Assignment 2

Submitted by

Major Arpit Guleria, 2022EET2804

Suchet Bhalla, 2022EET2088

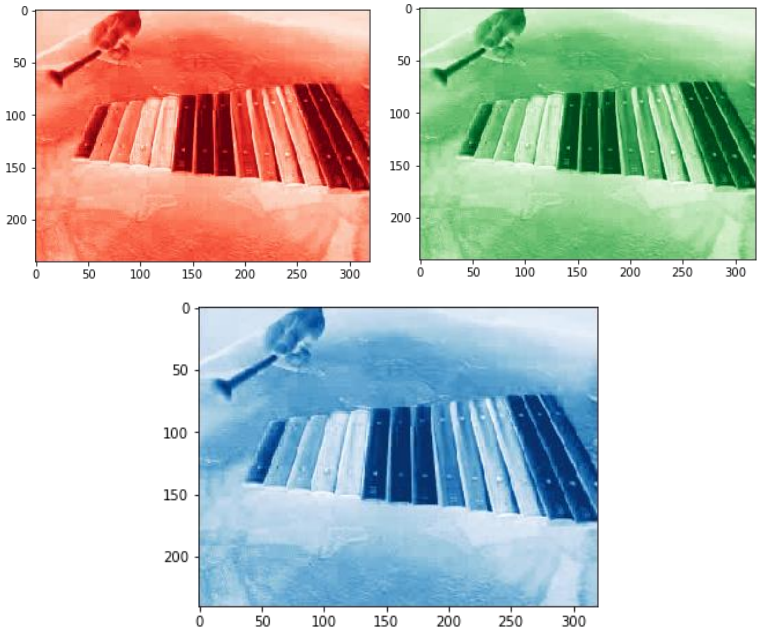
1. Luminance, Chrominance red & Chrominance blue, aka Y Cr Cb

Q. What is Y Cr Cb?

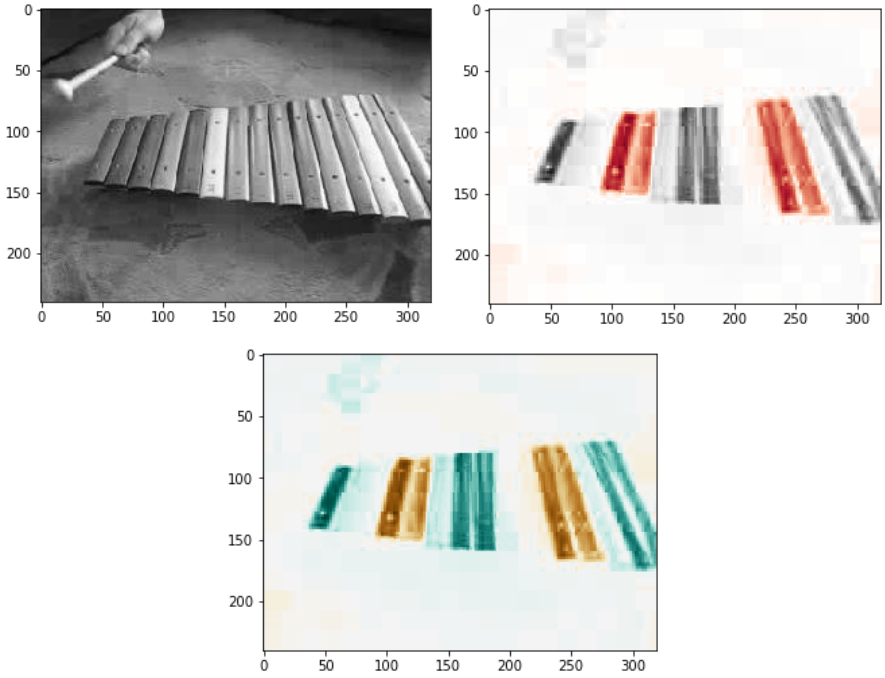
A. We know that an image can be represented as RGB channels. Similarly, YCrCb is a representation of an image. Most of the information is captured in the Y channel, as a grayscale image. The Cr & Cb channels contain information to add colour to the Y channel, to recreate the RGB image. The following images display these representations.

