ELL 786 Report Assignment-2

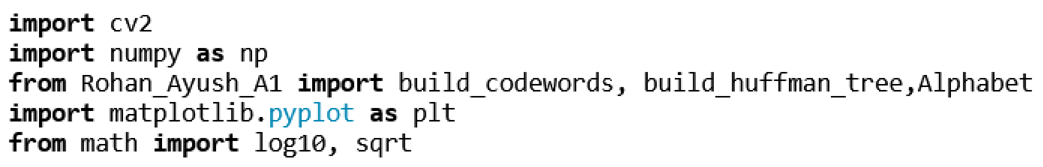
Rohan Mahala 2019MT60760

Ayush Verma 2019MT60749

<u>Video Encoding</u>

<u>Part-1</u> :-

- Importing all the valid functionalities to be used later.
- Also importing some functions from “Rohan_Ayush_A1”, performed in assignment 1 for Huffman encoding.

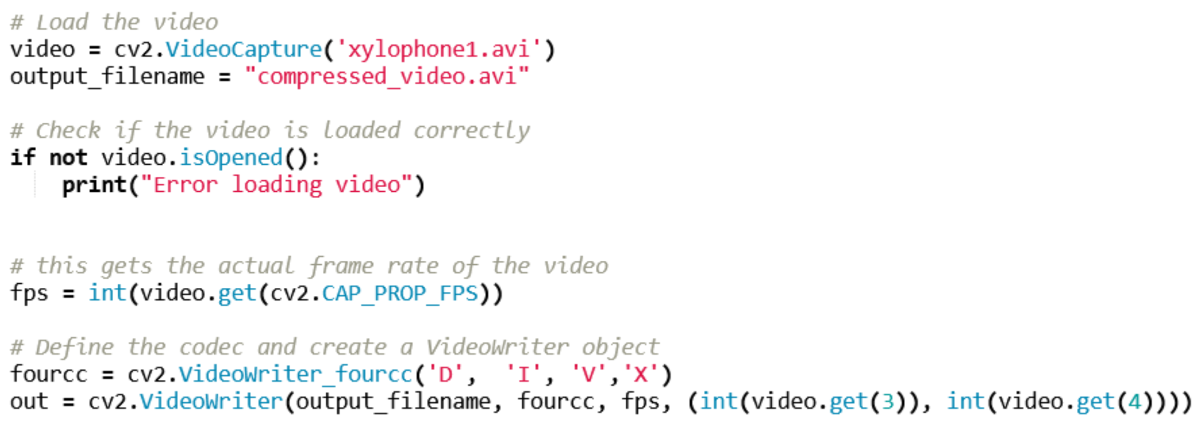


```
import cv2
import numpy as np
from Rohan_Ayush_A1 import build_codewords, build_huffman_tree,Alphabet
import matplotlib.pyplot as plt
from math import log10, sqrt
```

Figure 1 :- Source Code

Part-2 :-

- We open the input avi video file xylophone1 using the VideoCapture function provided by cv2.
- Checking if video is properly loaded.
- Determining the fps of the input video file.
- Naming the output filename as “compressed_video.avi”.
- Opening the output file to write the compressed video on it.

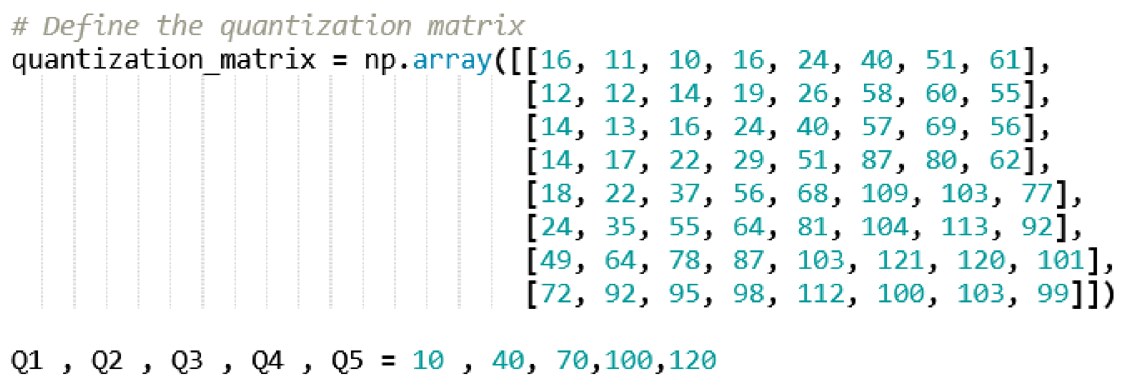


```python
# Load the video
video = cv2.VideoCapture('xylophone1.avi')
output_filename = "compressed_video.avi"

# Check if the video is loaded correctly
if not video.isOpened():
    print("Error loading video")


# this gets the actual frame rate of the video
fps = int(video.get(cv2.CAP_PROP_FPS))

# Define the codec and create a VideoWriter object
fourcc = cv2.VideoWriter_fourcc('D',  'I', 'V','X')
out = cv2.VideoWriter(output_filename, fourcc, fps, (int(video.get(3)), int(video.get(4))))
```

Figure 2 :- Source Code

Part-3 :-

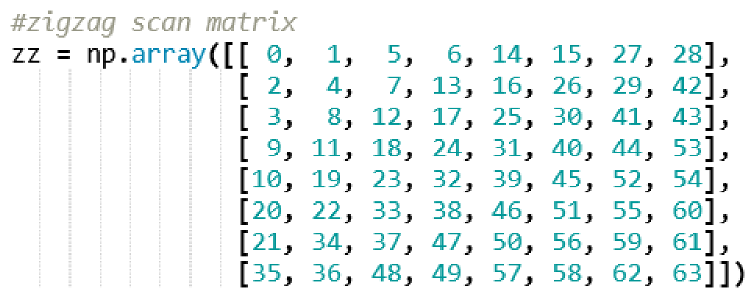
- Defining a quant_matrix that is used in quantization. Quantization is to reduce a range of numbers to a single small value, so we can use less bits to represent a large number.
- We also use 5 quantization factors , namely Q1, Q2, Q3, Q4& Q5, that will be used for generating the PNSR vs bitrate plots.
- After DCT, most of the energy(value) is concentrated to the top-left corner. After quantizing the transformed matrix, most data in this matrix may be zero.
- This is an important step to achieve compression.
- Due to quantization, coefficients are changed and hence it is a lossy compression.



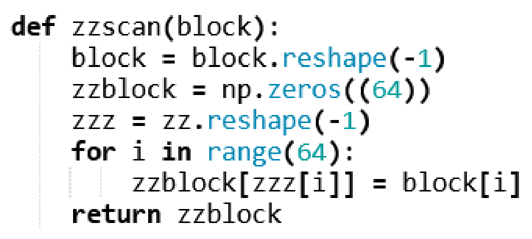
```
# Define the quantization matrix
quantization_matrix = np.array([[16, 11, 10, 16, 24, 40, 51, 61],
                                [12, 12, 14, 19, 26, 58, 60, 55],
                                [14, 13, 16, 24, 40, 57, 69, 56],
                                [14, 17, 22, 29, 51, 87, 80, 62],
                                [18, 22, 37, 56, 68, 109, 103, 77],
                                [24, 35, 55, 64, 81, 104, 113, 92],
                                [49, 64, 78, 87, 103, 121, 120, 101],
                                [72, 92, 95, 98, 112, 100, 103, 99]])

Q1 , Q2 , Q3 , Q4 , Q5 = 10 , 40, 70,100,120
```

<u>Part-4</u> :-

- Defining a zz matrix that is used in zigzag scanning. The value at any index in zz matrix indicates the iteration in which that index will be visited during the zigzag scan.
- zzscan function takes a block as input and returns the zigzag scan of the block by using the zz matrix.
- We use this zigzag function after DCT and quantization, when most AC values will be zero. By using zig-zag scan we can gather even more consecutive zeros.



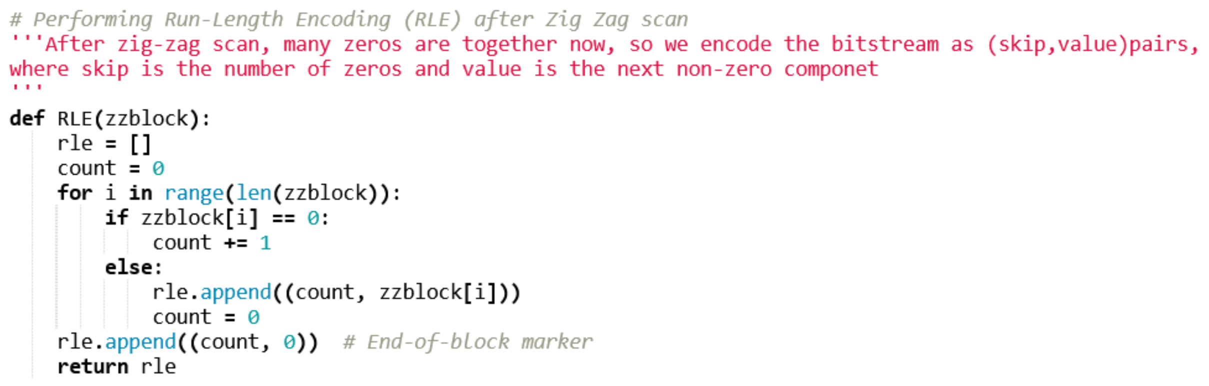
```
#zigzag scan matrix
zz = np.array([[ 0,  1,  5,  6, 14, 15, 27, 28],
               [ 2,  4,  7, 13, 16, 26, 29, 42],
               [ 3,  8, 12, 17, 25, 30, 41, 43],
               [ 9, 11, 18, 24, 31, 40, 44, 53],
               [10, 19, 23, 32, 39, 45, 52, 54],
               [20, 22, 33, 38, 46, 51, 55, 60],
               [21, 34, 37, 47, 50, 56, 59, 61],
               [35, 36, 48, 49, 57, 58, 62, 63]])
```



```
def zzscan(block):
    block = block.reshape(-1)
    zzblock = np.zeros((64))
    zzz = zz.reshape(-1)
    for i in range(64):
        zzblock[zzz[i]] = block[i]
    return zzblock
```

