IC: Project 2

Universidade de Aveiro
DETI

Daniel Silva, (102537) / João Andrade, (103361)
Tiago Santos, (89356)

Contents

1 Introduction

The goal of this project is to process audio and image files, beginning with basic image manipulations using OpenCV. We then focus on compressing audio files using a Golomb code, which will be used in a subsequent lossless audio codec.

2 Part I - Image Effects

This section consists of using OpenCV to edit image files, be it to extract a single color channel from an image or apply effects to it. To achieve this we relied heavily on the *at* function from OpenCV, which allows us to iterate through all the image pixels in an easier manner.

Since the first and second exercises consist of similar operations we grouped up everything into the ppm_effects.h file. All the methods in this file are part of a namespace, just like OpenCV.

We created a bash script to test all of our image processing functions at different values, all the results obtained can be found on our GitHub repository: `https://github.com/jrpfaria/IC/assignment2/result_images`

2.1 Extract

To extract a color channel from a given image we start by creating an empty single-channel image with the same size as the input image. Following that, we iterate through the pixels of this empty image setting their values to the values present on the desired channel of the original image.

Here is our implementation for the extract function:

```
static Mat extract(Mat image, char color) {
    Mat result = Mat::zeros(image.rows, image.cols, CV_8UC1);
    int ch = 0;
    switch(color){
        case 'b': ch = 0; break;
        case 'g': ch = 1; break;
        case 'r': ch = 2; break;
        default: break;
    }
    for (int i = 0; i < image.rows; i++)
        for (int j = 0; j < image.cols; j++)
            result.at<uchar>(i,j) = image.at<Vec3b>(i,j)[ch];
    return result;
}
```

2.1.1 Usage

```
../ppm_effects inputFile outputFile extract <b|g|r>
```

2.1.2 Results

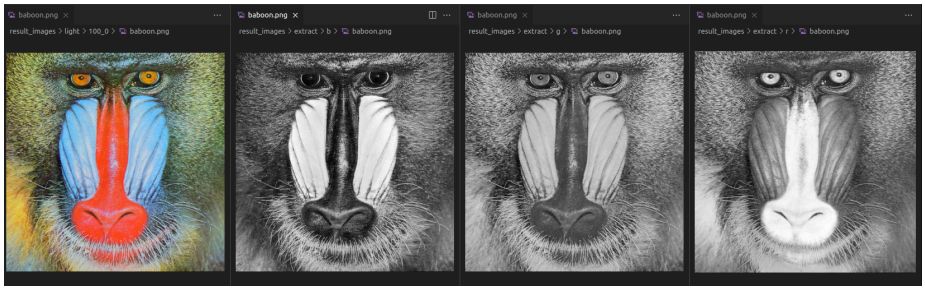

Figure 1: Results from extracting each of the BGR channels to baboon.ppm

2.2 Negative

In order to obtain the negative of an image we create an empty image with the same size as the original and iterate through each pixel. The BGR values of a pixel in the negative image is obtained from the complementary value of each one of its values in the original image.

```
static Mat negative(Mat image) {
    Mat result = Mat::zeros(image.rows, image.cols, CV_8UC3);
    for (int i = 0; i < image.rows; i++)
        for (int j = 0; j < image.cols; j++)
            for (int k = 0; k < 3; k++)
                result.at<Vec3b>(i,j)[k] =
                    0xFF - image.at<Vec3b>(i,j)[k];
    return result;
}
```

