Informação e Codificação

Second project report

Bruno Gomes, Diogo Silva

Universidade de Aveiro

Informação e Codificação

DEPARTAMENTO DE ELETRÓNICA
TELECOMUNICAÇÕES E INFORMÁTICA

Second project report

Bruno Gomes, Diogo Silva
(103320) brunofgomes@ua.pt, (104341) diogobranco.as@ua.pt

27/01/2025

Abstract

This report presents the development of efficient audio, image, and video encoders using predictive coding, Golomb coding, and transform coding techniques. The project is structured into multiple parts, starting with the implementation of a BitStream class for optimized bit-level file manipulation, followed by Golomb coding for efficient integer compression. In the audio domain, lossless and lossy predictive coding techniques are explored, incorporating adaptive parameter selection and bitrate control. For image and video coding, spatial and temporal prediction methods are employed, alongside intra-frame and inter-frame compression with motion compensation. . The report evaluates the compression ratio and processing time for each codec. Finally, challenges faced during development and potential improvements for future codec optimization are discussed.

Contents

Chapter 1

BitStream Class

1.1 Overview

The BitStream class is a core component of this project, responsible for efficient bit-level file operations. It provides an abstraction layer that allows reading and writing individual bits, bit sequences, and strings to and from files. This class is crucial for implementing efficient compression algorithms such as Golomb coding and predictive coding, which are fundamental for our project.

1.2 Key features:

1. **Constructor and Destructor**
   - Opens the file in binary mode (read or write);
   - Ensures the file is properly closed and flushed upon destruction.
2. **Bit-Level Operations**
   - **writeBit(bool bit):** Writes a single bit to the file;
   - **readBit():** Reads a single bit from the file.
3. **Multi-Bit Operations**
   - **writeBits(uint64_t value, int n):** Writes an N-bit integer to the file;
   - **readBits(int n):** Reads an N-bit integer from the file.
4. **String Operations**
   - **writeString(const std::string &str):** Stores a string as a series of bits;
   - **readString(size_t length):** Reads a string from the bitstream.

5. **Buffer Management and Padding**
    - **flushBuffer():** Ensures all buffered bits are written to the file;
    - **addPadding(uint64_t bitCount):** Adds padding bits to align to a byte boundary.
6. **File Positioning and EOF Handling**
    - **eof():** Checks if the end of the file has been reached;
    - **seek(std::streampos position):** Moves the file pointer to a specific position.

1.3 Tests and Results

To test if the BitStream class was correctly implemented, we developed a simple codec that takes an "input.txt" file that contains zeros and ones, encodes that file into a binary file, and finally the decoder decodes the binary file and the result should be a text file whose content should be exactly equal to the original input text file.

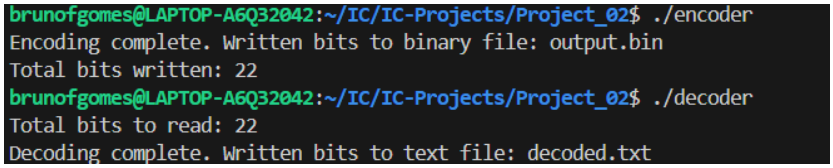

Figure 1.1: Text file decoding and encoding

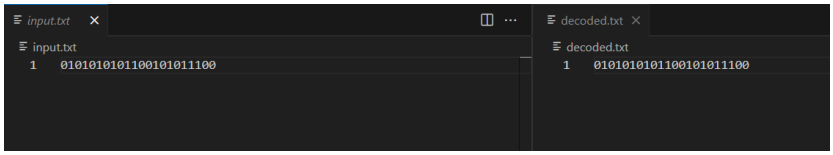

Figure 1.2: Original file and decoded file

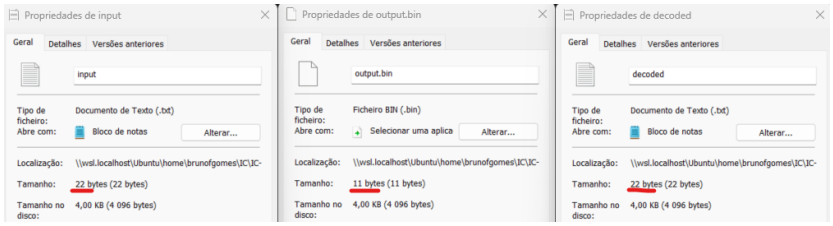

Figure 1.3: File sizes

Overall, these tests confirmed that our BitStream class was functioning properly, providing a solid foundation for the implementation of more advanced audio, image, and video codecs that rely on this critical class.

1.4 Conclusion

Basically, the BitStream class is a high-performance, reusable component that provides the bitwise precision required for efficient multimedia compression algorithms. Its design ensures optimal data storage, fast bit-level access, and robust error handling, making it a crucial building block for audio, image, and video encoding in this project.

Chapter 2

Golomb

2.1 Overview

The Golomb Class is a fundamental component of our compression system, enabling efficient entropy coding for predictive audio compression. It is designed to encode and decode integer sequences efficiently using Golomb coding, making it highly effective for data with geometric distributions.

2.2 Key features:

1. **Constructor**
    - Initializes the Golomb encoder with a specified divisor.
    - Optionally uses interleaving (default is true).
2. **Encoding Operations**
    - **encode(int value, BitStream &bitStream):** Converts an integer value into a Golomb-coded bit sequence, writing the result to a 'BitStream'. It uses the 'mapToPositive' function to map the value for encoding and writes both quotient and remainder values.
    - **mapToPositive(int value):** Maps signed values to non-negative integers for efficient coding, considering whether the interleaving is enabled or not. The method applies different transformations based on the 'interleave' flag.
3. **Decoding Operations**
    - **decode(BitStream &bitStream):** Reads and reconstructs an integer from a Golomb-coded bit sequence. The 'mapFromPositive' method is used to map the decoded value back to the signed integer.

- **mapFromPositive(int value):** Converts the non-negative integer back to a signed value, again considering the interleaving.