<u>Part-5</u> :-

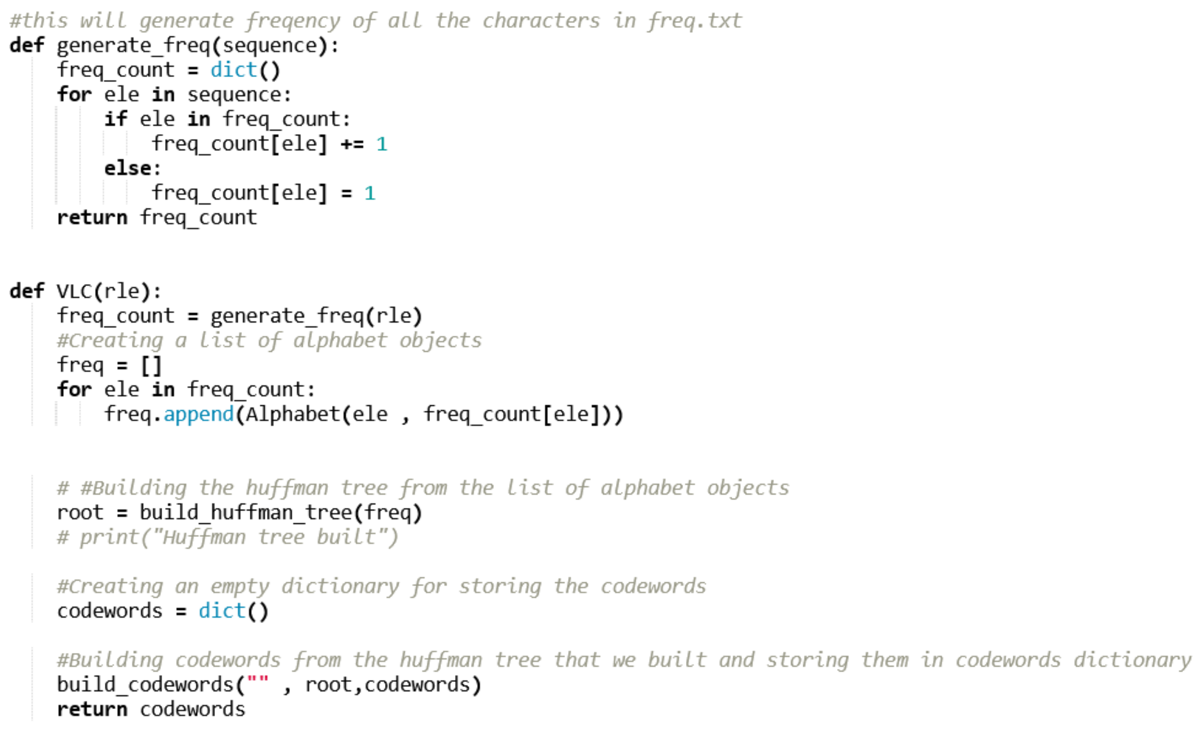
- After zig-zag scan, many zeros are together now, so we encode the bitstream as (skip, value) pairs, where skip is the number of zeros and value is the next non-zero component.
- Run Length Encoding(RLE) is used to achieve even greater compression ratio.



```python
# Performing Run-Length Encoding (RLE) after Zig Zag scan
'''After zig-zag scan, many zeros are together now, so we encode the bitstream as (skip,value)pairs,
where skip is the number of zeros and value is the next non-zero componet
'''
def RLE(zzblock):
    rle = []
    count = 0
    for i in range(len(zzblock)):
        if zzblock[i] == 0:
            count += 1
        else:
            rle.append((count, zzblock[i]))
            count = 0
    rle.append((count, 0))  # End-of-block marker
    return rle
```

Part-6 :-

- The run-length value, and the value of the non-zero coefficient which the run of zero coefficients precedes, i.e. (skip, value) pairs, are then combined and coded using a variable-length code (VLC).
- The VLC exploits the fact that short runs of zeros are more likely than long ones, and small coefficients are more likely than large ones.
- Huffman code is used as the variable-length code.



```python
#this will generate freqency of all the characters in freq.txt
def generate_freq(sequence):
    freq_count = dict()
    for ele in sequence:
        if ele in freq_count:
            freq_count[ele] += 1
        else:
            freq_count[ele] = 1
    return freq_count


def VLC(rle):
    freq_count = generate_freq(rle)
    #Creating a list of alphabet objects
    freq = []
    for ele in freq_count:
        freq.append(Alphabet(ele , freq_count[ele]))


    # #Building the huffman tree from the list of alphabet objects
    root = build_huffman_tree(freq)
    # print("Huffman tree built")

    #Creating an empty dictionary for storing the codewords
    codewords = dict()

    #Building codewords from the huffman tree that we built and storing them in codewords dictionary
    build_codewords("" , root,codewords)
    return codewords
```

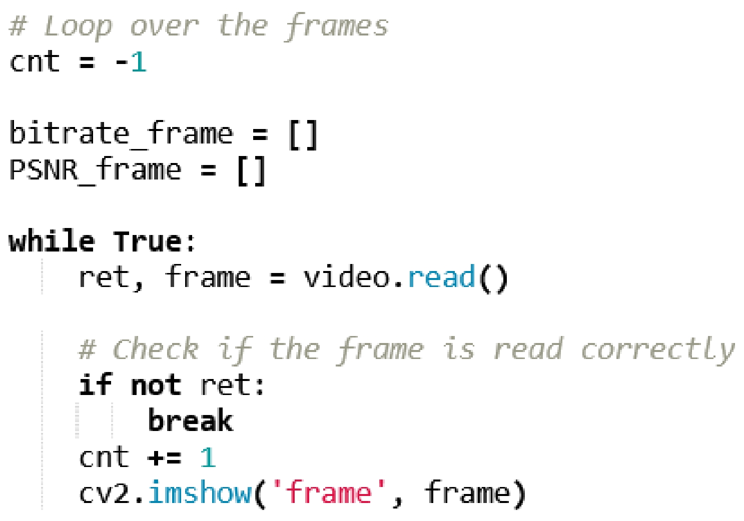
Part-7 :-

- The PSNR block computes the peak signal-to-noise ratio, in decibels, between two images.
- This ratio is used as a quality measurement between the original and a compressed image.
- The higher the PSNR, the better the quality of the compressed, or reconstructed image.
- Bitrate is the amount of data used to represent a unit of time in a digital audio or video file. It is usually measured in bits per second (bps) or kilobits per second (kbps), and it determines the quality and size of the audio or video file.

```
def PSNR(original, compressed):
    mse = np.mean((original - compressed) ** 2)
    if(mse == 0):  # MSE is zero means no noise is present in the signal .
                  # Therefore PSNR have no importance.
        return 100
    max_pixel = 255.0
    psnr = 20 * log10(max_pixel / sqrt(mse))
    return psnr

def calculate_bitrate(compression_ratio):
    bitrate = (int(video.get(3)) * int(video.get(4))) * 3 * 8 / compression_ratio
    return bitrate
```

<u>Part-8</u> :-

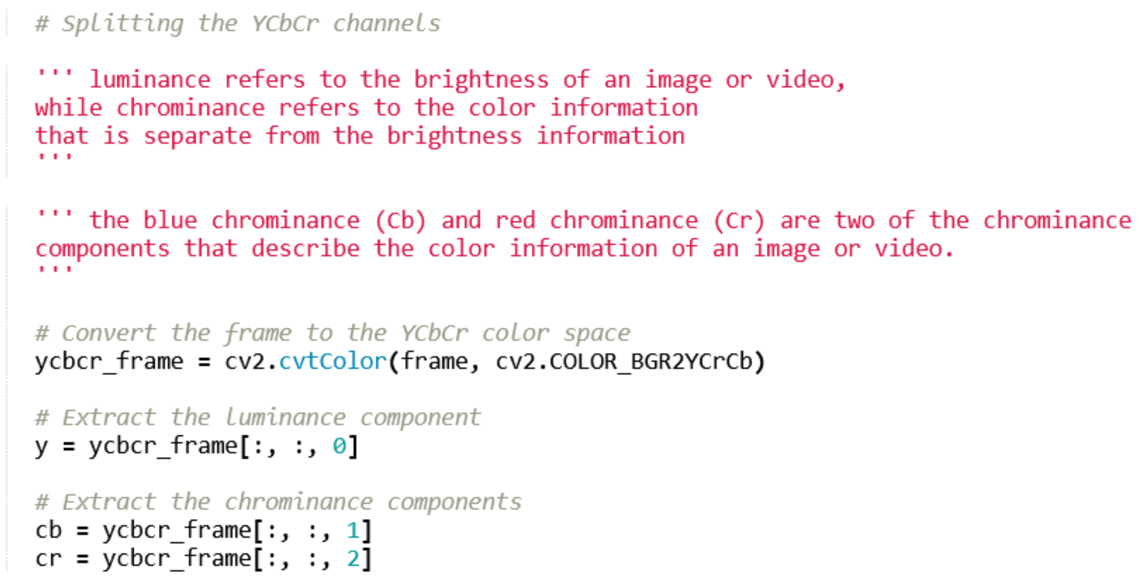
- Initializing certain parameters before starting the loop.
- “cnt” the frame count initialized to -1. The first frame will be “frame 0”, next “frame 1” and so on.
- Starting the while loop. This loop is over all the frames in the video file.
- Once in loop, we read the next frame, check if the read is successful and if Yes then increment the “cnt” counter.
- Displaying the frame read by imshow function of cv2.



```python
# Loop over the frames
cnt = -1

bitrate_frame = []
PSNR_frame = []

while True:
    ret, frame = video.read()

    # Check if the frame is read correctly
    if not ret:
        break
    cnt += 1
    cv2.imshow('frame', frame)
```

<u>Part-9</u> :-

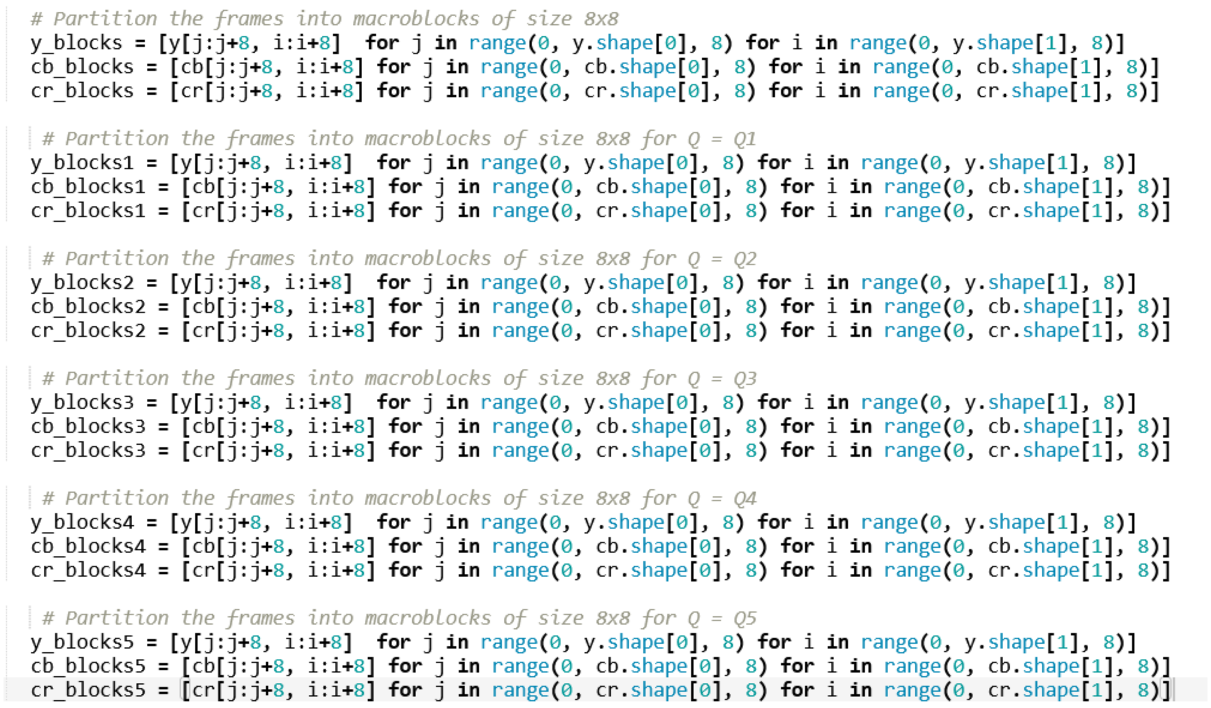
- Next, we split the frame into YCbCr channels.
- Luminance refers to the brightness of an image or video, while chrominance refers to the color information that is separate from the brightness information.
- The blue chrominance (Cb) and red chrominance (Cr) are two of the chrominance components that describe the color information of an image or video.
- After converting the frame to YCbCr color space, we extract the luminance, red chrominance and blue chrominance components.



```
# Splitting the YCbCr channels

''' luminance refers to the brightness of an image or video,
while chrominance refers to the color information
that is separate from the brightness information
'''

''' the blue chrominance (Cb) and red chrominance (Cr) are two of the chrominance
components that describe the color information of an image or video.
'''

# Convert the frame to the YCbCr color space
ycbcr_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2YCrCb)

# Extract the luminance component
y = ycbcr_frame[:, :, 0]

# Extract the chrominance components
cb = ycbcr_frame[:, :, 1]
cr = ycbcr_frame[:, :, 2]
```

