H.264 (MPEG-4 AVC) & JPEG Based Compression & Transmission System

L.B.I.P Thilakasiri – E/16/367

OBJECTIVES

1. To investigate the basic functions of an image coding system.
2. To investigate the basic functions of a block-based video coding system.
3. To investigate the Rate - Distortion optimisation techniques used in video coding.

INTRODUCTION

Hybrid video coding is an advanced coding technique which is derived from both predictive coding and transform coding. Hybrid video coding framework is commonly used in modern video coding standards, e.g., H.26x, MPEG2/4, AVS, HEVC, etc. In this project, a simplified hybrid video codec with coding tools like discrete cosine transformation (DCT), quantization, prediction, and entropy coding are implemented and the impact of each tool on the codec's performance is investigated. Quantization will be done with 3 levels of quantization values. The video coding component comprises of macro block-based coding, basic motion estimation and intra prediction as well. The third part of the project focuses on an Improved hybrid video codec. There, the quantisation process of the codec is expected to be optimized to facilitate transmission in fixed bandwidth environment and obtaining the best QP to meet the given bit-rate.

METHODOLOGY

Stage 1: Basic implementation: Image compression

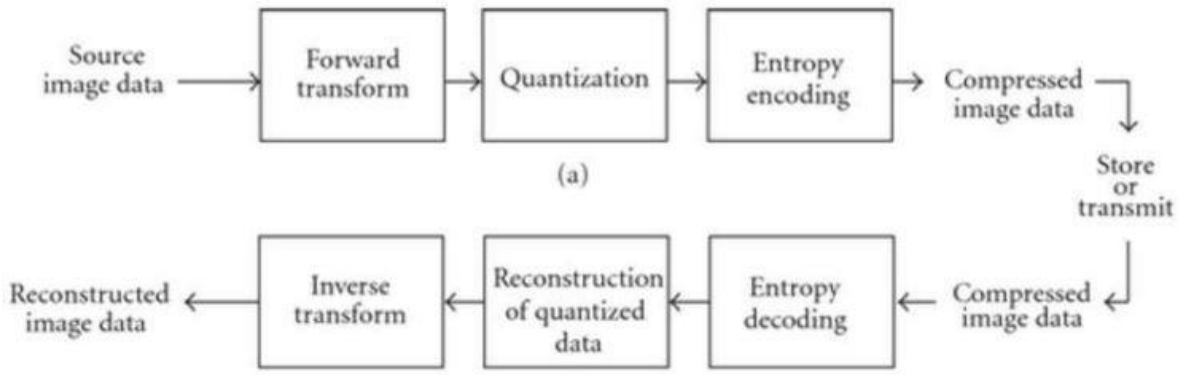


Figure 01: Basic Image Compression System

The targeted image compression system is shown in the above image and the above shown components were implemented individually.

1. Forward Transform

For the forward transform stage, many transforms can be used. Some of them are listed below.

- Walsh Transform
- Hadamard Transform
- Walsh-Hadamard Transform (WHT)
- Haar Transform
- Slant Transform
- KL Transform

The Karhunen-Loeve (KL) transform gives us the optimal transformation between the domains. But, this is not used in practice as there are no fast implementation and that can be an issue when working with large systems with high bitrate and high bandwidth systems.

On the hand, Discrete Cosine Transform (DCT) is usually used for this application as it can be easily coded to be executed fast and it gives us some additional benefits such as filtering out the DC components, AC components and pushing the higher frequency components towards the end of the bit vector.

$$C(u, v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y)\cos\left[\frac{(2x + 1)u\pi}{2N}\right]\cos\left[\frac{(2y + 1)v\pi}{2N}\right]$$

for $u, v = 0, 1, 2, \ldots, N - 1$, and

$$f(x, y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v)C(u, v)\cos\left[\frac{(2x + 1)u\pi}{2N}\right]\cos\left[\frac{(2y + 1)v\pi}{2N}\right]$$

for $x, y = 0, 1, 2, \ldots, N - 1$, where α is

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}} & \text{for } u = 0 \\ \sqrt{\frac{2}{N}} & \text{for } u = 1, 2, \ldots, N - 1. \end{cases}$$

Figure 02: Discrete Cosine Transform (DCT) Equations

The feature extraction by transforming and the way the data points are distributed such that it supports, difference coding or run length coding is another advantage of transforming the Time Domain Signal (Image) to Frequency Domain.

139	144	149	153	155	155	155	155
144	151	153	156	149	146	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

233.1	0.3	-9.8	-7.9	2.1	-0.1	-3.7	1.1
-25.5	-15.9	-3.5	-6.4	-2.9	2.1	-0.7	-1.5
-12.3	-8.5	-0.3	0.1	0.2	0.0	-1.1	-0.2
-6.4	-2.3	-0.4	2.2	0.9	-0.6	0.2	0.4
1.9	-2.2	-0.8	4.3	-0.1	-2.5	1.6	1.5
5.2	-2.0	-1.6	3.4	-0.8	-1.0	2.4	-0.6
2.0	-2.1	-3.3	2.1	-0.5	-0.6	2.3	-0.4
-0.6	0.5	-5.6	0.3	1.9	-0.2	0.2	-0.2

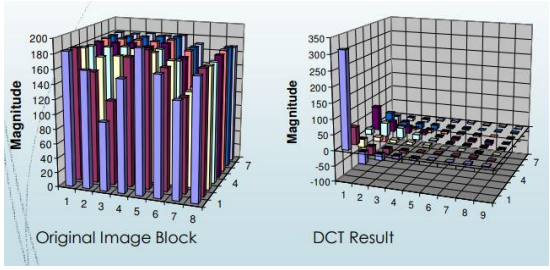


Figure 03: Example of How DCT Removes Redundancies

As shown above the amplitude of the highest frequency components (rare occurring ones) are attenuated as they do not represent a larger percentage of the essential information to recreate the image in a satisfactory level.

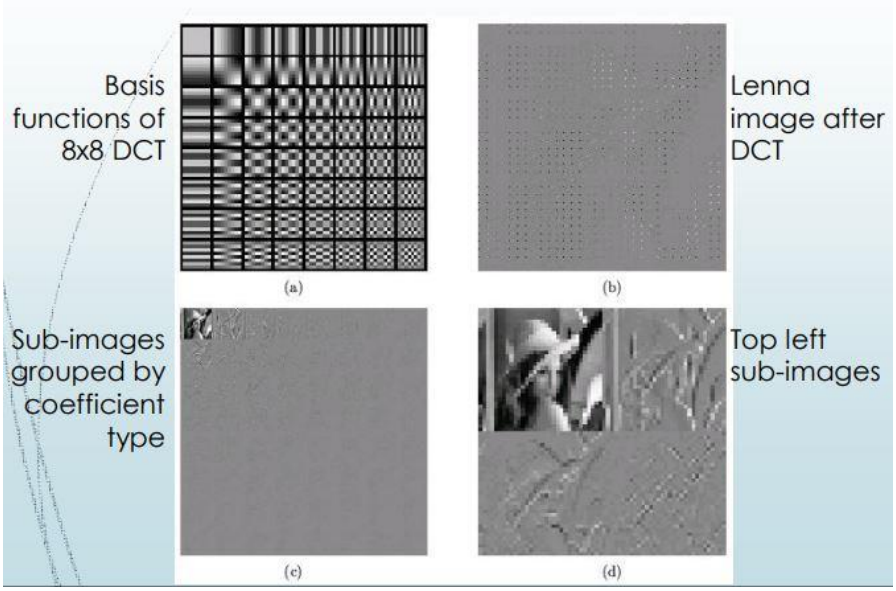


Figure 04: How DCT Transforms the Lenna Image

Thereafter, Quantization and Entropy Coding is performed. Once the Data is transmitted, Decoded and Dequantized, Inverse DCT is performed to recreate the image and display.

$$\textit{Inverse DCT}: \quad x(n) = \frac{1}{N}\sum_{k=0}^{N-1} w(k)F(k)\cos\left(\frac{\pi}{2N}k(2n+1)\right), \qquad 0 \le k < N$$

$$w(k) = \begin{cases} \frac{1}{2}, & k = 0 \\ 1, & 1 \le k < N \end{cases}$$

Figure 05: IDCT Equations

2. Quantization

Quantization is a method involved in lossy compression and it pushes the already existing values into a several windows of values and if these values are rounded, then it will further shrink the amount of information included. Quantization can be done in two ways, namely, Level Quantization and using a Quantization matrix. Both these methods were implemented in the project and results are compared in the implementation section of the report.

I. Quantization by levels

Here, three levels of quantization were chosen to match the required, High, Medium and Low-Quality outputs and they were, Q = 32, Q = 16, Q = 8 respectively.

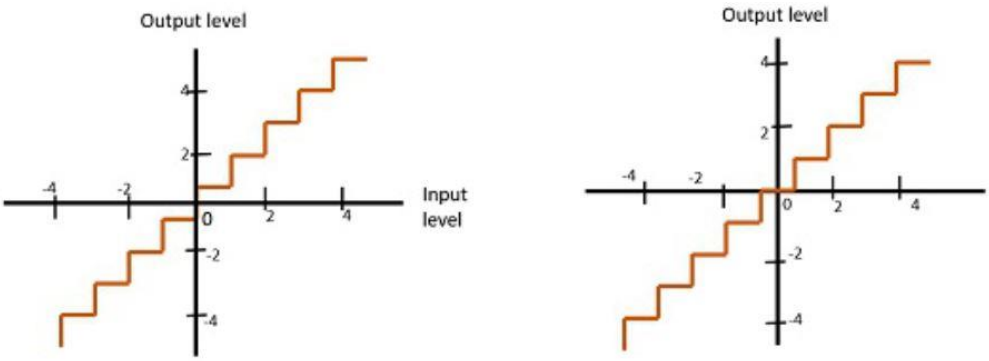


Figure 06: Quantization by Levels

Here the image values are rounded up and allocated into one of the chosen number of quantization levels. The higher the number of quantization levels, the sharper information it can preserve and the higher the memory requirement it will have. The opposite is true for low number of quantization levels.

II. Quantization by matrix

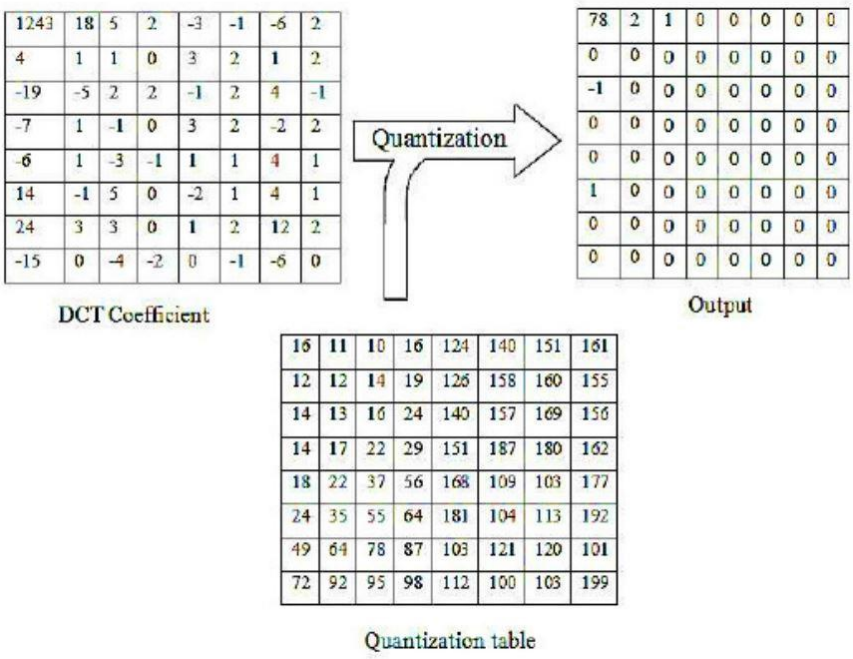


Figure 07: Quantization Process

In this approach, the DCT calculated matrix is quantized using another matrix of choice. There are several options corresponding to different quantization levels and Q10, Q50, Q90 quantization matrices were used in the implementation of the system. The first few elements are usually made smaller to preserve the low frequency data and the elements that are the furthest are made large to attenuate the amplitudes of the higher frequency components, which do not carry significant information useful to the image reconstruction. The following image shows the negligible

difference of the use of higher frequency components of an image. Therefore, zeroing them out does not affect the user experience greatly.

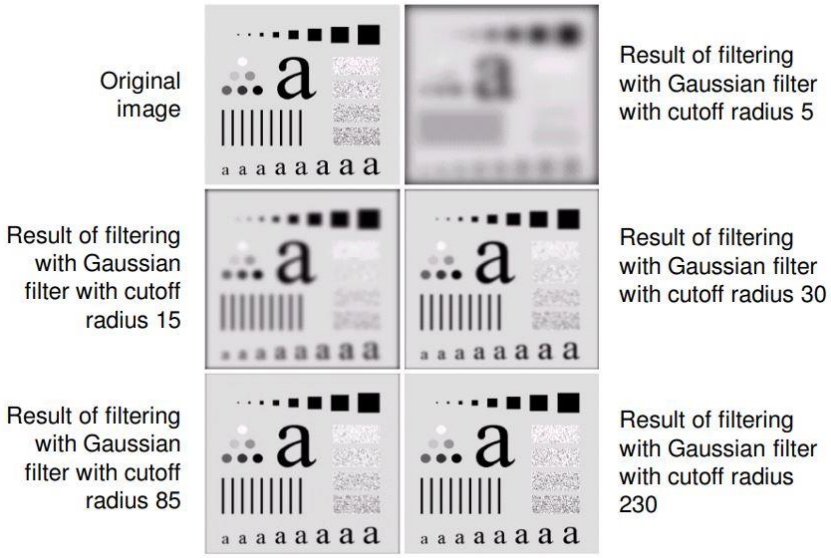


Figure 08: How Images are Perceived for Different Filters

These matrices are also called as "Quality Matrices". Q-10 means quantization matrix offering 10% quality of input image (i.e., quality of reconstructed image degrades). It has very high compression ratio. Similarly, Q-90 matrix offers very high quality (90%) but it has a drawback of less compression ratio. Therefore, Q10, Q50, Q90 corresponds to Low, Medium, High-Quality Compression respectively.

```
Q10 = ([[80,60,50,80,120,200,255,255],
        [55,60,70,95,130,255,255,255],
        [70,65,80,120,200,255,255,255],
        [70,85,110,145,255,255,255,255],
        [90,110,185,255,255,255,255,255],
        [120,175,255,255,255,255,255,255],
        [245,255,255,255,255,255,255,255],
        [255,255,255,255,255,255,255,255]])
```

Figure 09: Q10 Quantization Matrix

```
Q50 = ([[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],
        [72,92,95,98,112,100,130,99]])
```

Figure 10: Q50 Quantization Matrix

```
Q90 = ([[3,2,2,3,5,8,10,12],
        [2,2,3,4,5,12,12,11],
        [3,3,3,5,8,11,14,11],
        [3,3,4,6,10,17,16,12],
        [4,4,7,11,14,22,21,15],
        [5,7,11,13,16,12,23,18],
        [10,13,16,17,21,24,24,21],
        [14,18,19,20,22,20,20,20]])
```

Figure 11: Q90 Quantization Matrix

3. Entropy Coding

Entropy coding is a method of lossless coding and it is used regardless of media's specific characteristics. The data is input as the sequence and depending on the number of occurrences or the probability of each symbol, individual sequences are assigned to the existing ones, which then are used for encoding and transmission. The decoding is completely reversible as they are lossless and run-length, Huffman, Arithmetic coding schemes are examples for entropy coding. Huffman coding was used in this project as the encoding method.

Huffman Encoding

Huffman coding is a variable length type coding system that uses the statistical properties of each symbol to assign sequences for them. This is more efficient when the symbol probabilities vary widely and fewer bits are used for the more frequent symbols and opposite approach is taken for less frequent symbols. The algorithm is as follows.

1. Initialization: Put all symbols on a list sorted according to their frequencies.
2. Repeat until the list has only one symbol left:
    a. From the list pick two symbols with lowest frequency counts
    b. Form Huffman subtree that has these two symbols as child nodes and create a parent node.
    c. Assign the sum of children's frequency counts to the parent and insert it to the list such that the order is maintained.
    d. Delete the children from the list.
3. Assign a codeword for each lead based on the path from the root.

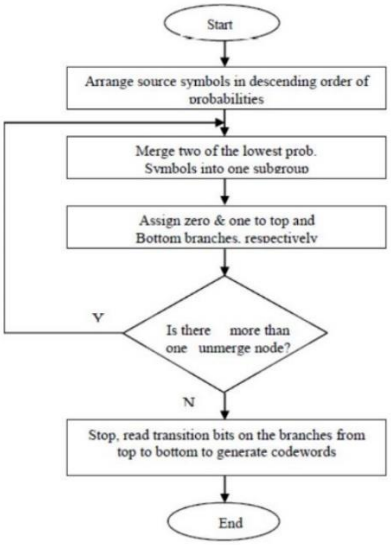


Figure 12: Huffman Encoding Flow Chart

Huffman is a greedy type algorithm and it yields the overall best solution. But this does not give us the optimal solution always. From the Shannon's information theorem, we can obtain the optimal rate for transmission and usually it's a bit lower than what we achieve from Huffman coding.

Run Length Coding

Run length is another coding scheme that was used in the implementation stage and it is useful when consecutive symbols in the sequence are identical. The basic idea can be interpreted as follows.

