Universidade de Aveiro

Information and Coding lab work nº2

Diogo Martins (108548) | Luís Sousa (108583) | Tomás Viana (108445)

Departamento de Eletrónica, Telecomunicações e Informática

16/11/2025

Index

Summary

The project consists of multiple C++ files with the purpose of manipulating audio and video files to analyze their properties. Part I is coded with the help of OpenCV, a computer vision library that mainly provides infrastructure to read, manipulate and display all kinds of image files, while Parts II-III are structured with Colomb coding, a lossless data compression method used for audio and image files, and Part IV uses both features.

Part 1

Exercise 1

For this exercise, we implemented **extract_channel.cpp**, a program able to extract a channel from any PPM image and create another with just one channel, reading and writing pixel by pixel using the OpenCV library.

The program begins by loading the input image as a 3-channel color image, where each pixel contains separate values for the blue, green, and red components (in that order). A new empty output image is then created, and the program iterates through the original image pixel by pixel, copying only the value of the selected color channel into the corresponding position of the output image. All other channel values are ignored. It is important to remember that OpenCV stores color images in the BGR format rather than the more common RGB format. This means:

- Channel 0 corresponds to the blue component;
- Channel 1 corresponds to the green component;
- Channel 2 corresponds to the red component.

The output is as follows:

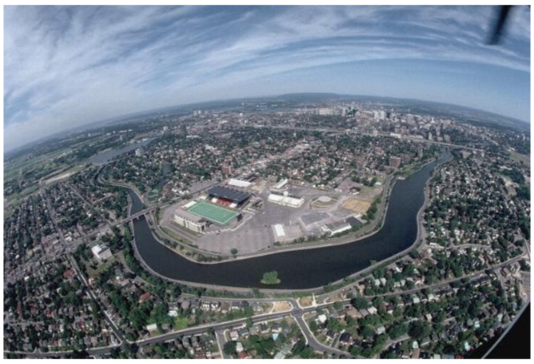


Figure 1 – Original arial image.

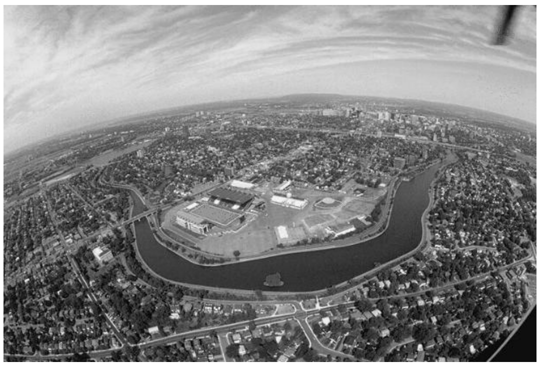


Figure 2 - Arial image with only channel 0.

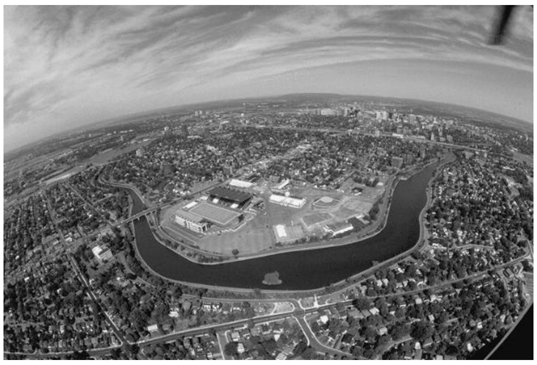


Figure 3 - Arial image with only channel 1.

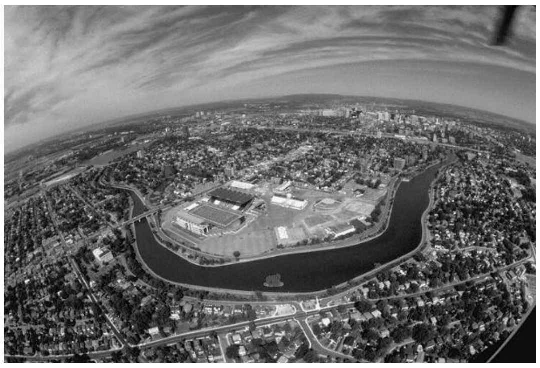


Figure 4 - Arial image with only channel 2.

