Project IC - Second Project

Universidade de Aveiro

Bernardo Marçal, Ricardo Machado, Rui Campos

Project IC - Second Project

Informação e Codificação

Universidade de Aveiro

Bernardo Marçal, Ricardo Machado, Rui Campos
(103236) bernardo.marcal@ua.pt, (102737) ricardo.machado@ua.pt
(103709) ruigabriel2@ua.pt

03 de dezembro de 2023

Índice

Capítulo 1

Introduction

In this project, we delve into the realm of digital signal processing and data compression, focusing on image and audio transformations. Building upon the foundational concepts explored earlier, such as image manipulation and intensity adjustments, we are now integrating the use of lossless audio codecs and Golomb coding to enhance our understanding of efficient data representation and compression.

The project takes a multidisciplinary approach, combining image processing techniques with audio compression methodologies. By integrating lossless audio codecs and Golomb coding, we aim to achieve optimal compression while preserving the quality and integrity of both image and audio data.

Throughout the project, we will employ a consistent image as our canvas for various transformations, ensuring clarity and visibility in the demonstration of each technique. Additionally, the incorporation of lossless audio codecs and Golomb coding introduces a deeper layer of complexity, fostering a comprehensive exploration of data compression methods.

The ultimate goal is to gain insights into the intricacies of lossless compression, understand the principles behind Golomb coding, and witness how these techniques contribute to efficient storage and transmission of digital multimedia content. This project aims to provide a hands-on experience in the integration of image and audio processing, contributing to a holistic understanding of multimedia data manipulation and compression

The project's repository can be viewed here: Github.

All the command codes to test are available at the README file.

Capítulo 2

Project Workthrough

2.1 Exercise 1

2.1.1 Source Code

```
    int channelNumber = atoi(argv[3]);
    Mat extractedChannel;
    extractChannel(sourceImage, extractedChannel, channelNumber);

    // Convert
    Mat bgrImage;
    cvtColor(extractedChannel, bgrImage, COLOR_GRAY2BGR);

    // save the BGR image
    imwrite(argv[2], bgrImage);

    cout << "Image saved in the directory opencv-bin" << endl;

    return 0;
}

void extractChannel(const Mat& input, Mat& output, int channelNumber) {
    // create a single-channel image
    output = Mat::zeros(input.size(), CV_8UC1);

    if (input.isContinuous()) {
        // calculate the total number of elements
        int totalElements = input.rows * input.cols * input.channels();

        // extract the specified channel
        for (int i = 0; i < totalElements; i += input.channels()) {
            output.data[i / input.channels()] = input.data[i + channelNumber - 1];
        }
    } else {
        // extract the specified channel
        for (int i = 0; i < input.rows; i++) {
            for (int j = 0; j < input.cols; j++) {
                output.at<uchar>(i, j) = input.at<Vec3b>(i, j)[channelNumber - 1];
            }
        }
    }
}
```

Fig. 1: Extract Channel Code

We created the function called *extractChannel* that takes an input image and extracts a specified channel from it, storing the result in an output image. This function supports both single-channel and multi-channel images. This function begins by creating an output image with the same size as the input image, but with a single channel (where *UC1* stands for unsigned char with 1 channel). Then, the code checks if the input image is stored in a contiguous block of memory. If it is, the function uses a more efficient method to extract the channel. If the image is not continuous, the function uses a nested loop to iterate through each pixel in the input image and extract the specified channel. In this case, it assumes that the input image is a three-channel image with *Vec3b*, and the specified channel is accessed using the index "*channelNumber* - 1". The extracted pixel value is then stored in the corresponding location in the output image.

Note: If the channel is equal to 1, we will have a transformation of the image in shades of blue; if the channel is equal to 2, we will have a transformation of the image in shades of green; finally, if the channel is equal to 3, we will have a transformation of the image in shades of red.