2.2.1 Usage

```
../ppm_effects inputFile outputFile negative
```

2.2.2 Results

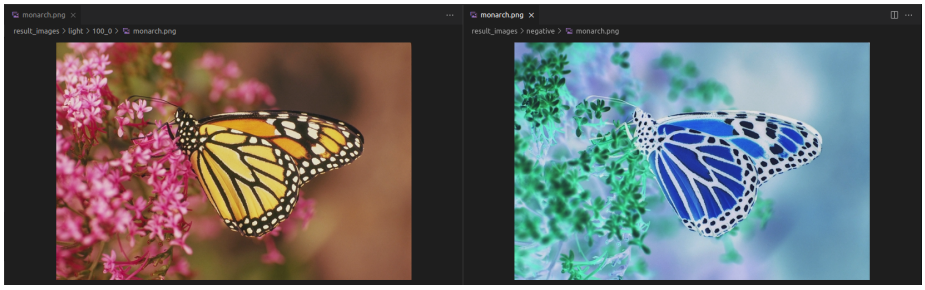

Figure 2: Result for monarch.ppm

2.3 Mirror

To achieve a mirroring effect, we create an empty image with the same size as the original image, and then we iterate through the original in a reversed manner for one of the axes, depending on the effect we want to achieve. This can be seen in the following code block:

```
static Mat mirrorHorizontally(Mat image) {
    Mat result = Mat::zeros(image.rows, image.cols, CV_8UC3);
    for (int i = 0; i < image.rows; i++)
        for (int j = 0; j < image.cols; j++)
            for (int k = 0; k < 3; k++)
                result.at<Vec3b>(i,j)[k] =
                    image.at<Vec3b>(i,image.cols - j)[k];
    return result;
}
static Mat mirrorVertically(Mat image) {
    Mat result = Mat::zeros(image.rows, image.cols, CV_8UC3);
    for (int i = 0; i < image.rows; i++)
        for (int j = 0; j < image.cols; j++)
            for (int k = 0; k < 3; k++)
                result.at<Vec3b>(i,j)[k] =
                    image.at<Vec3b>(image.rows - i,j)[k];
    return result;
}
```

2.3.1 Usage

```
../ppm_effects inputFile outputFile mirror <h | v>
```

2.3.2 Results

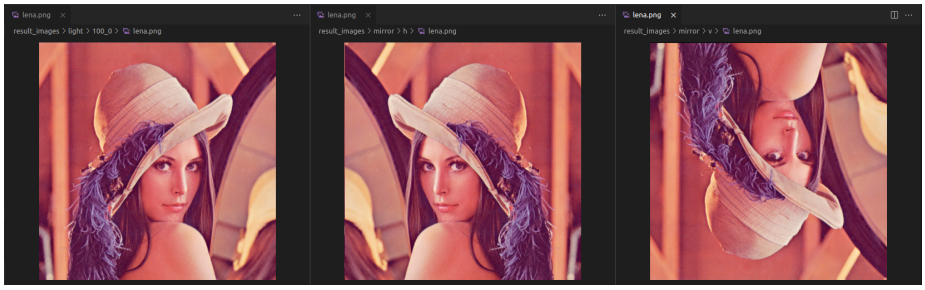

Figure 3: Results for mirroring lena.ppm horizontally and vertically

2.4 Rotate

To rotate an image we first adjust the input angle to ensure it is in the range [0; 360[. If the adjusted angle is either 90 or 270 degrees, we need to create an empty image with its width and height fields being swapped, if it is 180 degrees

we don't need to swap those fields; Additionally, if the angle is 0, there is no need to process the image.

To generate the rotated image, we iterate over each pixel of the original image and copy them into the corresponding pixel of the rotated image, as seen in the following code block:

```
static Mat rotate(Mat image, int angle) {
    ...
    switch (angle){
        case 90:
            for (int i = 0; i < height; i++)
                for (int j = 0; j < width; j++)
                    for (int k = 0; k < 3; k++)
                        result.at<Vec3b>(j, height - 1 - i)[k] =
                            image.at<Vec3b>(i, j)[k];
            break;
        case 180:
            for (int i = 0; i < height; i++)
                for (int j = 0; j < width; j++)
                    for (int k = 0; k < 3; k++)
                        result.at<Vec3b>(height-1-i,width-1-j)[k] =
                            image.at<Vec3b>(i, j)[k];
            break;
        case 270:
            for (int i = 0; i < height; i++)
                for (int j = 0; j < width; j++)
                    for (int k = 0; k < 3; k++)
                        result.at<Vec3b>(width - 1 - j, i)[k] =
                            image.at<Vec3b>(i, j)[k];
            break;
        default: break;
    }
    return result;
}
```

2.4.1 Usage

```
../ppm_effects inputFile outputFile rotate <angle>
```

2.4.2 Results

In the following image we can see the results from processing one of the images from our data set with angles from -270 to 270 degrees, at intervals of 90 degrees.