4. **Parameter Selection and Adaptation**
    - **setParameter(int m):** Adjusts the divisor dynamically for optimized compression.

2.3 Tests and Results

To verify the correctness of our Golomb Coding class, we conducted encoding and decoding tests using a dataset of predictive coding residuals. The following steps were performed:

1. Encode a series of integers using a fixed m.
2. Store the encoded bitstream in a binary file.
3. Decode the bitstream and read the reconstructed sequence,
4. Compare the original and reconstructed data for accuracy.

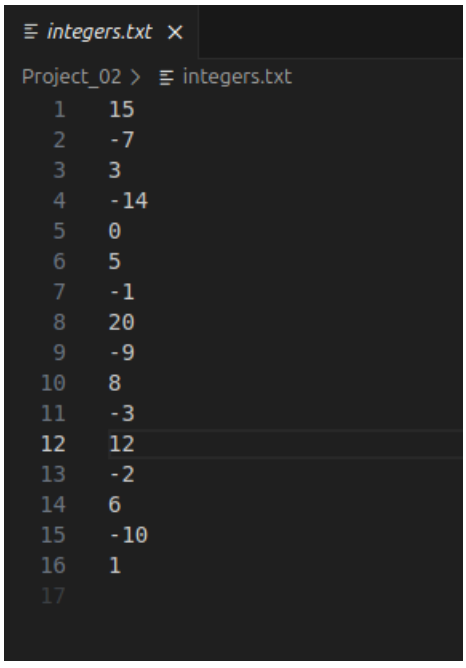

Figure 2.1: Integers test file

Figure 2.2: Running TestGolomb.cpp

Figure 2.3: File size comparisons

Chapter 3

Audio Encoding with Predictive Coding

3.1 Lossless Audio coding methodology

3.1.1 Predictive Coding Approach

We used a first order predictor to calculate the residuals of consecutive samples.

3.1.2 Determination of Golomb Parameter (m)

The optimal m is calculated based on the mean absolute residual value:

$$m = 2^{\lceil \log_2(\text{mean}+1) \rceil}$$

3.1.3 Preamble

Before encoding, we generate a preamble with important information about the audio file such as:

- Sample rate (32-bit)
- Number of channels (16-bit)
- Optimal M value (16-bit)
- Lossy flag (8-bit, 0 for lossless)
- Quantization (16-bit), only relevant for lossy compression mode.

3.1.4 Encoding with Golomb

Everything is encoded using Golomb encoding. The encoded bits are written to a BitSream After encoding all samples, padding is added to align the last byte

3.2 Lossless Coding Results

We ran the lossless codec for sample01.wav to sample07.wav and decoded them, obtaining the same audio as the original without any losses in information.

```
pastilhamas@pastilhamas-VirtualBox:~/IC-Projects/Project_02$ ./audio_codec encode audiofiles/sample01.wav encoded01.bin
Optimal M: 1024
pastilhamas@pastilhamas-VirtualBox:~/IC-Projects/Project_02$ ./audio_codec encode audiofiles/sample02.wav encoded02.bin
Optimal M: 1024
pastilhamas@pastilhamas-VirtualBox:~/IC-Projects/Project_02$ ./audio_codec encode audiofiles/sample03.wav encoded03.bin
Optimal M: 512
pastilhamas@pastilhamas-VirtualBox:~/IC-Projects/Project_02$ ./audio_codec encode audiofiles/sample04.wav encoded04.bin
Optimal M: 1024
pastilhamas@pastilhamas-VirtualBox:~/IC-Projects/Project_02$ ./audio_codec encode audiofiles/sample05.wav encoded05.bin
Optimal M: 256
pastilhamas@pastilhamas-VirtualBox:~/IC-Projects/Project_02$ ./audio_codec encode audiofiles/sample06.wav encoded06.bin
Optimal M: 128
pastilhamas@pastilhamas-VirtualBox:~/IC-Projects/Project_02$ ./audio_codec encode audiofiles/sample07.wav encoded07.bin
Optimal M: 2048
pastilhamas@pastilhamas-VirtualBox:~/IC-Projects/Project_02$
```