As expected, some images seem to have slightly different brightness which is proof that the program is working correctly. When we extract a channel from an image, we also pull the intensity of that channel and since it has no color on its own, it's just a number, OpenCV displays it as grayscale. Intensity represents brightness and each color contains different intensity values, which means that in the areas where a specific color from that channel is more prevalent and stronger, the image will seem brighter. In this image blue is a more prevalent color than green and red, so the channel 0 looks brighter than to the others.

Exercise 2

For this exercise, we implemented four programs, **negative_img.cpp**, **mirror_img.cpp**, **rotate_img.cpp**, **modifyLight_img.cpp**, which respectively create a negative version of an image, a mirrored one, rotates it by multiples of 90º and increases/decreases its intensity values (brightness). While no existing manipulation functions of OpenCV were used, we still used it to read and write the images.

Negative_img reads the input image and individually inverts the intensity of each pixel for each channel, writing them into the empty image. Channels store values between 0-255, so the expression to calculate the negative value is:

- neg_value = 255 - og_value

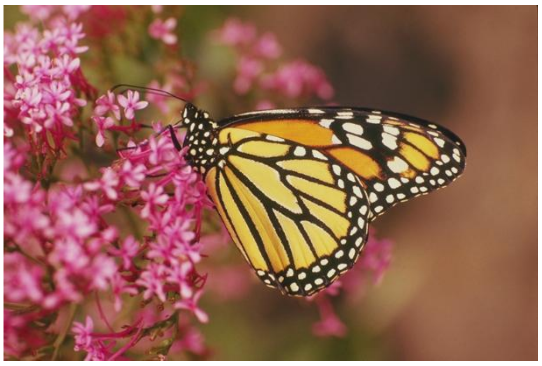


Figure 6 - Original butterfly.

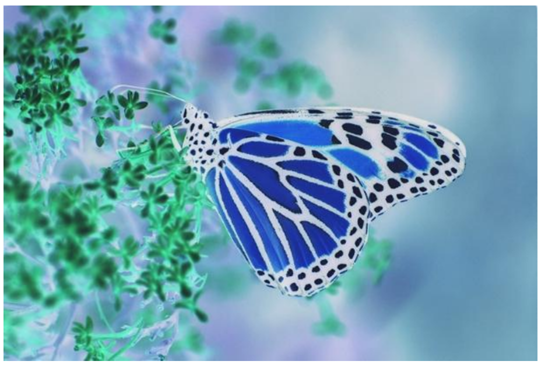


Figure 5 - Negative butterfly.

Mirror_img loads the input image and, according to whether the user requests a horizontal or a vertical flip, flips its columns or rows around respectively. In a horizontal flip the right columns switch with the left and in a vertical flip the upper rows switch with the lower. In code this switch can be done with these expressions:

- flip_col = width - og_col - 1
- flip_row = height - og_row - 1

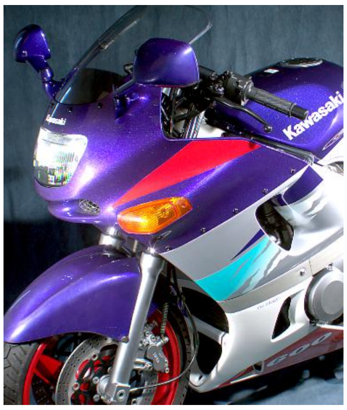


Figure 9 - Original bike.

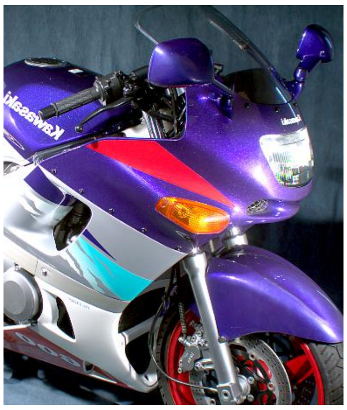


Figure 8 - Horizontally mirrored bike.

