Visual Media Compression

INFO-H516

Exercises report

BAKKALI Yahya (000445166)

May 2023

Université Libre de Bruxelles (ULB)

Contents

1 Introduction

The field of visual media compression plays a crucial role in various applications, ranging from multimedia streaming to efficient storage of large-scale visual data. In this report, we present the results and findings of the image and video compression exercises for the Visual Media Compression course. There are four tasks, each covering different aspects of compression techniques. Our first task was to implement an image encoder and decoder for image compression and decompression. The second and third tasks focused on video compression, where we designed a video encoder and decoder to efficiently compress and decompress video. In the second task we used only I-frames, while in the third task we extended the previous implementation by incorporating both I-frames and D-frames, with the aim of improving compression performance and reducing bitrate requirements. Finally, we compared the performance of the codecs implemented with existing compression standards: for image task, we used JPEG and JPEG2000, and for video tasks, H.264 and H.265.

2 Task 1

2.1 Description

In the first task of our report, we focused on simulating an image encoder and decoder. Our approach involved applying block-based coding, where we divided the image into blocks for efficient processing. We implemented the Discrete Cosine Transform (DCT) and Inverse DCT per block to convert the image data from the spatial domain to the frequency domain and vice versa. Quantization and inverse-quantization techniques were applied to reduce the precision of the frequency coefficients and subsequently reconstruct them. To optimize the encoding process, we implemented zigzag and inverse zigzag scan per block, which rearranged the coefficients in a specific order for better compression efficiency. Finally, we implemented entropy encoding/decoding to further reduce the data size by assigning shorter codes to more frequently occurring symbols.

2.2 Encoder implementation

2.2.1 Block-based coding

The first step was to divide the original image, with dimensions (W, H), into non-overlapping blocks of size (N, N). In our case, we have applied this step to the Lena image, whose dimensions are (256 , 256) and which has been divided into blocks of size (8 ,8).

To illustrate this process, the original Lena image was displayed alongside the split representation. The original image shows Lena in grayscale. In contrast, the split image of Lena contains a grid of blocks, with each block clearly delineated. Each block measured 8 by 8 pixels and contained a segment of Lena's image.

(a) Lena image (b) Lena image blocks

Figure 1: Split Lena image of size (256, 256) into blocks of size (8,8)

2.2.2 DCT

After dividing the image into blocks, the next step in block-based coding is to apply the Discrete Cosine Transform (DCT) to each block. The DCT converts the spatial domain pixel values into frequency domain coefficients. The two-dimensional DCT of an image block is simply the one-dimensional DCT performed along the rows and then along the columns. For our project we implement DCT-II, the formulas that we used comes from scipy [1] library. For the one-dimensional DCT the formula is the following:

$$y_k = 2 \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi k(2n+1)}{2N}\right) \qquad \text{for } k = 0, \ldots, N-1.$$

Where N is the number of elements in the array. Then, to normalize the value of y_k, we multiply it by a scaling factor f as follows:

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.fftpack.dct.html#scipy-fftpack-dct

$$f = \begin{cases} \sqrt{\frac{1}{4N}} & \text{if } k = 0, \\ \sqrt{\frac{1}{2N}} & \text{otherwise} \end{cases}$$

Let's consider an example using a block from the divided Lena image, where the block size is 8 by 8 pixels. We will calculate the DCT coefficients for the first block.