2.1.2 Results

In conclusion, the implementation of the program to extract a color channel from an image using the OpenCV library was successful. The program takes command line arguments for the input image file name, output image file name, and the channel number to be extracted (1 for blue, 2 for green, and 3 for red). The pixel-by-pixel extraction process ensures that the resulting single-channel image accurately represents the chosen color channel.

Now, let's take a look at the three images generated by the program:

Image extracted from the blue channel:

Fig. 3: Blue Pixel Image

Image extracted from the green channel:

Fig. 4: Green Pixel Image

Image extracted from the red channel:

Fig. 5: Red Pixel Image

These images effectively demonstrate the isolation of the specified color channels from the original images, showcasing the program's ability to manipulate and extract specific color information. The successful execution of the program enhances our understanding of image processing techniques using the OpenCV library.

2.2 Exercise 2

Throughout the following demonstrations, we will be working with the same image to facilitate a clear and visible understanding of various image transformations. Using a consistent image as a reference allows us to observe and compare the effects of each transformation method applied. From creating negatives and mirrored versions to rotating images and adjusting intensity values, this approach ensures a focused exploration of fundamental image processing techniques. The goal is to provide a comprehensible and insightful journey into the world of manual image transformations, enabling a better understanding of the underlying principles without the use of existing OpenCV functions.

Fig. 6: Original Image

2.3 Exercise 2 (a)

2.3.1 Source Code

```
    Mat negativeImage;
    createNegative(sourceImage, negativeImage);

    // save the negative image
    imwrite(argv[2], negativeImage);

    cout << "Negative image saved in the directory opencv-bin" << endl;

    return 0;
}

void createNegative(const Mat& input, Mat& output) {
    output = Mat::zeros(input.size(), input.type());

    for (int i = 0; i < input.rows; i++) {
        for (int j = 0; j < input.cols; j++) {
            for (int c = 0; c < input.channels(); c++) {
                output.at<Vec3b>(i, j)[c] = 255 - input.at<Vec3b>(i, j)[c];
            }
        }
    }
}
```

Fig. 7: Negative Version Code

We created a function called *createNegative* that takes an input image and generates the negative of that image, storing the result in an output image. The negative of an image is created by subtracting each pixel value from the maximum intensity. This function begins by creating an output image of the same size and type as the input image. This ensures that the output image has the same number of channels and bit depth as the input image. Then uses nested loops to iterate through each pixel of the input image and each channel of each pixel. Finnaly, inside the innermost loop, this code calculates the negative value for each channel of the current pixel. It subtracts the original pixel value from the maximum intensity value (255 for an 8-bit image) and assigns the result to the corresponding channel of the output image.

2.3.2 Results

In conclusion, the implementation of the program to create the negative version of an image without using existing functions of OpenCV was successful. The program successfully processed the input image, generating a negative version by subtracting each pixel value from the maximum intensity (255 for an 8-bit image). This manual approach effectively inverted the color information, producing the desired negative effect.

Now, let's take a look at the negative version of the original image:

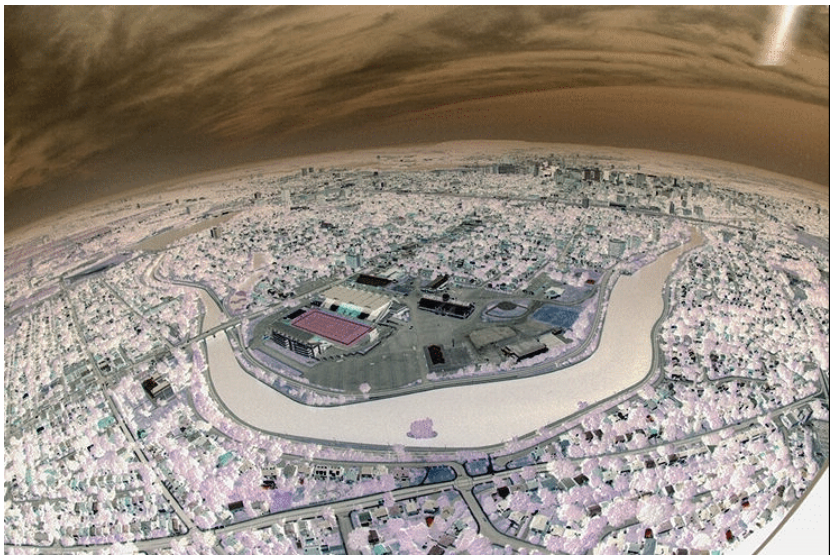

Fig. 8: Negative Version Image

The negative image showcases the successful execution of the program, where the colors are inverted, resulting in a visually distinct representation of the original photo. This implementation provides a basic yet fundamental insight into image processing techniques, illustrating the concept of negating pixel values to achieve a negative image effect.

2.4 Exercise 2 (b)

2.4.1 Source Code

```
    char orientation = argv[3][0];
    Mat mirroredImage;
    createMirrored(sourceImage, mirroredImage, orientation);

    // save the mirrored image
    imwrite(argv[2], mirroredImage);

    cout << "Mirrored image saved in the directory opencv-bin" << endl;

    return 0;
}

void createMirrored(const Mat& input, Mat& output, char orientation) {
    // create an output image with the same size and type as the input
    output = Mat::zeros(input.size(), input.type());

    if (orientation == 'h') {
        // mirror horizontally
        for (int i = 0; i < input.rows; i++) {
            for (int j = 0; j < input.cols; j++) {
                for (int c = 0; c < input.channels(); c++) {
                    output.at<Vec3b>(i, j)[c] = input.at<Vec3b>(i, input.cols - j - 1)[c];
                }
            }
        }
    } else if (orientation == 'v') {
        // mirror vertically
        for (int i = 0; i < input.rows; i++) {
            for (int j = 0; j < input.cols; j++) {
                for (int c = 0; c < input.channels(); c++) {
                    output.at<Vec3b>(i, j)[c] = input.at<Vec3b>(input.rows - i - 1, j)[c];
                }
            }
        }
    } else {
        cout << "Invalid orientation. Use 'h' for horizontal or 'v' for vertical." << endl;
    }
}
```

Fig. 9: Mirrored Version Code

We created a function called *createMirrored* that takes an input image, a character representing the desired orientation ('h' for horizontal or 'v' for vertical), and generates a mirrored version of the input image accordingly. The mirrored image is then stored in an output image. This function starts by creating an output image of the same size and type as the input image. The code checks the specified orientation ('h' for horizontal or 'v' for vertical) and performs the mirroring accordingly. If the orientation is 'h', it iterates through each pixel and channel, copying the corresponding pixel from the original image but

flipped horizontally. If the orientation is 'v', it does the same but flips vertically. If the orientation is neither 'h' nor 'v', it prints an error message indicating that the orientation is invalid.

Note: If the specified orientation are something different of 'h' or 'v', a message saying "Invalid orientation. Use 'h' for horizontal or 'v' for vertical."is displayed.

2.4.2 Results

In conclusion, the program successfully implemented the creation of mirrored versions of an image, both horizontally and vertically, without utilizing existing functions of OpenCV. The manual approach involved iterating through each pixel and channel, and either flipping horizontally or vertically, resulting in distinct mirrored effects.

Now, let's take a look at the two mirrored versions of the original image:

(a) Mirrored Horizontally:

Fig. 10: Mirrored Horizontally Version Image

(b) Mirrored Vertically:

Fig. 11: Mirrored Vertically Version Image

The program effectively produced these mirrored images, demonstrating the ability to manipulate pixel positions to achieve mirror effects. The horizontally mirrored version reflects each column across the vertical axis, while the vertically mirrored version reflects each row across the horizontal axis. This manual mirror operation showcases the foundational concepts of image transformation and enhances our understanding of image processing techniques.