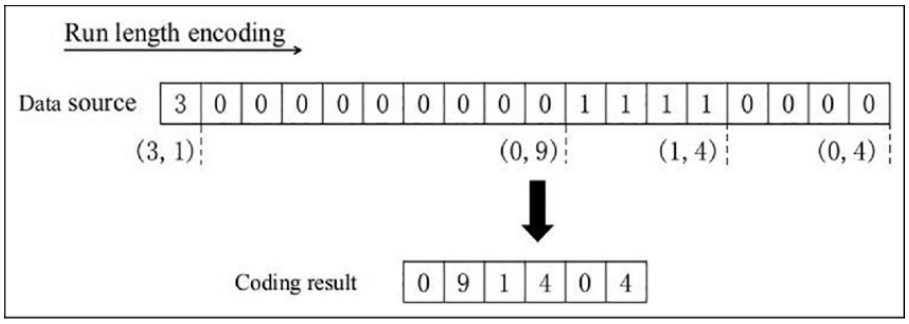


Figure 13: Run Length Coding Example

As shown above, the symbol is followed by the number of its occurrences and this is useful when coding the Zigzag pattern of the DCT computer matrix as it has many consecutive zeroes.

Stage 2: Basic implementation: Video compression

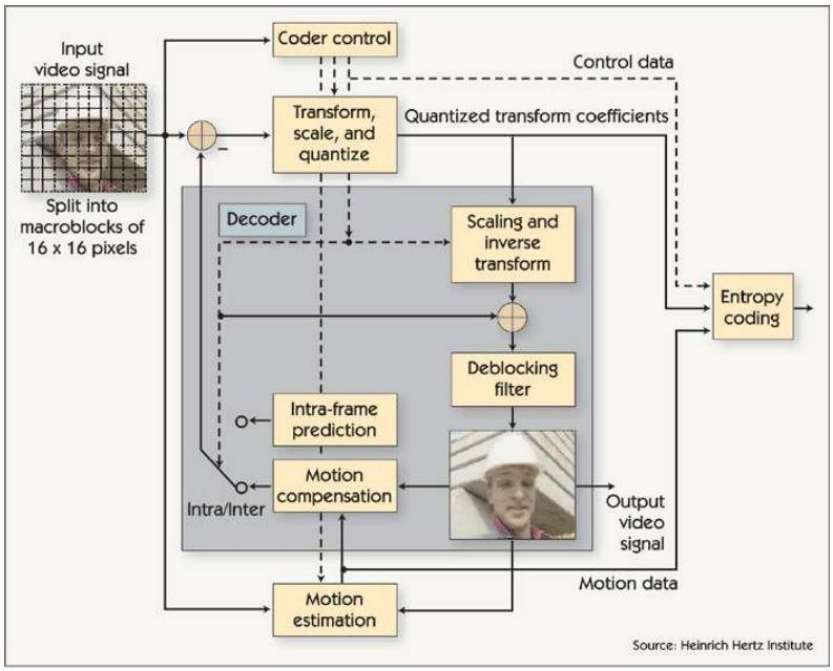


Figure 14: Block Diagram of H264 Hybrid Video Codec

All of the same concepts listed above were used in the implementation of the video coding system and some other techniques which are not listed above was used as well. Those methodologies are as follows.

4. Motion Estimation and motion Vector Generation

A video is a set of images put together and the same compression techniques can be used for video compression as well. But there are other methods that we can use to further reduce the bandwidth requirement and speed up the transmission process. The spatial redundancies of the frames are addressed by the coding methods mentioned above and the temporal redundancies should also be addressed to further reduce the memory requirements. Motion estimation and motion vector generation is used for this purpose. We consider a set of consecutive frames and we look for similarities between them to encode that information only so that the compression is better. We can use Current, Previous, and Future frames for this purpose and these are called I, P, B frames respectively.

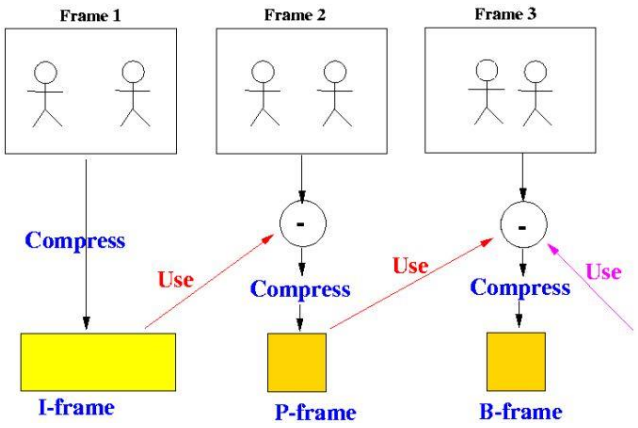


Figure 15: Use of IPB Frames for Coding

To extract these information, we use motion estimation techniques to obtain, which information and which blocks have been displaced in the future frames and by how much. Some of the Motion estimation techniques are listed below.

- Sequential Search
- Logarithmic Search
- Hierarchical Search

The frames are first divided into blocks of choice and these methods are used to figure out which block has ended up where in the next frame and the direction displacement vector corresponding to the movement is called as the Motion Vector and that is also recorded to be transmitted. This block matching process can be done across a limited number of neighbour frames and that is choice for the developer to make.

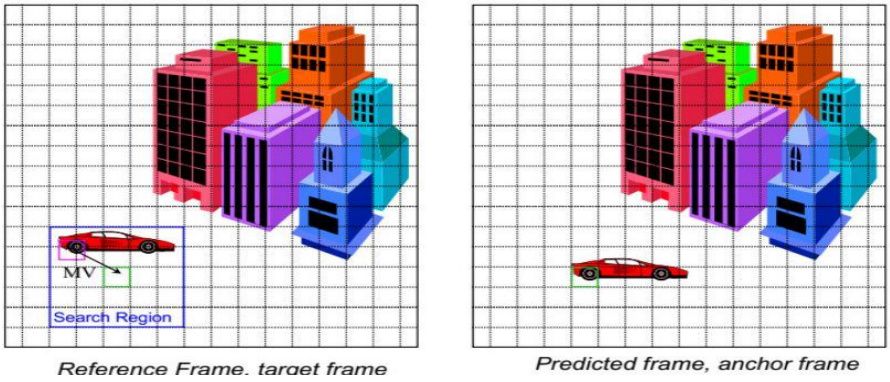


Figure 16: Motion Estimation and Motion Vector Search

Therefore, now we don't have to transmit all the frames one by one, instead we can just send the reference frame, the next frames motion information and the remaining difference between the reference frame and the next frame that is needed to properly recreate the frame on the receivers side. This difference is called the Residue and these operations have resulted in,

a. The original frame which the motions were obtained (Reference Frame)
b. Motion Vector
c. Residue

And we only have to transmit these three instead of the whole frame. Residue is also known as 'Control Data' and the Reference frame & Motion Vector is also known as 'Motion Data'. The best matching motion block is chosen by the calculation of loss functions and Mean Squared Error (MSE), Mean Absolute Difference (MAD), Sum of Absolute Differences (SAD) are used in common. SAD was used as the loss function in the implementation.

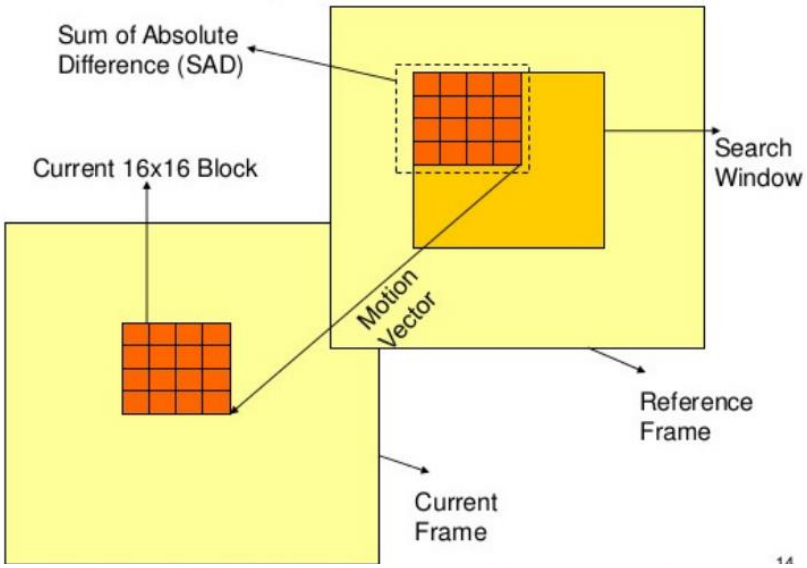


Figure 16: Block Matching and SAD Calculation

The residue is important for the acceptable recreation of the image and the created image with just the Blok Matching and Reference frame is called the 'Compensated Image'. As you can see below, it results in a blocky image and the residue must be added to create the required form of the next frame.

Figure 17: Frame 01, Frame 02 & Their Resultant Compensated Image of Frame 02

Stage 3: Improved Hybrid Video Codec

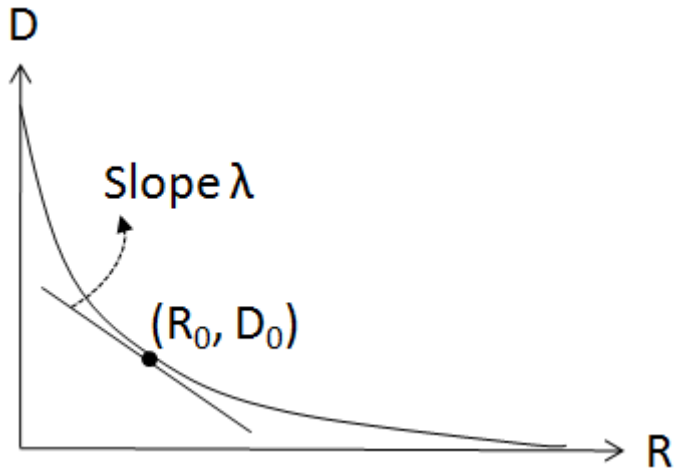


Figure 18: Rate Distortion Curve

Rate distortion curve shows that the more you compress something the more the distortion there will be. Therefore, finding an optimal point the best quality is achieved while the rate is not unnecessarily large, is important. For the distortion calculation, several quality metrics can be used and some of them are listed below.

- Mean Square Error (MSE)
- Peak Signal to noise Ratio (PSNR)
- Structural Similarity Index (SSIM)
- VQM (NTIA General Model)

The first three listed above was used in the implementation to obtain Rate-Distortion Curves of the system and the optimal point was chosen for the transmission system. These values were calculated for the different quantization levels that were discussed earlier and 10 different quantization levels were used to obtain the results.

IMPLEMENTATION & RESULTS

When I first started implanting the system, I used the Python Language and completed the Image compression component of the project. I created individual files for the different parts of the system and even though all of them worked when run individually, the output was not accurate when run together. During the debugging process, I realized that my Huffman encoding code was the one causing the inaccuracy. Therefore, I switched to Matlab as it has a built-in function for Huffman Coding and I have listed both my implementations in Matlab and Python in this report. The outputs of individual python scripts are also shown below as they all worked as intended when run individually.