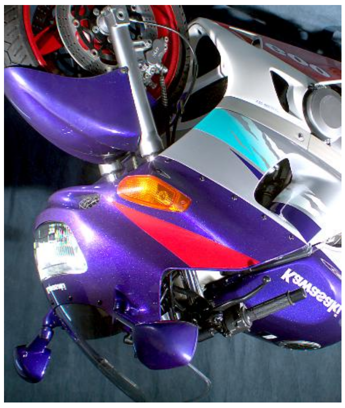


Figure 7 - Vertically mirrored bike.

Rotate_img reads the input image and rotates the full pixel grid that composes the image clockwise according to the angle requested by the user. That angle must be a multiple of 90º since with those degrees it is only necessary to swap coordinates and adjust indices, avoiding interpolation. For every 90º the grid suffers one rotation until it reaches full circle (360º) and switches the rows and columns accordingly:

- new_row = og_col
- new_col = og_row - new_row - 1

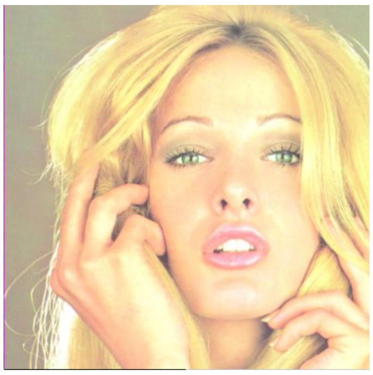


Figure 11 - Original girl.

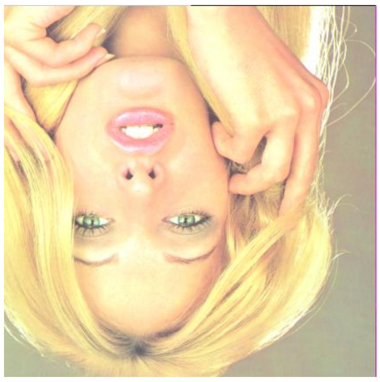


Figure 10 - 90º turned girl.

ModifyLight_img loads the input image and increases or decreases the brightness of each pixel by changing their intensity values, which can vary between -255 and 255. On all three channels the pixels get added to the intensity value input by the user, if the value is positive the image gets brighter, if the value is negative the image gets darker. If the resulting value is below 0 or above 255 it gets clamped, since the pixel can't be darker than black nor lighter than white.

- new_value = saturate_cast(og_value + int_value)

The output is as follows:

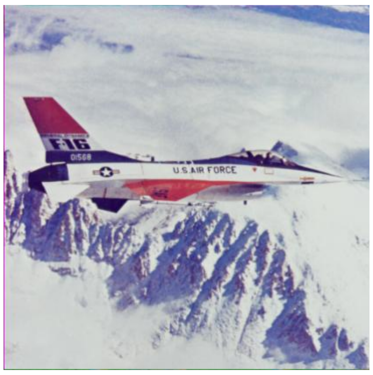


Figure 12 - Original airplane.

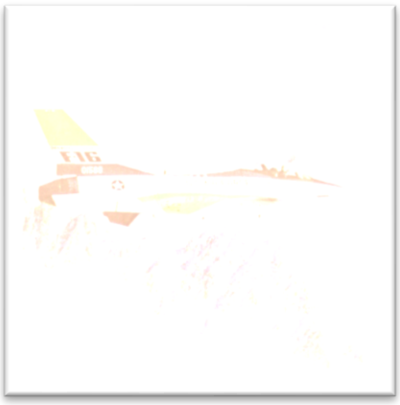


Figure 13 - Lightened airplane.

Part 2

Exercise 3

For this exercise, we implemented three programs, **golomb.cpp**, **golomb.h** and **golomb_test.cpp**, which consist of a codec that encodes bits into integers and decodes integers into bits, following standard Golomb code.