Frame (at t= 0)

Decomposition into RGB:



Decomposition into Y Cr Cb:



2. MPEG coding: Motion-compensated interframe prediction & Discrete Cosine Transform

Q. What is *predictive encoding*?

A. **Motivation**: A video is composed of a sequence of images aka *frames*. Typically, many frames are similar, i.e., there is *temporal redundancy* between (multiple) sub-sequences of (successive) frames. This redundancy can be utilized to store & transmit a video with lesser bits.

Concept: We label each frame as I, P or B. A frame which is kept unchanged is called an I-frame. A P-frame is composed of *error* signals, this error is the difference between the preceding I-frame & current I-frame. By adding this P-frame to the preceding I-frame, we can recreate the current I-frame. If there is temporal redundancy between 2 successive frames, then the mean error will be (about) 0 & standard deviation in error will also be low. Thus, by using a *variable length encoding*, a P-frame can be encoded with lesser bits.
A B-frame is formed from a preceding & a succeeding I-frame.
To further utilize *temporal redundancy*, we applied *predictive encoding* with *motion compensation*.

Q. What is *motion compensation*?

A. We shall explain this concept with an example. Consider 2 successive frames with high *temporal redundancy*, F_1 & F_2, of dimensions 240 x 320.

a. Macroblocks: We segregate F_1 & F_2 into macroblocks of size 16x16.

b. Motion vectors: Then, we displace each macroblock in F_1 horizontally & vertically, to best resemble a 16x16 sub-frame (not macroblock) in F_2. By doing this for each macroblock in F_1, we create a prediction of F_2, say F'_2. The displacement of each macro-blocks is called a motion vector. Now, we have a matrix of motion vectors of size 30 x 40.

c. P-frame: We calculate the error between F'_2 & F_2, say E.

a. And, By applying motion vectors to macroblocks in F_1, we create F'_2, then
b. By adding F'_2 & E, we recreate F_2 exactly.

Because, F_1 & F_2 have high *temporal redundancy*, we expect the error to be low & the *motion vectors* to be small.

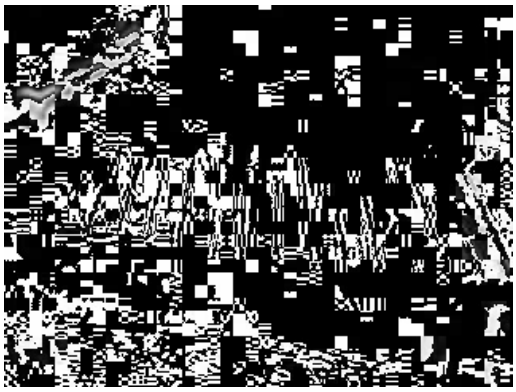
The following images demonstrate the result of motion compensation. **Note,** we performed *motion compensation* on 16x16 macroblocks, by using a variant of the *three step search algorithm,* with an initial step of 16.

Frame 0

Frame 1

Prediction

Errors



Observation: The error signals appear as rectangular blocks, because we chose to compress the frames (via DCT & iDCT) before *motion compensation*, with the aim of gaining better compression of video.

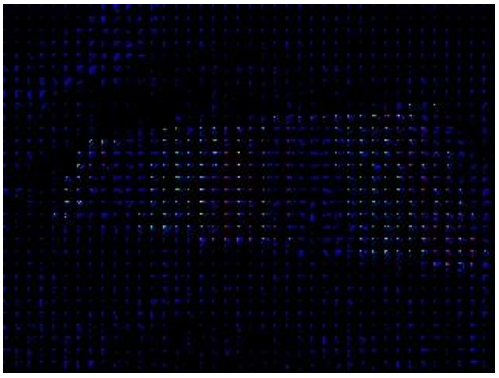
Q. What is DCT?

A. Th information in an image can be decomposed into its constituent, orthogonal co-sinusoidal signals. With DCT, we exploit *spatial redundancy* (within a frame) to achieve compression. We remove the high frequency components, & retain the low frequency components, still the difference is invisible to the human eye.
We performed DCT on every 8x8 block of a frame. For the aforesaid removal, we performed component-wise quantization, with the standard matrix Q_{50}.

The following images demonstrate the application of DCT & its inverse on an RGB frame.