Matlab Implementation

Stage 1: Basic implementation: Image compression

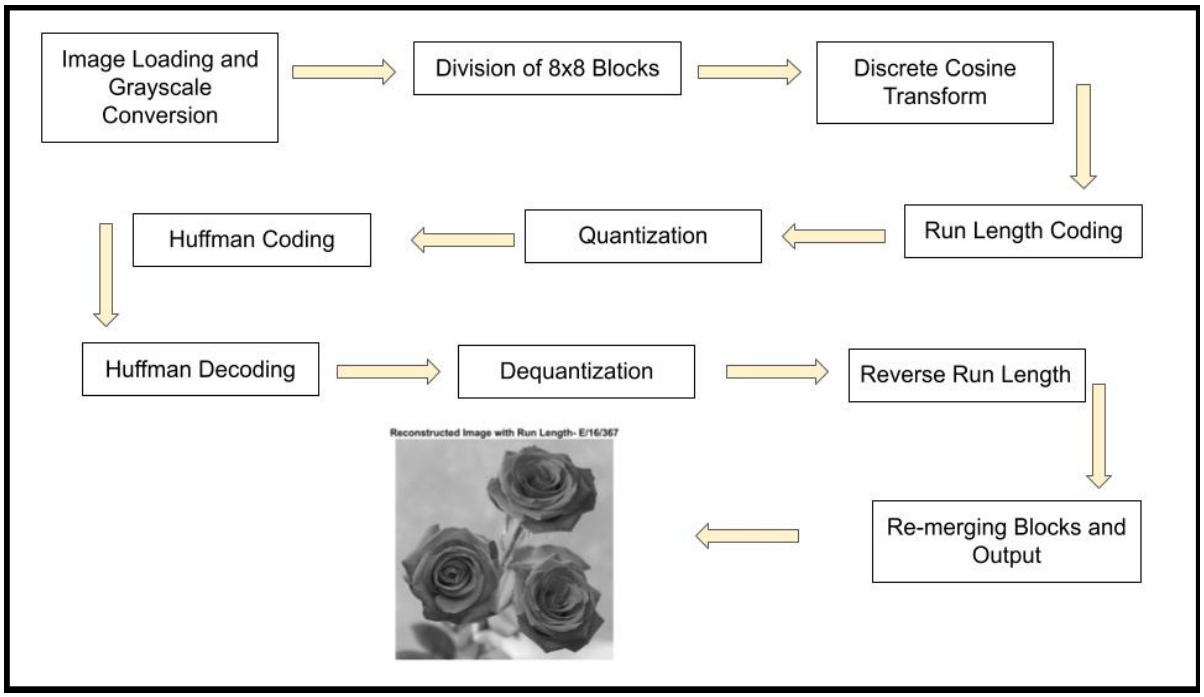


Figure 19: The Implemented Complete Image Compression System

Implemented Matlab Files

- main.m
- mainWithRunLength.m
- run_length_coding.m
- run_length_decoding.m

A. Reading the Image and Grayscale Conversion

Figure 20: Original Image and Grayscale Version

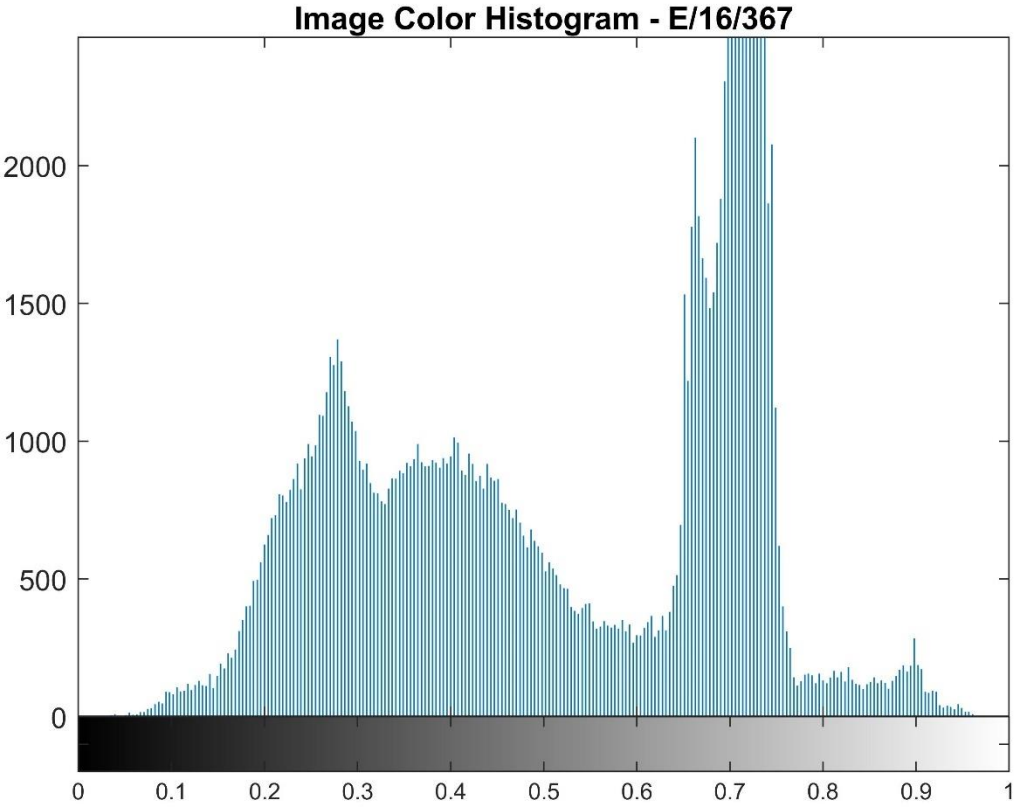


Figure 21: Grayscale Image Histogram

B. Division of 8x8 Blocks
C. Discrete Cosine Transform

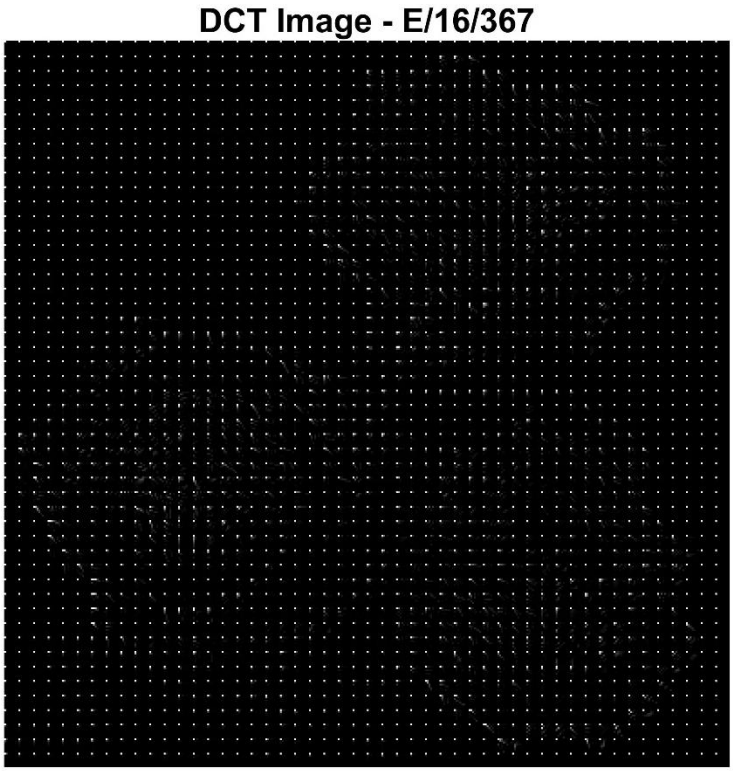


Figure 22: DCT Version

D. Quantization

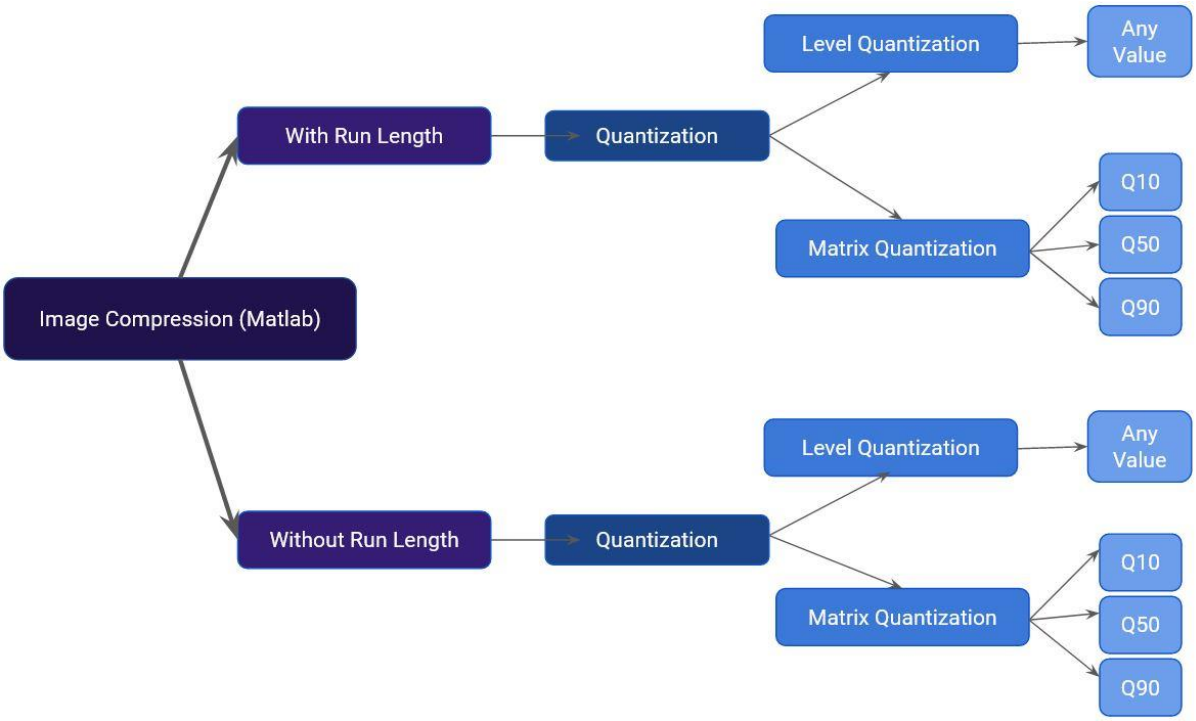


Figure 23: All the Different Forms of Coding and Quantization Possible in the Implemented System

Image Compression was implemented in two ways with one having the Run Length step and other without it. The following are the combinations of quantization and run length coding that can be achieved by the system which was implemented. User input was taken to determine the Quantization method and the Level as shown below.

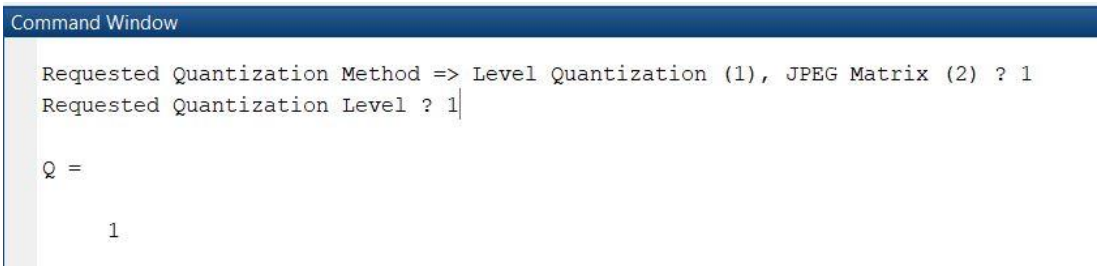


Figure 24: Taking User Input to Determine the Quantization Method and Level

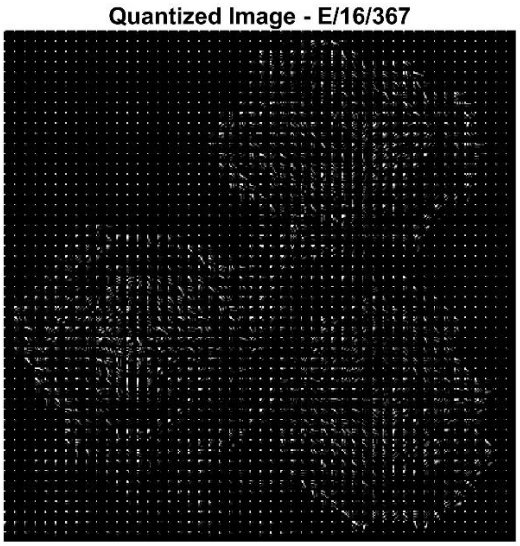


Figure 25: 8 Level Quantized Image from the 'With Run Length' Implementation

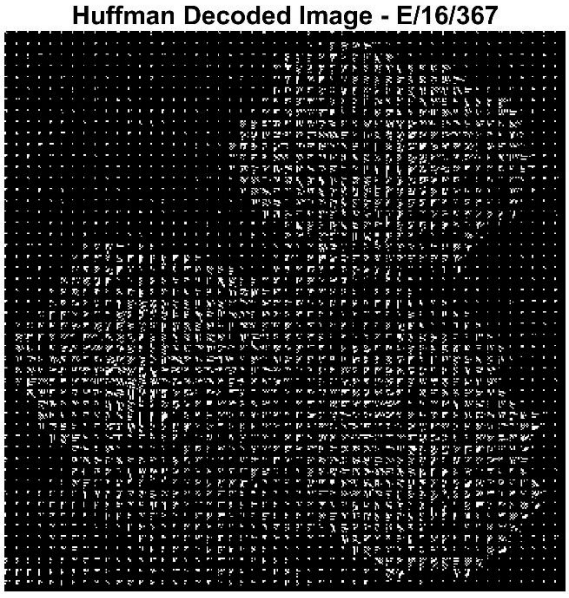


Figure 26: 8 Level Huffman Decoded Image from the 'With Run Length' Implementation

Quantization using levels saves a lot of information when quantization is done to the DCT converted image. Otherwise, we lose a huge amount of information by quantization before DCT application. That is shown below.

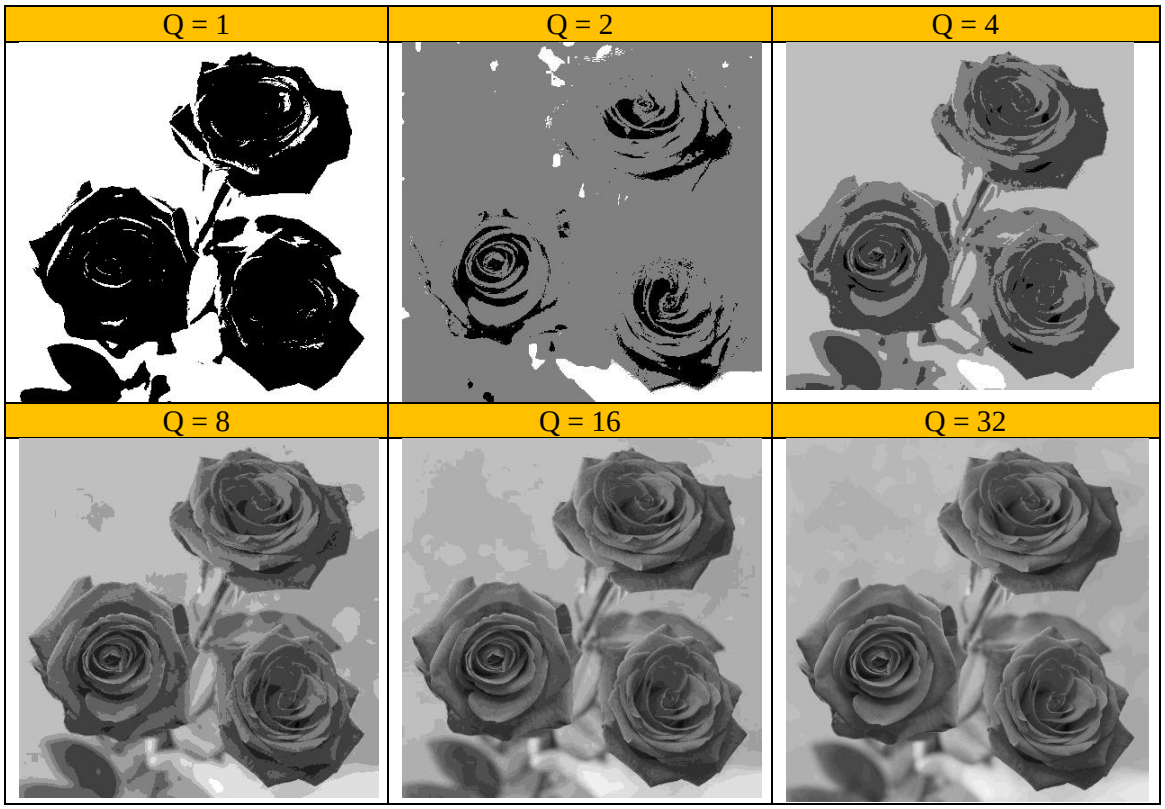
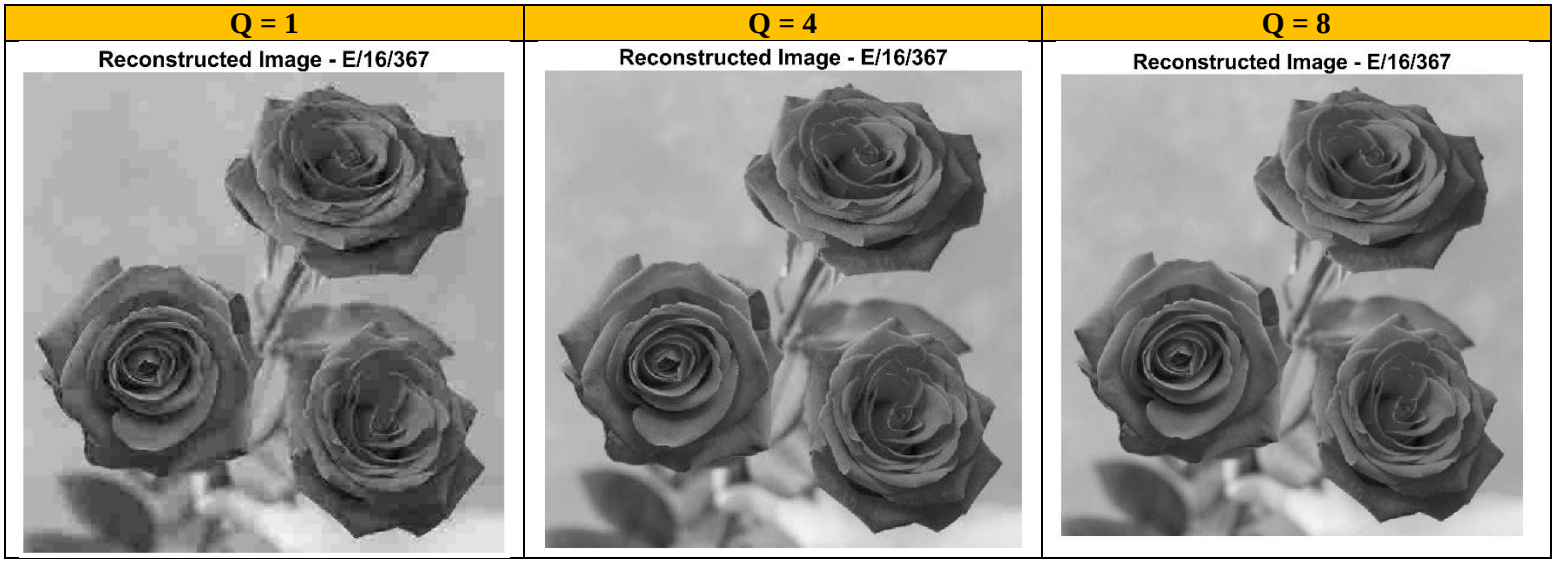


Figure 27: Gray Image for Different Level Quantization Levels if Quantization was Carried Out Before DCT Conversion

E. Re-merging Blocks and Output

(i) Level Quantization Outputs



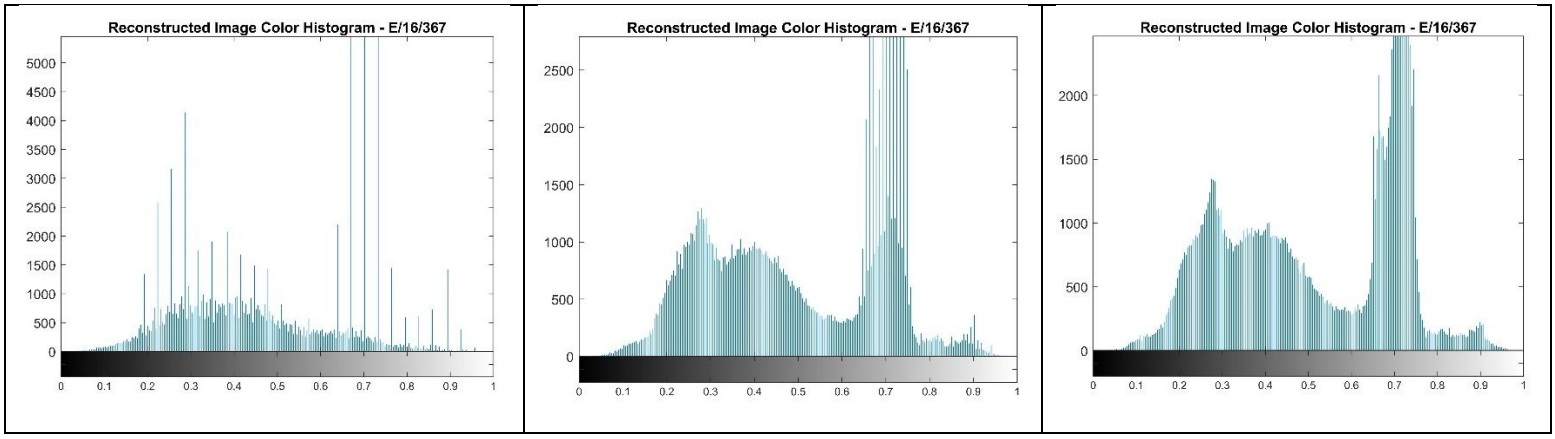


Figure 28: Reconstructed Images & Their Histograms for Different Level Quantization Levels

(ii) Matrix Quantization Outputs

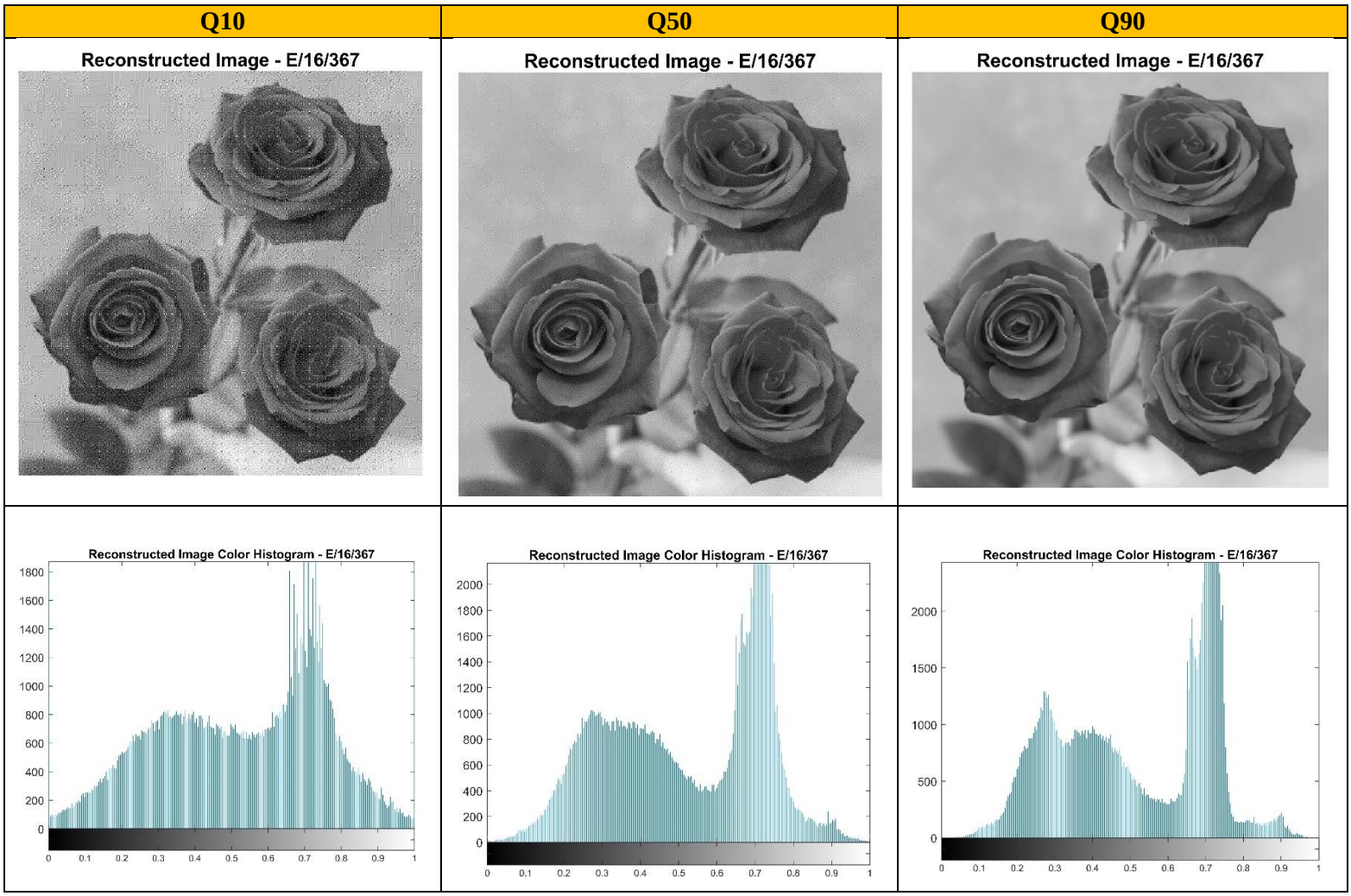


Figure 29: Reconstructed Images & Their Histograms for Different Matrix Quantization Levels

From the histograms we can observe that the low-quality quantized images have resulted in a slightly left shifted histograms which represents that their Black Color component has increased, i.e., the 0 value in grayscale or near zero values have increased decreasing the file size.