Golomb.h is the header file that defines all variables, structures, and functions required to work with Golomb codes. It specifies the two supported methods for handling negative numbers (either through a dedicated sign bit or via positive/negative interleaving), declares private Golomb-specific parameters such as the divisor m, the number of bits b, the cutoff threshold, and exposes the public interface used by golomb.cpp. This interface includes the constructor, accessor methods, and all functions responsible for encoding and decoding.

Golomb.cpp is the main file containing the full logic of the Golomb coding algorithm. The constructor consists of initializing all internal parameters based on the user-provided value of "m", the parameter that determines how the code splits a number. Specifically, it computes the number of bits required in the truncated binary representation and the cutoff which determines how the remainder is split between (b–1)-bit and b-bit encodings. The constructor also validates that m is strictly greater than zero and stores the negative-number handling mode selected by the user.

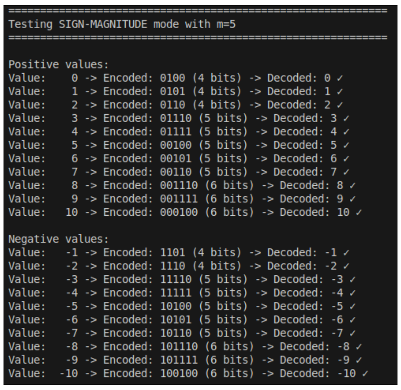
- $b = \text{ceil}(\log_2(m))$
- $c = 2^b - m$

The file also implements the two helper functions mapToUnsigned and mapToSigned which convert signed integers to an unsigned domain and vice-versa, necessary because Golomb coding operates on non-negative integers and the program supports two distinct strategies for incorporating negative values.

The **encoder** performs the full Golomb encoding. It begins by handling the sign (if using sign-magnitude mode), then computes the quotient and remainder of the mapped integer. The quotient is encoded using unary coding and the remainder is encoded using truncated binary coding, which ensures optimal bit length depending on whether the remainder falls below or above the cutoff. Conversely, the **decoder** function reverses the process by reading the optional sign bit, determining the quotient from the unary prefix, reconstructing the remainder using truncated binary logic, and finally mapping the reconstructed integer back to the appropriate signed integer. The function returns both the decoded value and the number of bits consumed, enabling sequential decoding of multiple values from a single bitstream.

There's also **Golomb_test.cpp**, a self-contained with the sole purpose of verifying the correctness of the Colomb code. This includes testing the negative-number modes, values of "m" and overall ensure the implementation behaves robustly. Only with this file we can visualize if our code is working as intended.

The output is as follows:

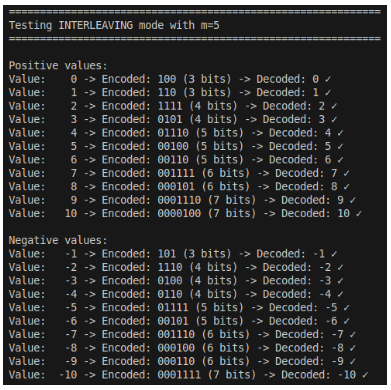


```
============================================================
Testing SIGN-MAGNITUDE mode with m=5
============================================================

Positive values:
Value:    0 -> Encoded: 0100 (4 bits) -> Decoded: 0 ✓
Value:    1 -> Encoded: 0101 (4 bits) -> Decoded: 1 ✓
Value:    2 -> Encoded: 0110 (4 bits) -> Decoded: 2 ✓
Value:    3 -> Encoded: 01110 (5 bits) -> Decoded: 3 ✓
Value:    4 -> Encoded: 01111 (5 bits) -> Decoded: 4 ✓
Value:    5 -> Encoded: 00100 (5 bits) -> Decoded: 5 ✓
Value:    6 -> Encoded: 00101 (5 bits) -> Decoded: 6 ✓
Value:    7 -> Encoded: 00110 (5 bits) -> Decoded: 7 ✓
Value:    8 -> Encoded: 001110 (6 bits) -> Decoded: 8 ✓
Value:    9 -> Encoded: 001111 (6 bits) -> Decoded: 9 ✓
Value:   10 -> Encoded: 000100 (6 bits) -> Decoded: 10 ✓

Negative values:
Value:   -1 -> Encoded: 1101 (4 bits) -> Decoded: -1 ✓
Value:   -2 -> Encoded: 1110 (4 bits) -> Decoded: -2 ✓
Value:   -3 -> Encoded: 11110 (5 bits) -> Decoded: -3 ✓
Value:   -4 -> Encoded: 11111 (5 bits) -> Decoded: -4 ✓
Value:   -5 -> Encoded: 10100 (5 bits) -> Decoded: -5 ✓
Value:   -6 -> Encoded: 10101 (5 bits) -> Decoded: -6 ✓
Value:   -7 -> Encoded: 10110 (5 bits) -> Decoded: -7 ✓
Value:   -8 -> Encoded: 101110 (6 bits) -> Decoded: -8 ✓
Value:   -9 -> Encoded: 101111 (6 bits) -> Decoded: -9 ✓
Value:  -10 -> Encoded: 100100 (6 bits) -> Decoded: -10 ✓
```