Figure 3.1: commands ran with optimal M for Golomb

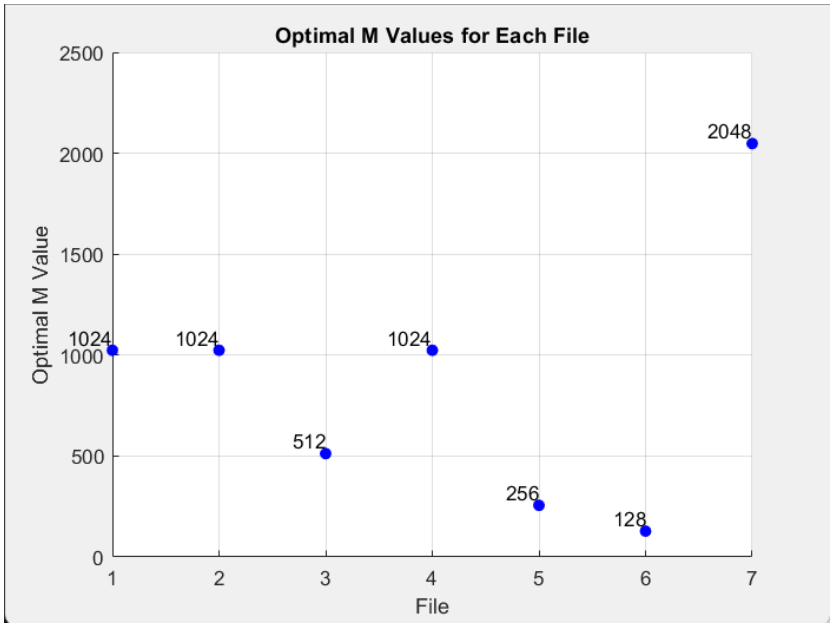

Figure 3.2: The optimal M chosen for each file

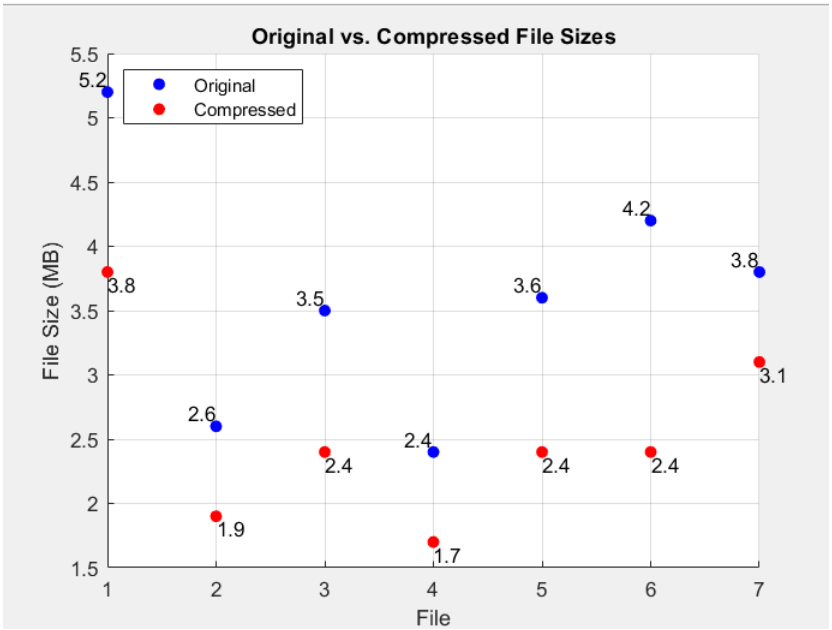

Figure 3.3: Comparison between the original audio files size and the lossless binary ones

3.3 Lossy Audio coding methodology

The lossy version of the codec follows a similar process to the lossless version but introduces quantization during the encoding and dequantization during decoding. The key differences occur in residual calculation,storage and reconstruction.

3.3.1 Extract audio

Same as lossless, load the wav file using SFML. Extract sample rate, number of channels and raw samples.

3.3.2 Compute residuals

Instead of storing the exact residual, we quantize it:

$$\text{residual} = \frac{\text{audioSamples}[i] - \text{prediction}}{2^{\text{quantizationlevel}}}$$

after computing the quantized residual we modify the original sample by reconstrucing it:

$$\text{audioSamples}[i] = \text{prediction} + \text{residual} \times 2^{\text{quantizationLevel}}$$

This process will reduce precision, meaning some audio details are lost, noise is introduced, but we can achieve a much higher compression.

3.3.3 Preamble

The preamble is the same as the lossless version but the Lossy Flag will be active and the Quantization level will be the one one we set as a parameter when encoding

3.3.4 Encoding with Golomb

Is the same as in the lossless version

3.3.5 Decoding and reconstruction

Residuals are decoded using Golomb, but unlike in the lossless version where we simply add the residual to the previous sample, in lossy we have to dequantize it before adding it back:

$$\text{residual} = \text{decoded residual} \times 2^{\text{quantizationLevel}}$$

3.3.6 Save Decoded audio

Same as in lossless, write the reconstructed samples back into an SFML SoundBuffer and save the output file.

3.4 Lossy Coding Results

The quantization works, as I will show in the next images, however I want to preface that there are some errors:

While decoding lossy binary files with a factor of "-q" $>= 9$ the decoding might output errors, this might mean that the quantization might be too much but its probably a fault with the way we calculate the residuals in the lossy version. I will show some examples with "-q" $>= 9$ but it might not work for all files when testing.

What this means is that while we are reducing greatly the size of the lossy encoded file compared to the lossless version, the output might not appear to have suffered great losses of information.