Stage 2: Basic implementation: Video compression

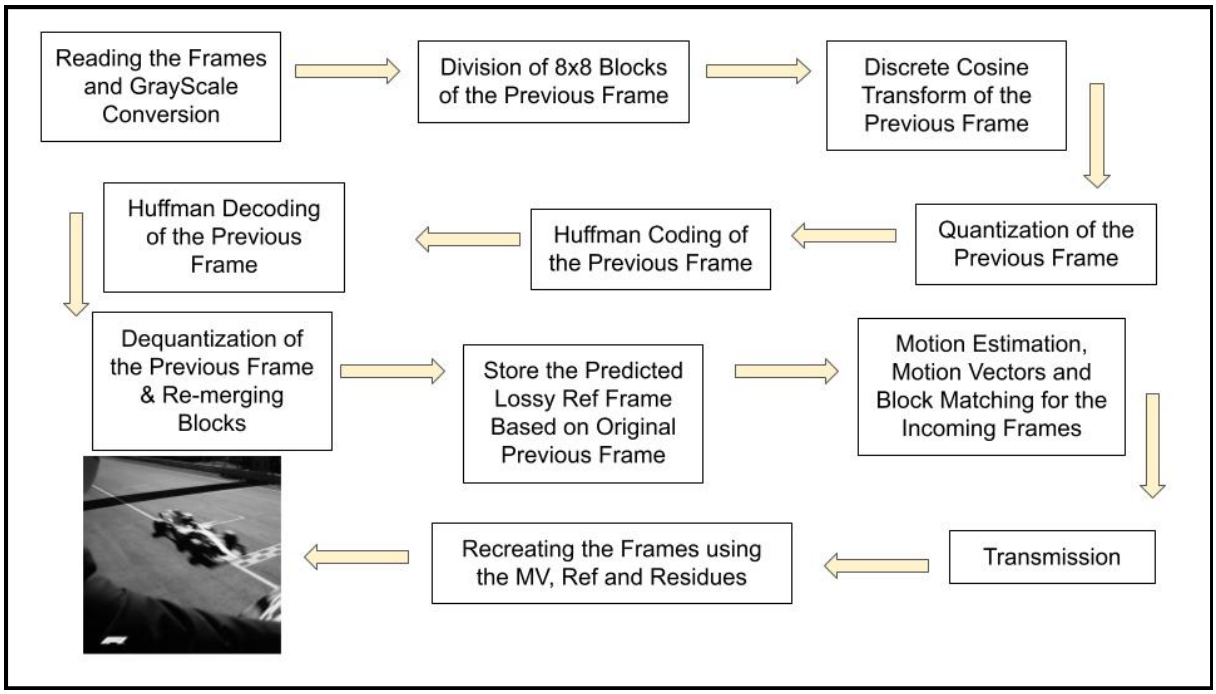


Figure 30: The Implemented Complete Video Compression System

Implemented Matlab Files

- main_vid.m
- motionVec_and_Residue.m
- SAD_Calc.m
- Quantization.m
- motion_estimation.m
- huffman_encode.m
- huffman_decode.m
- deQuantization.m
- compensated_img.m
- img_transmission_process.m

A. Reading the Frames and Grayscale Conversion

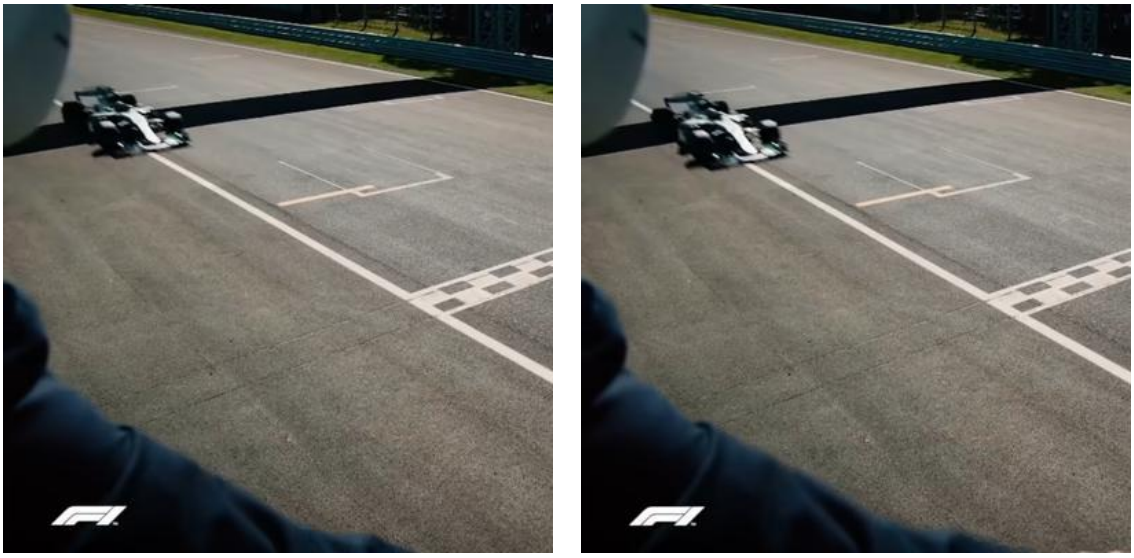


Figure 31: First & Second Frames Used for the System

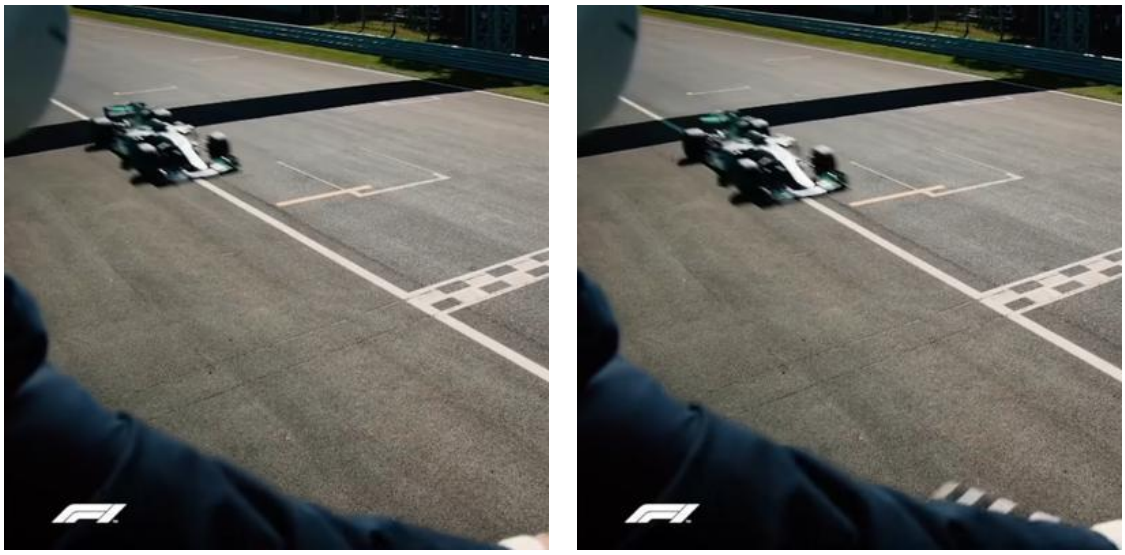


Figure 32: Third & Fourth Frames Used for the System

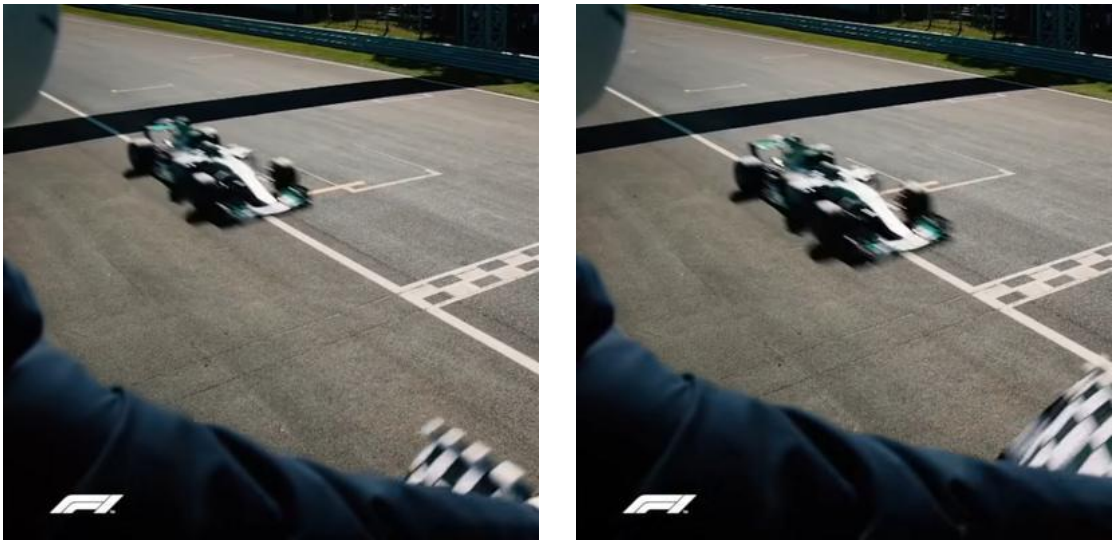


Figure 33: Fifth & Sixth Frames Used for the System

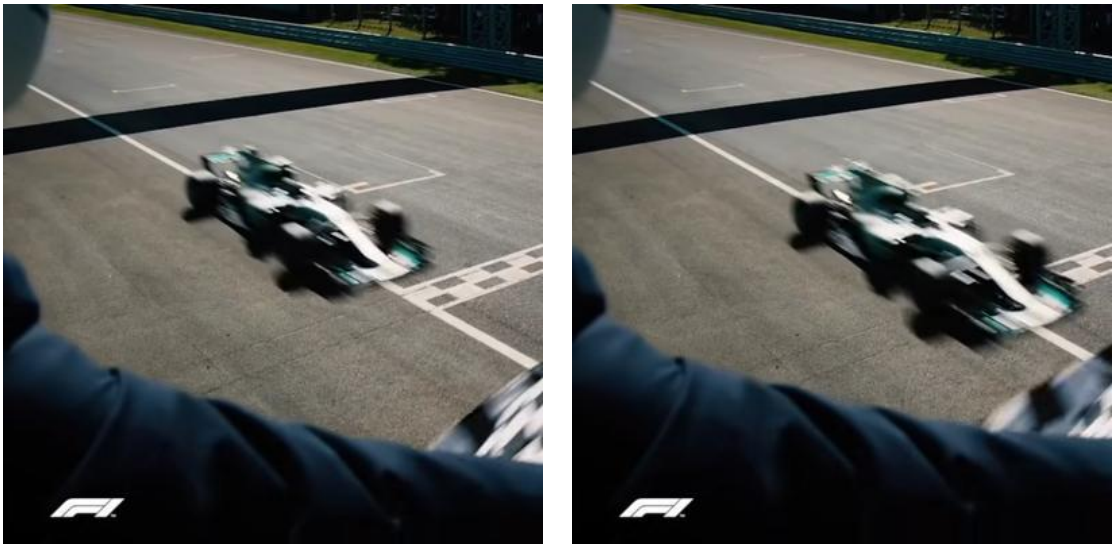


Figure 34: Seventh & Eighth Frames Used for the System

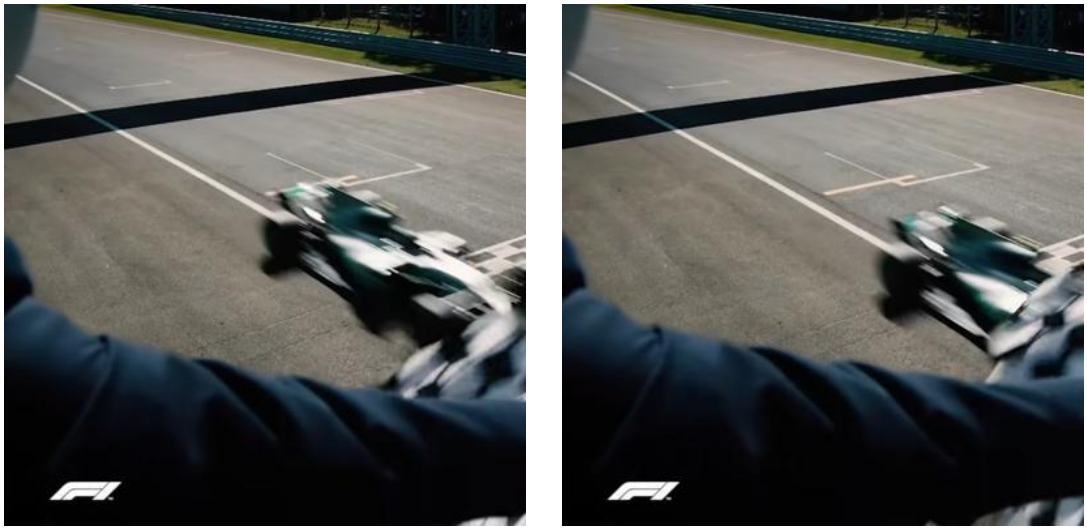


Figure 35: Nineth & Tenth Frames Used for the System

B. Choosing the Previous Frame as Reference

Dequantizing is a step in the decoding and recreation of the frames at the receiver's side. But, after quantization, the system can not recreate the same image at the decoder end. Therefore, if we had used the original image input to the system as the reference frame, the output will not be the same as we intended to as the residues and Motion estimations are done for the original frame. Therefore, to mitigate this issue, we use the quantized frame of the original as the reference. User input was also taken as before to determine which quantization method must be used and which level should be chosen.

```
Command Window

Requested Quantization Method => Level Quantization (1), JPEG Matrix (2) ? 1
Requested Quantization Level ? 1

Q =

     1
```

Figure 36: Taking User Input to Determine the Quantization Method and Level

As there are 10 frames, the intra coding is done only 9 times as the first frame is sent as it is without motion estimation. Those 10 intra frames are shown below.

Figure 37: Received Intra Frames Based on First & Second Frames

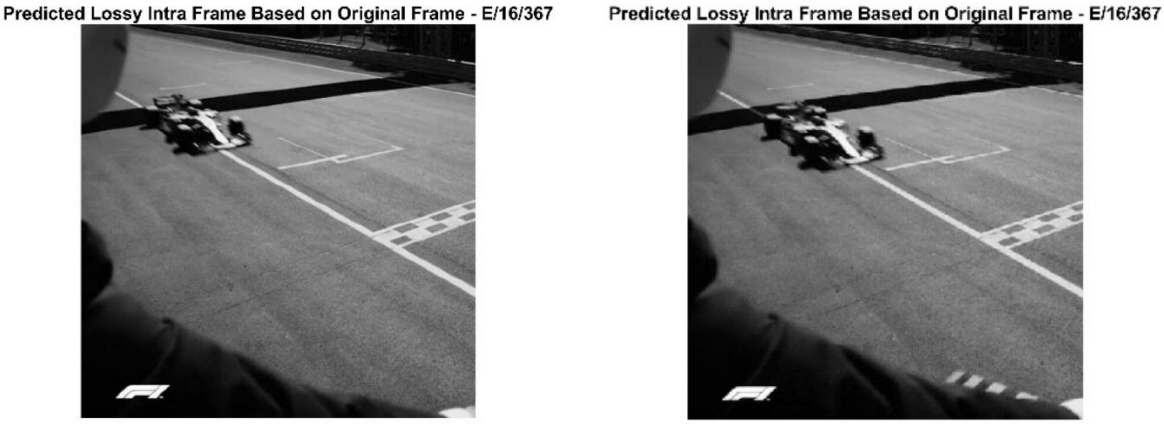


Figure 38: Received Intra Frames Based on Third & Fourth Frames

Figure 39: Received Intra Frames Based on Fifth & Sixth Frames

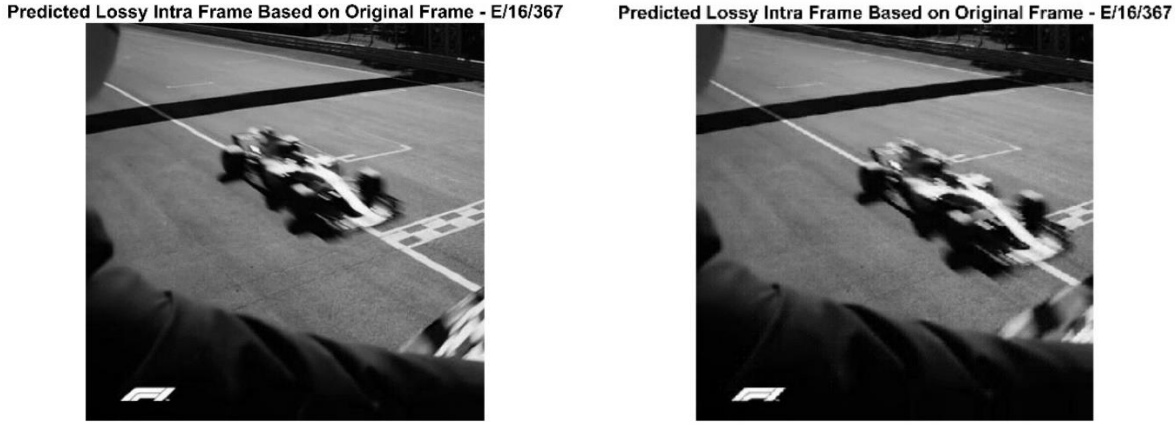


Figure 40: Received Intra Frames Based on Seventh & Eighth Frames

Figure 41: Received Intra Frames Based on Nineth & Tenth Frames

C. Store the Predicted Lossy Ref Frame Based on Original Previous Frame

As mentioned above, Motion prediction is done nine times and out of those, the received frames at the motion estimation stage is shown below.