Figure 14 - Sign-Magnitute test (m=5).

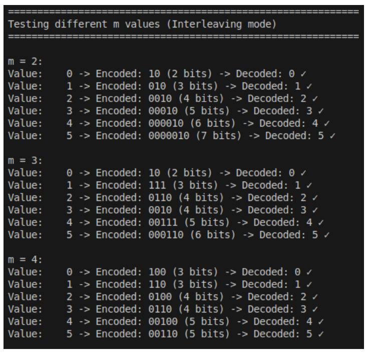


```
============================================================
Testing INTERLEAVING mode with m=5
============================================================

Positive values:
Value:    0 -> Encoded: 100 (3 bits) -> Decoded: 0 ✓
Value:    1 -> Encoded: 110 (3 bits) -> Decoded: 1 ✓
Value:    2 -> Encoded: 1111 (4 bits) -> Decoded: 2 ✓
Value:    3 -> Encoded: 0101 (4 bits) -> Decoded: 3 ✓
Value:    4 -> Encoded: 01110 (5 bits) -> Decoded: 4 ✓
Value:    5 -> Encoded: 00100 (5 bits) -> Decoded: 5 ✓
Value:    6 -> Encoded: 00110 (5 bits) -> Decoded: 6 ✓
Value:    7 -> Encoded: 001111 (6 bits) -> Decoded: 7 ✓
Value:    8 -> Encoded: 000101 (6 bits) -> Decoded: 8 ✓
Value:    9 -> Encoded: 0001110 (7 bits) -> Decoded: 9 ✓
Value:   10 -> Encoded: 0000100 (7 bits) -> Decoded: 10 ✓

Negative values:
Value:   -1 -> Encoded: 101 (3 bits) -> Decoded: -1 ✓
Value:   -2 -> Encoded: 1110 (4 bits) -> Decoded: -2 ✓
Value:   -3 -> Encoded: 0100 (4 bits) -> Decoded: -3 ✓
Value:   -4 -> Encoded: 0110 (4 bits) -> Decoded: -4 ✓
Value:   -5 -> Encoded: 01111 (5 bits) -> Decoded: -5 ✓
Value:   -6 -> Encoded: 00101 (5 bits) -> Decoded: -6 ✓
Value:   -7 -> Encoded: 001110 (6 bits) -> Decoded: -7 ✓
Value:   -8 -> Encoded: 000100 (6 bits) -> Decoded: -8 ✓
Value:   -9 -> Encoded: 000110 (6 bits) -> Decoded: -9 ✓
Value:  -10 -> Encoded: 0001111 (7 bits) -> Decoded: -10 ✓
```

Figure 15 - Interleaving test (m=5).