Original

After DCT



After iDCT (with Q 50)

Q. To encode the frames (video), why do we perform a zig-zag scan?

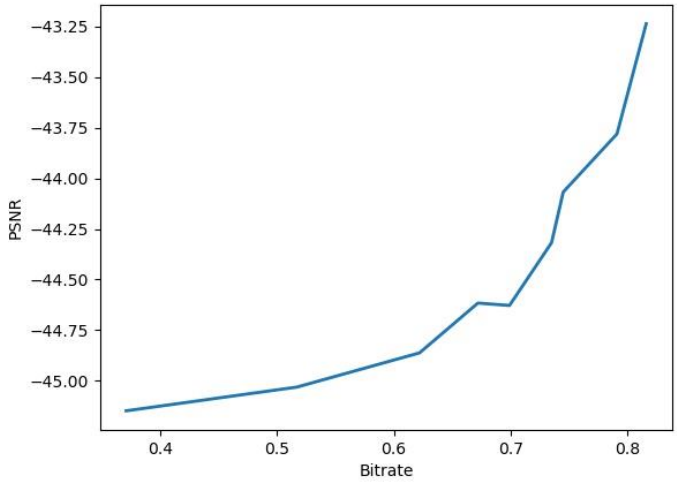
A. Before storage, we convert each frame to a stream of binary data. After quantization, most coefficients in a frame are zero. To consolidate a long run of zeroes, we scan a frame in a zig-zag fashion.

After this scan, we applied *run length coding*, then our custom *variable length coding*.

Result: With this encoding, we achieved a maximum compression of 48 times, on average, with the quantization matrix Q_{10}. The compression achieved for Y, Cr, Cb frames is different; 6, 67, 72, respectively.
The size of each uncompressed frame is 230.4 KB (= 240 rows x 320 cols x 3 channels). After compression it is 4.8 KB.
The size of an uncompressed video should be 16 MB (= 230.4 KB x 69 frames). The size of the encoded video is 331 KB (~ 4.8 KB x 69 frames).

3. PSNR vs Bitrate

In the following graph, we varied the degree of quantization with the matrices Q_{10} to Q_{90}, to obtain different qualities of reconstruction, hence PSNRs.



References:

1. "Image Compression & the Discrete Cosine Transform" by Ken Cabeen & Peter Gent, College of the Redwoods, Eureka, California
2. "An Investigation of Block Searching Algorithms for Video Frame Codecs" by Jerome Casey, Technological University Dublin, Dublin, Ireland
3. "Search Algorithms for Block Matching in Motion Estimation" by Deepak Turaga, Mohamed Alkanhal, Carnegie Mellon University, Pittsburgh, Pennsylvania