3.4.1 sample01.wav results

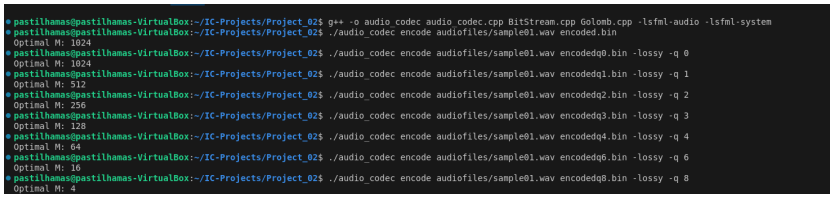

Figure 3.4: Commands that were ran for lossy encoding of sample01.wav

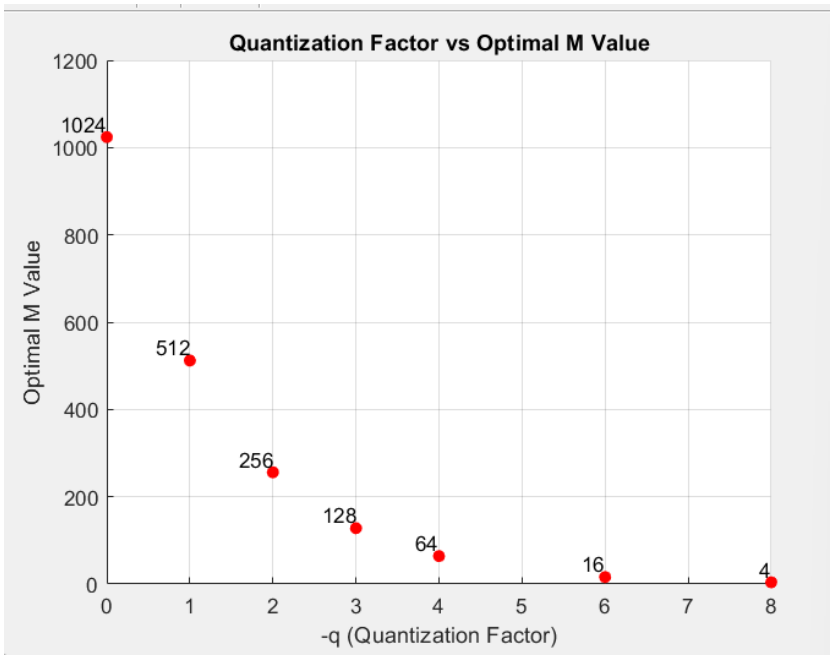

Figure 3.5: Optimal M value for varying -q in sample01.wav

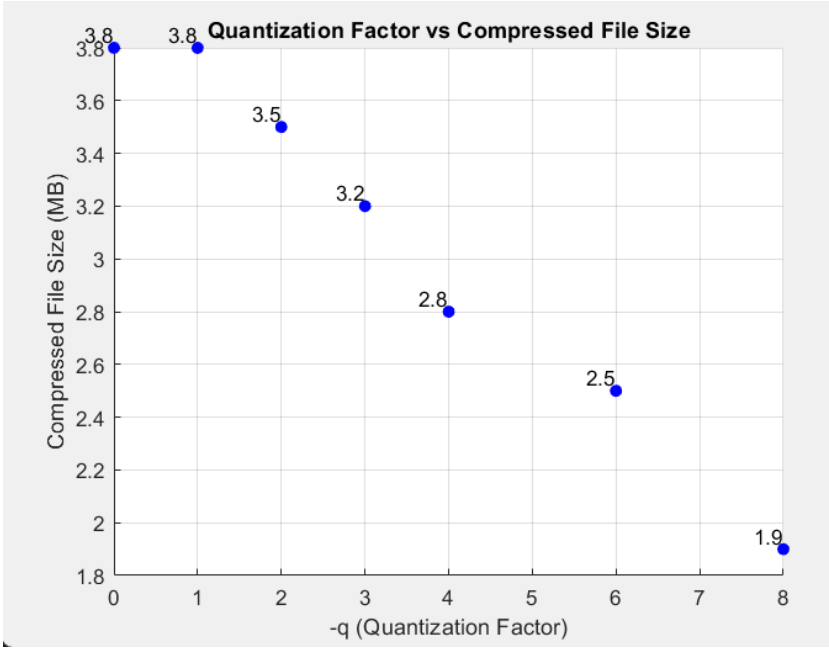

Figure 3.6: sample01.wav compressed file size for varying -q

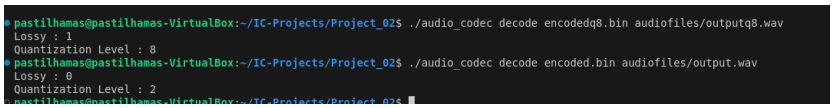

Figure 3.7: decompression of lossless and lossy with -q=8

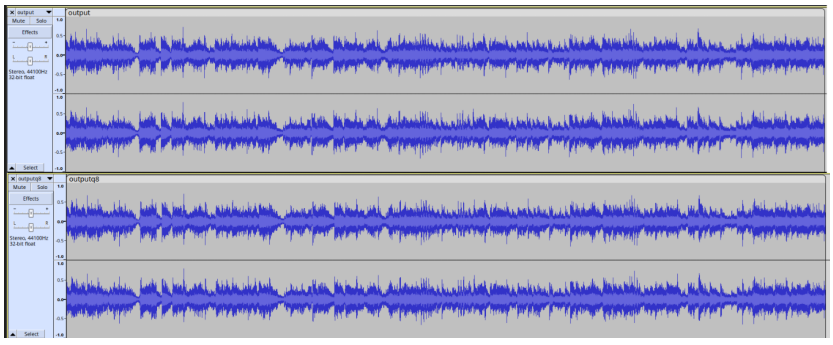

Figure 3.8: waveform comparison between a lossless and lossy with -q=8

While the compressed version of the lossy was 1.9 MB compared to the 3.8 MB of the lossless version, the output remained largely the same with no perceptible by waveform or acoustically by listening losses of information.

While we could quantize more aggressively, resulting in even smaller compressed file sizes, as I said in the beginning, I cannot guarantee that the output file will be listenable.