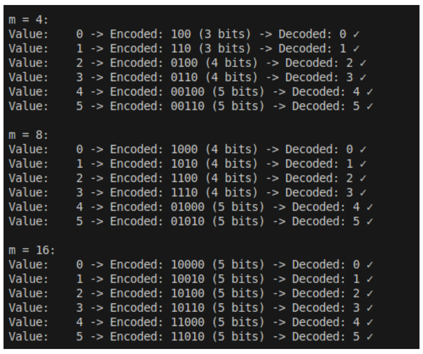


```
============================================================
Testing different m values (Interleaving mode)
============================================================

m = 2:
Value:    0 -> Encoded: 10 (2 bits) -> Decoded: 0 ✓
Value:    1 -> Encoded: 010 (3 bits) -> Decoded: 1 ✓
Value:    2 -> Encoded: 0010 (4 bits) -> Decoded: 2 ✓
Value:    3 -> Encoded: 00010 (5 bits) -> Decoded: 3 ✓
Value:    4 -> Encoded: 000010 (6 bits) -> Decoded: 4 ✓
Value:    5 -> Encoded: 0000010 (7 bits) -> Decoded: 5 ✓

m = 3:
Value:    0 -> Encoded: 10 (2 bits) -> Decoded: 0 ✓
Value:    1 -> Encoded: 111 (3 bits) -> Decoded: 1 ✓
Value:    2 -> Encoded: 0110 (4 bits) -> Decoded: 2 ✓
Value:    3 -> Encoded: 0010 (4 bits) -> Decoded: 3 ✓
Value:    4 -> Encoded: 00111 (5 bits) -> Decoded: 4 ✓
Value:    5 -> Encoded: 000110 (6 bits) -> Decoded: 5 ✓

m = 4:
Value:    0 -> Encoded: 100 (3 bits) -> Decoded: 0 ✓
Value:    1 -> Encoded: 110 (3 bits) -> Decoded: 1 ✓
Value:    2 -> Encoded: 0100 (4 bits) -> Decoded: 2 ✓
Value:    3 -> Encoded: 0110 (4 bits) -> Decoded: 3 ✓
Value:    4 -> Encoded: 00100 (5 bits) -> Decoded: 4 ✓
Value:    5 -> Encoded: 00110 (5 bits) -> Decoded: 5 ✓
```

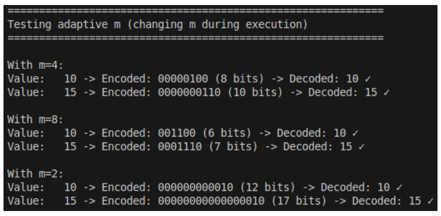


```
m = 4:
Value:    0 -> Encoded: 100 (3 bits) -> Decoded: 0 ✓
Value:    1 -> Encoded: 110 (3 bits) -> Decoded: 1 ✓
Value:    2 -> Encoded: 0100 (4 bits) -> Decoded: 2 ✓
Value:    3 -> Encoded: 0110 (4 bits) -> Decoded: 3 ✓
Value:    4 -> Encoded: 00100 (5 bits) -> Decoded: 4 ✓
Value:    5 -> Encoded: 00110 (5 bits) -> Decoded: 5 ✓

m = 8:
Value:    0 -> Encoded: 1000 (4 bits) -> Decoded: 0 ✓
Value:    1 -> Encoded: 1010 (4 bits) -> Decoded: 1 ✓
Value:    2 -> Encoded: 1100 (4 bits) -> Decoded: 2 ✓
Value:    3 -> Encoded: 1110 (4 bits) -> Decoded: 3 ✓
Value:    4 -> Encoded: 01000 (5 bits) -> Decoded: 4 ✓
Value:    5 -> Encoded: 01010 (5 bits) -> Decoded: 5 ✓

m = 16:
Value:    0 -> Encoded: 10000 (5 bits) -> Decoded: 0 ✓
Value:    1 -> Encoded: 10010 (5 bits) -> Decoded: 1 ✓
Value:    2 -> Encoded: 10100 (5 bits) -> Decoded: 2 ✓
Value:    3 -> Encoded: 10110 (5 bits) -> Decoded: 3 ✓
Value:    4 -> Encoded: 11000 (5 bits) -> Decoded: 4 ✓
Value:    5 -> Encoded: 11010 (5 bits) -> Decoded: 5 ✓
```

