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):
        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):
        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))


def bytes2int(bytes_, byteorder='big'):
    return int.from_bytes(bytes_, byteorder)


```

```python
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,  11,  12,  14,  12,  10,  16,  14,
            13,  14,  18,  17,  16,  19,  24,  40,
            26,  24,  22,  22,  24,  49,  35,  37,
            29,  40,  58,  51,  61,  60,  57,  51,
            56,  55,  64,  72,  92,  78,  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, 26,
            26, 47, 99, 66, 56, 66, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99,
            99, 99, 99, 99, 99, 99, 99, 99], dtype=np.int32)
    else:
        raise ValueError((
            f"component should be either 'lum' or 'chr', "
            f"but '{component}' was found."))
    if 0 < quality < 50:
        q = np.minimum(np.floor(50/quality * q + 0.5), 255)
    elif 50 <= quality <= 100:
        q = np.maximum(np.floor((2 - quality/50) * q + 0.5), 1)
    else:
        raise ValueError("quality should belong to (0, 100].")
    return q.astype(np.int32)


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

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

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


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


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


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


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

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

def main():
    pass

```

```python

if __name__ == '__main__':
    main()
    arrMatrix = np.array([[[1, 2],
                           [3, 4]],
                           [[5, 2],
                           [3, 4]]])
    print(averageMatrix(arrMatrix))
```