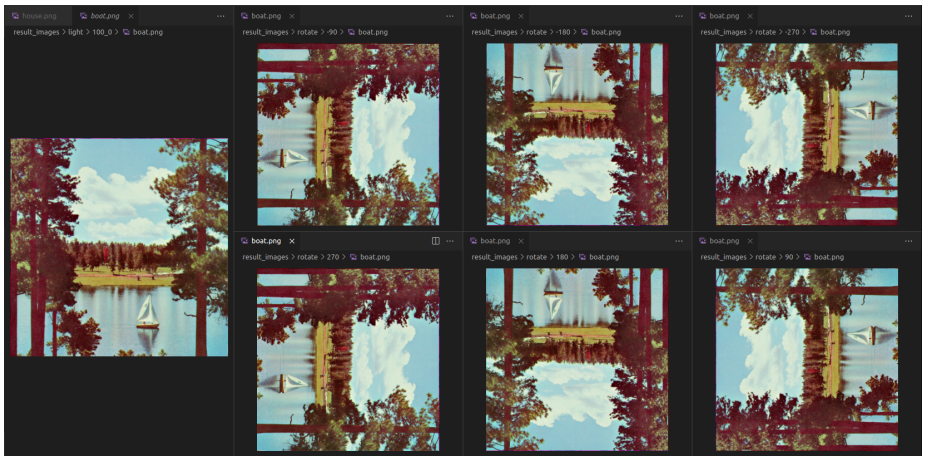

Figure 4: Results from rotating boat.ppm

2.5 Intensity

To add or remove the light from an image we came up with an expression that transforms the BGR values of a pixel in a function of logarithmic progression, this allows us to brighten and darken images without getting white or black areas on our images, keeping the ratio of the colors.

```
static Mat changeIntensity(Mat image, float intensity) {
    Mat result = Mat::zeros(image.rows, image.cols, CV_8UC3);
    for (int i = 0; i < image.rows; i++)
        for (int j = 0; j < image.cols; j++)
            for (int k = 0; k < 3; k++)
                result.at<Vec3b>(i,j)[k] =
                    adaptIntensity(image.at<Vec3b>(i,j)[k],
    intensity);
    return result;
}
```

With *adaptIntensity* having the following expression:

$$result = \begin{cases} v - (1-i) * v & : \ i \leq 0 \\ (i-1) * (255-v) + v & : \ i > 1 \end{cases}$$

Where v is the value of a BGR channel for a pixel and i being the intensity.

2.5.1 Usage

```
../ppm_effects inputFile outputFile light <[0; 2]>
```

2.5.2 Results

In the following image we can see the results from processing one of the images from our data set with light intensity levels from 0 to 200%, at intervals of 12.5%.

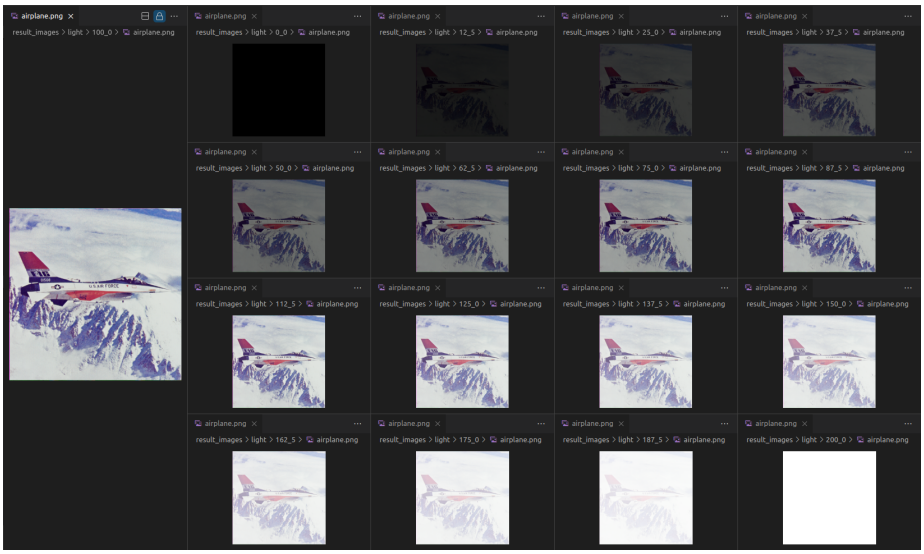

Figure 5: Result from changing intensity in an image

3 Part II - Golomb Code

3.1 Context

Golomb coding is a lossless data compression method that creates a code of variable length of each value to be encoded. Golomb code represents an integer **i** by dividing it into two parts: a quotient **q** and a remainder **r**. The quotient is encoded in unary coding, and the remainder is encoded in truncated binary coding.