3.4.2 sample02.wav results

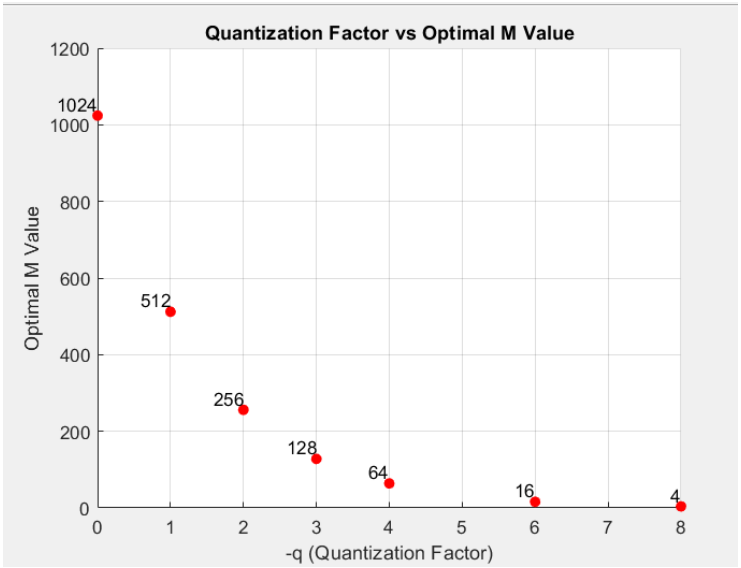

Figure 3.9: Optimal M value for varying -q in sample02.wav

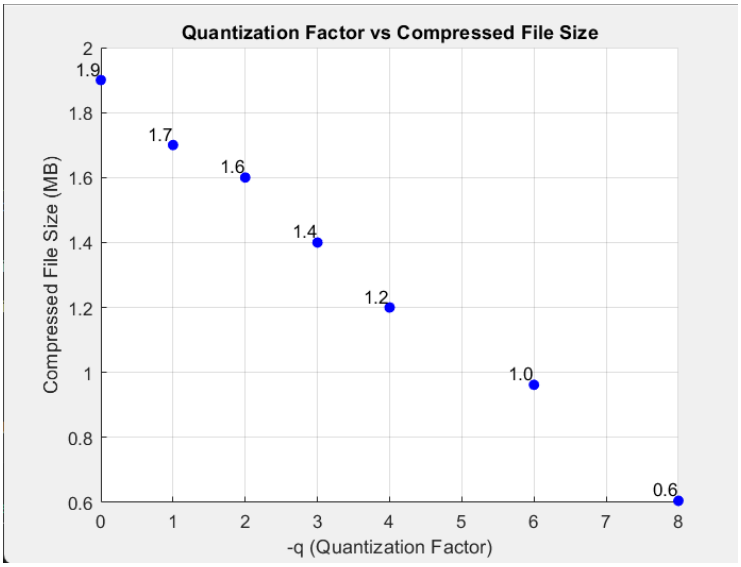

Figure 3.10: sample02.wav compressed file size for varying -q

3.4.3 sample03.wav results

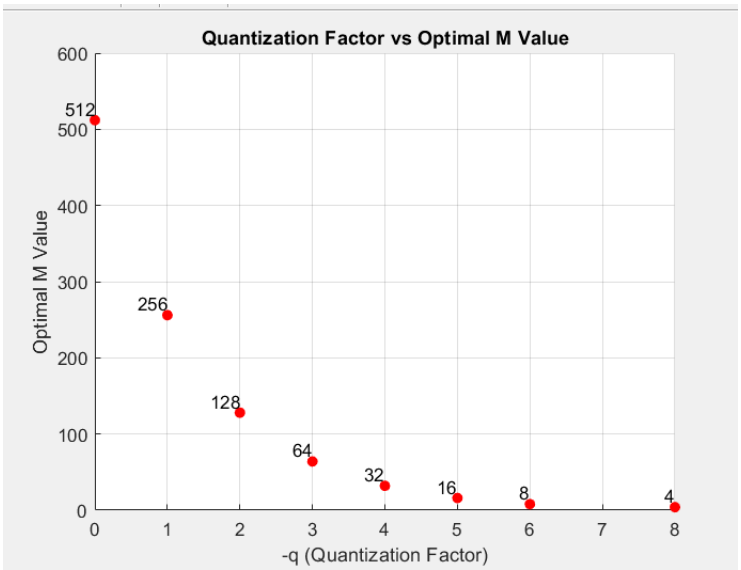

Figure 3.11: Optimal M value for varying -q in sample03.wav

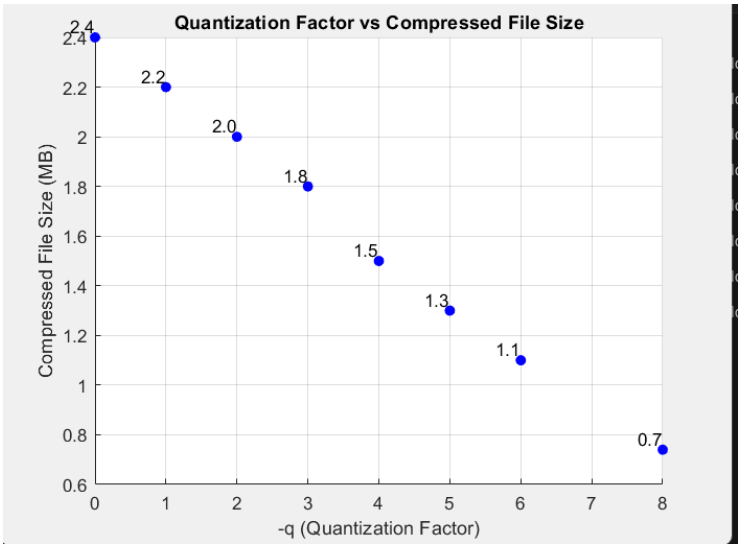

Figure 3.12: sample03.wav compressed file size for varying -q

3.4.4 sample05.wav results

Most of the other samples show the same results, with higher -q we choose and lower M value for the Golomb and get a smaller compressed file than the previous version while keeping most information. I will now show one example where we can see how the hard quantization ruins the information on the output file

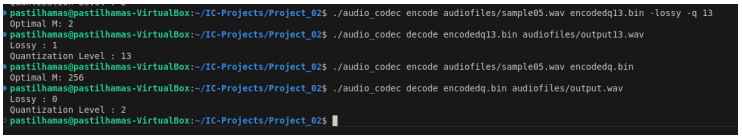

Figure 3.13: Commands we ran

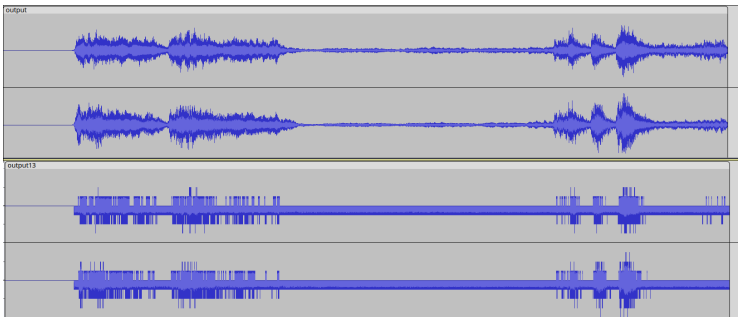

Figure 3.14: waveform comparison between lossless and lossy with a -q of 13

As we can see, in this waveform, we can notice losses of information and the introduction of noise.

Figure 3.15: file size comparison between lossless and lossy with a -q of 13

Chapter 4

Image and Video Coding with Predictive Coding

4.1 Lossless Image coding

4.1.1 Predictive Coding Approach

A spatial predictor is applied to estimate each pixel's intensity using its left neighbor. The prediction error (residual) is then calculated as:

$$prediction\_error = current\_pixel - predicted\_pixel$$

Since natural images contain smooth regions, most residuals are small values centered around zero, making them highly compressible using Golomb coding.