<u>Part-10</u> :-

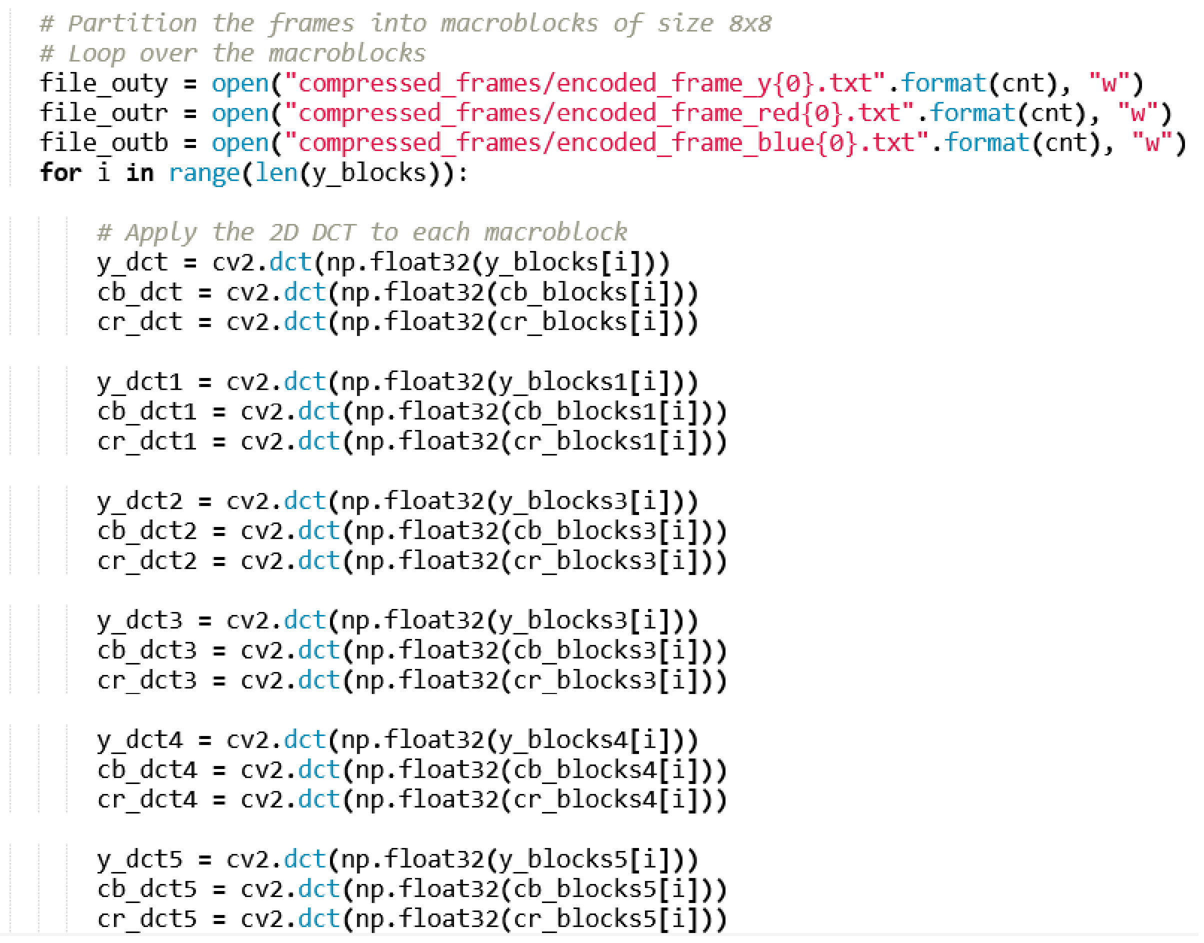
- Next, we partition the current frame into macroblocks of size 8x8.
- Creating multiple copies as we use multiple quantization schemes.



```
# Partition the frames into macroblocks of size 8x8
y_blocks = [y[j:j+8, i:i+8]  for j in range(0, y.shape[0], 8) for i in range(0, y.shape[1], 8)]
cb_blocks = [cb[j:j+8, i:i+8] for j in range(0, cb.shape[0], 8) for i in range(0, cb.shape[1], 8)]
cr_blocks = [cr[j:j+8, i:i+8] for j in range(0, cr.shape[0], 8) for i in range(0, cr.shape[1], 8)]

 # Partition the frames into macroblocks of size 8x8 for Q = Q1
y_blocks1 = [y[j:j+8, i:i+8]  for j in range(0, y.shape[0], 8) for i in range(0, y.shape[1], 8)]
cb_blocks1 = [cb[j:j+8, i:i+8] for j in range(0, cb.shape[0], 8) for i in range(0, cb.shape[1], 8)]
cr_blocks1 = [cr[j:j+8, i:i+8] for j in range(0, cr.shape[0], 8) for i in range(0, cr.shape[1], 8)]

 # Partition the frames into macroblocks of size 8x8 for Q = Q2
y_blocks2 = [y[j:j+8, i:i+8]  for j in range(0, y.shape[0], 8) for i in range(0, y.shape[1], 8)]
cb_blocks2 = [cb[j:j+8, i:i+8] for j in range(0, cb.shape[0], 8) for i in range(0, cb.shape[1], 8)]
cr_blocks2 = [cr[j:j+8, i:i+8] for j in range(0, cr.shape[0], 8) for i in range(0, cr.shape[1], 8)]

 # Partition the frames into macroblocks of size 8x8 for Q = Q3
y_blocks3 = [y[j:j+8, i:i+8]  for j in range(0, y.shape[0], 8) for i in range(0, y.shape[1], 8)]
cb_blocks3 = [cb[j:j+8, i:i+8] for j in range(0, cb.shape[0], 8) for i in range(0, cb.shape[1], 8)]
cr_blocks3 = [cr[j:j+8, i:i+8] for j in range(0, cr.shape[0], 8) for i in range(0, cr.shape[1], 8)]

 # Partition the frames into macroblocks of size 8x8 for Q = Q4
y_blocks4 = [y[j:j+8, i:i+8]  for j in range(0, y.shape[0], 8) for i in range(0, y.shape[1], 8)]
cb_blocks4 = [cb[j:j+8, i:i+8] for j in range(0, cb.shape[0], 8) for i in range(0, cb.shape[1], 8)]
cr_blocks4 = [cr[j:j+8, i:i+8] for j in range(0, cr.shape[0], 8) for i in range(0, cr.shape[1], 8)]

 # Partition the frames into macroblocks of size 8x8 for Q = Q5
y_blocks5 = [y[j:j+8, i:i+8]  for j in range(0, y.shape[0], 8) for i in range(0, y.shape[1], 8)]
cb_blocks5 = [cb[j:j+8, i:i+8] for j in range(0, cb.shape[0], 8) for i in range(0, cb.shape[1], 8)]
cr_blocks5 = [cr[j:j+8, i:i+8] for j in range(0, cr.shape[0], 8) for i in range(0, cr.shape[1], 8)]
```

Part-11 :-

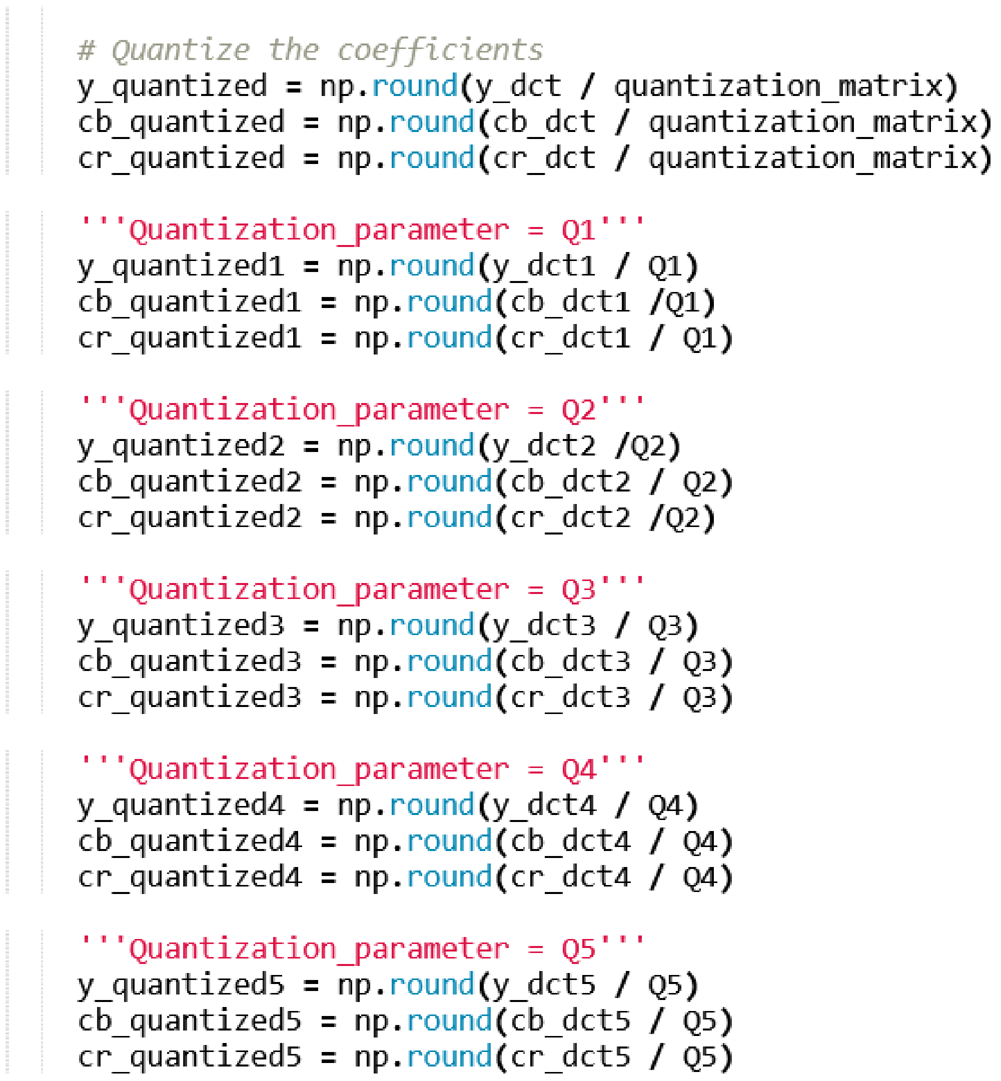
- Opening the file to write encoding of the compressed frames into the “compressed_frames” folder.
- Looping over all the macroblocks created for the current frame.
- We apply the Discrete Cosine Transform(DCT) to each of the macroblocks.



```
# Partition the frames into macroblocks of size 8x8
# Loop over the macroblocks
file_outy = open("compressed_frames/encoded_frame_y{0}.txt".format(cnt), "w")
file_outr = open("compressed_frames/encoded_frame_red{0}.txt".format(cnt), "w")
file_outb = open("compressed_frames/encoded_frame_blue{0}.txt".format(cnt), "w")
for i in range(len(y_blocks)):

    # Apply the 2D DCT to each macroblock
    y_dct = cv2.dct(np.float32(y_blocks[i]))
    cb_dct = cv2.dct(np.float32(cb_blocks[i]))
    cr_dct = cv2.dct(np.float32(cr_blocks[i]))

    y_dct1 = cv2.dct(np.float32(y_blocks1[i]))
    cb_dct1 = cv2.dct(np.float32(cb_blocks1[i]))
    cr_dct1 = cv2.dct(np.float32(cr_blocks1[i]))

    y_dct2 = cv2.dct(np.float32(y_blocks3[i]))
    cb_dct2 = cv2.dct(np.float32(cb_blocks3[i]))
    cr_dct2 = cv2.dct(np.float32(cr_blocks3[i]))

    y_dct3 = cv2.dct(np.float32(y_blocks3[i]))
    cb_dct3 = cv2.dct(np.float32(cb_blocks3[i]))
    cr_dct3 = cv2.dct(np.float32(cr_blocks3[i]))

    y_dct4 = cv2.dct(np.float32(y_blocks4[i]))
    cb_dct4 = cv2.dct(np.float32(cb_blocks4[i]))
    cr_dct4 = cv2.dct(np.float32(cr_blocks4[i]))

    y_dct5 = cv2.dct(np.float32(y_blocks5[i]))
    cb_dct5 = cv2.dct(np.float32(cb_blocks5[i]))
    cr_dct5 = cv2.dct(np.float32(cr_blocks5[i]))
```