$$\begin{bmatrix} 137 & 136 & 133 & 136 & 138 & 134 & 134 & 132 \\ 137 & 136 & 133 & 136 & 138 & 134 & 134 & 132 \\ 138 & 133 & 134 & 134 & 136 & 132 & 130 & 130 \\ 133 & 133 & 133 & 130 & 134 & 133 & 128 & 125 \\ 129 & 133 & 130 & 130 & 133 & 131 & 132 & 128 \\ 131 & 133 & 130 & 122 & 132 & 131 & 130 & 130 \\ 131 & 130 & 130 & 130 & 132 & 131 & 128 & 130 \\ 131 & 132 & 130 & 130 & 131 & 131 & 130 & 128 \end{bmatrix} \xrightarrow{\text{DCT}} \begin{bmatrix} 1056 & 7 & -3 & 7 & 0 & -4 & -2 & 3 \\ 15 & 3 & -2 & 1 & 4 & 2 & -2 & 0 \\ 6 & -1 & 1 & -1 & 3 & 2 & -1 & -2 \\ -1 & -4 & 3 & 0 & -2 & -3 & -2 & 1 \\ -1 & 0 & -3 & 0 & -2 & 1 & -2 & -1 \\ -2 & 2 & -1 & 0 & 1 & 0 & 3 & 0 \\ -1 & 1 & 3 & 1 & -1 & -1 & 1 & 2 \\ 1 & -1 & -1 & -2 & 3 & 2 & -1 & -1 \end{bmatrix}$$

Figure 2: Example of DCT execution on the first block

2.2.3 Quantization

Once the DCT coefficients have been calculated for each block, the next step in the encoding is quantization. Quantization reduces the precision of the DCT coefficients to achieve compression. The process involves dividing the DCT coefficients by a quantization matrix, which controls the level of compression and determines the amount of information discarded. The quantization matrix used for this project is the following:

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Consider the first block of Lena's split image, which has been transformed using DCT. The DCT coefficients for this block have been obtained, and we will now apply quantization to this block. By dividing each DCT coefficient in the first block by the corresponding entry in the quantization matrix, we obtain the quantized DCT coefficients. The higher values in the quantization matrix result in greater loss of precision and hence more compression.

$$\begin{bmatrix} 1056 & 7 & -3 & 7 & 0 & -4 & -2 & 3 \\ 15 & 3 & -2 & 1 & 4 & 2 & -2 & 0 \\ 6 & -1 & 1 & -1 & 3 & 2 & -1 & -2 \\ -1 & -4 & 3 & 0 & -2 & -3 & -2 & 1 \\ -1 & 0 & -3 & 0 & -2 & 1 & -2 & -1 \\ -2 & 2 & -1 & 0 & 1 & 0 & 3 & 0 \\ -1 & 1 & 3 & 1 & -1 & -1 & 1 & 2 \\ 1 & -1 & -1 & -2 & 3 & 2 & -1 & -1 \end{bmatrix} \xrightarrow{/\ \mathrm{Q}} \begin{bmatrix} 66 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 3: Example of quantization execution on the first block

2.2.4 Zigzag scan

After quantization, the next step is the zigzag scanning. Zigzag scanning is performed on the quantized DCT coefficients of each block to rearrange them in a specific order, optimizing the compression efficiency. To illustrate the zigzag, we will use the quantized DCT coefficients of the first block of Lena's split image. Figure 4 shows the original order of the quantized DCT coefficients and the zigzag scanning order to be followed.

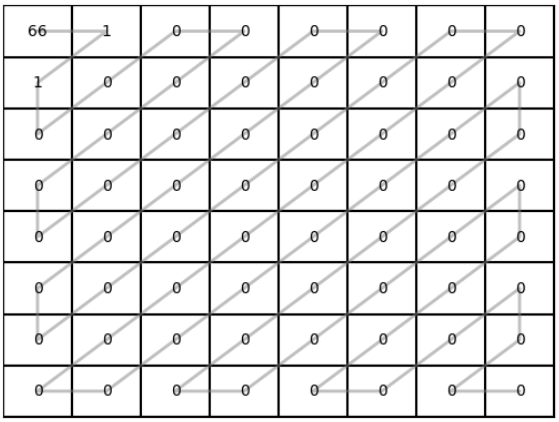

Figure 4: Zigzag scannig on the first block

The resulting zigzag scanning order of quantized DCT coefficients is as follows:

$$\begin{bmatrix} 66 & 1 & 1 & EOB \end{bmatrix}$$

As the result shows, all trailing zeros are removed and the EOB delimiter is inserted to separate the blocks. The EOB will be used by the decoder to reconstruct the original block. For technical

reasons, we replace the EOB with inf in our implementation.