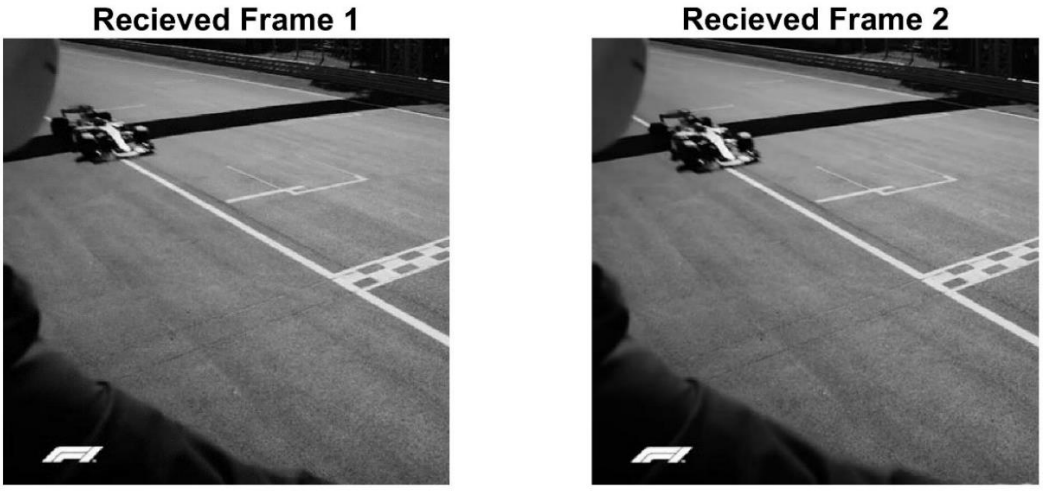


Figure 42: Received Frames for the First Motion Estimation

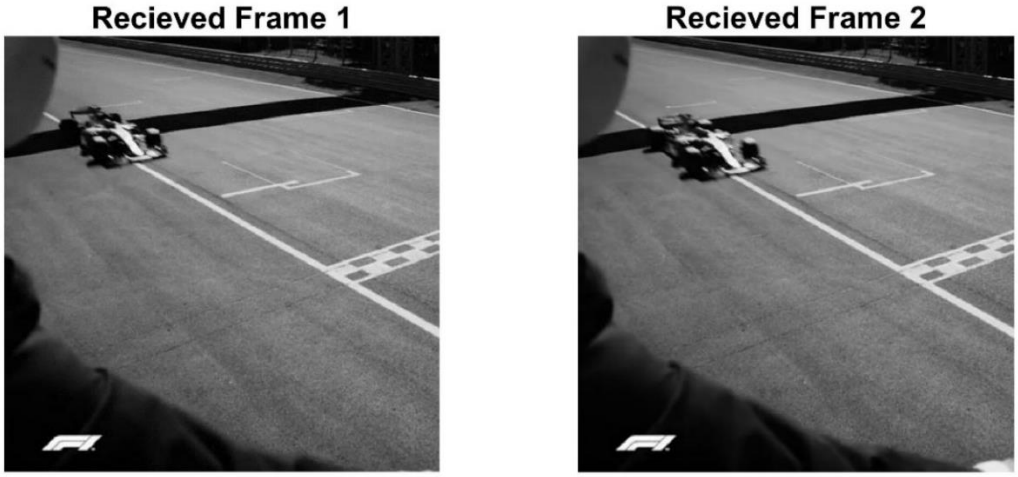


Figure 43: Received Frames for the Second Motion Estimation

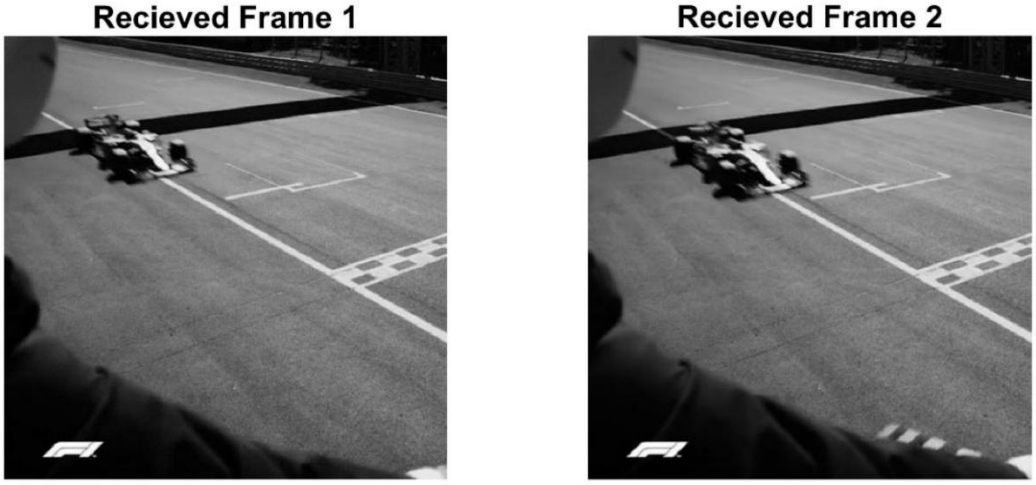


Figure 44: Received Frames for the Third Motion Estimation

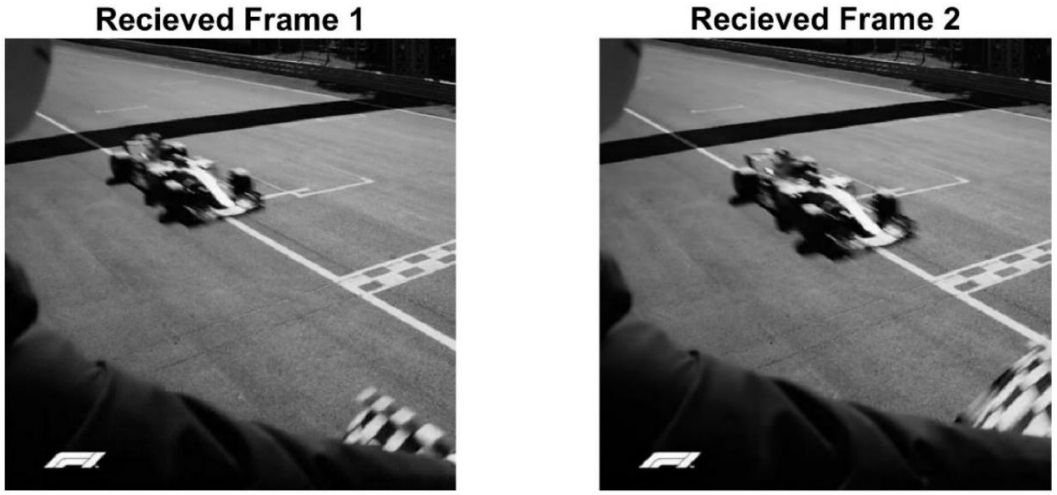


Figure 45: Received Frames for the Fourth Motion Estimation

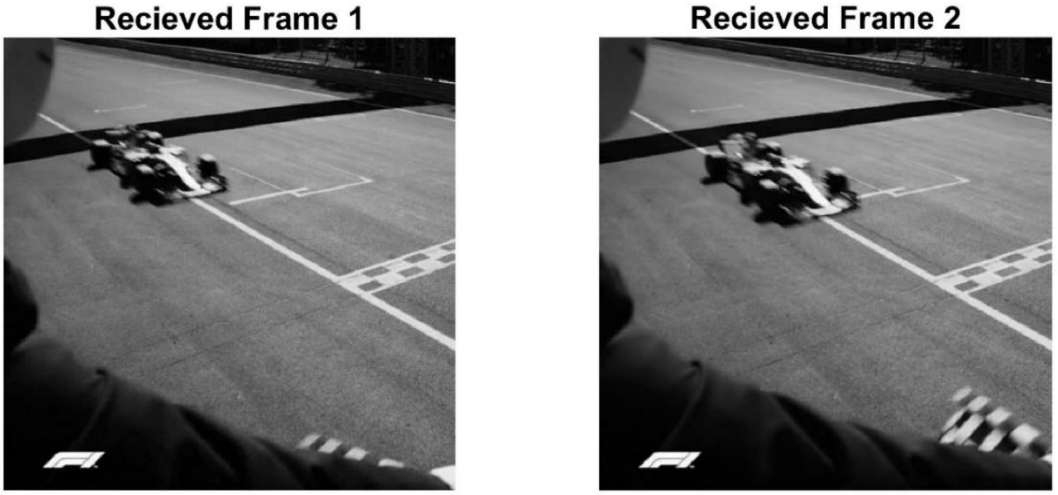


Figure 46: Received Frames for the Fifth Motion Estimation

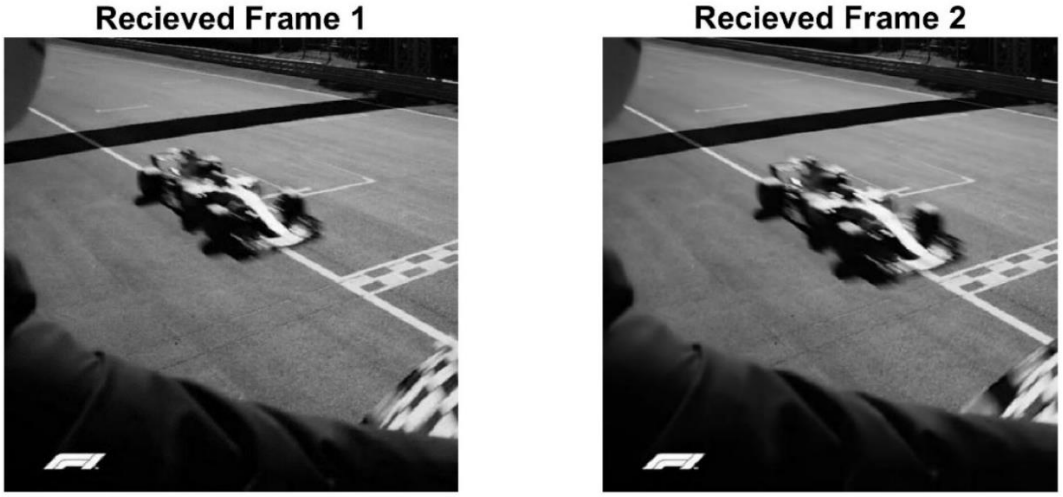


Figure 47: Received Frames for the Sixth Motion Estimation

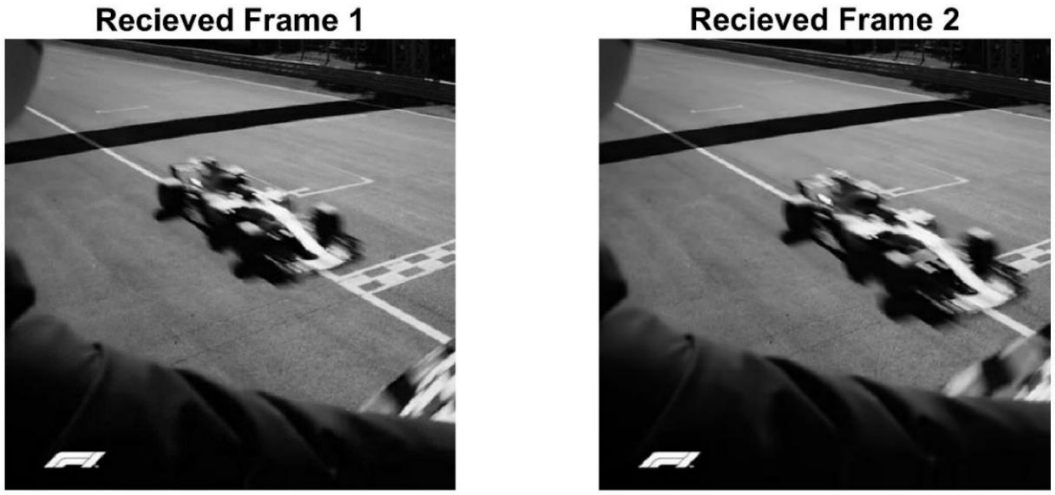


Figure 48: Received Frames for the Seventh Motion Estimation

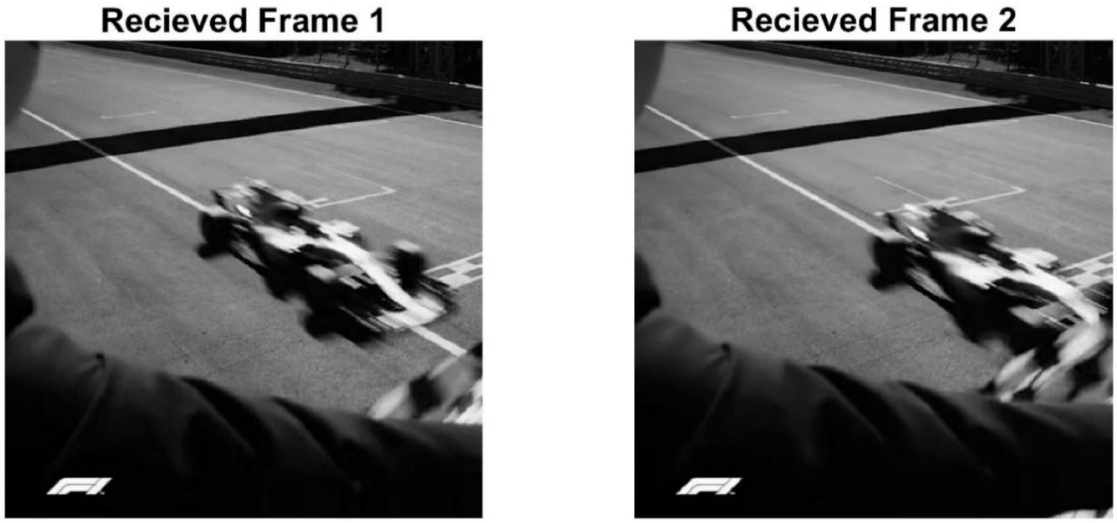


Figure 49: Received Frames for the Eighth Motion Estimation

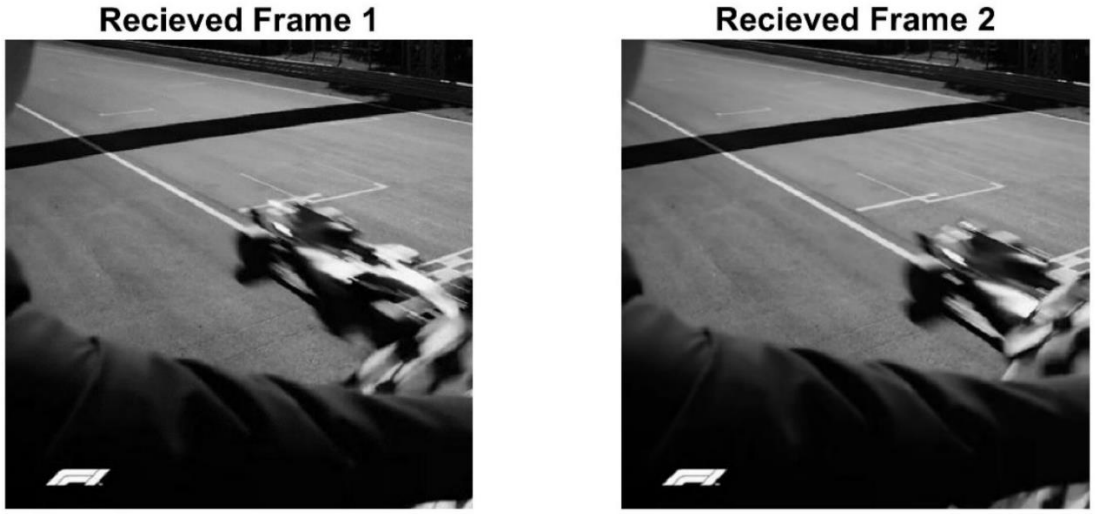


Figure 50: Received Frames for the Nineth Motion Estimation

D. Motion Estimation - Motion Vectors and Block Matching
E. Recreating the Frames using the MV, Ref and Residues

We can observe the distortion in the compensated image as expected.