Figure 16 & 17 - Interleaving test (multiple m).



```
============================================================
Testing adaptive m (changing m during execution)
============================================================

With m=4:
Value:   10 -> Encoded: 00000100 (8 bits) -> Decoded: 10 ✓
Value:   15 -> Encoded: 0000000110 (10 bits) -> Decoded: 15 ✓

With m=8:
Value:   10 -> Encoded: 001100 (6 bits) -> Decoded: 10 ✓
Value:   15 -> Encoded: 0001110 (7 bits) -> Decoded: 15 ✓

With m=2:
Value:   10 -> Encoded: 000000000010 (12 bits) -> Decoded: 10 ✓
Value:   15 -> Encoded: 00000000000000010 (17 bits) -> Decoded: 15 ✓
```

Figure 18 – Adaptive test.

Part 3

Exercise 4

For this exercise we implemented a lossless audio codec composed of **audio_codec.cpp**, **audio_encode.cpp**, **audio_decode.cpp** and **audio_codec.h**, that can encode any 16-bit PCM WAV and decodes it back to a WAV without any major losses. These programs both uses Golomb code and logic from some programs in Lab work nº1.

Audio_codec.h is the header file, so as standard it defines all interfaces needed by both encoder and decoder such as the AudioBuffer which serves as a container for audio in memory, the BitReader that defines tools to read streams of bits in multiple ways and the BitWriter that writes those bits. The API for both encoders and decoders is also defined here.

Audio_codec.cpp contains most of the codec logic, including all the implementation of what header defines and the encoder and decoder logic. The WAV reader loads the audio and, if it is valid, fills in the data that it needs, such as number of channels, sample rate, bits per sample, while also computing the number of audio samples. WAV writer builds a valid WAV file with the data that should be already filled in the struct. It starts by writing some requirements (WAV files need to start with “RIFF”, “WAVE” and “fmt”) and then stores some of the data directly, while other data needs to be computed first:

- ByteRate = SampleRate x NumChannels x BitsPerSample/8
- BlockAlign = NumChannels x BitsPerSample/8
- ChunkSize = 4 + (8 + Subchunk1Size) + (8 + dataSize)

The BitReader and BitWriter are bitstream classes that pack and unpacks bits into/from bytes, necessary since our codec produces sequences of bits but WAV files work with bytes. So, the BitWriter acts during the encoding of the audio and the BitReader during the decoding of the block bytes.

Audio_encoder.cpp is a wrapper that merely serves the purpose of handling the user input, parses some optional flags and calls the encodeFile() function from **audio_codec.cp**, where the actual logic behind the encoder lies, so that it can build the executable. **Audio_decoder.cpp** is also a wrapper for the decoder that serves a similar purpose to the previous file, it handles the user input, parses some optional flags and calls the decodeFile() function from **audio_codec.cp**, where the actual logic behind the decoder lies.

The output is as follows:

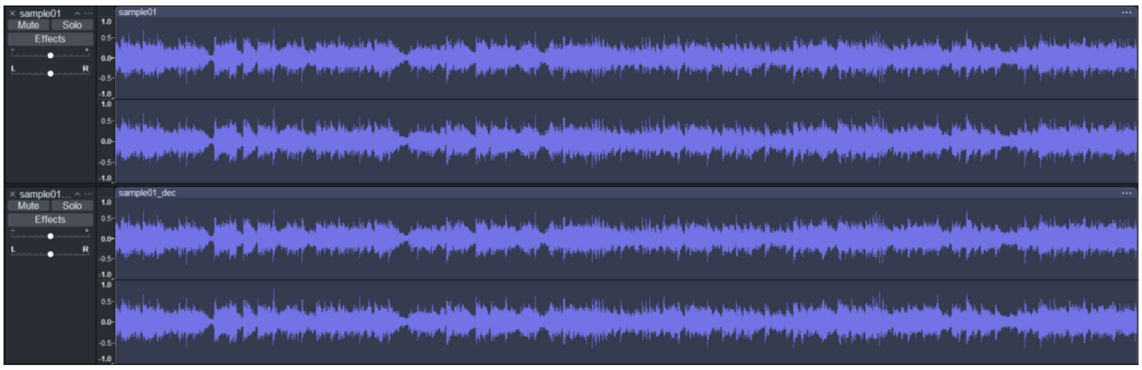


Figure 19 - Comparison between the original waveform and the decoded.