2.2.5 Entropy coding

After the zigzag scanning stage, the next step is entropy coding. For this project, we used Huffman coding, a widely used entropy coding technique that assigns variable-length codes to symbols according to their probability of occurrence. We apply Huffman entropy coding to the zigzag scanned quantized DCT coefficients of each block. By applying Huffman's entropy coding to the previous result of the first block, we obtain the following result:

$$\begin{bmatrix} 66 & 1 & 1 & EOB \end{bmatrix} \xrightarrow{\text{Huffman}} \text{b`\textbackslash xcb\textbackslash xe4'}$$

Figure 5: Example of Huffman encoding on the first block

2.3 Decoder implementation

2.3.1 Entropy decoding

Entropy decoding is the process of converting the variable-length Huffman codes back into the corresponding zigzag scanned quantized DCT coefficients. By using the Huffman codebook, which contains the mapping between codes and coefficients, the compressed data is decoded to obtain the original blocks.

2.3.2 Inverse zigzag scan

By applying inverse zigzag scanning, the coefficients are placed back into their respective positions within the block, recreating the original quantized DCT coefficients.

2.3.3 Inverse quantization

Both previous steps are lossless, however, b the inverse quantization is a lossy one. Inverse quantization aims to restore the quantized coefficients to their original values by multiplying them with the corresponding entries from the quantization matrix used during compression.

Let's use the example block we coded in the section above. Once we have applied entropy

decoding and zigzag scanning, we get the same original block. We now perform inverse quantization on this block.

$$\begin{bmatrix} 66 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{\times \text{Q}} \begin{bmatrix} 1056 & 11 & 0 & 0 & 0 & 0 & 0 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 6: Example of inverse quantization execution on the first block

Figure 7 shows the difference between the original DCT coefficient for the first block and that obtained by inverse quantization. We can clearly see that this is a lossy step, as most of the values could not be restored.

$$\begin{bmatrix} 1056 & 7 & -3 & 7 & 0 & -4 & -2 & 3 \\ 15 & 3 & -2 & 1 & 4 & 2 & -2 & 0 \\ 6 & -1 & 1 & -1 & 3 & 2 & -1 & -2 \\ -1 & -4 & 3 & 0 & -2 & -3 & -2 & 1 \\ -1 & 0 & -3 & 0 & -2 & 1 & -2 & -1 \\ -2 & 2 & -1 & 0 & 1 & 0 & 3 & 0 \\ -1 & 1 & 3 & 1 & -1 & -1 & 1 & 2 \\ 1 & -1 & -1 & -2 & 3 & 2 & -1 & -1 \end{bmatrix} \neq \begin{bmatrix} 1056 & 11 & 0 & 0 & 0 & 0 & 0 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 7: Difference between original DCT coefficients (left) and the decoded (right) for first block

2.3.4 Inverse DCT

After inverse quantization, the decoder performed inverse DCT. The inverse DCT converts the frequency domain coefficients back into the spatial domain pixel values, reconstructing the image block as close as possible to the original. For the one-dimensional inverse DCT the formula is the following:

$$y_k = x_0 + 2\sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi(2k+1)n}{2N}\right) \qquad \text{for } k = 0, \ldots, N-1.$$

Where N is the number of elements in the array. For the inverse DCT of normalized values, this formula is used instead:

$$y_k = \frac{x_0}{\sqrt{N}} + \sqrt{\frac{2}{N}} \sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi(2k+1)n}{2N}\right) \text{ for } k = 0, \ldots, N-1.$$

By applying the inverse DCT on the decoded first block we obtained the results below:

$$\begin{bmatrix} 1056 & 11 & 0 & 0 & 0 & 0 & 0 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{\text{IDCT}} \begin{bmatrix} 136 & 136 & 135 & 134 & 134 & 133 & 132 & 132 \\ 136 & 135 & 135 & 134 & 133 & 133 & 132 & 132 \\ 135 & 135 & 134 & 134 & 133 & 132 & 132 & 131 \\ 134 & 134 & 133 & 133 & 132 & 131 & 131 & 131 \\ 133 & 133 & 133 & 132 & 131 & 131 & 130 & 130 \\ 133 & 132 & 132 & 131 & 130 & 130 & 129 & 129 \\ 132 & 132 & 131 & 131 & 130 & 129 & 129 & 128 \\ 132 & 132 & 131 & 130 & 130 & 129 & 128 & 128 \end{bmatrix}$$