Part-12 :-

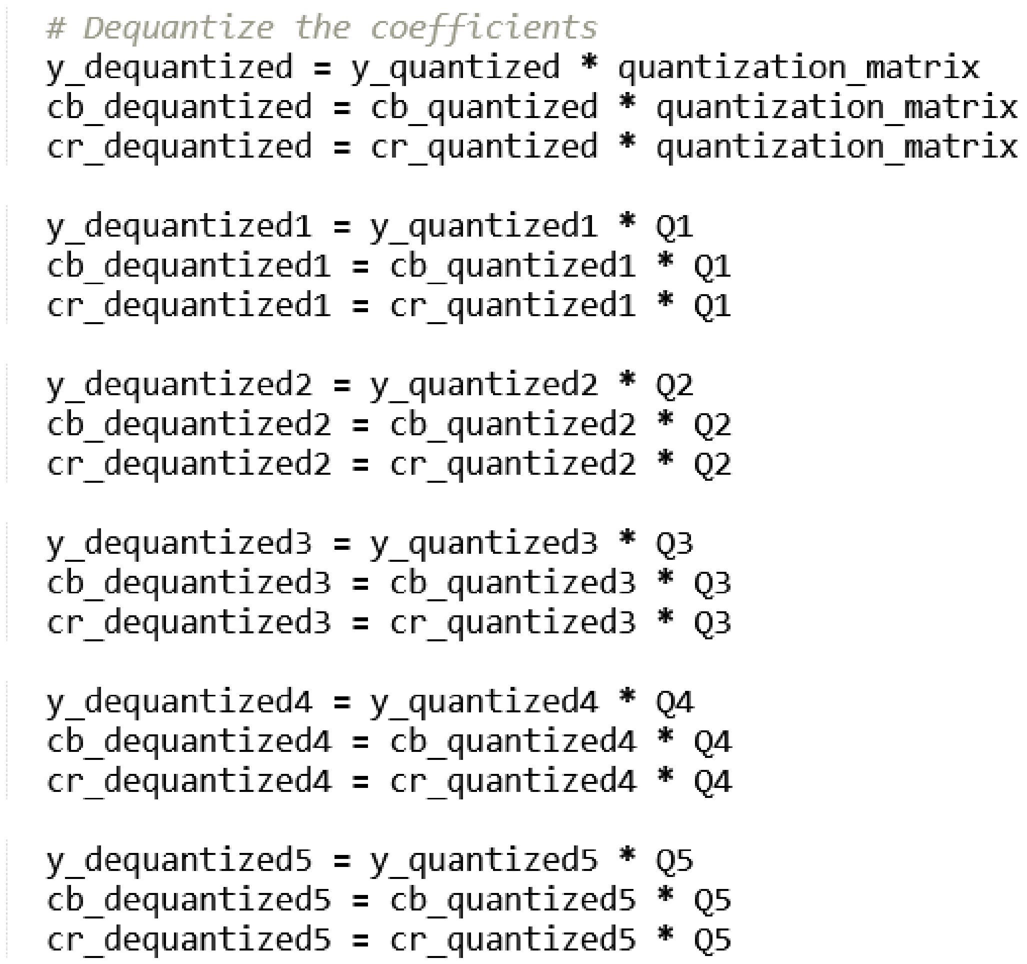
- Quantizing the coefficients by dividing by the quant_matrix and the quantization factors introduced in Part-3. This is done to achieve compression.
- This makes the elements other than the top-left corner of the macroblock zero.



```python
# Quantize the coefficients
y_quantized = np.round(y_dct / quantization_matrix)
cb_quantized = np.round(cb_dct / quantization_matrix)
cr_quantized = np.round(cr_dct / quantization_matrix)

'''Quantization_parameter = Q1'''
y_quantized1 = np.round(y_dct1 / Q1)
cb_quantized1 = np.round(cb_dct1 /Q1)
cr_quantized1 = np.round(cr_dct1 / Q1)

'''Quantization_parameter = Q2'''
y_quantized2 = np.round(y_dct2 /Q2)
cb_quantized2 = np.round(cb_dct2 / Q2)
cr_quantized2 = np.round(cr_dct2 /Q2)

'''Quantization_parameter = Q3'''
y_quantized3 = np.round(y_dct3 / Q3)
cb_quantized3 = np.round(cb_dct3 / Q3)
cr_quantized3 = np.round(cr_dct3 / Q3)

'''Quantization_parameter = Q4'''
y_quantized4 = np.round(y_dct4 / Q4)
cb_quantized4 = np.round(cb_dct4 / Q4)
cr_quantized4 = np.round(cr_dct4 / Q4)

'''Quantization_parameter = Q5'''
y_quantized5 = np.round(y_dct5 / Q5)
cb_quantized5 = np.round(cb_dct5 / Q5)
cr_quantized5 = np.round(cr_dct5 / Q5)
```

<u>Part-13</u> :-

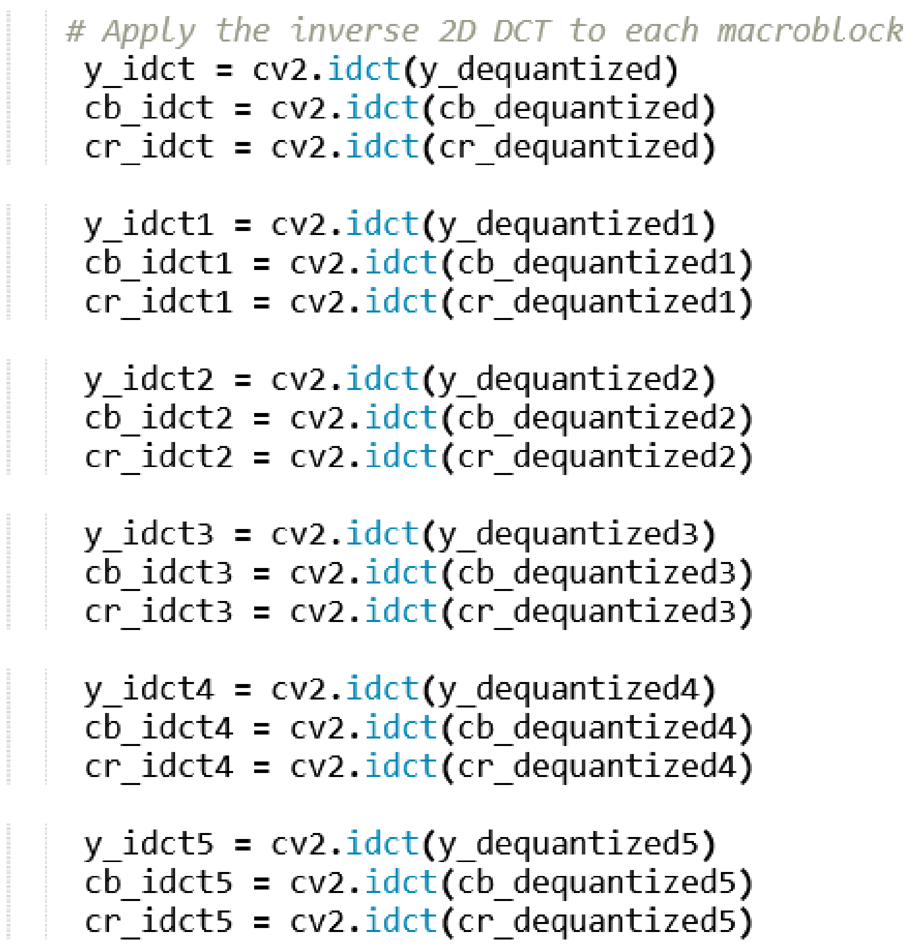
- Dequantizing by multiplying by the quant_matrix and the quantization factors for reconstruction.



```
# Dequantize the coefficients
y_dequantized = y_quantized * quantization_matrix
cb_dequantized = cb_quantized * quantization_matrix
cr_dequantized = cr_quantized * quantization_matrix

y_dequantized1 = y_quantized1 * Q1
cb_dequantized1 = cb_quantized1 * Q1
cr_dequantized1 = cr_quantized1 * Q1

y_dequantized2 = y_quantized2 * Q2
cb_dequantized2 = cb_quantized2 * Q2
cr_dequantized2 = cr_quantized2 * Q2

y_dequantized3 = y_quantized3 * Q3
cb_dequantized3 = cb_quantized3 * Q3
cr_dequantized3 = cr_quantized3 * Q3

y_dequantized4 = y_quantized4 * Q4
cb_dequantized4 = cb_quantized4 * Q4
cr_dequantized4 = cr_quantized4 * Q4

y_dequantized5 = y_quantized5 * Q5
cb_dequantized5 = cb_quantized5 * Q5
cr_dequantized5 = cr_quantized5 * Q5
```

Part-14 :-

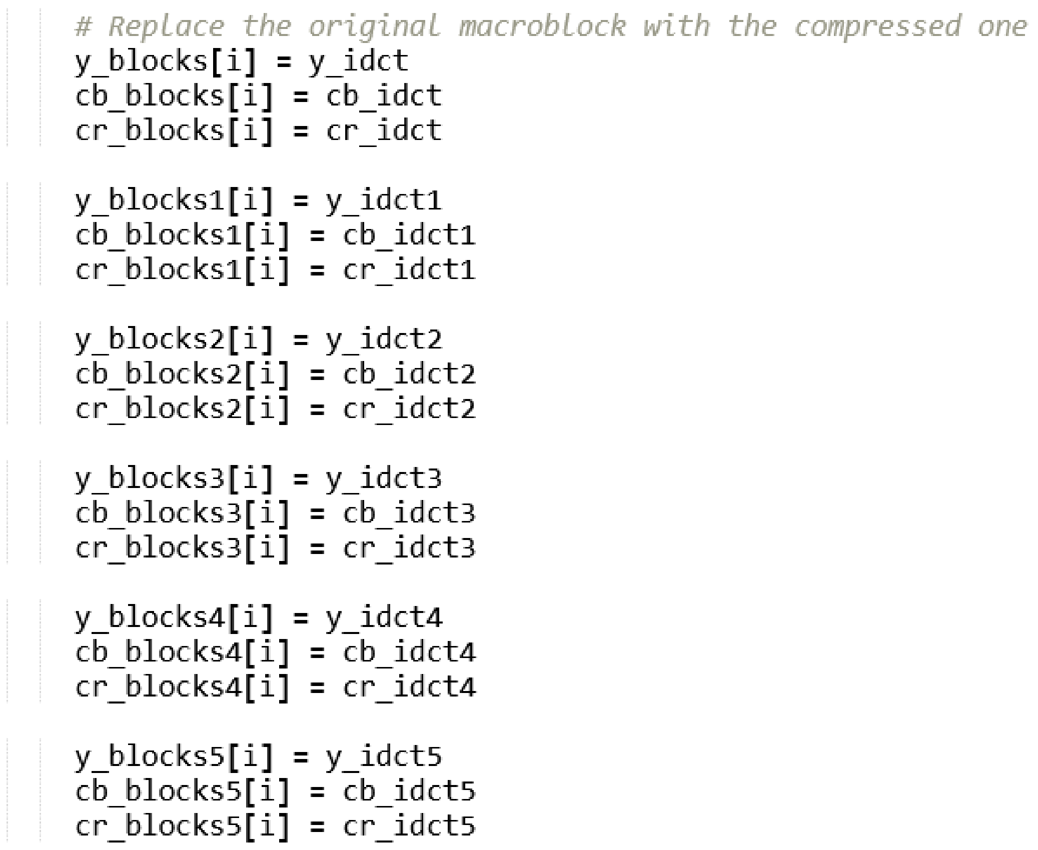
- Applying the inverse Discrete Cosine Transformation (DCT) to reconstruct the original frame macroblocks.



```
# Apply the inverse 2D DCT to each macroblock
 y_idct = cv2.idct(y_dequantized)
 cb_idct = cv2.idct(cb_dequantized)
 cr_idct = cv2.idct(cr_dequantized)

 y_idct1 = cv2.idct(y_dequantized1)
 cb_idct1 = cv2.idct(cb_dequantized1)
 cr_idct1 = cv2.idct(cr_dequantized1)

 y_idct2 = cv2.idct(y_dequantized2)
 cb_idct2 = cv2.idct(cb_dequantized2)
 cr_idct2 = cv2.idct(cr_dequantized2)

 y_idct3 = cv2.idct(y_dequantized3)
 cb_idct3 = cv2.idct(cb_dequantized3)
 cr_idct3 = cv2.idct(cr_dequantized3)

 y_idct4 = cv2.idct(y_dequantized4)
 cb_idct4 = cv2.idct(cb_dequantized4)
 cr_idct4 = cv2.idct(cr_dequantized4)

 y_idct5 = cv2.idct(y_dequantized5)
 cb_idct5 = cv2.idct(cb_dequantized5)
 cr_idct5 = cv2.idct(cr_dequantized5)
```