(Please zoom in if not clearly visible)

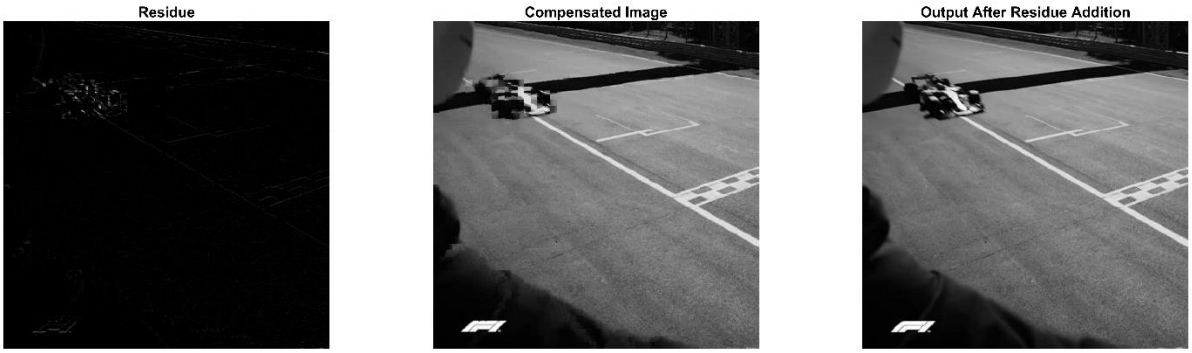


Figure 51: Residue, Compensated & Recreated Images for the First Motion Estimation

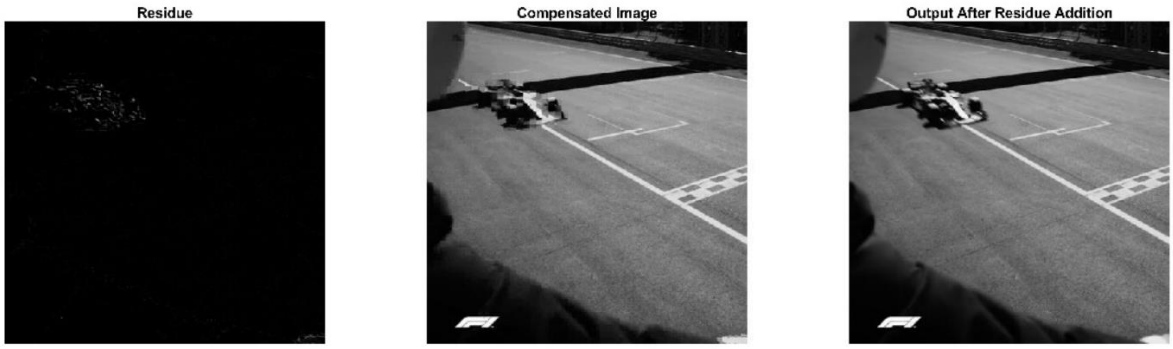


Figure 52: Residue, Compensated & Recreated Images for the Second Motion Estimation

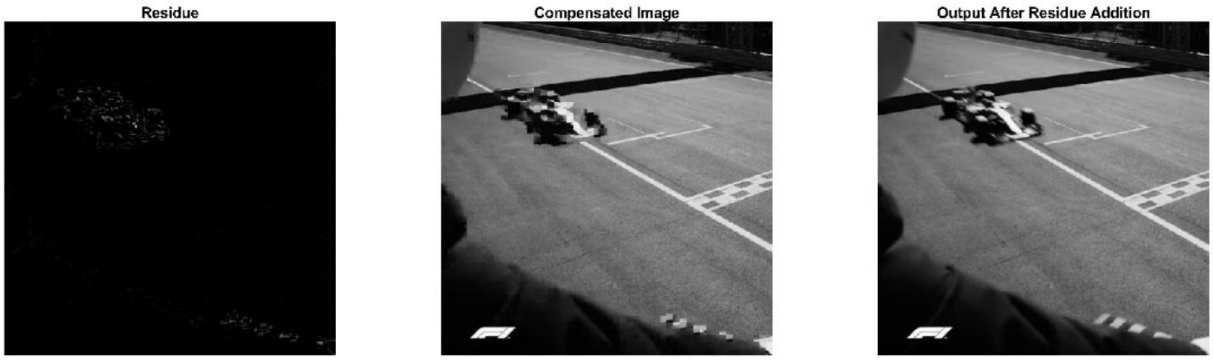


Figure 53: Residue, Compensated & Recreated Images for the Third Motion Estimation

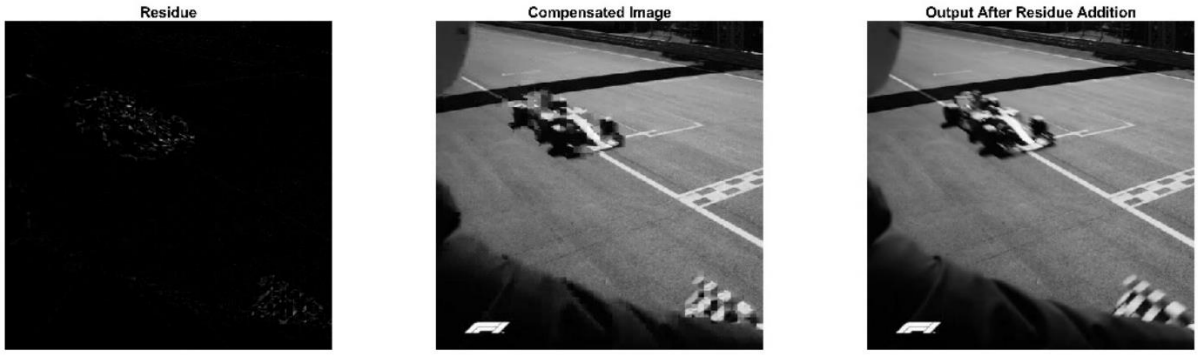


Figure 54: Residue, Compensated & Recreated Images for the Fourth Motion Estimation

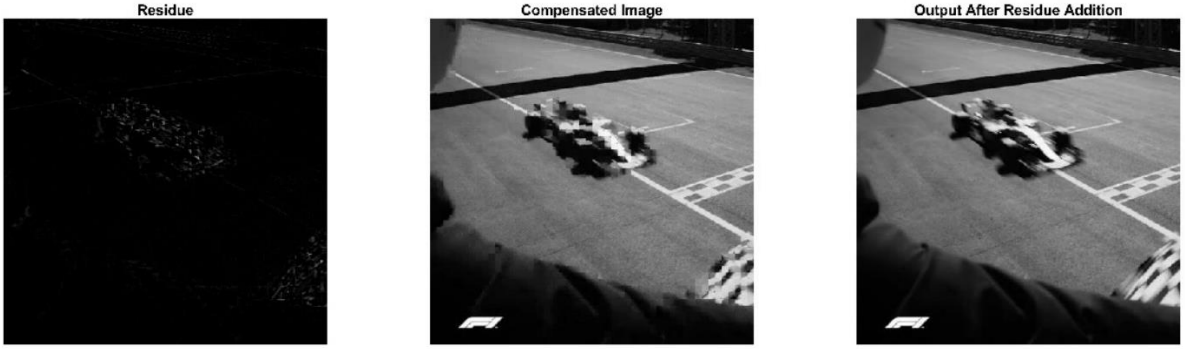


Figure 55: Residue, Compensated & Recreated Images for the Fifth Motion Estimation

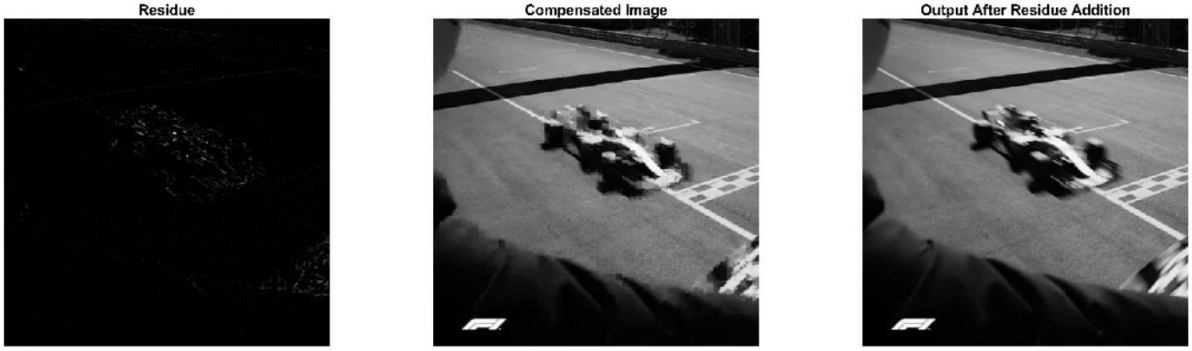


Figure 56: Residue, Compensated & Recreated Images for the Sixth Motion Estimation

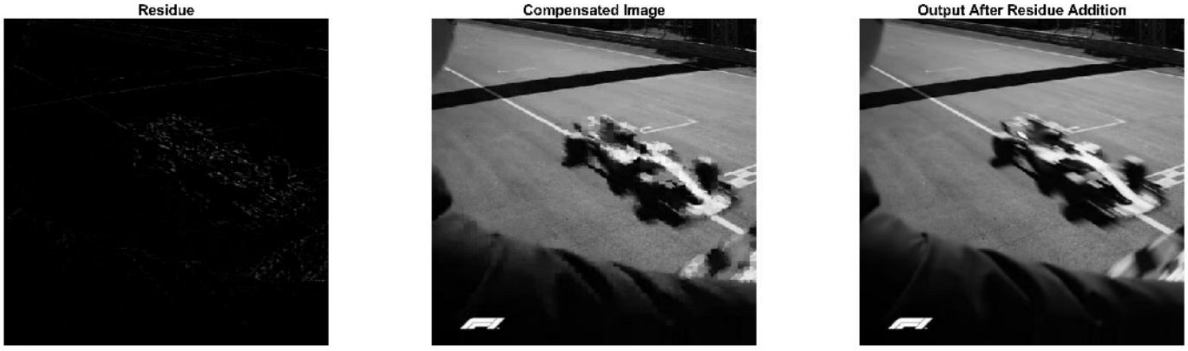


Figure 57: Residue, Compensated & Recreated Images for the Seventh Motion Estimation

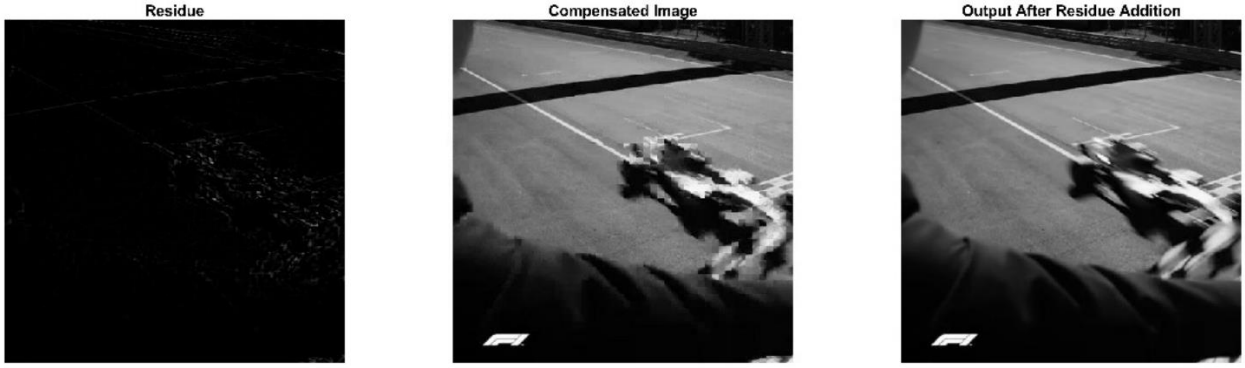


Figure 58: Residue, Compensated & Recreated Images for the Eighth Motion Estimation

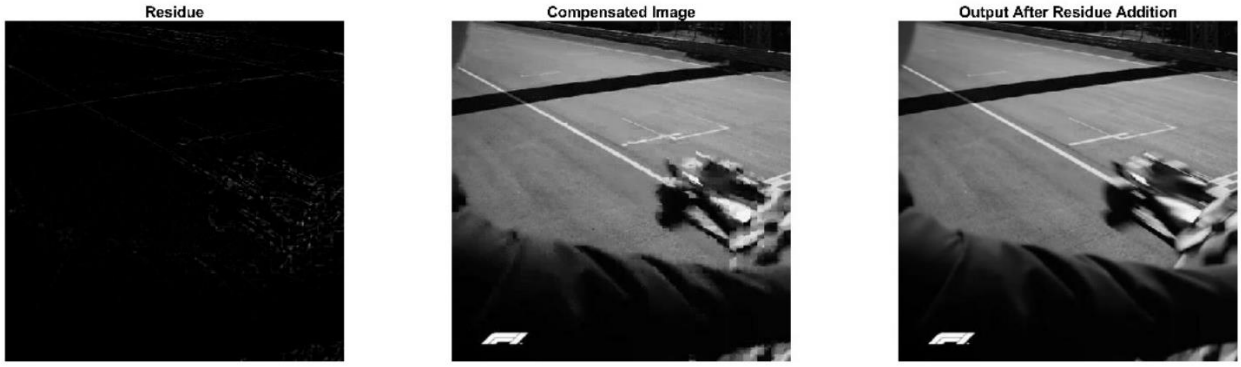


Figure 59: Residue, Compensated & Recreated Images for the Nineth Motion Estimation