Figure 8: Example of inverse DCT execution on the first block

Figure 9 shows the difference between the original the first block and that obtained by decoder. We can see that the values of the decoded are close to the original one even there was some data losses with quantization.

$$\begin{bmatrix} 137 & 136 & 133 & 136 & 138 & 134 & 134 & 132 \\ 137 & 136 & 133 & 136 & 138 & 134 & 134 & 132 \\ 138 & 133 & 134 & 134 & 136 & 132 & 130 & 130 \\ 133 & 133 & 133 & 130 & 134 & 133 & 128 & 125 \\ 129 & 133 & 130 & 130 & 133 & 131 & 132 & 128 \\ 131 & 133 & 130 & 122 & 132 & 131 & 130 & 130 \\ 131 & 130 & 130 & 130 & 132 & 131 & 128 & 130 \\ 131 & 132 & 130 & 130 & 131 & 131 & 130 & 128 \end{bmatrix} \neq \begin{bmatrix} 136 & 136 & 135 & 134 & 134 & 133 & 132 & 132 \\ 136 & 135 & 135 & 134 & 133 & 133 & 132 & 132 \\ 135 & 135 & 134 & 134 & 133 & 132 & 132 & 131 \\ 134 & 134 & 133 & 133 & 132 & 131 & 131 & 131 \\ 133 & 133 & 133 & 132 & 131 & 131 & 130 & 130 \\ 133 & 132 & 132 & 131 & 130 & 130 & 129 & 129 \\ 132 & 132 & 131 & 131 & 130 & 129 & 129 & 128 \\ 132 & 132 & 131 & 130 & 130 & 129 & 128 & 128 \end{bmatrix}$$

Figure 9: Difference between original first block (left) and the decoded first block (right)

2.3.5 Block merging

After performing the inverse DCT on each block, the decoder moves on to the merging of blocks. During encoding, the image was divided into smaller blocks to facilitate efficient encoding processes. Now, in the decoding phase, these blocks are merged together to reconstruct the full-sized image.

2.4 Evaluation

2.4.1 Objective

As Figure 10 displays, a higher quantization scale leads to lower values of PSNR thus the image quality will be reduced. Particularly, the range 0.1 to 1 for the quantization scale gives optimal values for PSNR, therefore the image quality will be acceptable.

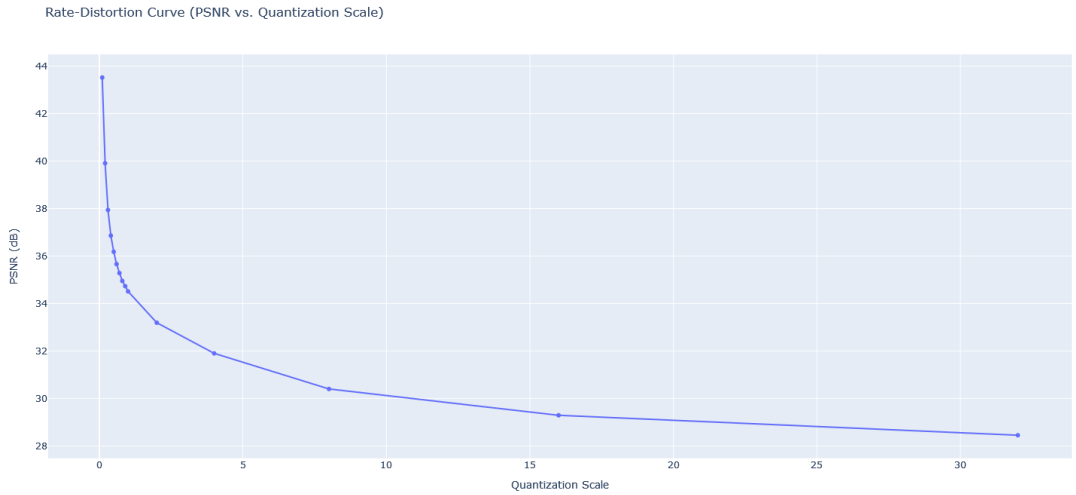

Figure 10: Rate-Distortion Curve (PSNR vs. scale)