Part-15 :-

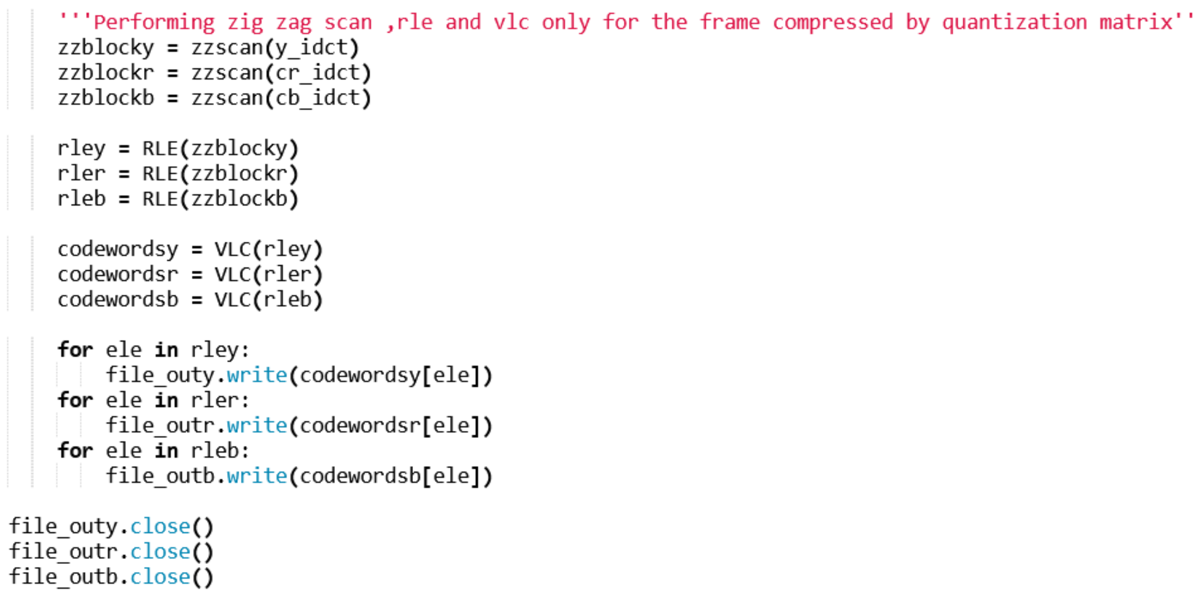
- Then replacing the original macroblocks with the reconstructed ones.



```
        # Replace the original macroblock with the compressed one
        y_blocks[i] = y_idct
        cb_blocks[i] = cb_idct
        cr_blocks[i] = cr_idct

        y_blocks1[i] = y_idct1
        cb_blocks1[i] = cb_idct1
        cr_blocks1[i] = cr_idct1

        y_blocks2[i] = y_idct2
        cb_blocks2[i] = cb_idct2
        cr_blocks2[i] = cr_idct2

        y_blocks3[i] = y_idct3
        cb_blocks3[i] = cb_idct3
        cr_blocks3[i] = cr_idct3

        y_blocks4[i] = y_idct4
        cb_blocks4[i] = cb_idct4
        cr_blocks4[i] = cr_idct4

        y_blocks5[i] = y_idct5
        cb_blocks5[i] = cb_idct5
        cr_blocks5[i] = cr_idct5
```

Part-16 :-

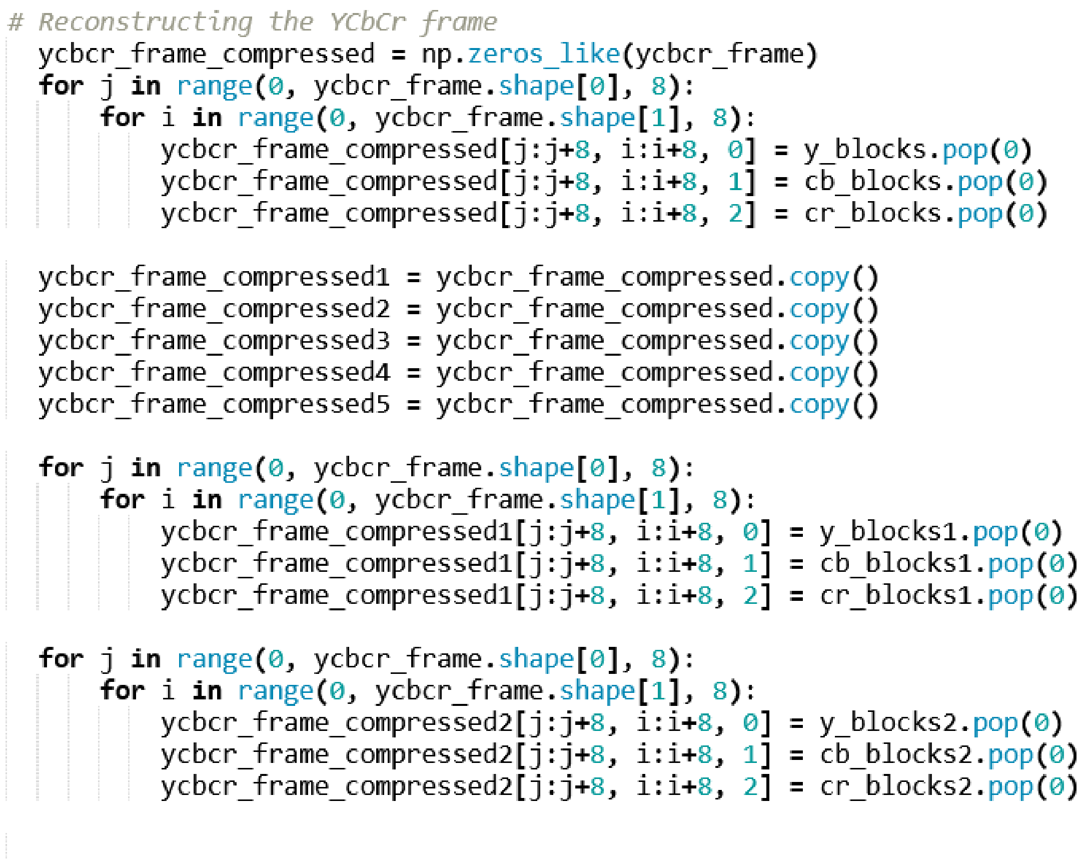
- Performing the zig zag scan on the reconstructed macroblock.
- Run length encoding the sequence obtained by zig zag scan.
- Encoding the RLE using variable-length encoding (Huffman) and generating and writing the codewords.
- After looping out of the, macroblocks of the current frame, we close the files opened for writing the encoding.



```
    '''Performing zig zag scan ,rle and vlc only for the frame compressed by quantization matrix'''
    zzblocky = zzscan(y_idct)
    zzblockr = zzscan(cr_idct)
    zzblockb = zzscan(cb_idct)

    rley = RLE(zzblocky)
    rler = RLE(zzblockr)
    rleb = RLE(zzblockb)

    codewordsy = VLC(rley)
    codewordsr = VLC(rler)
    codewordsb = VLC(rleb)

    for ele in rley:
        file_outy.write(codewordsy[ele])
    for ele in rler:
        file_outr.write(codewordsr[ele])
    for ele in rleb:
        file_outb.write(codewordsb[ele])

file_outy.close()
file_outr.close()
file_outb.close()
```

Part-17 :-

- Combining all the reconstructed macroblocks together.
- Then moving back from the YCbCr color space to BGR color space to finally obtain the compressed frame.
- Displaying the compressed frame.