3.2 Code

In our Golomb class, we leverage the previously developed BitStream class for efficient bit read and write operations. To streamline the implementation, this class abstains from directly handling the BitStream and instead relies on a pre-existing BitStream instance provided through a reference.

Additionally, the class incorporates two parameters: 'method' and 'm'. These elements play a crucial role in ensuring accurate encoding and decoding processes. The 'method' value determines how to handle the existence of negative values, while the 'm' field defines the divisor for the operation.

Golomb code can only be used to encode values such that $i \in \mathbb{N}_0$. With this limitation in mind, if we want to encode negative values we can either resort to sign and magnitude coding, by encoding the absolute value of the integer while representing its sign with a sign bit, or value intervals, by encoding the negative values as $2i - 1$ (odd values) and the positive values as $2i$ (even values).

3.2.1 Encoding

```
void encode(int i) {
    int q, r;

    if (method)
        // Encode the sign bit
        bs.write(i < 0);
    else
        // Positive values at even (2x)
        // negative values at odd (-2x-1)
        i = (i < 0) ? abs(i << 1) - 1 : (i << 1);

    // Encode the absolute value of i
    i = abs(i);
    q = floor(i / m);
    r = i % m;

    // Encode the quotient: unary code
    for (int j = 0; j < q; j++)
        bs.write(1);
    bs.write(0);

    // Encode the remainder with the appropriate number of bits
    int b = ceil(log2(m));
    int u = (1 << b) - m;
    (r < u) ? bs.write(r, b-1) : bs.write(r+u, b);
}
```

The remainder in Golomb code is coded represented in binary. Additionally, if the value of **m** is not a power of 2, we can reduce the amount of bits used if we resort to truncated binary code.

To achieve that we first define

$$b = \lceil log(m) \rceil;\ u = 2^b - m$$

The remainder values are represented by $b-1$ bits if they are lower than u. Otherwise, the sum of $r+u$ is used and represented with b bits.

3.2.2 Decoding

```
int decode() {
    int i = 0;

    // Decode the signal of the integer.
    int s = 1;
    if (method){
        s = bs.read() ? -1 : 1;
    }

    int q = 0, r = 0;

    // Decode the quotient from the unary code.
    while (bs.read()) q++;

    // Decode the remainder with the appropriate number of bits
    int b = ceil(log2(m));
    int u = (1 << b) - m;

    for (int j = 0; j < b-1; j++)
        r = (r << 1) | bs.read();
    if (r>=u)
        r = ((r << 1) | bs.read()) - u;

    // Reconstruct the integer.
    i = s * (q*m + r);

    // Even values to positive (x/2),
    // odd values to negative (-(x+1)/2)
    if (!method){
        s = i % 2;
        i = s == 1 ? -((i + 1) >> 1) : i >> 1;
    }

    return i;
}
```

To decode the remainder, we read the first $b-1$ bits following the terminator for the unary code. If the obtained value is greater or equal to u it means that the value is written in truncated binary code. If that is the case, we read one additional bit and subtract u to get the value of the remainder.

4 Part III - Audio Codec

To implement an audio codec using Golomb coding, we use predictive coding to calculate the difference between each sample and the previous sample, and then save the values to a file using Golomb coding.

4.1 Finding the ideal value of m

To minimize the average code length per symbol, we must find the ideal 'm' value. for that, we created the idealM method in our Golomb class, which was

discussed in 3.

This method starts by fitting the predicted residuals into a geometric function. This is done by calculating the average value of the residuals, a. Since the expected value of a geometric function is $1/p$, we can calculate the p value with $1/a$. We then use the following formula to calculate the ideal 'm' value:

$$m = -log(2-p)/log(1-p)$$

4.1.1 Code

```
static int idealM(vector<int> values) {
    double e = 0;
    for (auto i: values) {
        int j = (i < 0) ? abs(i << 1) - 1 : (i << 1);
        e += abs(j);
    }
    e /= values.size();
    double p = 1;
    p /= e;
    int m = ceil(-log(2-p)/log(1-p));
    return m;
}
```

4.2 Encoder

In order to encode an audio file we created audio_encoder.cpp. This code makes use of Golomb in order to compress a WAV file.