4.1.2 Golomb Encoding of Prediction Residuals

Once the prediction residuals are computed, they are encoded using Golomb coding. The efficiency of Golomb coding relies on choosing an optimal parameter **m**, which determines the encoding length.

To ensure efficient compression, the optimal **m** is dynamically computed from the mean absolute residual value:

$$m = max(1, round(mean\_residual\_magnitude))$$

This adaptive selection ensures that the encoded residuals require the fewest possible bits, improving compression efficiency.

4.1.3 Decoding and Reconstruction

The decoding process retrieves the encoded residuals from the bitstream using Golomb decoding and reconstructs the image using the same predictive model:

$$reconstructed\_pixel = prediction\_error + predicted\_pixel$$

Since the same predictor is applied symmetrically, the original pixel values are perfectly recovered, maintaining lossless reconstruction.

4.1.4 Tests and Results

For the image **airplane.ppm** we got the following results:

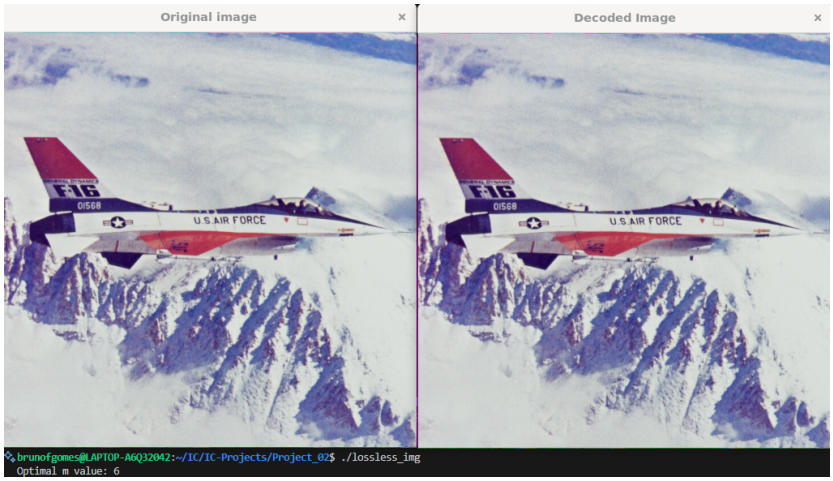

Figure 4.1: Original and Decoded airplane.ppm images

As we can observe, the images are basically the same and the chosen **m** value for the residuals of this image was 6, which makes sense since the whole image contains uniform colors.

Regarding file sizes for this image we got these values:

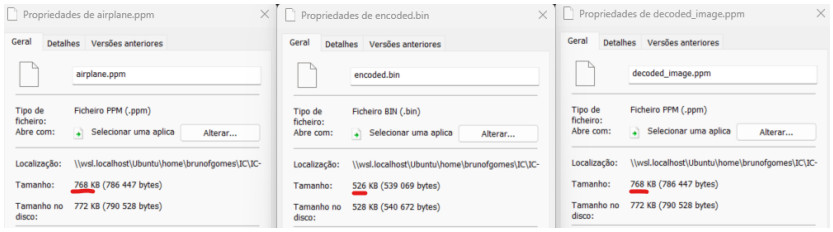

Figure 4.2: Original, binary and decoded files

For this specific image we managed to get a compression ratio of 68.5%. This means that the binary file size is 68.5% of the original file.

$$(526/768) * 100 \approx 68.5\%$$

For the image **baboon.ppm** we got the following results:

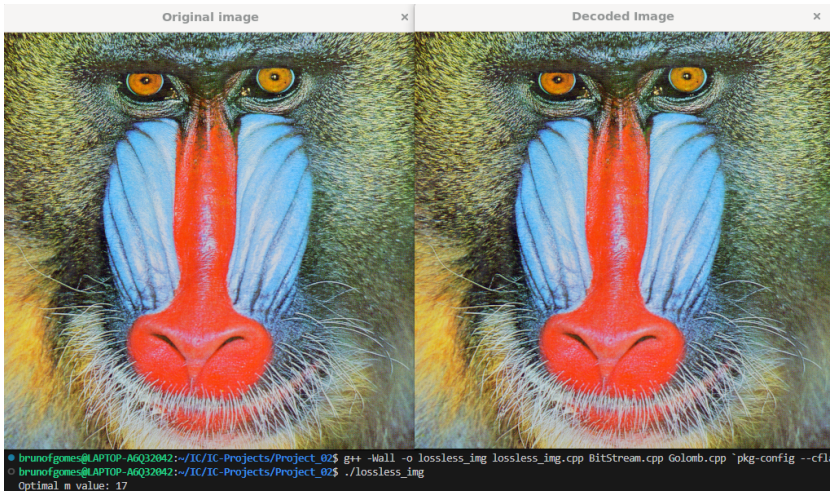

Figure 4.3: Original and Decoded baboon.ppm images

The original and decoded images are the same, however, the **m** value here is higher when compared to to the airplane.ppm image, because this image contains more vibrant colors, more details and contrasts, this causes the residual values to become higher, therefore requiring a higher **m** value to encode these values.

The file sizes for this image are:

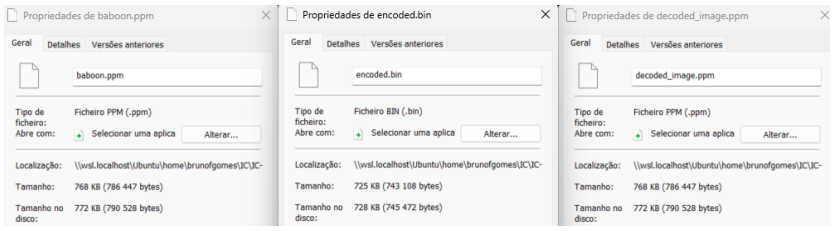

Figure 4.4: Original, binary and decoded files

The compression ratio is lower when compared to the previous image, due to the aspects referred above, here the binary file size is **94.4%** of the original file size.