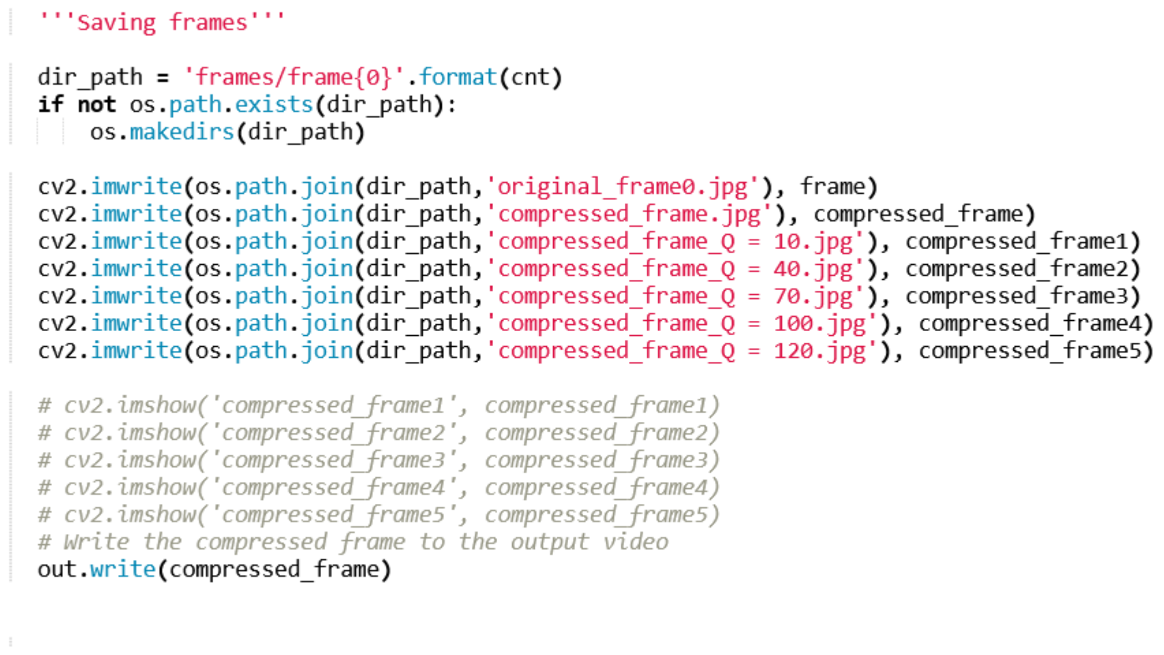


```python
# Reconstructing the YCbCr frame
ycbcr_frame_compressed = np.zeros_like(ycbcr_frame)
for j in range(0, ycbcr_frame.shape[0], 8):
    for i in range(0, ycbcr_frame.shape[1], 8):
        ycbcr_frame_compressed[j:j+8, i:i+8, 0] = y_blocks.pop(0)
        ycbcr_frame_compressed[j:j+8, i:i+8, 1] = cb_blocks.pop(0)
        ycbcr_frame_compressed[j:j+8, i:i+8, 2] = cr_blocks.pop(0)

ycbcr_frame_compressed1 = ycbcr_frame_compressed.copy()
ycbcr_frame_compressed2 = ycbcr_frame_compressed.copy()
ycbcr_frame_compressed3 = ycbcr_frame_compressed.copy()
ycbcr_frame_compressed4 = ycbcr_frame_compressed.copy()
ycbcr_frame_compressed5 = ycbcr_frame_compressed.copy()

for j in range(0, ycbcr_frame.shape[0], 8):
    for i in range(0, ycbcr_frame.shape[1], 8):
        ycbcr_frame_compressed1[j:j+8, i:i+8, 0] = y_blocks1.pop(0)
        ycbcr_frame_compressed1[j:j+8, i:i+8, 1] = cb_blocks1.pop(0)
        ycbcr_frame_compressed1[j:j+8, i:i+8, 2] = cr_blocks1.pop(0)

for j in range(0, ycbcr_frame.shape[0], 8):
    for i in range(0, ycbcr_frame.shape[1], 8):
        ycbcr_frame_compressed2[j:j+8, i:i+8, 0] = y_blocks2.pop(0)
        ycbcr_frame_compressed2[j:j+8, i:i+8, 1] = cb_blocks2.pop(0)
        ycbcr_frame_compressed2[j:j+8, i:i+8, 2] = cr_blocks2.pop(0)
```

```python
for j in range(0, ycbcr_frame.shape[0], 8):
    for i in range(0, ycbcr_frame.shape[1], 8):
        ycbcr_frame_compressed3[j:j+8, i:i+8, 0] = y_blocks3.pop(0)
        ycbcr_frame_compressed3[j:j+8, i:i+8, 1] = cb_blocks3.pop(0)
        ycbcr_frame_compressed3[j:j+8, i:i+8, 2] = cr_blocks3.pop(0)

for j in range(0, ycbcr_frame.shape[0], 8):
    for i in range(0, ycbcr_frame.shape[1], 8):
        ycbcr_frame_compressed4[j:j+8, i:i+8, 0] = y_blocks4.pop(0)
        ycbcr_frame_compressed4[j:j+8, i:i+8, 1] = cb_blocks4.pop(0)
        ycbcr_frame_compressed4[j:j+8, i:i+8, 2] = cr_blocks4.pop(0)

for j in range(0, ycbcr_frame.shape[0], 8):
    for i in range(0, ycbcr_frame.shape[1], 8):
        ycbcr_frame_compressed5[j:j+8, i:i+8, 0] = y_blocks5.pop(0)
        ycbcr_frame_compressed5[j:j+8, i:i+8, 1] = cb_blocks5.pop(0)
        ycbcr_frame_compressed5[j:j+8, i:i+8, 2] = cr_blocks5.pop(0)

# Convert the compressed frame back to the BGR color space
compressed_frame = cv2.cvtColor(ycbcr_frame_compressed, cv2.COLOR_YCrCb2BGR)

compressed_frame1 = cv2.cvtColor(ycbcr_frame_compressed1, cv2.COLOR_YCrCb2BGR)

compressed_frame2 = cv2.cvtColor(ycbcr_frame_compressed2, cv2.COLOR_YCrCb2BGR)

compressed_frame3 = cv2.cvtColor(ycbcr_frame_compressed3, cv2.COLOR_YCrCb2BGR)

compressed_frame4 = cv2.cvtColor(ycbcr_frame_compressed4, cv2.COLOR_YCrCb2BGR)

compressed_frame5 = cv2.cvtColor(ycbcr_frame_compressed5, cv2.COLOR_YCrCb2BGR)

cv2.imshow('compressed_frame', compressed_frame)
```

Part-18 :-

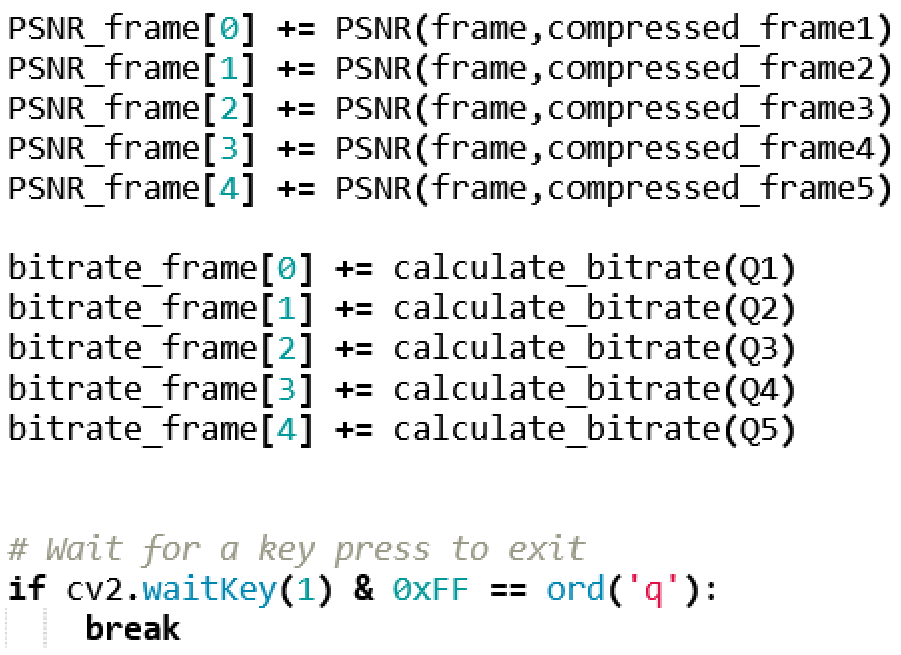
- Saving the frames in the “frames” folder.



```
'''Saving frames'''

dir_path = 'frames/frame{0}'.format(cnt)
if not os.path.exists(dir_path):
    os.makedirs(dir_path)

cv2.imwrite(os.path.join(dir_path,'original_frame0.jpg'), frame)
cv2.imwrite(os.path.join(dir_path,'compressed_frame.jpg'), compressed_frame)
cv2.imwrite(os.path.join(dir_path,'compressed_frame_Q = 10.jpg'), compressed_frame1)
cv2.imwrite(os.path.join(dir_path,'compressed_frame_Q = 40.jpg'), compressed_frame2)
cv2.imwrite(os.path.join(dir_path,'compressed_frame_Q = 70.jpg'), compressed_frame3)
cv2.imwrite(os.path.join(dir_path,'compressed_frame_Q = 100.jpg'), compressed_frame4)
cv2.imwrite(os.path.join(dir_path,'compressed_frame_Q = 120.jpg'), compressed_frame5)

# cv2.imshow('compressed_frame1', compressed_frame1)
# cv2.imshow('compressed_frame2', compressed_frame2)
# cv2.imshow('compressed_frame3', compressed_frame3)
# cv2.imshow('compressed_frame4', compressed_frame4)
# cv2.imshow('compressed_frame5', compressed_frame5)
# Write the compressed frame to the output video
out.write(compressed_frame)
```

<u>Part-19</u> :-

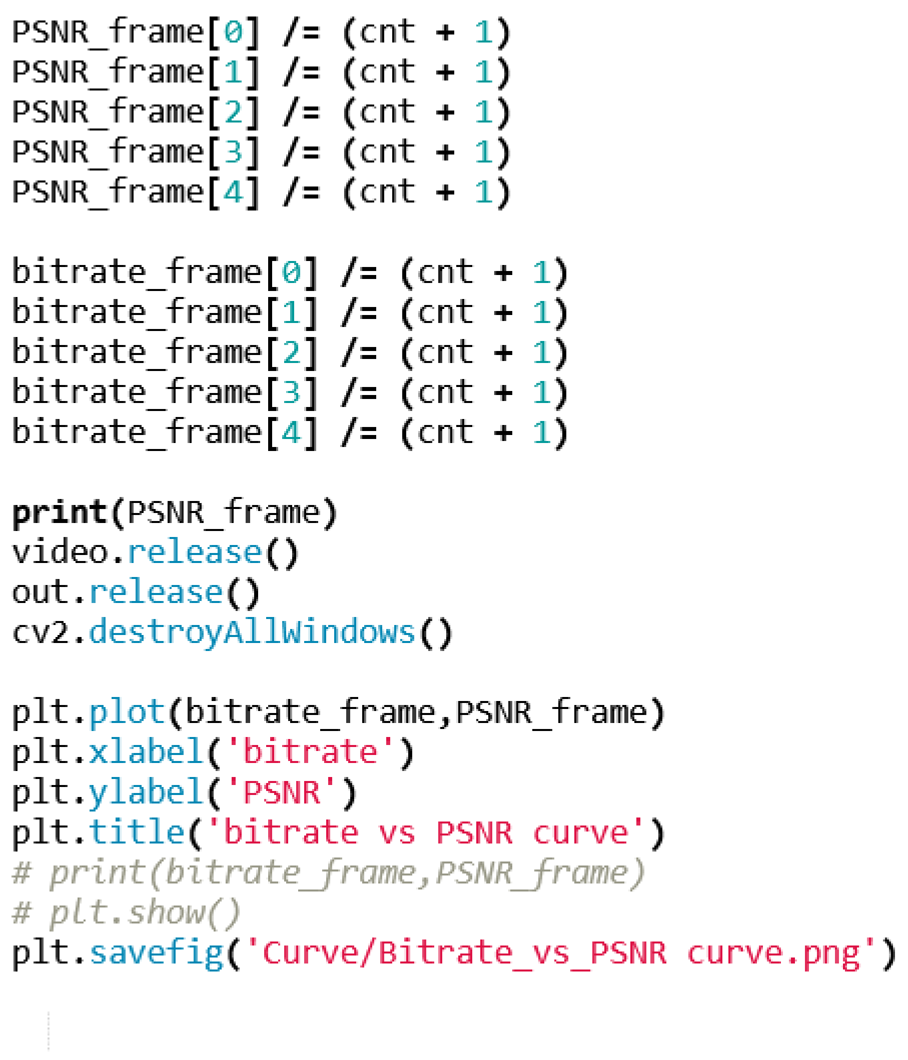
- Summing the PNSR and bitrate values.
- Waiting for the next frame.



```
PSNR_frame[0] += PSNR(frame,compressed_frame1)
PSNR_frame[1] += PSNR(frame,compressed_frame2)
PSNR_frame[2] += PSNR(frame,compressed_frame3)
PSNR_frame[3] += PSNR(frame,compressed_frame4)
PSNR_frame[4] += PSNR(frame,compressed_frame5)

bitrate_frame[0] += calculate_bitrate(Q1)
bitrate_frame[1] += calculate_bitrate(Q2)
bitrate_frame[2] += calculate_bitrate(Q3)
bitrate_frame[3] += calculate_bitrate(Q4)
bitrate_frame[4] += calculate_bitrate(Q5)


# Wait for a key press to exit
if cv2.waitKey(1) & 0xFF == ord('q'):
    break
```

<u>Part-20</u> :-

- Calculating the average PNSR and bitrate value.
- Plotting the PNSR vs bitrate graph.



```
PSNR_frame[0] /= (cnt + 1)
PSNR_frame[1] /= (cnt + 1)
PSNR_frame[2] /= (cnt + 1)
PSNR_frame[3] /= (cnt + 1)
PSNR_frame[4] /= (cnt + 1)

bitrate_frame[0] /= (cnt + 1)
bitrate_frame[1] /= (cnt + 1)
bitrate_frame[2] /= (cnt + 1)
bitrate_frame[3] /= (cnt + 1)
bitrate_frame[4] /= (cnt + 1)

print(PSNR_frame)
video.release()
out.release()
cv2.destroyAllWindows()

plt.plot(bitrate_frame,PSNR_frame)
plt.xlabel('bitrate')
plt.ylabel('PSNR')
plt.title('bitrate vs PSNR curve')
# print(bitrate_frame,PSNR_frame)
# plt.show()
plt.savefig('Curve/Bitrate_vs_PSNR curve.png')
```