Source Code:

```
#Libraries
import cv2
from sys import exit
import numpy as np
import matplotlib.pyplot as plt
from os.path import getsize
from os import remove
from Seg import *
from runlen import *
from VLC import *

#Function definitions

#Extracts frames from a video with 'name' (which is in current directory), into the list
'empty_list'
def extract_frames(name, empty_list):

    #opens video
    capture = cv2.VideoCapture(name)

    #If the file is absent from the current-directory, terminate this program
    if capture.isOpened() == False:
        print("Error: File not is current directory")
        exit(0)

    #Stores each frame in a list
    while capture.isOpened():

        ret, frame = capture.read()

        #If there is a frame, then store it
        if ret == True:
            #cv2 stores frames as BGR. Here I convert to RGB; dtype = uint8
            empty_list.append(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))

        #Else, I have read all frames
        else:
            break

    capture.release()
    del capture

#1. Color-converts an RGB image to YCrCb
#2. Returns the converted frame
def rgb2ycrcb(old):
```

```
    """
    SOURCE of the formulae:
https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html
    """
    rows, cols, depth = old.shape
    new = np.zeros((rows, cols, depth), dtype = int)
    #Y
    new[:, :, 0] = 0.299*old[:, :, 0] + 0.587*old[:, :, 1] + 0.114*old[:, :, 2]
    #Cr
    new[:, :, 1] = (old[:, :, 0] - new[:, :, 0])*0.713 + 128
    #Cb
    new[:, :, 2] = (old[:, :, 2] - new[:, :, 0])*0.564 + 128


    #multiplication by float converts the array's data-type (uint8) to 'float 64'

    return new.astype(int)

#Converts all frames from RGB to YCrCb, &  stores these in the empty_list
def convert_to_YCrCb(frames, empty_list):
    #To store the result separately
    rows, cols, depth = frames[0].shape

    for i in range(len(frames)):
        empty_list.append( rgb2ycrcb(frames[i]) )

#creates the T matrix
def create_T():
    from math import cos, pi
    N = 8
    T = np.zeros((N, N))

    x = round(1/np.sqrt(N), 5)
    Pi = round(pi, 4)
    del pi
    sqrt2 = round(np.sqrt(2), 5)

    for i in range (N):
        for j in range(N):
            if i == 0:
                T[i, j] = round(x, 4)
            else:
                T[i, j] = round(x * sqrt2 * round(cos((2*j + 1)*i*Pi/(2*N)), 4), 4)

    return T

#Performs DCT on 1 frame; inplace
def dct(processed_frame):
```

```
    rows, cols, depth = processed_frame.shape

    #Performs DCT on all the 8-by-8 blocks
    #1. subtracts 128
    processed_frame[:, :, :] = processed_frame[:, :, :] - 128
    for i in range(0, rows, 8):
        for j in range(0, cols, 8):
            for k in range(0, depth, 1):
                #Performs T.M.T'
                    processed_frame[i:i+8, j:j+8, k] = T @ processed_frame[i:i+8, j:j+8, k] @
T.transpose()
                    #Performs quantization
                    #This if-else enables different quantizations for the luma & chromas
                    #luma
                    if (k == 0):
                        processed_frame[i:i+8, j:j+8, k] = np.divide(processed_frame[i:i+8, j:j+8,
k], Q_50)
                    #chromas
                    else:
                        processed_frame[i:i+8, j:j+8, k] = np.divide(processed_frame[i:i+8, j:j+8,
k], Q_10)

#Performs iDCT on 1 frame; inplace
def idct(processed_frame):
    rows, cols, depth = processed_frame.shape
    #Performs iDCT on all the 8-by-8 blocks
    for i in range(0, rows, 8):
        for j in range(0, cols, 8):
            for k in range(0, depth, 1):
                #Performs de-quantization
                if (k == 0):
                    processed_frame[i:i+8, j:j+8, k] = np.multiply(Q_50, processed_frame[i:i+8,
j:j+8, k])
                else:
                    processed_frame[i:i+8, j:j+8, k] = np.multiply(Q_10, processed_frame[i:i+8,
j:j+8, k])
                #Performs T'.M.T
                processed_frame[i:i+8, j:j+8, k] = T.transpose() @ processed_frame[i:i+8,
j:j+8, k] @ T
    #Adds 128
    processed_frame[:, :, :] = processed_frame[:, :, :] + 128

#1. Converts an image in YCrCb to RGB