For the image **bike3.ppm** we got the following results:

Figure 4.5: Original and Decoded bike3.ppm images

Since we are dealing with a high definition image the **m** value will also be higher, for this image the chosen value is 10.

The file sizes for this image are:

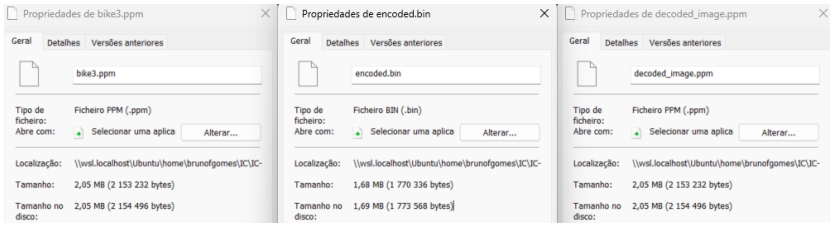

Figure 4.6: Original, binary and decoded files

In this example, the compressed binary file size is approximately **82%** of the original file. This reflects the codec difficulty in compressing more vibrant images.

Finally, we tested the image **tulips.ppm** and we got these results:

Figure 4.7: Original and Decoded tulips.ppm images

The chosen **m** value is 7 because this image contains more constant colors and has a lower definition.

The encoded and decoded file sizes are the following:

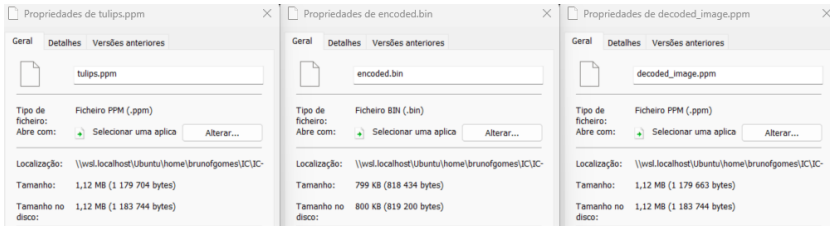

Figure 4.8: Original, binary and decoded files

As seen above, the binary file size if approximately **69.4%** of the original file, the compression here is higher when comparing with the two previous images, because of the more constant tones and lower resolution of the image.

4.2 Intra-Frame Video Coding

4.2.1 Introduction

Intra-frame video coding is a technique used to compress individual video frames independently using spatial prediction and entropy coding. Unlike inter-frame coding, which relies on differences between consecutive frames, intra-frame coding treats each frame as a standalone image. This method is crucial for I-frames (intra-coded frames) in video compression, ensuring that certain frames can be fully reconstructed without referencing past or future frames.

In this implementation, predictive coding is used to estimate pixel values based on neighboring pixels within the same frame, followed by Golomb coding

to efficiently encode the prediction residuals. This technique exploits spatial redundancy to reduce the amount of data required to store each frame.

4.2.2 Spatial Prediction for Intra-Frame Compression

- Each pixel in a frame is predicted using its left and top neighbors, reducing redundancy.
- The prediction residual is stored instead of the actual pixel value:

$$prediction\_error = current\_pixel - predicted\_pixel$$

- If a pixel is in the first row or first column, only one neighboring pixel is available for prediction.
- Otherwise, a weighted average of left and top neighbors is used.

By storing only the differences (residuals) instead of raw pixel values, the codec reduces entropy, making the data more compressible.

4.2.3 Golomb Coding for Residual Compression

The encoding and decoding of prediction residuals are performed in the same manner as in the lossless image codec, so there is no need to elaborate further on this topic.

4.2.4 Frame-by-Frame Decoding

- The decoder reads the encoded residuals from the bitstream and reconstructs each frame using the same prediction model.
- The original pixel value is restored as:

$$reconstructed\_pixel = predicted\_pixel + decoded\_residual$$

- The decoded frames are written to a **YUV4MPEG** file (decoded_video.y4m), preserving the original video format.

4.2.5 Quantization

It is also possible to perform lossy compression, by specifying a quantization level that will be propagated to all frames of the video, therefore reducing the file size with minimal quality losses.

This is achieved by dividing the residuals by the quantization level during encoding, and then multiplying the residuals by the quantization level during decoding to preserve the correct value scale.

4.2.6 Tests and Results

In this part, the decoded video file sizes are always larger than the original video file when no quantization is applied. This is incorrect, as the decoded file should always be equal or smaller than the original. However, we soon identified that this issue stems from how we handle YUV files. Specifically, we are not processing them correctly, which leads to noise (interpolation) being introduced in the file structure to align with our working method. In this project, we always work with three channels of the same dimensions, but YUV files do not follow this structure.

Firstly, we tested the **football_cif.y4m** video file, and we got the following results:

- With a quantization level of **1**:
  - **Original video size:** 38 612 KB
  - **Binary file size:** 54 580 KB
  - **Decoded video size:** 77 222 KB
  - **Codec processing time:** 35.46 seconds
  - **Optimal m value:** 6

  The binary file size is approximately **70%** of the size of the decoded video. This video is particularly hard to encode due to the high variation of pixels in each frame, this causes an increase in the range of residuals that need to be encoded.

- With a quantization level of **10**:
  - **Original video size:** 38 612 KB
  - **Binary file size:** 16 731 KB
  - **Decoded video size:** 77 222 KB
  - **Codec processing time:** 18.43 seconds
  - **Optimal m value:** 1

  In this test, we encode and decode the same video but with a quantization level of 10, which results in a drastic decrease in the size of the binary file, this expected due to the lesser range of residuals encoded. Since the residuals values reduce, the m value will also usually reduce. The binary file size is **21.7%** of the decoded video size, the compression is massive and there is only a marginal quality loss of the video.

The next video we tested was the **akiyo_cif.y4m** video file and we got these results:

- With a quantization level of **1**:
  - **Original video size:** 44 552 KB
  - **Binary file size:** 49 018 KB
  - **Decoded video size:** 89 102 KB
  - **Codec processing time:** 36.35 seconds
  - **Optimal m value:** 4

  The frames of this video present a more uniform pixel variation which makes it easier to predict pixels, therefore resulting in a more efficient process of encoding residuals. The binary file size is **55%** of the decoded video file, which means we compressed more than half of the video file.