Firstly it checks if the audio file is valid and has a proper format, following that, if the user requests the program to do so, it proceeds to present its info (frames, samples per second, and channels).

The code then starts by reading the samples from the input file and storing them in the **samples** vector. Having done that, it calculates the prediction error samples, representing the difference between each sample and its preceding sample. These prediction error samples are then stored in the **pred** vector. Finally, employing the idealM method from the Golomb.cpp file, the code determines the ideal Golomb parameter for the prediction error samples.

```
size_t nChannels { static_cast<size_t>(sfhIn.channels()) };
size_t nFrames { static_cast<size_t>(sfhIn.frames()) };

vector<short> samples(nChannels * nFrames);
sfhIn.readf(samples.data(), nFrames);

vector<int> pred;
pred.push_back(samples[0]);
for (int i = 1; i < int(nChannels * nFrames); i++) {
    pred.push_back(samples[i]-samples[i-1]);
}
int m = Golomb::idealM(pred);
```

After this, the code creates a **BitStream** object called **bitstreamOutput** and opens it for writing. It then proceeds to write the following, on the new

file: the ideal Golomb parameter **m**, the number of channels **nChannels**, the number of frames **nFrames**, the sample rate, and the mode of compression (lossy or lossless). If lossless compression is used, it also writes the value of m. Otherwise, it writes the number of samples in each block (this value is provided by the user when running the program).

```
BitStream bitstreamOutput { argv[argc-1], 0 };
bitstreamOutput.write(nChannels,16);
bitstreamOutput.write(nFrames,32);
bitstreamOutput.write(sfhIn.samplerate(),16);
bitstreamOutput.write(method);
bitstreamOutput.write(lossy);
if (lossy) bitstreamOutput.write(blockSize,16);
else bitstreamOutput.write(m, 16);
```

Finally, we encode the prediction error samples and write them to the output file, depending on the mode used:

- lossless: creates a Golomb encoder object and uses it to encode all of the prediction error samples;
- lossy: for each block of predictor residuals, calculate the ideal m. Then, calculate the number of bits needed to represent that block in Golomb using the *getEncodedLength* method. If the number of bits is less than the defined value (considering the average bit rate provided by the user), then the remaining bits are added to a *reserve* variable. If the number of bits needed is more than the defined value, more bits can be used, depending on how many there are in the *reserve*. If even with *reserve* bits the size is too great, we need to calculate the predictors again for this block, but this time with one less bit for each residual. This process is repeated until an appropriate block size is achieved. We then write the m value and the number of bits removed from the file, followed by the block predictor residuals in Golomb coding.

```
if (!lossy) {
    Golomb g = Golomb(bitstreamOutput, m, method);
    for (auto p: pred) {
        g.encode(p);
    }
}
else {
    double blockBits = (double(blockSize)/(2*sfhIn.samplerate()))*
    averageBitRate;
    int reserve = 0;
    int lastValue = 0;
    for (int i = 0; i < int(nChannels * nFrames); i+=blockSize) {
        vector<int> predBlock(blockSize);
        bool valid = false;
        int bitsRemoved = 0;
        while (!valid) {
            int nBits = 0;
            for (int j = 0; (j < blockSize) && (i+j < int(nChannels
     * nFrames)); j++) {
```

```
                int p = (samples[i+j]-lastValue)>>bitsRemoved;
                predBlock[j] = p;
                lastValue = lastValue + (p<<bitsRemoved);
            }
            m = Golomb::idealM(predBlock);
            Golomb g = Golomb(bitstreamOutput, m, method);
            for (auto p: predBlock) {
                nBits += g.getEncodedLength(p);
            }
            if (nBits<=blockBits) {
                reserve += blockBits-nBits;
                valid = true;
            }
            else if (nBits<=blockBits+reserve) {
                reserve -= nBits-blockBits;
                valid = true;
            }
            else bitsRemoved++;
        }
        bitstreamOutput.write(m,16);
        bitstreamOutput.write(bitsRemoved,16);
        Golomb g = Golomb(bitstreamOutput, m, method);
        for (auto p: predBlock) g.encode(p);
    }
}
```

4.2.1 Usage

```
../audio_encoder -ab averageBitRate -bs blockSize -m method
    inputFile outputFile -v
```

where *method* is a boolean value to determine how to deal with negative values. To use lossless compression, omit the -ab and -bs parameters.