#2. Returns the converted frame
def ycrcb2rgb(old):
    """
    R ← Y + 1.403·(Cr−delta)
```

```
    B ← Y + 1.773·(Cb−delta)
    G ← Y − 0.714·(Cr−delta) − 0.344·(Cb−delta)
    delta = 128 for 8 bit images
    """
    rows, cols, depth = old.shape
    new = np.zeros((rows, cols, depth), dtype = int)
    #R
    new[:, :, 0] = old[:, :, 0] + 1.403*(old[:, :, 1] - 128)
    #B
    new[:, :, 2] = old[:, :, 0] + 1.773*(old[:, :, 2] - 128)
    #G
    new[:, :, 1] = old[:, :, 0] - 0.714*(old[:, :, 1] - 128) - 0.344*(old[:, :, 2] - 128)

    return np.uint8(new)

#Converts all frames from YCrCb to RGB, &  stores these in the list 'results'
def convert_to_RGB(processed_frames, results):
    rows, cols, depth = processed_frames[0].shape
    for i in range(len(processed_frames)):
        results.append(ycrcb2rgb(processed_frames[i]))

#Makes a (colored) video from frames (at fps)
def makevideo(frames, fps, isColor = True):
    rows = frames[0].shape[0]
    cols = frames[0].shape[1]
    writer = cv2.VideoWriter("7. Compressed_video.avi",
cv2.VideoWriter_fourcc(*"DIVX"), \
                        fps, (cols, rows), isColor = isColor)
    for frame in frames:
        writer.write(cv2.cvtColor(frame, cv2.COLOR_RGB2BGR))

    writer.release()
    del writer

#1. Motion Compensation: Shifts macro-blocks (16x16) (in the previous-frame) to
recreate the next-frame
#2. Returns the prediction
#3. The arguments are grayscale-images
def predict(fprev, fnext):

    #init
    rows, cols = fprev.shape
    pred = np.copy(fprev)

    #To store the motion vectors
    mv = []

    #size of macro-block
```

```
size = 16

#For each 16x16 block
for i in range(0, rows, size):
   for j in range(0, cols, size):

      #Each block
      block = np.copy(fprev[i:i+size, j:j+size])

      #The new-position will be stored here ; motion vector = (new - old) position
      ip = i
      jp = j

      #The max. step  by which a block can displace
      step = 16

      #condition for termination
      while step > 1:

         #list to store error-values & new-coordinates
         lst = []
         coords = []
         #init
         for k in range(9):
            lst.append(0xFFFFFFFF)
            coords.append( [-1, -1] )

         #Count of the iteration no. It is used to index into the list
         count = 0

         #Creates [-step, 0 , +step]
         ranger = range(-1 * step, step<<1, step)
         #Search the 9 blocks
         for r in ranger:
            for c in ranger:
               #first-element of the block is fnext[idx, jdx]
               fr = ip + r
               fc = jp + c
               #last element is fnext[idx+size-1, jdx+size-1]
               lr = fr + size
               lc = fc + size

               #boundary checks
               if fr < 0 or fc < 0:
                  count+=1
                  continue
               if lr >= rows or lc >= cols:
                  count+=1
```

```
                    continue

                #calculation
                lst[count] = np.linalg.norm(block - fnext[fr:lr, fc:lc], ord = 2)
                coords[count] = [fr, fc]
                count += 1

            #Search complete, i.e., I am out of the (previous) nested 'for' loops

            #For next search, decrease the window-size
            step = int(step /2)

            #'min_' = index (in list 'lst') of the min. error
            min_ = lst.index( min(lst) )
            #Sets the new position
            ip, jp = coords[min_]

            #clean up
            del lst, coords, ranger, count, min_

        #Out of the 'while' loop
        #Shifts this block to the suitable position (ip, jp)
        pred[ip:ip+size, jp:jp+size] = fprev[i:i+size, j:j+size]
        del block
        mv.append((ip - i, jp - j))
        #Onto the next block

    #Prediction created, i.e., out of the nested 'for' loops
    del step, i, j, ip, jp, fr, fc, lr, lc

    return pred, mv

#1. Applies the motion-vectors stored in 'mv' onto the (gray-scale) image called 'frame'
#2. Returns the result (prediction)
def displace(frame, mv):
    #init
    rows, cols = frame.shape
    pred = np.copy(frame)

    #size of macroblock
    size = 16

    jmp = int(cols/size)