Figure 11 verifies the image quality according to the instances' size. As we can see, the PSNR value has a direct relationship with the file size. The files which have a larger size, also have the larger PSNR values. The reason for that can be explained by the fact that when there is a large number of pixels, a lower number (portion) of pixels will be affected by the noise so big-size files will give the images with better quality.

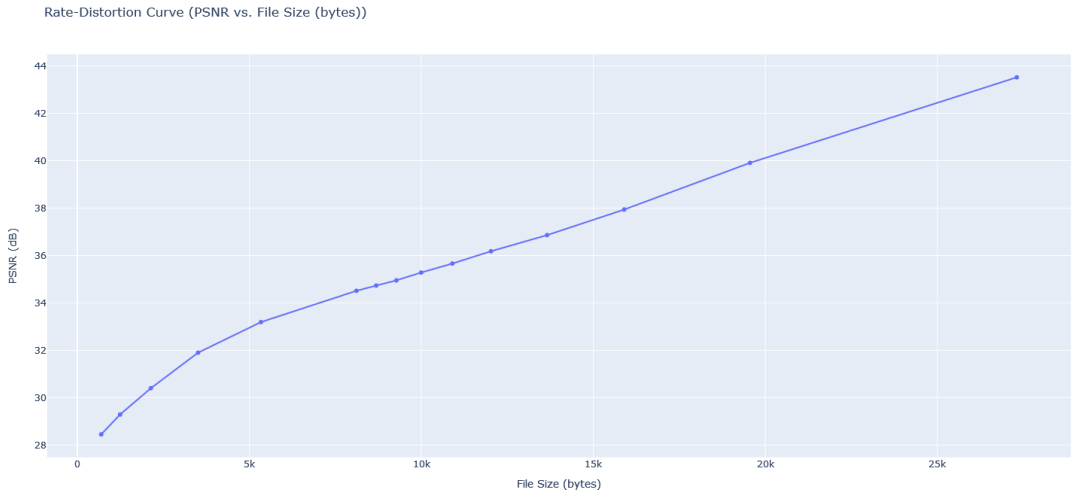

Figure 11: Rate-Distortion Curve (PSNR vs. size)

Figure 12 plots the PSNR by taking into consideration the BPP. As we can see there is a concordance between this Figure and Figure 11. We can claim the same explanation here too.

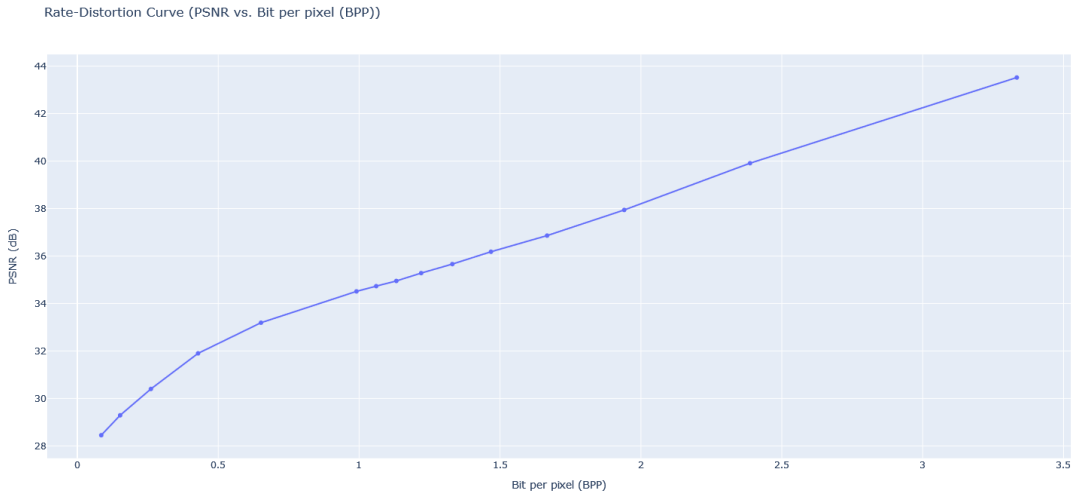

Figure 12: Rate-Distortion Curve (PSNR vs. bpp)