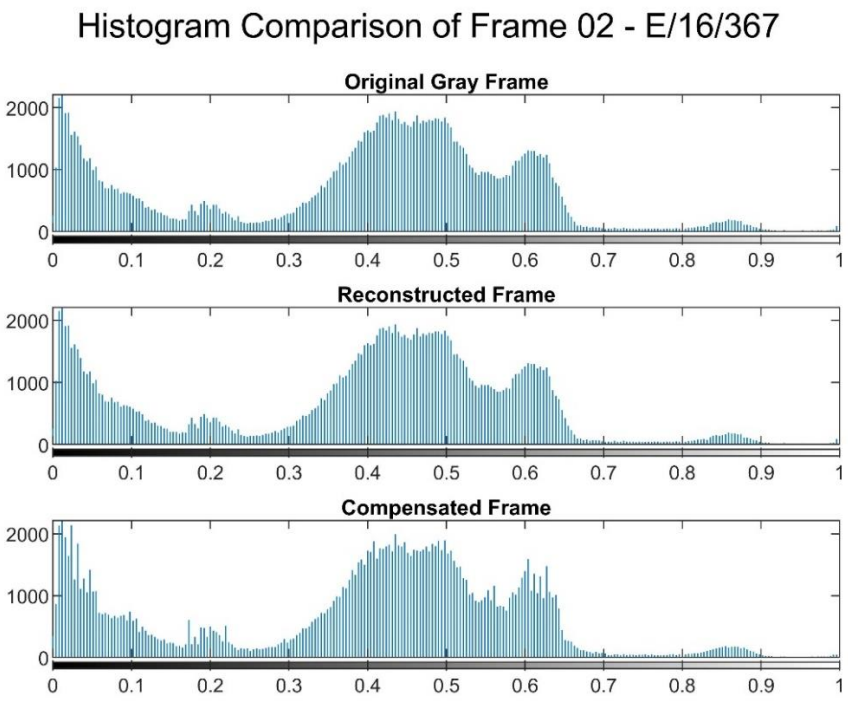


Figure 60: Histogram Comparison for Frame 02

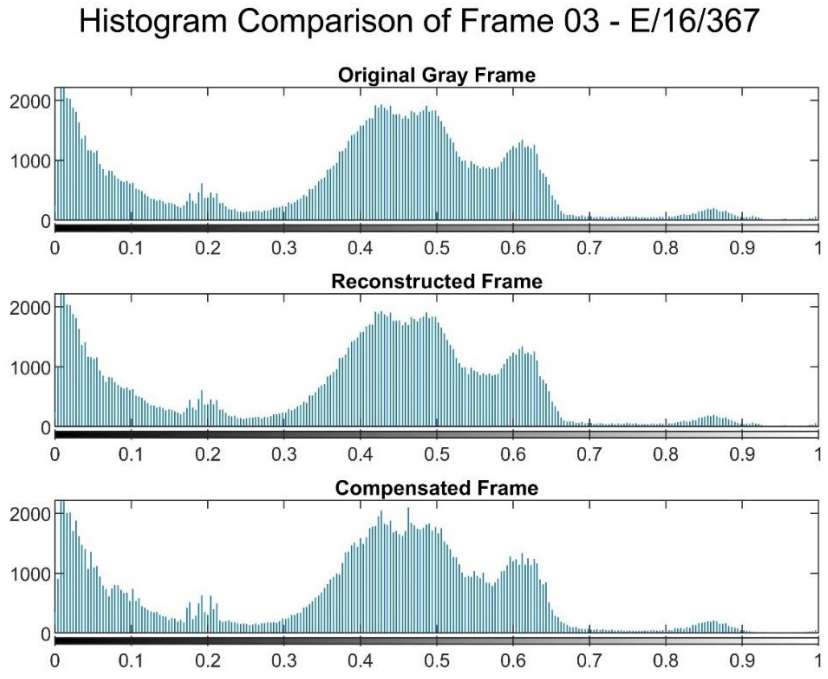


Figure 61: Histogram Comparison for Frame 03

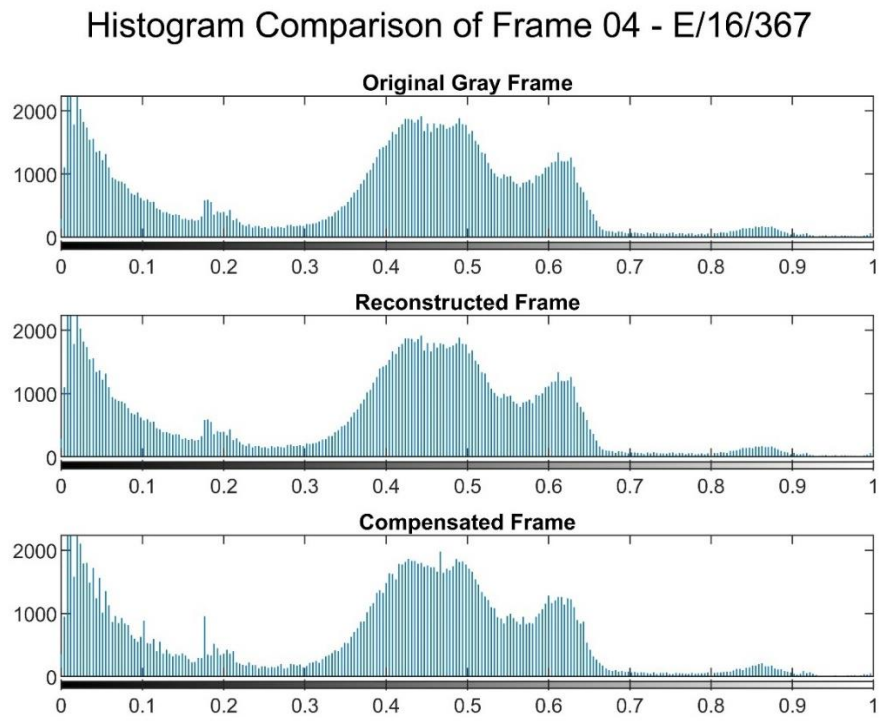


Figure 62: Histogram Comparison for Frame 04

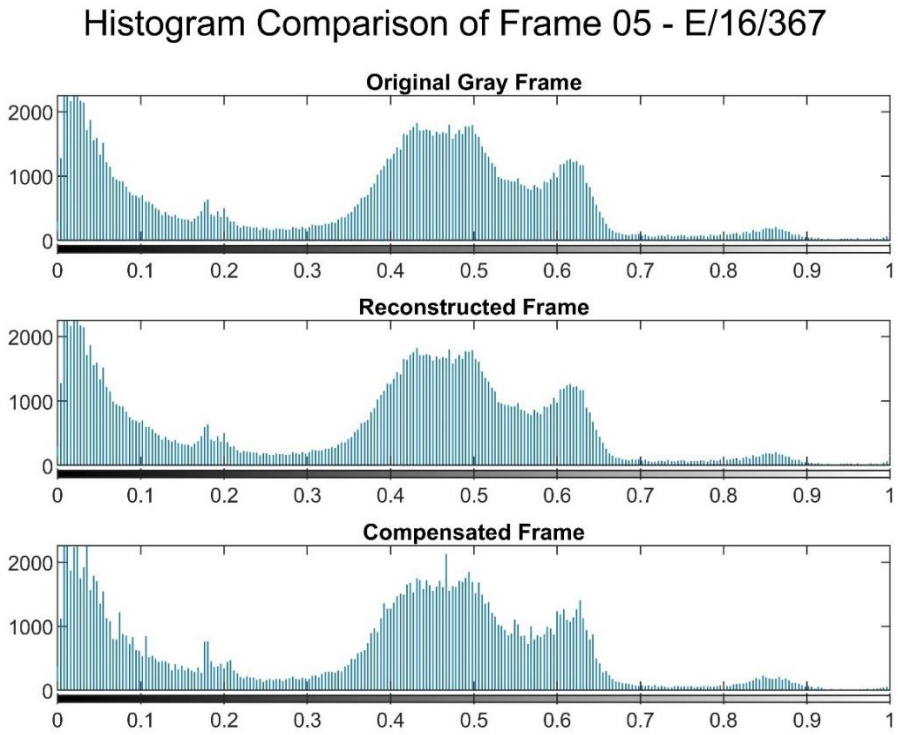


Figure 63: Histogram Comparison for Frame 05

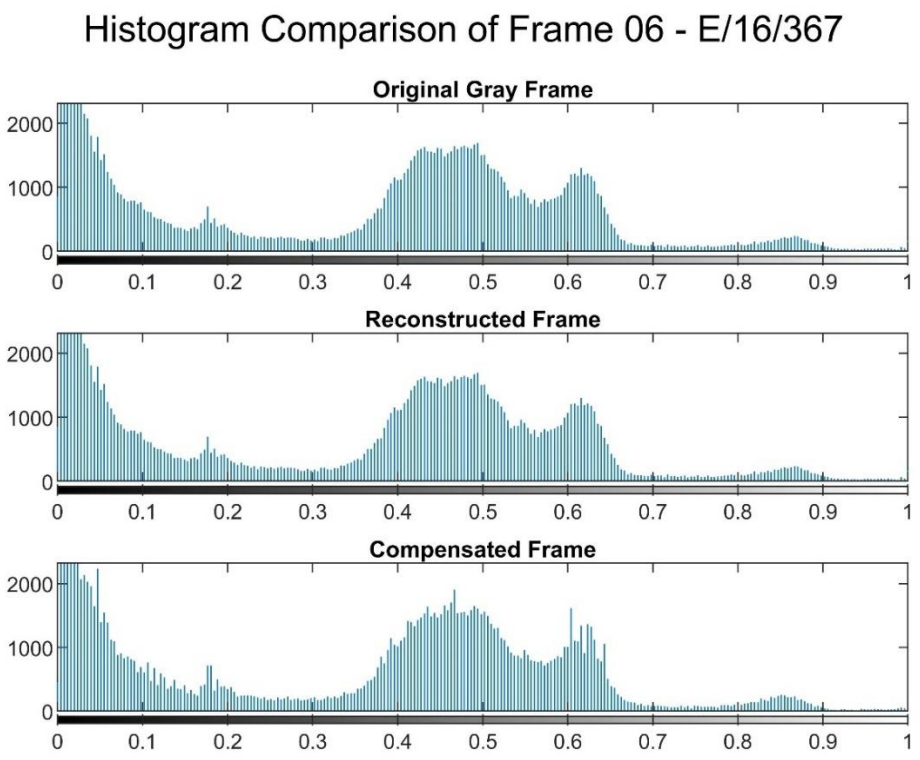


Figure 64: Histogram Comparison for Frame 06

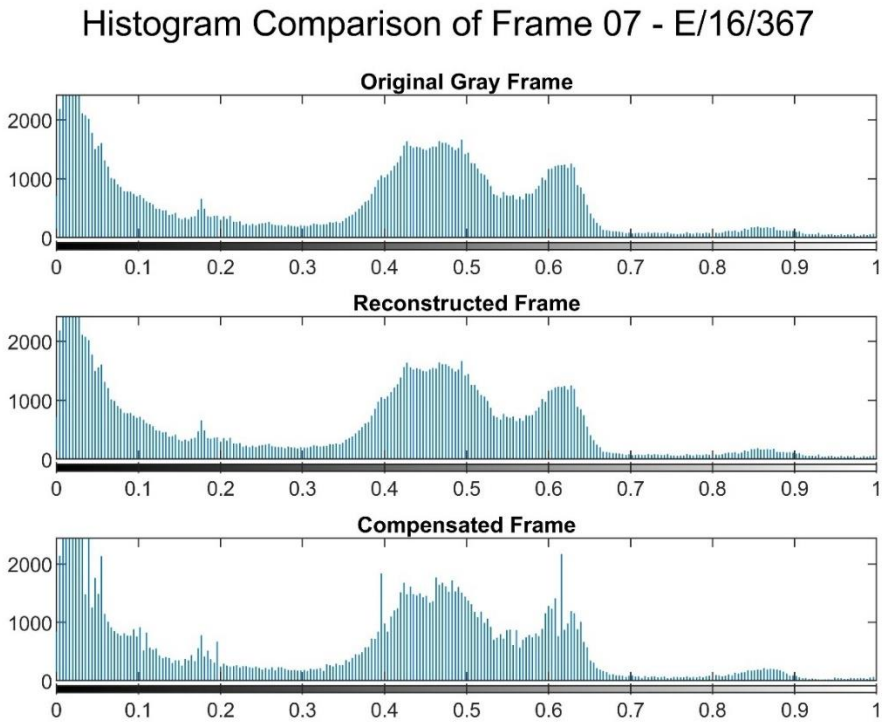


Figure 65: Histogram Comparison for Frame 07

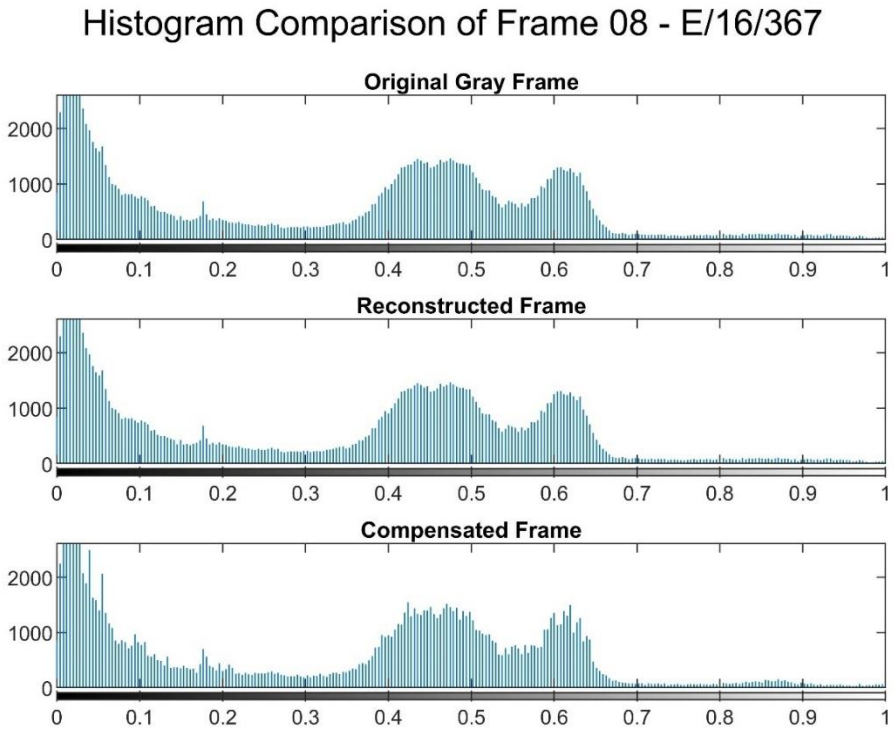


Figure 66: Histogram Comparison for Frame 08

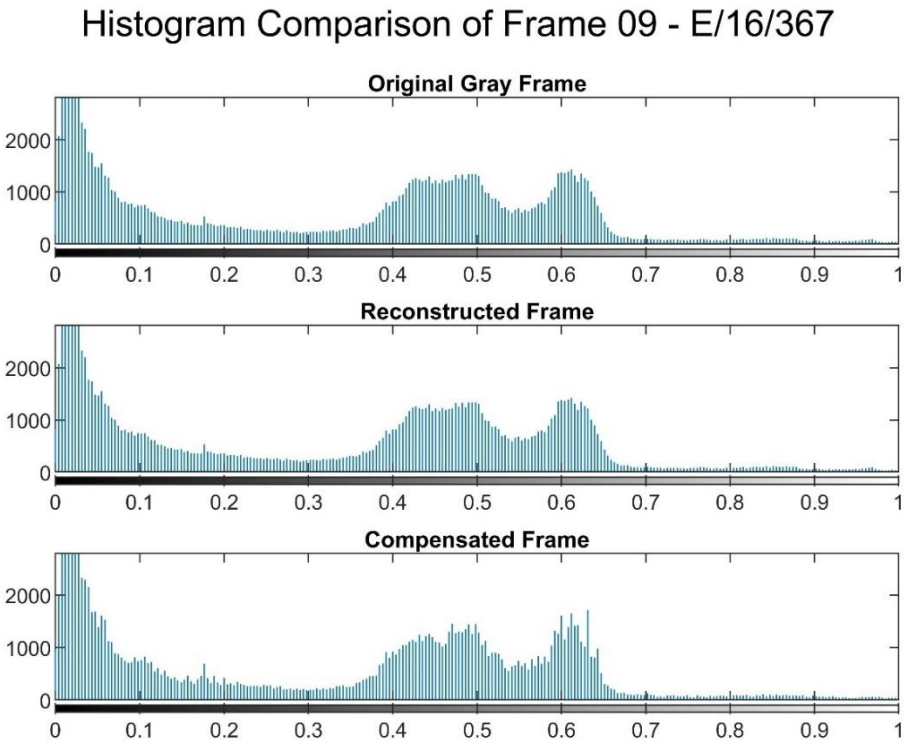


Figure 67: Histogram Comparison for Frame 09

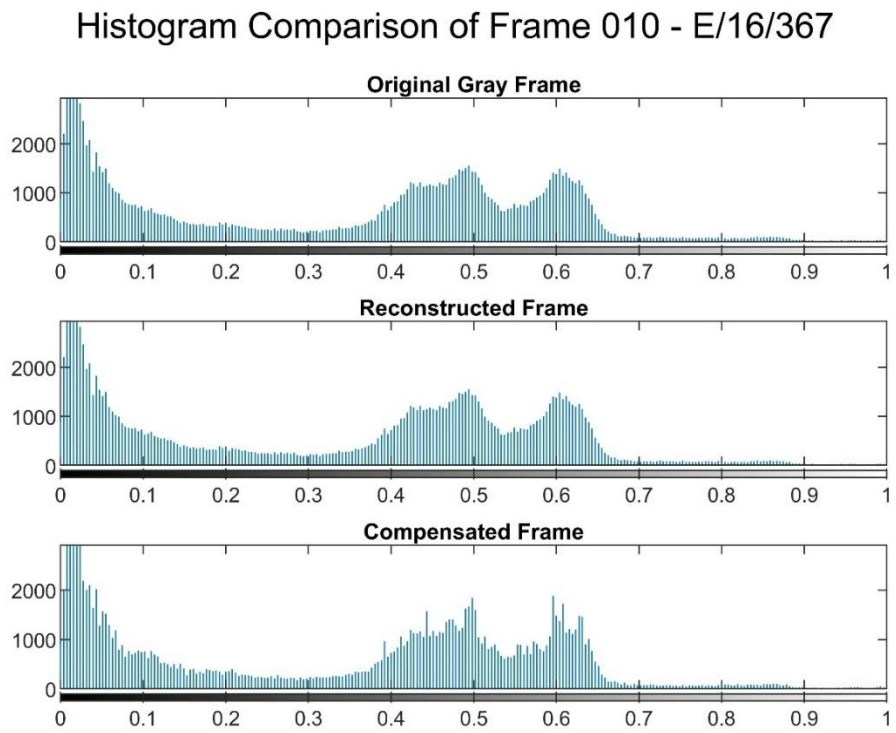


Figure 68: Histogram Comparison for Frame 10

Observing the histograms, we can come into the conclusion that, the data loss in compression is negligible as the histograms do not show significant difference for this quantization level.

Stage 3: Improved Hybrid Video Codec

Rate Distortion curves were created by calculating MSE, PSNR, SSIM as mentioned above in the methodology section and the plots for the obtained values are given below.

Table 01: MSE for Different Quantization Levels

Quantization level	File Size (kB)	MSE
20	18.2	1.0542
16	18.1	1.2936
8	17.9	2.2359
5	17.2	4.8946
4	16.8	6.1141
3	15.6	9.5738
2	14.6	18.4629
1	10.0	50.2919

Table 02: PSNR for Different Quantization Levels

Quantization level	File Size (kB)	PSNR	Distortion (1/PSNR)
20	18.2	47.9016	0.0209
16	18.1	47.0127	0.0213
8	17.9	44.6363	0.0224
5	17.2	41.2337	0.0243
4	16.8	40.2675	0.0248
3	15.6	38.3200	0.0261
2	14.6	35.4678	0.0282
1	10.0	31.1158	0.0321

Table 03: SSIM for Different Quantization Levels

Quantization level	File Size (kB)	SSIM	Distortion (1/SSIM)
20	18.2	0.9959	1.0041
16	18.1	0.9950	1.0051
8	17.9	0.9910	1.0091
5	17.2	0.9784	1.0221
4	16.8	0.9727	1.0281
3	15.6	0.9589	1.0428
2	14.6	0.9312	1.0739
1	10.0	0.8620	1.1601