    #For each block
    for i in range(0, rows, size):
        for j in range(0, cols, size):
            dx, dy = mv[ int( (j + i*jmp)/size) ]
            ip = i + dx
```

```
            jp = j + dy
            pred[ip:ip + size, jp: jp + size] = frame[i:i + size, j:j + size]

    return pred

#1. Image compression: Performs DCT & iDCT on all frames in the list 'frames'
#2. Returns the compressed frame*s*, as a list
def compress(frames):

    compressed_frames = []
    convert_to_YCrCb(frames, compressed_frames)

    for i in range(len(frames)):
        dct(compressed_frames[i])
        idct(compressed_frames[i])
    return compressed_frames

"""
1. Performs motion-compensated, predictive encoding on all 'frames'
2. interval = Interval b/w I-frames
3. Returns 2 lists of
    a. the I & P frames
    b. the motion-vectors
"""
def encode(frames, interval):

    #These lists store the final results
    errors = []
    mvs = []
    output = np.zeros((240,320,3))
    frame_1 = []
    frame_2 = []
    frame_3 = []
    runlen1 = []
    runlen2 = []
    runlen3 = []
    VLC_encode1 = []
    VLC_encode2 = []
    VLC_encode3 = []
    VLC_decode1 = []
    VLC_decode2 = []
    VLC_decode3 = []
    runlen_inv1 = []
    runlen_inv2 = []
    runlen_inv3 = []

    #This list stores the compressed frames
    compressed_frames = compress(frames)
```

```
    #This list stores the YCrCb images
    processed_frames = []
    convert_to_YCrCb(frames, processed_frames)

    for i in range( len(processed_frames) ):
        #If i != a multiple of 'interval', create a prediction
        if i % interval:
            #pred, mv = predict(prev, next)
            pred, mv = predict(compressed_frames[i-1][:, :, 0], compressed_frames[i][:, :, 0])

            #Equation: diff = frame - pred (=> frame = pred + diff)
            dct(processed_frames[i])
            diff = np.copy(processed_frames[i])
            diff[:, :, 0] = compressed_frames[i][:, :, 0] - pred

            errors.append(diff)
            mvs.append(mv)

        #Else, store the (DCT of the) I-frame
        else:
            dct( processed_frames[i] )
            errors.append( np.copy(processed_frames[i]) )
            mvs.append(-1)

    for a in range(0, 240, 8):
        for b in range(0, 320, 8):
            for i in range(0,64):
                frame_1.append(segregator(processed_frames[0][:, :, 0],a,b)[i])
                frame_2.append(segregator(processed_frames[0][:, :, 1],a,b)[i])
                frame_3.append(segregator(processed_frames[0][:, :, 2],a,b)[i])
    runlen1 = runlen_code(frame_1)
    runlen2 = runlen_code(frame_2)
    runlen3 = runlen_code(frame_3)
    VLC_encode1 = VLC(runlen1)
    VLC_encode2 = VLC(runlen2)
    VLC_encode3 = VLC(runlen3)
    print(" Length of a frame[0]-Y after DCT and Quantisation : ",len(frame_1))
    print(" Length of a frame[0]-Y after Run length Encoding : ",len(runlen1))
    a1 = len(frame_1)/len(runlen1)
    print(" Implies reduction by the factor : ",a1)

print("============================================================
====")
    print(" Length of a frame[0]-Cr after DCT and Quantisation : ",len(frame_2))
    print(" Length of a frame[0]-Cr after Run length Encoding : ",len(runlen2))
    a2 = len(frame_2)/len(runlen2)
    print(" Implies reduction by the factor: ",a2)
```

```
print("============================================================
====")
    print(" Length of a frame[0]-Cb after DCT and Quantisation : ",len(frame_3))
    print(" Length of a frame[0]-Cb after Run length Encoding : ",len(runlen3))
    a3 = len(frame_3)/len(runlen3)
    print(" Implies reduction by the factor : ",a3)

print("============================================================
====")
    print("Average Reduction of frame[0] by factor of : ",(a1+a2+a3)/3)

print("============================================================
====")
    VLC_decode1 = inv_VLC(VLC_encode1)
    VLC_decode2 = inv_VLC(VLC_encode2)
    VLC_decode3 = inv_VLC(VLC_encode3)
    runlen_inv1 = runlen_decode(VLC_decode1)
    runlen_inv2 = runlen_decode(VLC_decode2)
    runlen_inv3 = runlen_decode(VLC_decode3)
    output[:, :, 0] = combiner(runlen_inv1)
    output[:, :, 1] = combiner(runlen_inv2)