2.4.2 Subjective

(a) 3.5 BPP

(b) 2 BPP

(c) 1 BPP

(d) 0.5 BPP

Figure 13: Lena image compressed with different scales

Consider Figure 13a, Figure 13b, Figure 13c and Figure 13d. As we can see, Figure 13a has the best quality among these figures as it has the higher value of bpp. Figure 13d has the lowest bpp value and therefore it has the worst image quality. This visualization corresponds entirely with

the results that we verified in the previous subsection. The difference between the bpp values of Figure 13a and Figure 13b is not too much, thus it is really hard to distinguish which of them has a better quality. On the other hand, it is easy to distinguish the difference between Figure 13c and Figure 13d from Figure 13a and Figure 13b as there exists a high difference of bpp between these groups of images.

3 Task 2

3.1 Description

This task involves simulating a video encoder and decoder using an all I-picture approach within a Group of Pictures (GOP) structure. In this scenario, the GOP consists of several frames, specifically I-pictures only.

3.2 Implementation

As part of our video encoder implementation, we designed a specialized function to facilitate reading of the video file. This function enabled us to access and extract individual frames from the video sequence seamlessly. We then iterated over each frame, applying the image codec we had developed for Task 1.

In terms of decoder implementation, we iterate over each frame of the coded images and decode it using the corresponding entropy codebook. The image codec from Task 1 was also used here, and another specialized function was implemented to recreate the original video using the decoded images.

3.3 Evaluation

3.3.1 Objective

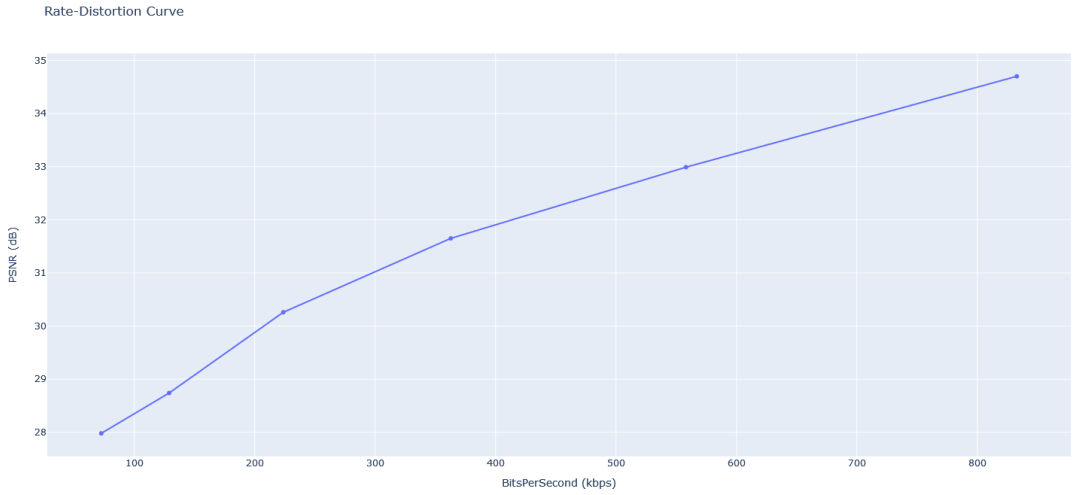

Figure 14: Rate-Distortion Curve (PSNR vs. BitsPerSecond)

Figure 14 checks the PSNR values according to the number of bits of videos per second for task 2. As we can see higher bps values lead to higher PSNR values and so on higher video quality. The reason for that is similar to our explanation for task 1. When we have higher bps values, it means that each second contains more frames so the noise will affect the lower portion of frames.