2.5 Exercise 2 (c)

2.5.1 Source Code

```
    int rotationDegrees = atoi(argv[3]);
    Mat rotatedImage;
    rotateImage(sourceImage, rotatedImage, rotationDegrees);

    // save the rotated image
    imwrite(argv[2], rotatedImage);

    cout << "Rotated image saved in the directory opencv-bin" << endl;

    return 0;
}

void rotateImage(const Mat& input, Mat& output, int degrees) {
    // validate degrees (ensure it's a multiple of 90)
    if (degrees % 90 != 0) {
        cout << "Invalid rotation degrees. Please use a multiple of 90." << endl;
        return;
    }

    // calculate the number of rotations (0, 1, 2, or 3)
    int numRotations = (degrees / 90) % 4;

    // create an output image with the same size and type as the input
    output = Mat::zeros(input.size(), input.type());

    Mat rotatedInput = input.clone();  // make a copy of the input matrix

    for (int r = 0; r < numRotations; ++r) {
        // rotate 90 degrees clockwise
        transpose(rotatedInput, output);
        flip(output, output, 1);
        rotatedInput = output.clone();  // clone the rotated image to rotatedInput for the next iteration
    }
}
```

Fig. 12: Rotate Image Code

We created a function called *rotateImage* that takes an input image, a reference to an output image, and an angle in degrees. The function rotates the input image by the specified angle and stores the result in the output image. This code first checks whether the specified rotation angle is a multiple of 90 degrees. If it's not, the function prints an error message saying "Invalid rotation degrees. Please use a multiple of 90."and returns, as non-multiple-of-90-degree rotations can result in complex transformations that are not handled by the current implementation. Then calculates the number of 90-degree rotations needed based on the specified degrees. It then creates an output image with the same size and type as the input image and makes a copy of the input image using the clone function. Finnaly, this code enters a loop that performs the

specified number of 90-degree clockwise rotations. Inside the loop, it uses the OpenCV functions *transpose* and *flip* to rotate the image 90 degrees clockwise. The *transpose* function swaps the rows and columns of the input matrix, and the *flip* function performs a horizontal flip. The result is stored in the output image. After each rotation, the rotated image is cloned back into *rotatedInput* for the next iteration. This process repeats until the desired number of rotations is achieved.

2.5.2 Results

In conclusion, the program successfully implemented the rotation of an image by a multiple of 90 degrees without relying on existing OpenCV functions. The manual rotation process involved transposing the image matrix and performing a horizontal flip, iteratively applied for rotations of 90, 180, and 270 degrees. This approach effectively altered the spatial orientation of the image, producing visually distinct results.

Now, let's take a look at the rotated versions of the original image:

(a) Rotated by 90 degrees:

Fig. 13: Rotate 90 degrees Image Version

(b) Rotated by 180 degrees:

Fig. 14: Rotate 180 degrees Image Version

(c) Rotated by 270 degrees:

Fig. 15: Rotate 270 degrees Image Version

The program successfully demonstrated the rotation of the image at different angles, creating unique visual effects. These rotated images highlight the manual manipulation of pixel positions to achieve the desired rotation, providing a fundamental understanding of image transformation techniques. This implementation contributes to the exploration of basic image processing concepts outside the scope of existing OpenCV functions.

2.6 Exercise 2 (d)

2.6.1 Source Code

```
    double intensityFactor = atof(argv[3]);
    Mat adjustedImage;
    adjustIntensity(sourceImage, adjustedImage, intensityFactor);

    // save the adjusted image
    imwrite(argv[2], adjustedImage);

    cout << "Adjusted intensity image saved in the directory opencv-bin" << endl;

    return 0;
}

void adjustIntensity(const Mat& input, Mat& output, double factor) {
    // validate factor (ensure it's within a valid range)
    if (factor < -255.0 || factor > 255.0) {
        cout << "Invalid intensity factor. Please use a factor between -255.0 and 255.0." << endl;
        return;
    }

    // calculate the alpha and beta values for addWeighted
    double alpha, beta;

    if (factor >= 0.0) {
        alpha = 1.0 + factor / 255.0;
        beta = 0.0;
    } else {
        alpha = 1.0 + factor / 255.0;
        beta = 0.0;
    }

    // apply the intensity adjustment using addWeighted
    addWeighted(input, alpha, Mat::zeros(input.size(), input.type()), 0.0, beta, output);
}
```