Implemented Matlab Files

- quality.m

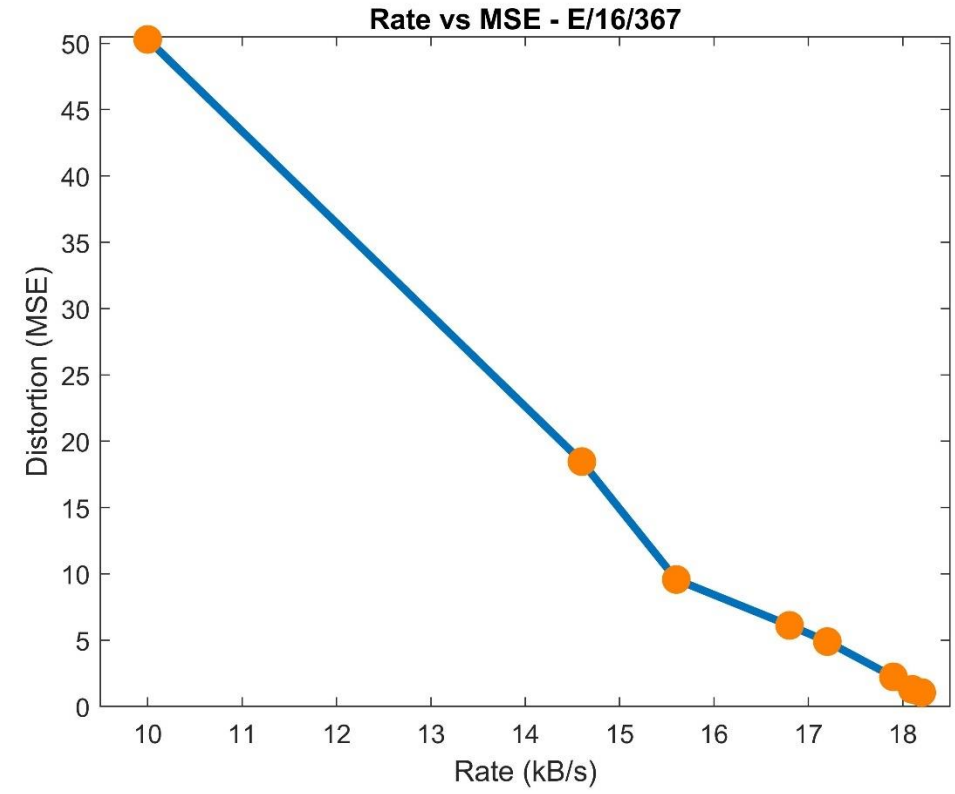


Figure 52: Rate vs MSE

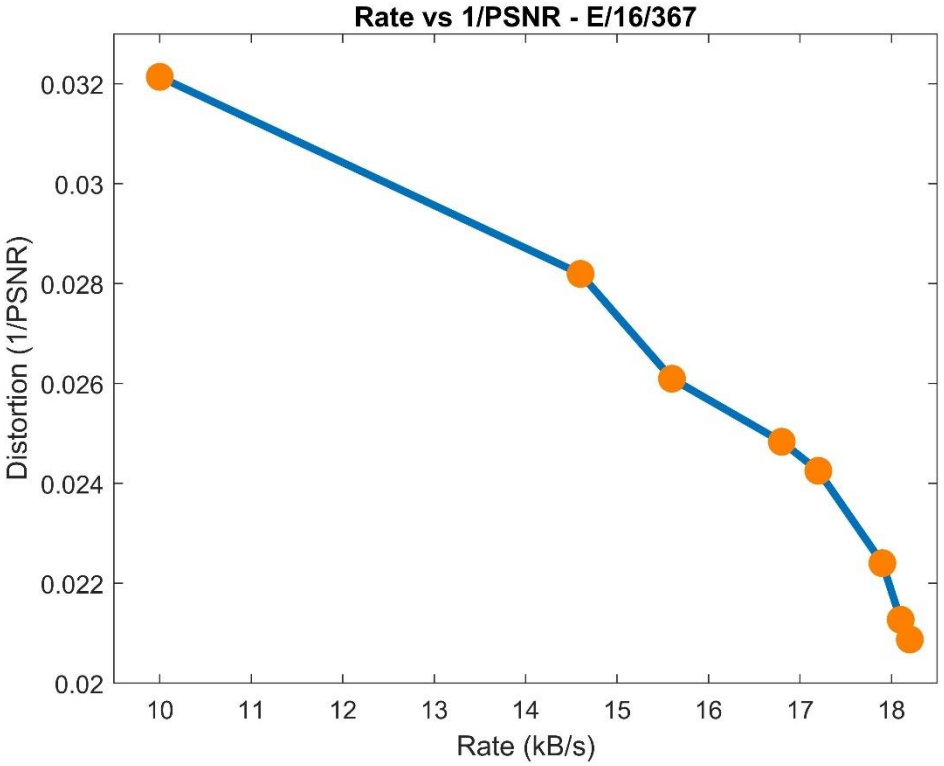


Figure 53: Rate vs 1/PSNR

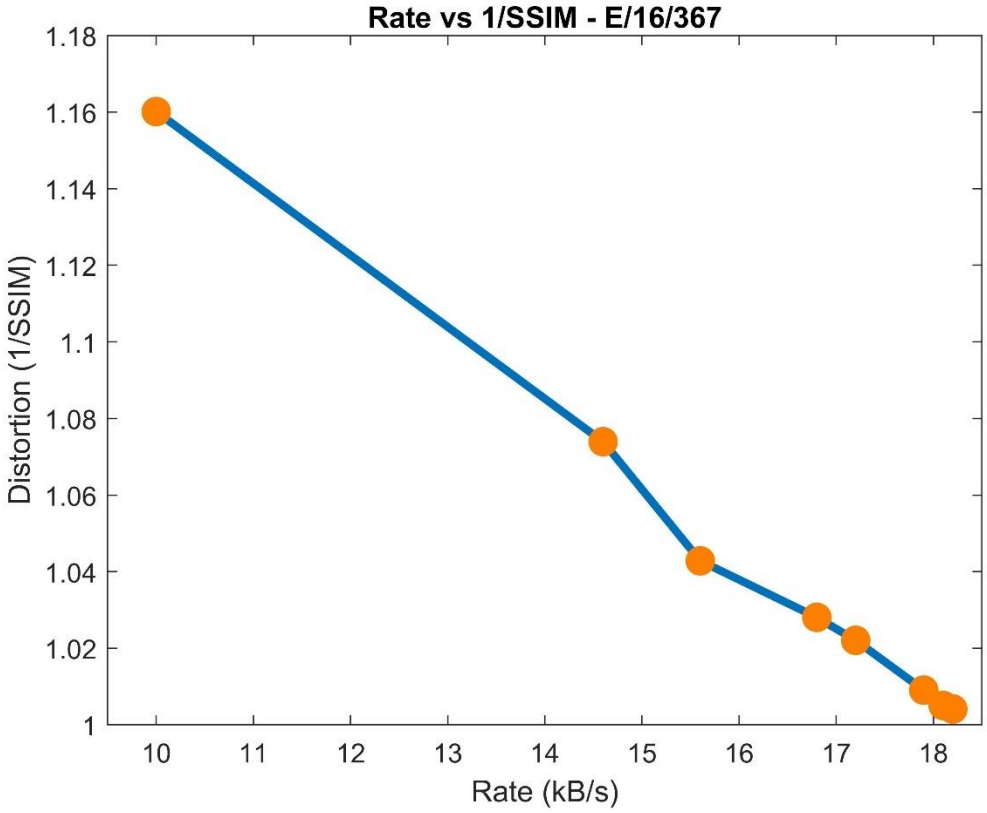


Figure 53: Rate vs 1/SSIM

The SSIM graph does not have distortion less than 1 and it has flattened out around Distortion = 1. This is true as, the maximum value SSIM can get is 1 and 1/SSIM cannot be smaller than 1 as well.

Observing the Rate Distortion Curves, it can be concluded that the **15.5 kB/s rate** is the best approach for optimal transmission.

Python Implementation

NOTE: My first attempt was using the Python Language and I created individual files for the different parts of the system. Even though all of them worked when run individually, the output was not accurate when run together. During the debugging process, I realized that my Huffman encoding code was the one causing the inaccuracy. Therefore, I switched to Matlab and Python implementation of those working modules are as follows.

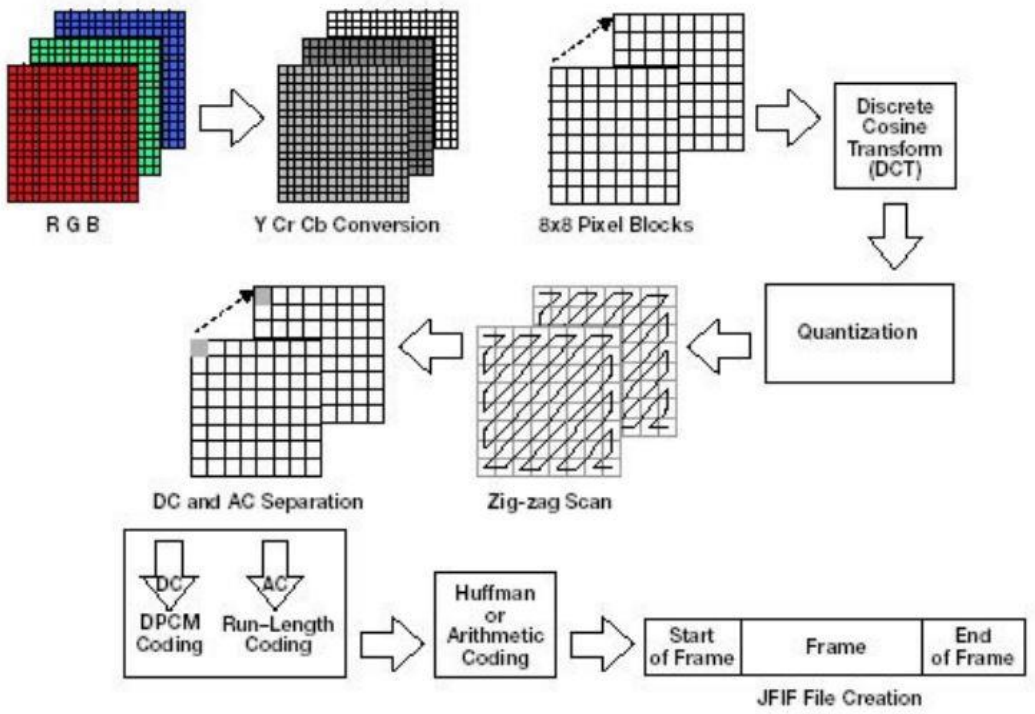


Figure 54: JPEG Pipeline

Python Implementation was done according to the above JPEG pipeline and all of the steps shown above were implemented.

Figure 54: Grayscale Image that was Used for Python Implementation

Libraries Used

```
import cv2
from skimage.io import imread
from skimage.color import rgb2gray
import numpy as np
import matplotlib.pylab as plt
from scipy.fftpack import dct, idct
import PIL
from PIL import Image
import time
import collections
import datetime
import re
```

Figure 55: Libraries Used

8x8 Window Blocking

```
def block_Div(path):

    row = 8
    col = 8
    windowed = []
    img = imread(path)
    #gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    for r in range(0,img.shape[0] - row, row):
        for c in range(0,img.shape[0] - col, col):
            windowed.append(img[r:r+row,c:c+col])
    return windowed
```

Figure 56: 8x8 Window Blocking Code

Discrete Cosine Transform

```
def DCT(block_array):

    imF = []
    for i in range (len(block_array)):
        imF.append(dct(dct(block_array[i].T, norm='ortho').T, norm='ortho'))

    imF = np.around(imF)

    return imF
```

Figure 57: DCT Function

Quantization

```
def quantization(array):

    level = int(input("Enter the Required Quantization Level (1 or 2 or 3):"))

    l1_norm = 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]])
    quantized = []
    quant = [[]]
    quant = np.array(quant)
    if level == 1:
        for i in range (len(array)):
            quantized.append(np.array(array[i])/l1_norm)
            #print(array[i]/l1_norm)
##    for j in range (len(quantized)):
##       quant =  np.concatenate((quant, quantized[i]), axis=0)
    quantized = np.around(quantized)
    print('Quantized:', quantized)

    #np.concatenate((a, b), axis=0)
##            array = Image.fromarray(array)
##            array = array.convert("L")
##            array.save('DCTBeforeQuantizing.jpg')
##            quantized = array.quantize(level)
##            quantized = quantized.convert("L")
##            quantized.save('DCTAfterQuantizing.jpg')

    return quantized
```

Figure 58: Quantization Function

ZigZag Code

```
File  Edit  Format  Run  Options  Window  Help
import numpy as np

def zigzag(AC):
    AC_list = []
    for i in range (len(AC)):
        AC_list.append((AC[i]).tolist())
    AC_list = np.array(AC_list)
    #AC_list = np.array(list(map(int, AC_list)))
    print(type(AC_list))
    print(AC)
    AC_row = np.concatenate([np.diagonal(AC_list[::-1,:],
        1+i)[::(2*(i % 2)-1)] for i in range(1-AC_list.shape[0], AC_list.shape[0])])
    #print('acrow:',AC_row)
    AC_row = AC_row.tolist()
    AC_row = list(map(int, AC_row))
    AC_row.insert(0,int(AC_list[0][0]))
    #print('acrow:',AC_row)

    return AC_row


bb = [[1,2,3],[4,5,6],[7,8,9]]
b = zigzag(np.array(bb))
print('ZigZag BitStream:', b)
```

Figure 59: ZigZag Function

```
<class 'numpy.ndarray'>
[[1 2 3]
 [4 5 6]
 [7 8 9]]
ZigZag BitStream: [1, 2, 4, 7, 5, 3, 6, 8, 9]
>>>
```

Figure 60: Output of ZigZag Function for a Test Case

DC sequence separation, Difference Coding & Reverse Difference Coding

File Edit Format Run Options Window Help

```
def reverse_differential(dc_decoded):
    dc_rev = []
    for i in range (len(dc_decoded)):
        #print(dc_rev)
        if i == 0:
            dc_rev.append(dc_decoded[i])
        else:
            #print(dc_rev[i-1],dc_decoded[i])
            dc_rev.append(dc_rev[i-1] - dc_decoded[i])

    return dc_rev

def differential_coding (quantized):
    DC = []
    print('Recieved DC BitStream: ',quantized)
    for i in range (len(quantized)):
        if i == 0:
            DC.append(quantized[i])
        else:
            DC.append(quantized[i-1] - quantized[i])
    #DC = list(map(int, DC))
    print('Difference Coded DC BitStream:',DC)
    return DC

dc = [1,2,3,4,5,6,7,8,9]
c = differential_coding(dc)
x = print('Final:',reverse_differential(c))
```

Figure 61: DC sequence separation & Difference Coding Function

```
Recieved DC BitStream:  [1, 2, 3, 4, 5, 6, 7, 8, 9]
Difference Coded DC BitStream: [1, -1, -1, -1, -1, -1, -1, -1, -1]
Final: [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Figure 62: Output of DC sequence separation & Difference Coding Function for a Test Case

Re-combining the separated AC and DC sequences

```
File  Edit  Format  Run  Options  Window  Help
def ac_dc_combine(ac_rev, dc_rev):

    arr_new = []
    print('Recieved AC BitStream:',ac_rev,'\n')
    print('Recieved DC BitStream:',dc_rev,'\n')
    #arr_new = np.array(arr_new)
    for i in range (len(dc_rev)):
        temp1 = []
        for j in range(8):
            temp2 = []
            temp2.append(dc_rev[i])
            for k in range (7):
                temp2.append(ac_rev[i*8*8+k])
            temp1.append(temp2)
        arr_new.append(temp1)
        #print(arr_new[i],'\n')
    #print(arr_new)

    return(arr_new)


dc_rev = [1,2,3,4,5]
ac_rev = []
ac_rev = ac_rev + [1]*63 + [2]*63 + [3]*63 + [4]*63 + [5]*63
arr_new = ac_dc_combine(ac_rev, dc_rev)
print('Combined DC-AC BitStream:',arr_new,'\n')
```

Figure 63: Function Combining the Separated AC and DC sequences

```
Recieved AC BitStream: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5
, 5, 5, 5]

Recieved DC BitStream: [1, 2, 3, 4, 5]

Combined DC-AC BitStream: [[[1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1],
[1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1], [1
, 1, 1, 1, 1, 1, 1, 1]], [[2, 2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2, 2], [2
, 2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2, 2], [2,
2, 2, 2, 2, 2, 2, 2]], [[3, 3, 3, 3, 3, 3, 3, 3], [3, 3, 3, 3, 3, 3, 3, 3], [3, 3, 3, 3, 3, 3, 3, 3], [3,
3, 3, 3, 3, 3, 3, 3], [3, 3, 3, 3, 3, 3, 3, 3], [3, 3, 3, 3, 3, 3, 3, 3], [3, 3, 3, 3, 3, 3, 3, 3], [3, 3,
3, 3, 3, 3, 3, 3]], [[4, 4, 4, 4, 4, 4, 4, 4], [4, 4, 4, 4, 4, 4, 4, 4], [4, 4, 4, 4, 4, 4, 4, 4], [4, 4,
4, 4, 4, 4, 4, 4], [4, 4, 4, 4, 4, 4, 4, 4], [4, 4, 4, 4, 4, 4, 4, 4], [4, 4, 4, 4, 4, 4, 4, 4], [4, 4, 4,
4, 4, 4, 4, 4]], [[5, 5, 5, 5, 5, 5, 5, 5], [5, 5, 5, 5, 5, 5, 5, 5], [5, 5, 5, 5, 5, 5, 5, 5], [5, 5, 5,
5, 5, 5, 5, 5], [5, 5, 5, 5, 5, 5, 5, 5], [5, 5, 5, 5, 5, 5, 5, 5], [5, 5, 5, 5, 5, 5, 5, 5], [5, 5, 5, 5,
5, 5, 5, 5]]]

>>>
```

Figure 64: Output of the Function Combining the Separated AC and DC sequences for a Test Case

IDCT

```
def IDCT(path):
    img = np.asarray(Image.open(path),np.uint8)
    im1 = idct(idct(img.T, norm='ortho').T, norm='ortho')
    im1 = Image.fromarray(im1)
    im1 = im1.convert("L")
    im1.save('Output.jpg')
    return None
```

Figure 65: IDCT Function

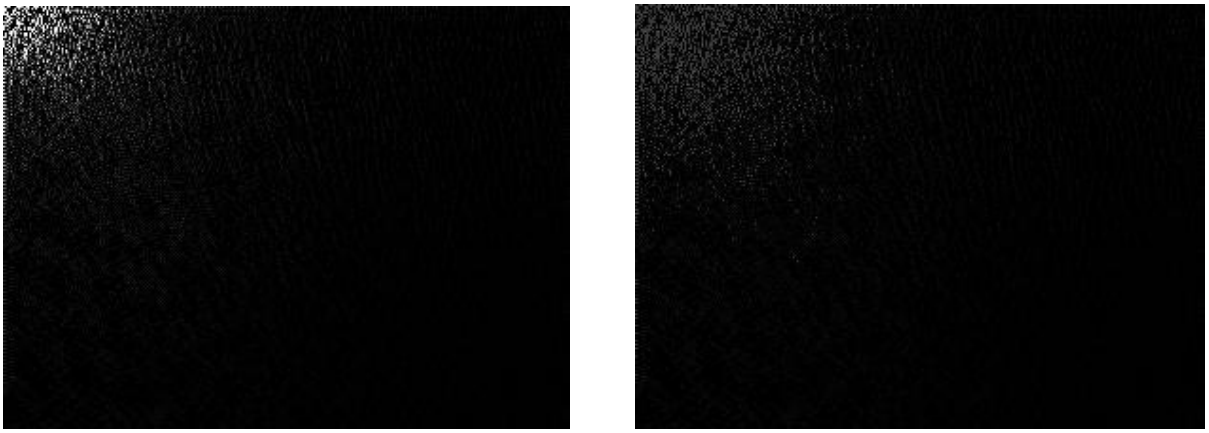


Figure 66: DCT Output and Quantized DCT Output for the ‘Tiger’ Image