4.3 Decoder

The decoder has the objective of decoding audio files that have been encoded using Golomb and predictive coding. The code achieves this by reading the header information from the input file, decoding the prediction error samples, reconstructing the original audio samples, and writing the reconstructed audio samples to the output file. Like the encoder, this is possible using both the lossless and lossy methods.

To achieve this the code starts by opening the input file, reading the header info, and checking which method the user wishes to use:

```
BitStream bitstreamInput = BitStream(argv[argc-2], 1);
size_t nChannels = bitstreamInput.readInt(16);
size_t nFrames = bitstreamInput.readInt(32);
int samplerate = bitstreamInput.readInt(16);
int method = bitstreamInput.read();
bool lossy = bitstreamInput.read();
int blockSize = 0;
int m = 0;
```

```
if (lossy) blockSize = bitstreamInput.readInt(16);
else m = bitstreamInput.readInt(16);
```

After checking if the file is in the correct format, it decodes the prediction error samples from the input file and stores them in the **pred** vector. The way it does it depends on the mode chosen:

- lossless: creates a Golomb decoder object and uses it to decode all of the prediction error samples;
- lossy: iterates over the encoded audio data in blocks of **blockSize** samples. For each block, it reads the ideal Golomb parameter **m** and *removedBits* for that block from the input file.

```
vector<int> pred;
if (!lossy) {
    Golomb g = Golomb(bitstreamInput, m, method);
    for (int i = 0; i < int(nChannels * nFrames); i++) {
        pred.push_back(g.decode());
    }
}
else {
    for (int i = 0; i < int(nChannels * nFrames); i+=blockSize) {
        m = bitstreamInput.readInt(16);
        Golomb g = Golomb(bitstreamInput, m, method);
        int bitsRemoved = bitstreamInput.readInt(16);
        for (int j = 0; (j < blockSize) && (i+j < int(nChannels *
    nFrames)); j++) {
            pred.push_back(g.decode()<<bitsRemoved);
        }
    }
}
```

Finally, it reconstructs the original audio samples by adding the prediction error samples to the previous sample and writes it in the output file.

```
vector<short> samples(nChannels * nFrames);
samples[0] = pred[0];
for (int i = 1; i < int(nChannels * nFrames); i++) {
    samples[i] = samples[i-1] + pred[i];
}
sfhOut.writef(samples.data(), nFrames);
```

4.3.1 Usage

```
../audio_decoder inputFile outputFile
```

4.4 Results

To test the code we used different audio files and proceeded to encode and decode each one comparing the audio waves and the size of its compressed version.

4.4.1 Lossless

In the following images, you can see the results for compressing samples[0-3]. In blue there's the original of each sample followed by its compressed version, in red.

