFile):
109     stat = pd.read_csv(csvFile)
110     return stat
111
112
113 def dataInterpreation(dataFrame):
114     df = dataFrame
115     qualities = df["quality"].unique()
116     qualitySize = {}
117     qualityTime = {}
118     for quality in qualities:
119         qualitySize[quality] = int(df[df["quality"] == quality]["newSize"].mean())
120         qualityTime[quality] = round(df[df["quality"] == quality]["time"].mean(),
3)
121     stdSize = int(df[df["stdHuffmanTables"] == True]["newSize"].mean())
122     stdTime = round(df[df["stdHuffmanTables"] == True]["time"].mean(), 3)
123     nonStdSize = int(df[df["stdHuffmanTables"] == False]["newSize"].mean())
124     nonStdTime = round(df[df["stdHuffmanTables"] == False]["time"].mean(), 3)
125
126     plt.rcParams["figure.figsize"] = [10, 5]
127
128     fig, (ax1, ax3) = plt.subplots(1, 2)
129     ax2 = ax1.twinx()
130
131     fig.suptitle("Comparaison des compressions en fonction du facteur de qualité")
132
133     width = 0.25
134
135     initialSize = 786486
136     xaxis = list(qualitySize.keys())
137     yaxisSize = np.array(list(qualitySize.values()))
138     yaxisTime = np.array(list(qualityTime.values()))
139     yaxisRatio = (initialSize - yaxisSize) / yaxisTime
140
141     color1 = "tab:red"
142     color2 = "tab:blue"
143     color3 = "tab:green"
144
145     ax1.bar(
146         np.arange(len(qualitySize)) - width,
147         yaxisSize,
148         width,
149         tick_label=xaxis,
150         color=color1,
151         label="Taille après compression",
152     )
153     ax2.bar(
154         np.arange(len(qualityTime)),
155         yaxisTime,
156         width,
157         tick_label=xaxis,
158         color=color2,
159         label="Temps de compression",
160     )

```

```

161     ax3.bar(
162         np.arange(len(qualitySize)),
163         yaxisRatio,
164         width,
165         tick_label=xaxis,
166         color=color3,
167         label="Octets gagnés par seconde",
168     )
169
170     ax1.legend(loc="upper left")
171     ax2.legend(loc="upper left", bbox_to_anchor=(0, 0.9))
172     ax3.legend(loc="upper right")
173
174     ax3.yaxis.tick_right()
175
176     ax1.set_xlabel("Facteur de qualité")
177     ax1.set_ylabel("Taille (en octets)", color=color1)
178     ax2.set_ylabel("Temps (en secondes)", color=color2)
179     ax3.set_xlabel("Facteur de qualité")
180     ax3.set_ylabel("Taille gagné par unité de temps (octets.Hz)", color=color3)
181     ax3.yaxis.set_label_position("right")
182
183     plt.savefig("./data/treated/compressionComparaison", transparent=True)
184
185     plt.rcParams["figure.figsize"] = [7, 5]
186     plt.clf()
187
188     fig = plt.figure()
189     ax1 = fig.add_subplot(111)
190     ax2 = ax1.twinx()
191
192     fig.suptitle(
193         "Comparaison des compressions en fonction des tables de Huffman utilisées"
194     )
195
196     yaxisSize = [stdSize, nonStdSize]
197     yaxisTime = [stdTime, nonStdTime]
198
199     labels = ["Tables Standards", "Tables Optimales"]
200     ax1.bar(
201         np.arange(2) - width / 2,
202         yaxisSize,
203         width,
204         tick_label=labels,
205         color=color1,
206         label="Taille après compression",
207     )
208     ax2.bar(
209         np.arange(2) + width / 2,
210         yaxisTime,
211         width,
212         tick_label=labels,
213         color=color2,
214         label="Temps de compression",
215     )
216
217     ax1.legend(loc="upper center", bbox_to_anchor=(0.45, 1))
218     ax2.legend(loc="upper center", bbox_to_anchor=(0.45, 0.9))

```