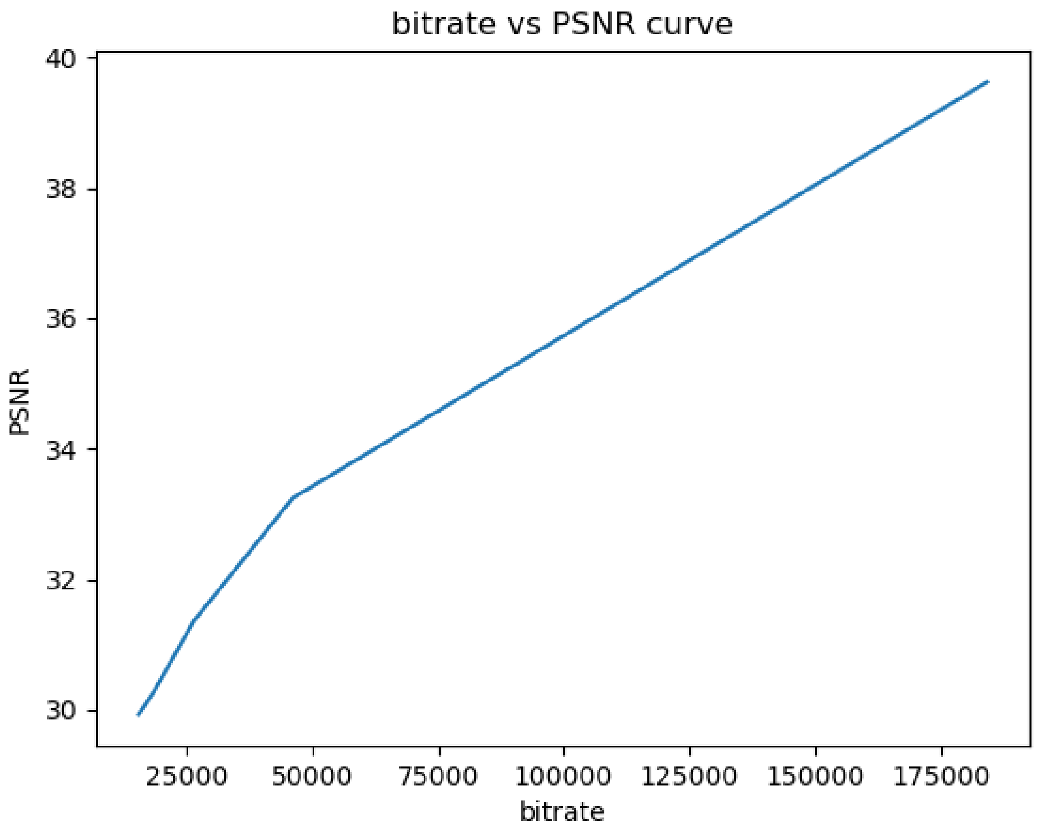


Figure :- PNSR vs bitrate curve

Figure :- Original Frame

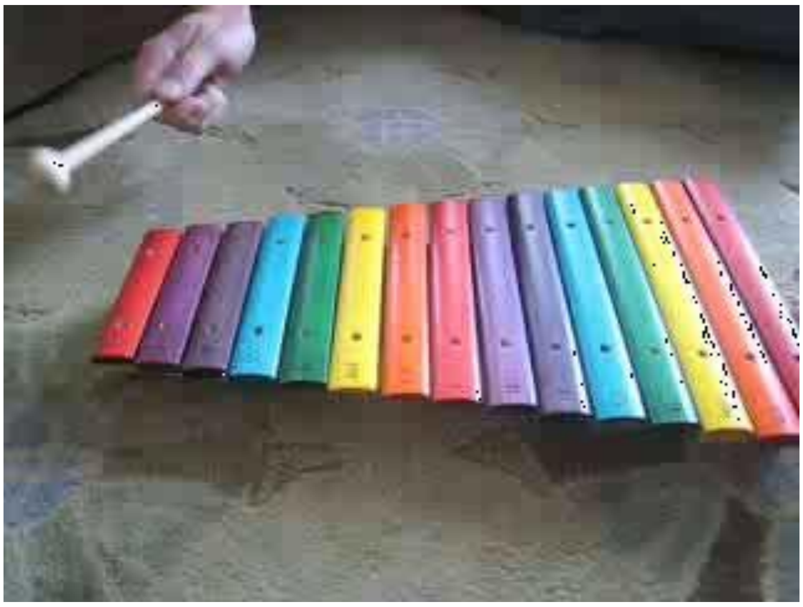


Figure :- Compressed frame

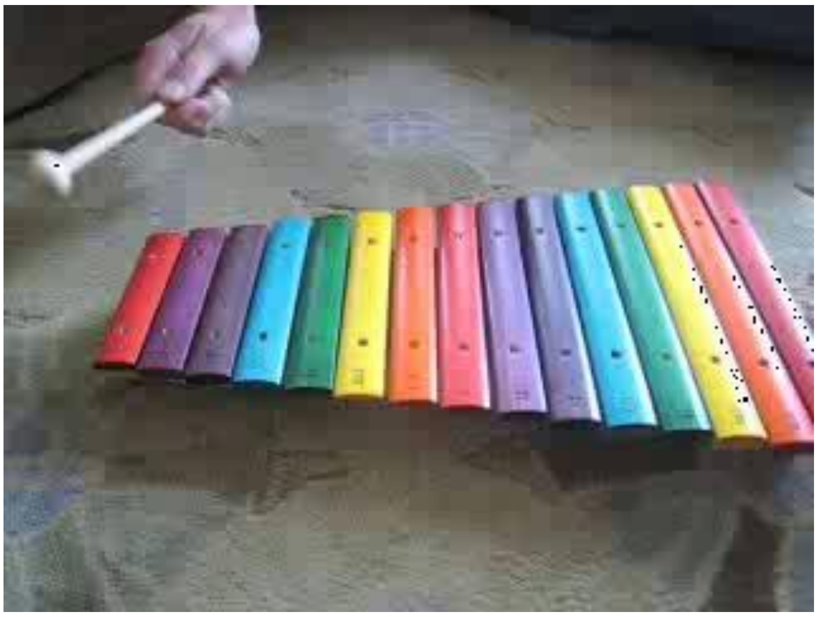


Figure :- Compressed frame with quantization factor = 10

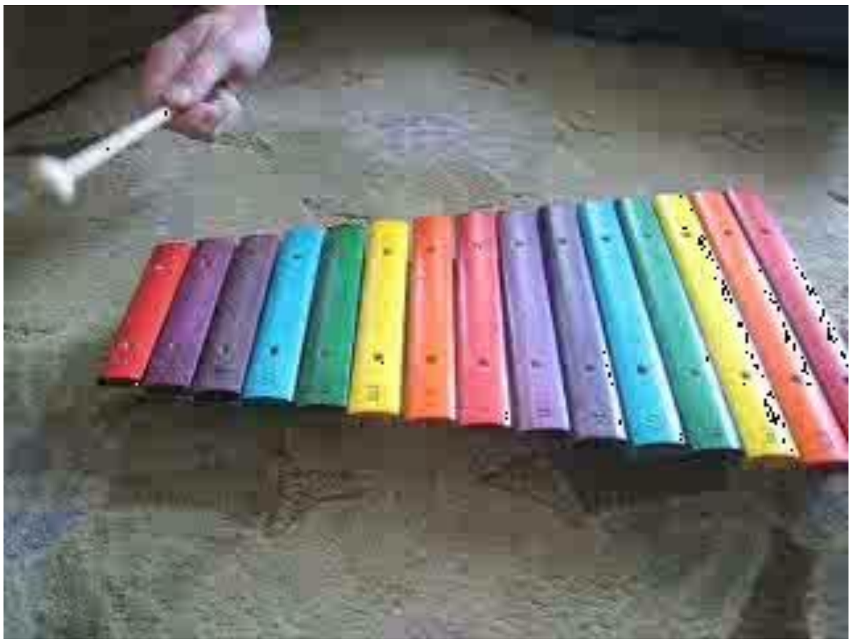


Figure :- Compressed frame with quantization factor = 40

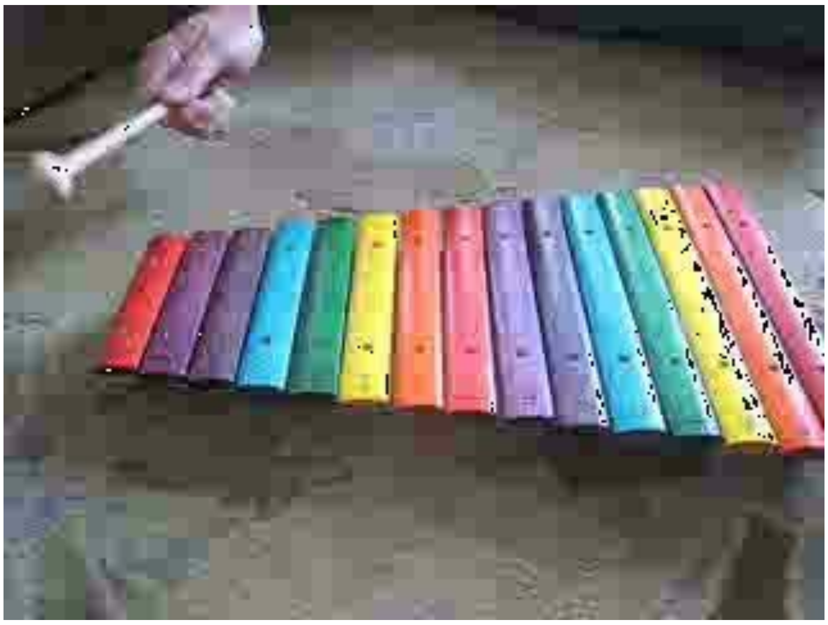


Figure :- Compressed frame with quantization factor = 70

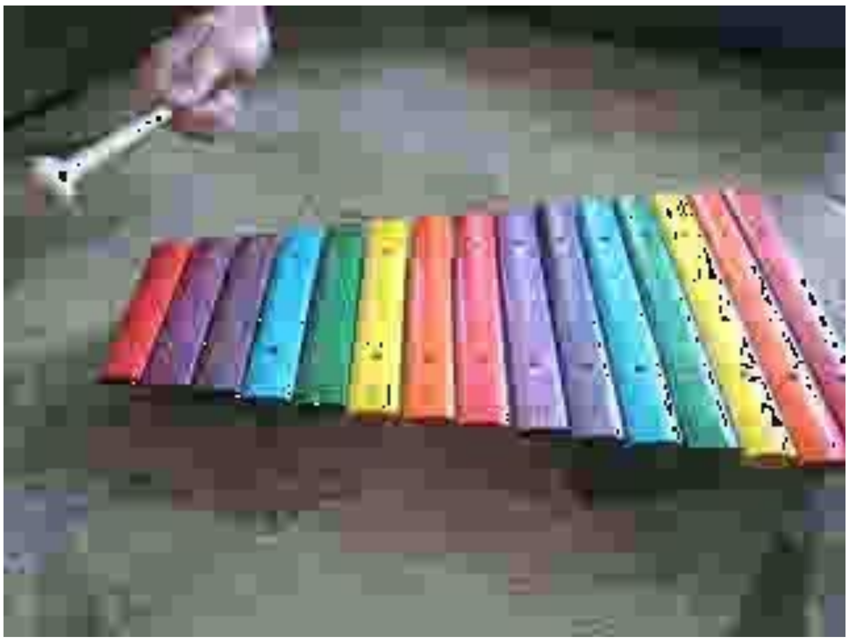


Figure :- Compressed frame with quantization factor = 100

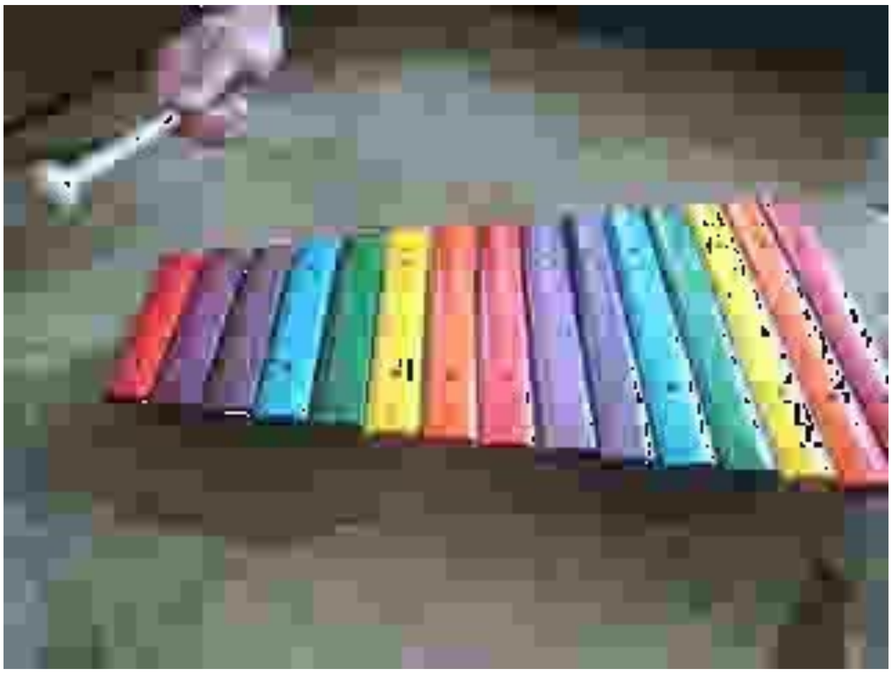


Figure :- Compressed frame with quantization factor = 120

Motion-Compensated Interframe Prediction

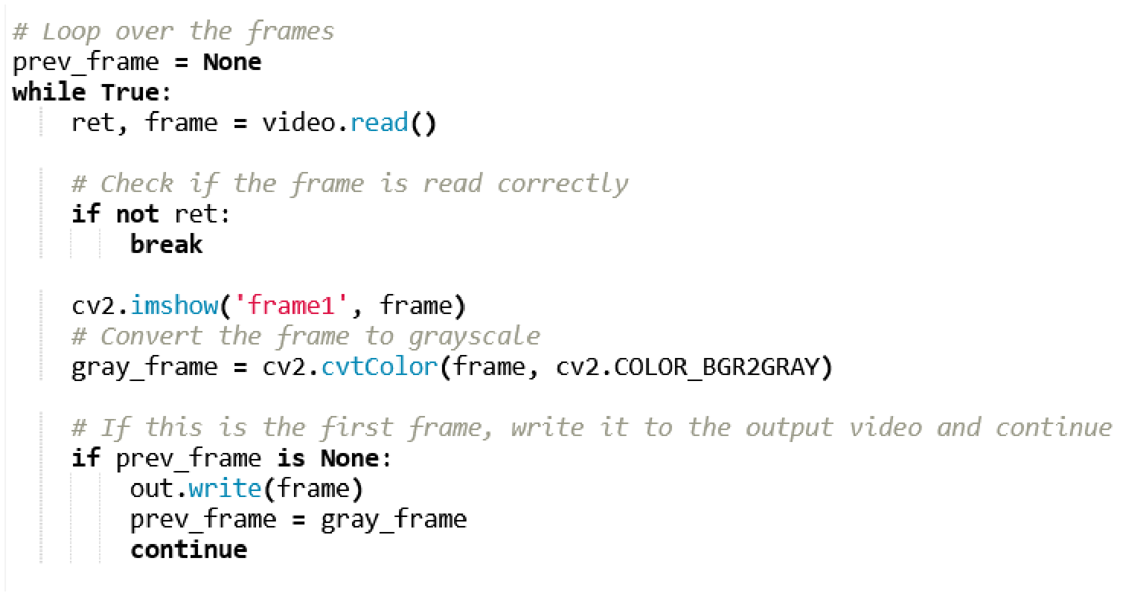
Part-1 :-

- Importing the required functionalities.
- Setting up the block size as 16
- Opening the video file.
- Determining the video file properties.
- Opening the file for output.

```python
import cv2
import numpy as np

# Set the block size for motion compensation
block_size = 16

# Open the input video
video = cv2.VideoCapture('xylophone1.avi')

# Get the video properties
frame_width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))
fps = int(video.get(cv2.CAP_PROP_FPS))

# Create a VideoWriter object for the output video
fourcc = cv2.VideoWriter_fourcc(*'DIVX')
out = cv2.VideoWriter('video.avi', fourcc, fps, (frame_width, frame_height))
```

<u>Part-2</u> :-

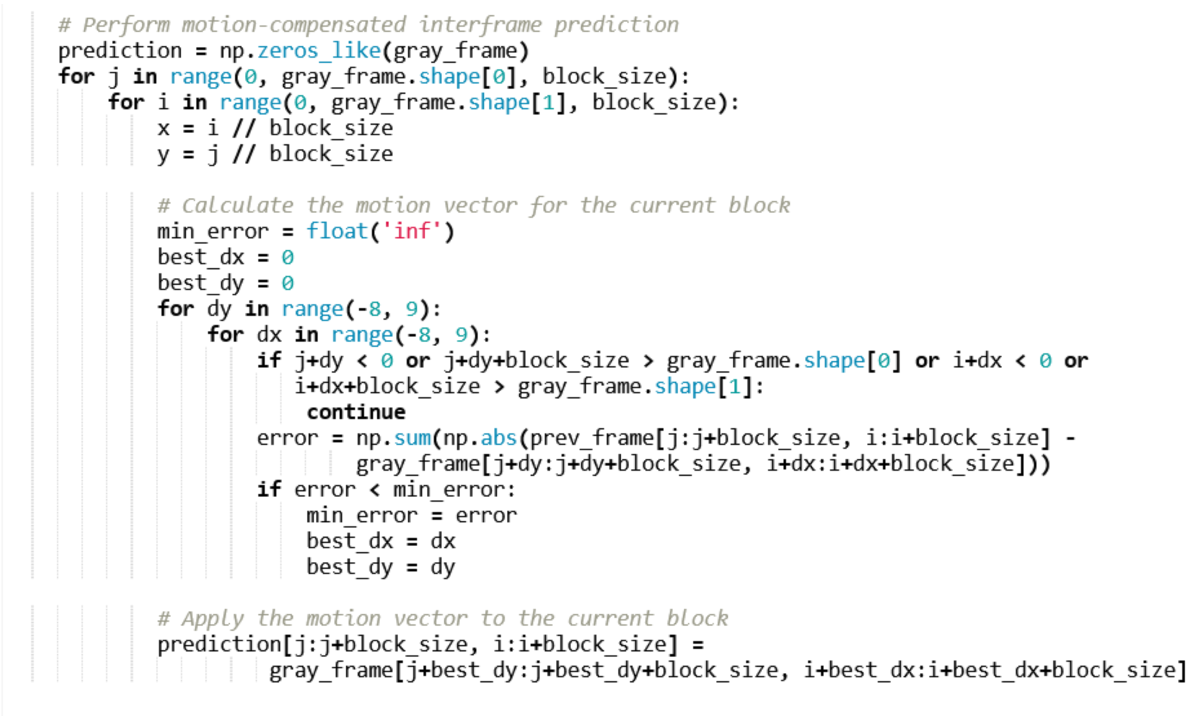
- Looping over the frames.
- Reading the next frame in the loop and checking if correctly read.
- Displaying the current frame.
- Converting the frame to grayscale



```python
# Loop over the frames
prev_frame = None
while True:
    ret, frame = video.read()

    # Check if the frame is read correctly
    if not ret:
        break

    cv2.imshow('frame1', frame)
    # Convert the frame to grayscale
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # If this is the first frame, write it to the output video and continue
    if prev_frame is None:
        out.write(frame)
        prev_frame = gray_frame
        continue
```

Part-3 :-

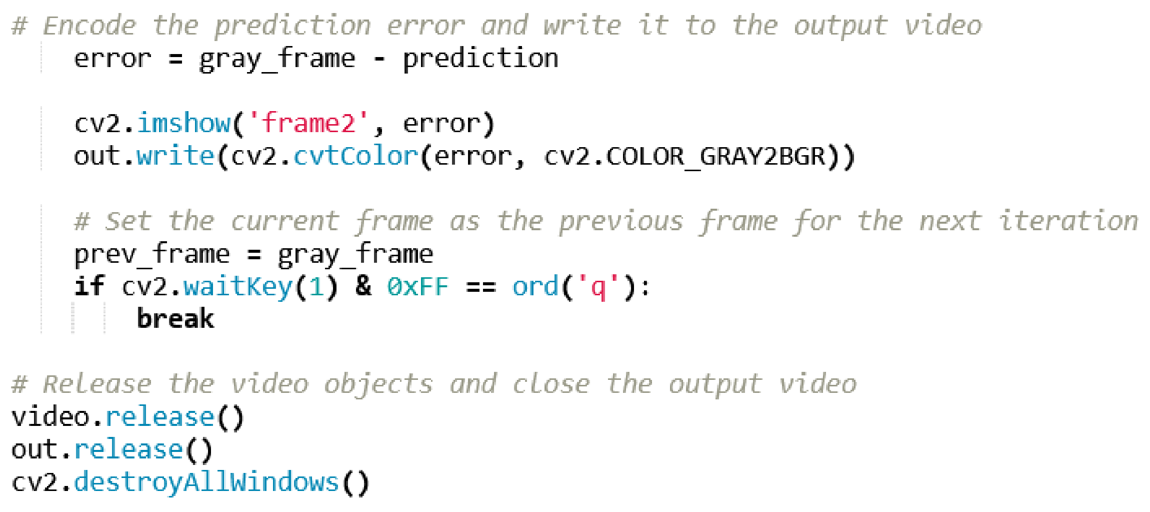