Part 4

Exercise 5

For this exercise we implemented a lossless image codec composed of **image_codec.cpp**, **image_encode.cpp**, **image_decode.cpp** and **image_codec.h**, that can encode any 8-bit grayscale image and decode it back to an image without losing any pixel information. Like the audio files, this codec uses Golomb coding and pixel prediction techniques similar to those explored in the previous work.

Image_codec.h is the header file that defines the public interface required by both the encoder and the decoder. It specifies the ImageCodec class, which is responsible for all prediction and Golomb-based compression/decompression operations and defines standard predictors of PNG and JPEG lossless compression such as:

- Previous pixel
- Above pixel
- Average
- Paeth
- JPEG-LS
- Gradient

Image_codec.cpp is once again where the main code resides. The image is processed pixel by pixel and for each one a prediction is calculated using neighboring pixels already processed, and instead of encoding the raw pixel value the encoder computes a residual (the difference between the actual pixel and the predicted value). All residuals are analyzed to determine an optimal Golomb parameter "m" which improves compression efficiency. The encoded bitstream starts with a fixed header where the image width, height, predictor mode and selected "m" value are stored so that the decoder can properly restore the image.

Image_encode.cpp is a wrapper for the encoder that simply serves the purpose of handling user input, loading the input image with OpenCV and calling the encode() function from image_codec.cpp, where the actual encoder logic lies. The produced bitstream is then packed into bytes and written into a binary file. At the end, the program prints basic compression statistics, including original size, compressed size and compression ratio.

Image_decode.cpp is also a wrapper for the decoder and serves a similar purpose. It reads the encoded binary file, unpacks the stored bits and calls the decode() function from image_codec.cpp. The decoder reconstructs the image by applying the same predictor and Golomb decoding logic and outputs a fully restored grayscale image, which is then saved using OpenCV.

The output is as follows:

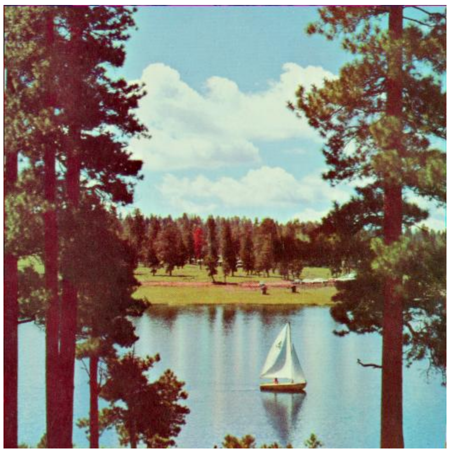


Figure 20 - Original boat (to be decoded).

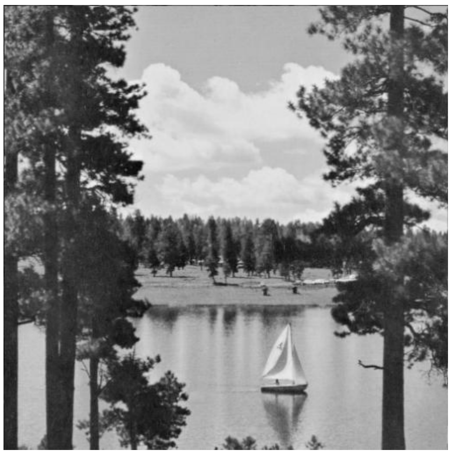


Figure 21 - Decoded boat.

Original size: 262144 bytes

Compressed size: 191928 bytes

Compression ratio: 1.36584 : 1

The decoded image (Figure 21 – Decoded boat) is grayscale because the codec is designed only for 8-bit grayscale images. When the encoder loads the input, color channels are discarded, and only luminance information is retained. This does not constitute a loss of information—all structural and luminance data remains fully intact in the decoded image.