```

219 ax1.set_ylabel("Taille (en octets)", color=color1)
220 ax2.set_ylabel("Temps (en secondes)", color=color2)
221
222
223 plt.savefig("./data/treated/compressionComparaison2", transparent=True)
224
225
226 def energyCompaction(imgPath):
227     img = cv2.imread(imgPath)
228
229     imgYCrCb = cv2.cvtColor(
230         img, cv2.COLOR_RGB2YCrCb
231     ) # Convert RGB to YCrCb (Cb applies V, and Cr applies U).
232
233     Y, Cr, Cb = cv2.split(padding(imgYCrCb, 8, 8))
234     Y = Y.astype("int") - 128
235     blocks_Y = utils.divide_blocks(Y, 8, 8)
236     dctBlocks_Y = np.zeros_like(blocks_Y)
237     for i in range(len(blocks_Y)):
238         dctBlocks_Y[i] = dct(
239             dct(blocks_Y[i], axis=0, norm="ortho"), axis=1, norm="ortho"
240         )
241     avg_Y = utils.averageMatrix(blocks_Y)
242     avgDct_Y = utils.averageMatrix(dctBlocks_Y)
243
244     x = np.random.randint(blocks_Y.shape[0])
245     arr1 = blocks_Y[x]
246     arr2 = dctBlocks_Y[x]
247
248     fig, (ax1, ax2) = plt.subplots(1, 2)
249
250     valueMax, valueMin = max(np.max(arr1), np.max(arr2)), min(
251         np.min(arr1), np.min(arr2)
252     )
253     # fig.suptitle('Matrice de la luminance de "villeLyon.jpg"')
254
255     ax1.matshow(arr1, cmap="cool", vmin=valueMin, vmax=valueMax)
256     ax1.set_title("avant DCT")
257
258     ax2.matshow(arr2, cmap="cool", vmin=valueMin, vmax=valueMax)
259     ax2.set_title("après DCT")
260
261     for i in range(arr1.shape[0]):
262         for j in range(arr1.shape[1]):
263             cNormal = int(arr1[i, j])
264             cDct = int(arr2[i, j])
265             ax1.text(i, j, str(cNormal), va="center", ha="center")
266             ax2.text(i, j, str(cDct), va="center", ha="center")
267     plt.savefig("./data/energyCompaction.png", transparent=True)
268
269
270 def rgbToYCbCr_channel_bis():
271     img = cv2.imread("./data/villeLyon.jpg") # Read input image in BGR format
272
273     imgYCrCb = cv2.cvtColor(
274         img, cv2.COLOR_BGR2YCrCb
275     ) # Convert RGB to YCrCb (Cb applies V, and Cr applies U).
276

```

```

277 Y, Cr, Cb = cv2.split(imgYCrCb)
278
279 # Fill Y and Cb with 128 (Y level is middle gray, and Cb is "neutralized").
280 onlyCr = imgYCrCb.copy()
281 onlyCr[:, :, 0] = 128
282 onlyCr[:, :, 2] = 128
283 onlyCr_as_bgr = cv2.cvtColor(
284     onlyCr, cv2.COLOR_YCrCb2BGR
285 ) # Convert to BGR - used for display as false color
286
287 # Fill Y and Cr with 128 (Y level is middle gray, and Cr is "neutralized").
288 onlyCb = imgYCrCb.copy()
289 onlyCb[:, :, 0] = 128
290 onlyCb[:, :, 1] = 128
291 onlyCb_as_bgr = cv2.cvtColor(
292     onlyCb, cv2.COLOR_YCrCb2BGR
293 ) # Convert to BGR - used for display as false color
294
295 cv2.imshow("img", img)
296 cv2.imshow("Y", Y)
297 cv2.imshow("onlyCb_as_bgr", onlyCb_as_bgr)
298 cv2.imshow("onlyCr_as_bgr", onlyCr_as_bgr)
299 cv2.waitKey()
300 cv2.destroyAllWindows()
301
302 cv2.imwrite("./data/treated/villeLyon_Y.jpg", Y)
303 cv2.imwrite("./data/treated/villeLyon_Cb.jpg", onlyCb_as_bgr)
304 cv2.imwrite("./data/treated/villeLyon_Cr.jpg", onlyCr_as_bgr)
305
306
307 if __name__ == "__main__":
308     # compare()
309     # rgbToYCbCr_channel_bis()
310     # energyCompaction("./data/villeLyon.jpg")
311     # test = np.array([[93, 90, 83, 68, 61, 61, 46, 21],
312     #                  [102, 92, 95, 77, 65, 60, 49, 32],
313     #                  [69, 55, 47, 57, 65, 60, 72, 65],
314     #                  [55, 55, 40, 42, 23, 1, 11, 38],
315     #                  [55, 57, 47, 53, 35, 59, -2, 26],
316     #                  [64, 41, 42, 55, 60, 57, 25, -8],
317     #                  [77, 87, 58, -2, -5, 14, -10, -35],
318     #                  [38, 14, 33, 33, -21, -23, -43, -34]])
319     # print(dct(dct(test, axis=0, norm="ortho"), axis=1, norm='ortho'))
320
321     # stat = compare(qualities = list(range(1, 101, 10)), subsamples=['4:4:4',
322     # '4:2:0', '4:1:1', '4:2:2'], useStdHuffmanTable=[True, False],
323     DeleteFilesAfterward=True)
324     # write_stat_csv("./data/treated/stat.csv", stat)
325     stat = csv_to_stat("./data/treated/stat.csv")
326     dataInterpreation(stat)

```