    output[:, :, 2] = combiner(runlen_inv3)
    cv2.imwrite("9. RLC_VLC_encoded_decoded.jpeg", np.uint8(output))


    del processed_frames, compressed_frames, interval

    return errors, mvs


"""
1. Reconstructs frames from I, P frames & motion vectors
2. interval = Interval b/w I-frames
3. Returns the reconstructed-frames
"""
def decode(errors, mvs, interval):

    #This list store the final result
    recon = []

    #For pretty-printing
    pp = 1

    for i in range( len(errors) ):
        if i % interval:
            #displace the image
            disp = displace(errors[i-1][:, :,0], mvs[i])
```

```
            #add the errors to the displaced image; Here I have reconstructed Y of the frame 'i'
            temp = disp + errors[i][:, :,0]
            #Perform idct on the chroma channels; Here, the first channel is the error in
prediction
            idct(errors[i])
            #Set the first channel = Y
            errors[i][:, :, 0] = temp
            if pp:
                cv2.imwrite("4. Prediction_of_frame_1_from_frame_0.jpeg",\
                        disp)
                cv2.imwrite("5. Reconstructed_frame_1.jpeg",\
                        temp)
                pp -= 1
            del disp
        else:
            idct( errors[i] )

        recon.append(errors[i])

    del errors, mvs
    return recon

def psnr(org, recon):
    #1 Peak value
    peak = org.max()
    #2 Frobenius norm
    diff = np.linalg.norm(org - recon)
    #3 MSE
    rows, cols, depth = org.shape
    N = rows * cols * depth
    mse = np.square(diff)/N
    #4 PSNR = 10*log_10 (ratio)
    ratio = np.square(peak)/mse
    return round(10 * np.log10(ratio), 3)

#Displays the PSNR vs Bitrate
def comparixion(frames):
    #init
    global Q_50
    global Q_10
    safe = np.copy(Q_50)
    Q_10 = np.copy(Q_50)

    rows, cols, depth = frames[0].shape
    N = rows * cols * depth

    #selects the 1st frame
    org = frames[0]
```

```
img = rgb2ycrcb(org)

#results will be stored here
psnrs = []
sizes = []

# [1, 4] => greater compression; [6, 9] => lesser compression
for i in range(1, 10):
    if i < 5:
        Q_50 = (5/i * safe).astype(int)
    elif i > 5:
        Q_50 = ((10-i)/5 * safe).astype(int)
    elif i == 5:
        Q_50 = np.copy(safe)

    Q_10 = Q_50

    #creates a new copy
    temp = np.copy(img)

    #performs compression
    dct(temp)
    idct(temp)

    #stores the result as RGB
    recon = ycrcb2rgb(temp)

    name = "Comp_Q" + str(i) + "0.jpeg"
    cv2.imwrite( name , cv2.cvtColor(recon, cv2.COLOR_RGB2BGR))
    #
    psnrs.append(psnr(org, recon))
    sizes.append( round(getsize(name) * 8 / N, 3))
    #
    remove(name)
    del recon, temp, name

# plot
fig, ax = plt.subplots()

ax.plot(sizes, psnrs, linewidth=2.0)
plt.xlabel("Bitrate")
plt.ylabel("PSNR")
#ax.set(xlim=(0, 8), xticks=np.arange(1, 8), ylim=(0, 8), yticks=np.arange(1, 8))

plt.savefig("8. PSNR_vs_Bitrate.jpeg")
#print(psnrs, sizes)
Q_50 = safe
del org, img, psnrs, sizes, fig, ax, rows, cols, depth, N
```

```
#Global variables
#The quantization matrices Q50 & Q10
Q_50 = 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]
      ])

Q_10 = 5*np.copy(Q_50)

T = create_T()

#Main
frames = []
extract_frames("xylophone1.wm", frames)
cv2.imwrite("1. Original_frame_0.jpeg", cv2.cvtColor(frames[0],
cv2.COLOR_RGB2BGR))

#Interval b/w I-frames
interval = 23
#1. errors = I, P frames
#2. mvs = a list of motion-vectors
errors, mvs = encode(frames, interval)
cv2.imwrite("2. After_DCT.jpeg", np.uint8(errors[0]) )
cv2.imwrite("6. Error_in_motion_compensation_between_frames_0_&_1.jpeg",
np.uint8(errors[1][:, :, 0]))

#Frames reconstructed from I, P frames & motion-vectors
recon = decode(errors, mvs, interval)

#Color-conversion from YCrCb to RGB
results = []
convert_to_RGB( recon, results)
cv2.imwrite("3. Compressed_frame_0.jpeg", cv2.cvtColor(results[0],
cv2.COLOR_RGB2BGR))

#Stores the video at 15 FPS
makevideo(results, 17, isColor = True)

#Displays the PSNR
comparixion(frames)

#Clean up
```

```
del frames, errors, mvs, recon, results, interval
del T, Q_50, Q_10
del cv2, exit, np, plt

print("Sucess! Compressed video loaded into current directory")
```