- With a quantization level of **10**:
  - **Original video size:** 44 552 KB
  - **Binary file size:** 15 599 KB
  - **Decoded video size:** 89 102 KB
  - **Codec processing time:** 19.31 seconds
  - **Optimal m value:** 1

  The encoding and decoding process is way faster due to the range of residuals also being reduced, this caused a decrease in the binary file size that stores the residuals in almost half of the time of processing. The binary file size is approximately **18%** of the decoded video file.

With the video **flower_cif.y4m** we got these results:

- With a quantization level of **1**:
  - **Original video size:** 37 127 KB
  - **Binary file size:** 62 044 KB
  - **Decoded video size:** 74 252 KB
  - **Codec processing time:** 34.0 seconds
  - **Optimal m value:** 15

  This video features a diverse range of colors, making the choice of an m value of 15 appropriate. With multiple colors present in each frame, predicting pixel values becomes more challenging, resulting in a wider range of residual values. Overall, the frames lack large uniform areas of pixels, significantly complicating the encoding of residuals. The binary file size is **83.6%** of the decoded file size, only a small amount of compression was achieved.

- With a quantization level of **10**:
  - **Original video size:** 37 127 KB
  - **Binary file size:** 30 860 KB
  - **Decoded video size:** 74 252 KB
  - **Codec processing time:** 20.0 seconds
  - **Optimal m value:** 1

  With a quantization level of 10 we managed to reduce even more the binary file size, resulting in a bigger compression ratio, however, the compression is not as high when comparing with the other previously analyzed videos, due to the intricacies that were referenced above. The binary file size is **42%** of the decoded file size.

The final video that we tested was **foreman_cif.y4m**, and we got the following results:

- With a quantization level of **1**:
  - **Original video size:** 44 552 KB
  - **Binary file size:** 61 512 KB
  - **Decoded video size:** 89 102 KB
  - **Codec processing time:** 39.30 seconds
  - **Optimal m value:** 6

  In this example, the distribution of pixels is more or less uniform and does not require a high value of m to encode the residuals. The binary file size is **69%** of the decoded video file size.

- With a quantization level of **10**:
  - **Original video size:** 44 552 KB
  - **Binary file size:** 18 561 KB
  - **Decoded video size:** 89 102 KB
  - **Codec processing time:** 20.67 seconds
  - **Optimal m value:** 1

  With quantization we managed get a better compressed binary file with the loss of minimal quality. The binary file size is **21%** of the decoded video file size.

Note: The decoded video files are consistently twice the size of the original files due to improper handling of files that follow the YUV structure.

4.3 Inter-Frame Video Coding

4.3.1 Introduction

Inter-frame video coding is a technique that exploits temporal redundancy by encoding only the differences between consecutive frames, rather than encoding each frame independently. This is essential for P-frames (predicted frames) in video compression, as they depend on previously encoded I-frames (intra-coded frames) for reconstruction.

This implementation uses motion estimation to find the best-matching blocks between the current frame and a reference frame, reducing the amount of data required to encode a video sequence. The motion vectors and residuals are then encoded using Golomb coding to maximize compression efficiency.

4.3.2 I-Frame Encoding (Intra-Frame Compression)

- Every iframe_interval frames, a keyframe (I-frame) is encoded using spatial prediction and predictive coding, just like in intra-frame coding.
- The residuals (difference between the predicted and actual pixel values) are encoded using Golomb coding, with an optimal m value computed dynamically with a formula that was already presented before, the only difference here is that instead of calculating an optimal m based on all residuals of all frames, in this specific case each frame will have its own optimal m based on the residuals of that specific frame. The m value is encoded in the header of each frame, along with the motion vectors.

4.3.3 P-Frame Encoding (Inter-Frame Compression)

- Instead of storing the entire frame, motion estimation is used to find the best-matching blocks in the previous frame (reference frame).
- The encoder searches within a search_range to find the best displacement vector (motion vector) for each block.

4.3.4 Decoding and Reconstruction

- I-frames are decoded independently using intra-frame prediction.
- P-frames are reconstructed using motion compensation:
    1. Motion vectors are read from the bitstream.
    2. The reference frame is used to generate an approximation of the current frame.
    3. The decoded residuals are added back to the motion-compensated pixels, ensuring reconstruction accuracy.

4.3.5 Tests and Results

All these tests were made with the following arguments:

Parameter	**Value**
Quantization level	variable
I-frame interval	15
Block size	16
Search range	8

Table 4.1: Video Encoding Parameters

The only change we made was to the quantization levels.

Starting with the video **football_cif.y4m**, the results were:

- With a quantization level of **1**:
  - **Original video size:** 38 612 KB
  - **Binary file size:** 53 920 KB
  - **Decoded video size:** 77 222 KB
  - **Codec processing time:** 6:49 min

  As we can see in the codec processing time, this is extremely inefficient and requires an optimization, comparing this result with the intra-frame video coding result we were only able to encode more 660 KB (54 580 - 53 920). For the time spent in the coding an decoding process, the saved space does not benefit us much.

- With a quantization level of **10**:
  - **Original video size:** 38 612 KB
  - **Binary file size:** 17 106 KB
  - **Decoded video size:** 77 222 KB
  - **Codec processing time:** 6:33 min

  Here the compression ratio is almost the same when comparing to our intra-frame video codec, again the biggest problem is the time it takes to perform the processing.

Due to the long processing times, we decided to test only one video and compare it to our intra-frame video codec. The compression gains were minimal.

An area for future improvement would be to optimize the codec's information processing.

.

Chapter 5

Appendices

https://github.com/BrunoGomes22/IC-Projects/tree/main/Project_02

Work Contributions

Both authors contributed equally towards the realization of this project, asking for 50% each.