- Performing the motion-compensated interframe prediction.
- Looping trough the blocks of the frame and calculating the motion vector of the block by a search block algorithm.
- This uses the temporal and spatial redundancy in the consecutive frames.



```python
# Perform motion-compensated interframe prediction
prediction = np.zeros_like(gray_frame)
for j in range(0, gray_frame.shape[0], block_size):
    for i in range(0, gray_frame.shape[1], block_size):
        x = i // block_size
        y = j // block_size

        # Calculate the motion vector for the current block
        min_error = float('inf')
        best_dx = 0
        best_dy = 0
        for dy in range(-8, 9):
            for dx in range(-8, 9):
                if j+dy < 0 or j+dy+block_size > gray_frame.shape[0] or i+dx < 0 or
                   i+dx+block_size > gray_frame.shape[1]:
                    continue
                error = np.sum(np.abs(prev_frame[j:j+block_size, i:i+block_size] -
                        gray_frame[j+dy:j+dy+block_size, i+dx:i+dx+block_size]))
                if error < min_error:
                    min_error = error
                    best_dx = dx
                    best_dy = dy

        # Apply the motion vector to the current block
        prediction[j:j+block_size, i:i+block_size] =
                 gray_frame[j+best_dy:j+best_dy+block_size, i+best_dx:i+best_dx+block_size]
```

Part-4 :-

- Encoding the predication error and writing it to the output vector.
- Displaying the reconstructed frame.
- Setting the current frame as the previous frame.
- Waiting for the next frame.
- After successfully iterating through all the frames of the video we release the video and close the output video file.



```
# Encode the prediction error and write it to the output video
    error = gray_frame - prediction

    cv2.imshow('frame2', error)
    out.write(cv2.cvtColor(error, cv2.COLOR_GRAY2BGR))

    # Set the current frame as the previous frame for the next iteration
    prev_frame = gray_frame
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the video objects and close the output video
video.release()
out.release()
cv2.destroyAllWindows()
```