4 Task 3

4.1 Description

This task is similar to task 2. However it considers that a Group of Pictures is made up of of several frames, containing both I-frame and D-frame.

4.2 Implementation

The implementation differs from task 2 in that, for D-frame, we don't code the actual frame but its difference from the decoded I-frame of the GOP to which this frame belongs.

4.3 Evaluation

4.3.1 Objective

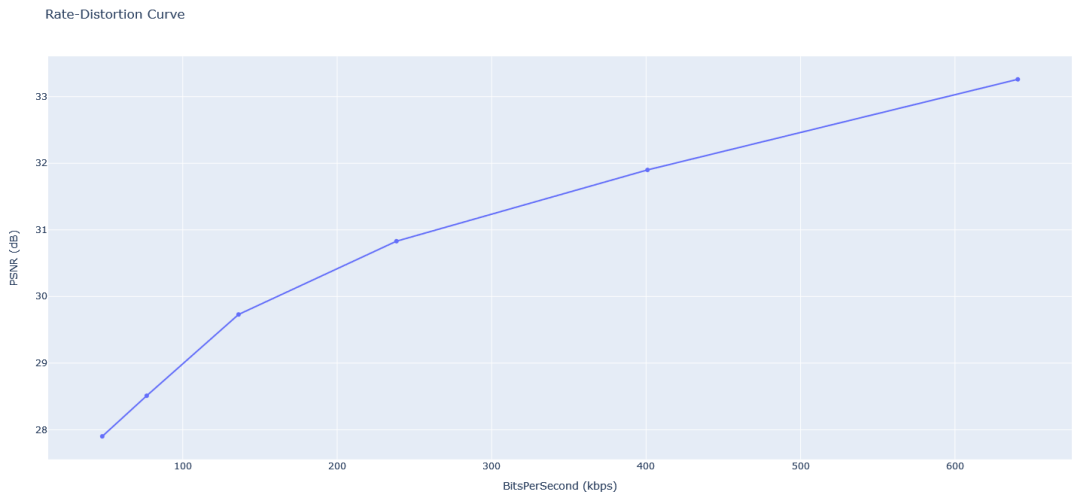

Figure 15: Rate-Distortion Curve (PSNR vs. BitsPerSecond)

Figure 15 checks the PSNR values according to the number of bits of videos per second for task 3. As we can see higher bps values lead to higher PSNR values and so on higher video quality. The reason for that can be considered the same as our explanation for task 2.

5 Task 5

5.1 Description

The objective of this task is to compare the codecs implemented for image and video compression with existing standard codecs, both objectively and subjectively. For image compression, we will

evaluate the performance of our codecs against established ones such as JPEG and JPEG2000. Similarly, for video compression, we will assess the effectiveness of our codecs in comparison to widely-used standards like h.264 and h.265.

5.2 Implementation

In our implementation, we utilized the Python Imaging Library (PIL) for testing the JPEG [2] and JPEG2000 [3] codecs. For video compression, we employed the Moviepy library. It offers support for a wide range of video codecs [4], including h.264 and h.265.