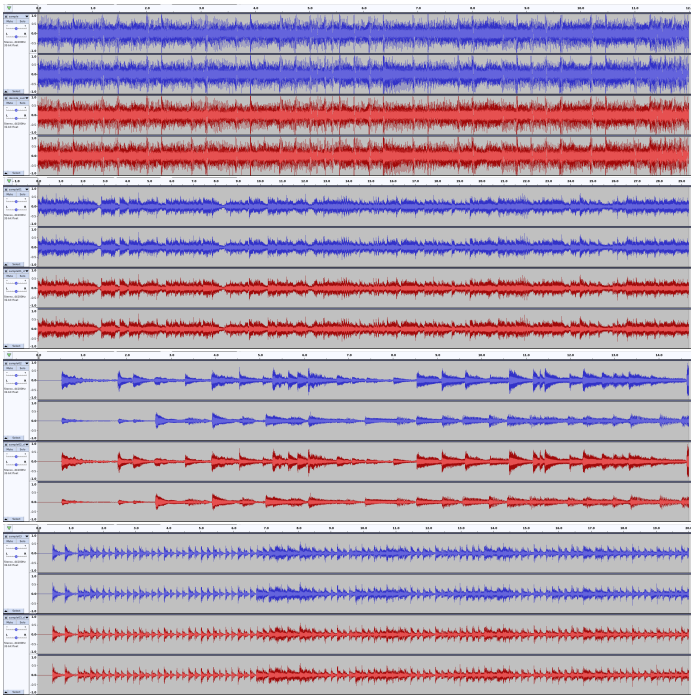

Figure 6: Results for lossless compression of WAV files

Size of Audio Files		
File	Original Size (MB)	Compressed Size (MB)
sample.wav	2.1	2.0
sample01.wav	5.2	4.2
sample02.wav	2.6	2.2
sample03.wav	3.5	2.8
sample04.wav	2.4	2.1
sample05.wav	3.6	3.1

Table 1: Obtained compression for the sample files

From these results, we can see that despite not having changes in the quality of the audio, the size of the compressed files is lower than the original.

4.4.2 Lossy

In the following images, you can see the results for compressing samples[0-3] with an average bitrate of 500KBps and a block size of 5000, represented as in the previous section.

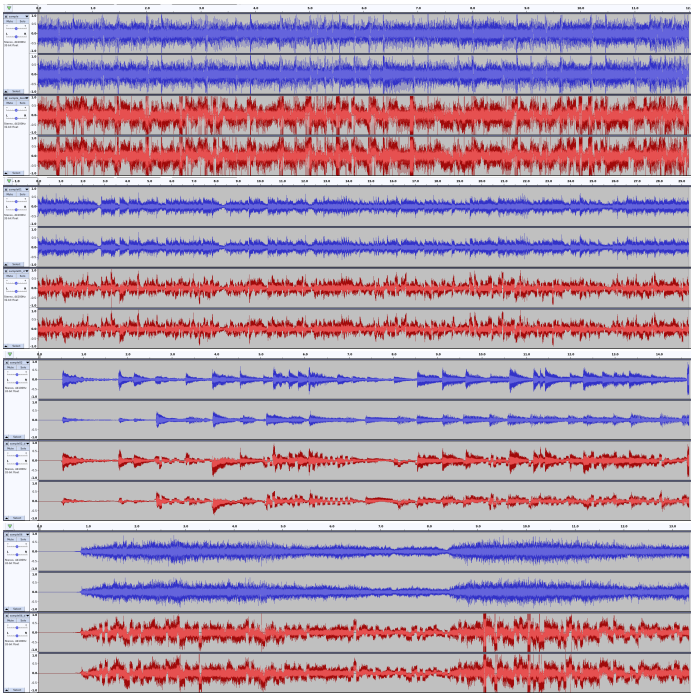

Figure 7: Results for lossy compression of WAV files

Size of Audio Files		
File	Original Size (MB)	Compressed Size (MB)
sample.wav	2.1	0.751
sample01.wav	5.2	1.8
sample02.wav	2.6	0.898
sample03.wav	3.5	1.3
sample04.wav	2.4	0.818
sample05.wav	3.6	1.3

Table 2: Obtained compression for the sample files

Unlike the results achieved in 4.4.1, where audio quality remained unchanged, our exploration into lossy compression revealed a noticeable decrease in audio quality. Despite this, we were able to decrease a greater reduction in the compressed file size.

4.4.3 Results with wav_cmp

For Lossless compression:

```
Channel 1:
L2 Norm: 0          LInf Norm: 0          SNR: inf dB
Channel 2:
L2 Norm: 0          LInf Norm: 0          SNR: inf dB
Average L2 Norm across all channels: 0
Average SNR across all channels: inf dB
```

For Lossy compression:

```
Channel 1:
L2 Norm: 0.269058     LInf Norm: 1.9946     SNR: -0.221488 dB
Channel 2:
L2 Norm: 0.267828     LInf Norm: 1.99521    SNR: -0.161076 dB
Average L2 Norm across all channels: 0.268443
Average SNR across all channels: -0.191282 d
```

5 Conclusion

With this assignment we had the opportunity to explore image processing functions, as well as data compression.

About Part I, using OpenCV has given us a better understanding of how an image is represented in data. It was also a great opportunity to explore some aspects of the C++ language.

Regarding the representation of data through varying length codes, exploring Golomb and its utility for data compression was a very entertaining challenge that has fostered our interest in the topic.

6 Group Contribution

We consider this project's development to have been shared equally amongst all members of the group.