5.3 Evaluation

5.3.1 Objective

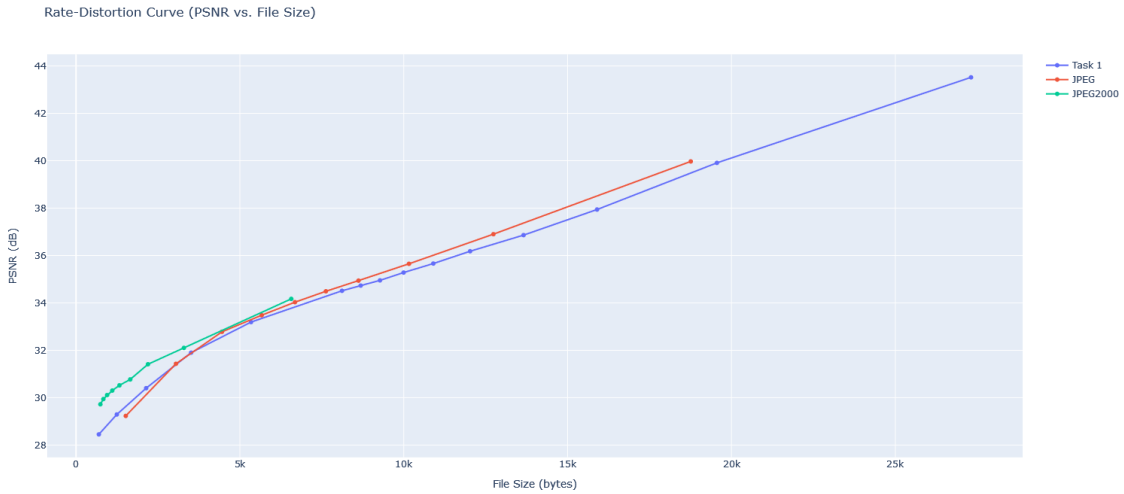

Figure 16: Rate-Distortion Curve (PSNR vs. File size)

As Figure 16 shows JPEG2000 gives always better PSNR values. The second position belongs to JPEG and the last one belongs to our implementation. But as you can see there is only a little difference between JPEG and our implementation.

[2] https://pillow.readthedocs.io/en/stable/handbook/image-file-formats.html#jpeg-saving

[3] https://pillow.readthedocs.io/en/stable/handbook/image-file-formats.html#jpeg-2000-saving

[4] https://zulko.github.io/moviepy/ref/VideoClip/VideoClip.html#moviepy.video.VideoClip.VideoClip.write_videofile

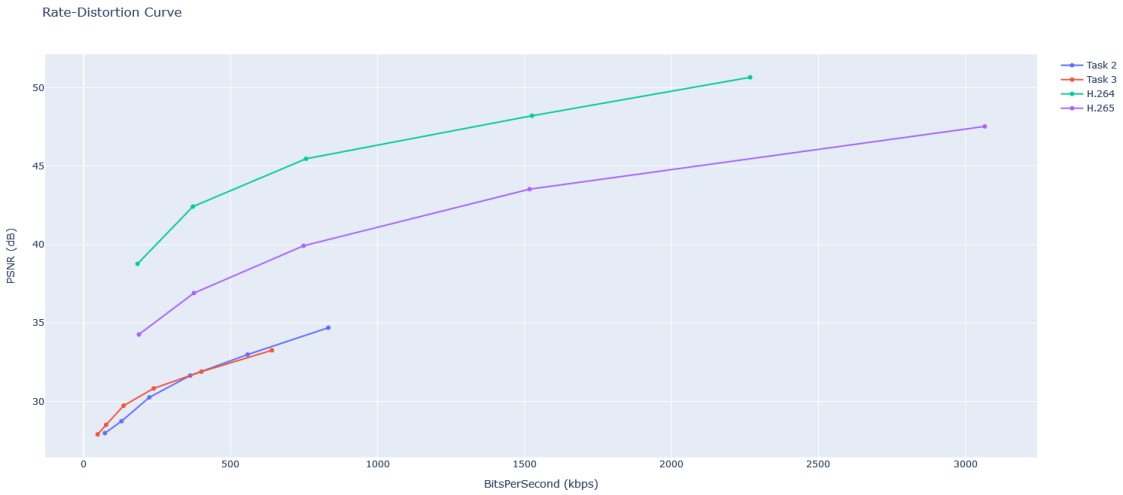

Figure 17: Rate-Distortion Curve (PSNR vs. BitsPerSecond)

As Figure 17 shows H.264 gives always better PSNR values. The second position belongs to H.265. Our implementation for task 2 gives better results for the videos which has bps higher than 500 while for bps lower than 500, task 3 implementation is better.

5.3.2 Subjective

(a) Original (65.536 bytes)

(b) Task 1 (3055 bytes)

(c) JPEG (3048 bytes)

(d) JPEG2000 (2194 bytes)

Figure 18: Lena image compressed with different scales

Task 1 codec and JPEG standard give a similar compressed image for the same file size, with a slight difference in some image blocks. However, JPEG-2000 gives a less well-compressed, but smaller image. JPEG2000's quantization tend to preserve important regions of the image.