Fig. 16: Increase/Decrease Intensity Code

We created a function called *adjustIntensity* that takes an input image, a reference to an output image, and a scaling factor (factor). The function adjusts the intensity of each pixel in the input image using the OpenCV function *addWeighted*. The function includes a validation step to ensure that the intensity factor is within a valid range (between -255.0 and 255.0). This function begins by checking whether the specified intensity factor is within a valid range (between -255.0 and 255.0). If the factor is outside this range, the function prints an error message saying "Invalid intensity factor. Please use a factor between -255.0 and 255.0."and returns without processing the image. The code calculates the *alpha* and *beta* values needed for the *addWeighted* function. If the factor is non-negative, the *alpha* is set to 1.0 + factor / 255.0, and *beta* is set to 0.0. If the factor is negative, the values are set the same way. This calculation determines how much the input image and an empty matrix are weighted

before being added to the output image. Finally, the code applies the intensity adjustment using the OpenCV function *addWeighted*. This function combines the input image with a scaled version of an empty matrix, and the result is stored in the output image. The *alpha* and *beta* values control the contribution of the input image and the empty matrix, respectively.

2.6.2 Results

In conclusion, the program successfully implemented intensity adjustment in images, allowing for both an increase (more light) and a decrease (less light) in intensity values without relying on existing OpenCV functions. The manual adjustment process involved iterating through each pixel and channel, applying a scaling factor to modulate the intensity values, resulting in visually perceptible changes in brightness.

Now, let's examine the results of the intensity adjustment:

(a) Increased intensity (Factor: 60):

Fig. 17: Increase Intensity Imagem (Factor = 60)

(b) Increased intensity (Factor: 200):

Fig. 18: Increase Intensity Imagem (Factor = 200)

(c) Decreased intensity (Factor: -60):

Fig. 19: Decrease Intensity Imagem (Factor = -60)

(d) Decreased intensity (Factor: -200):

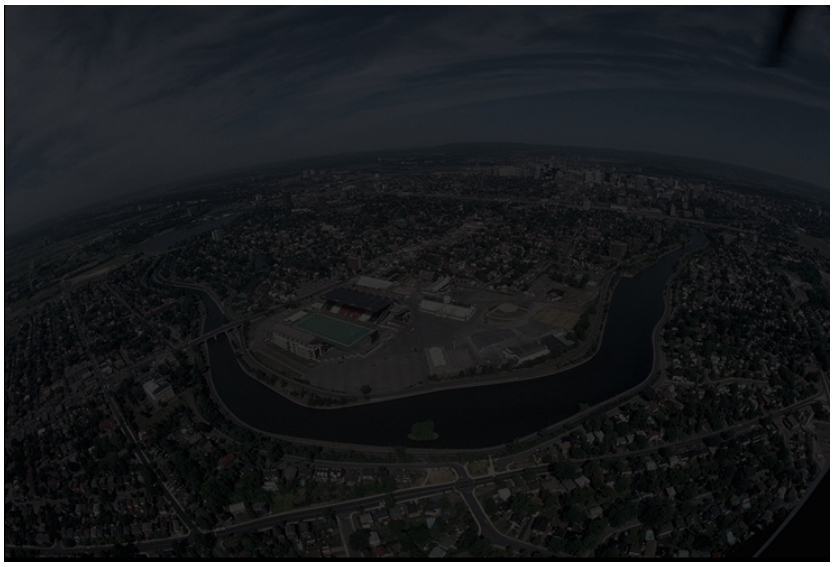

Fig. 20: Decrease Intensity Imagem (Factor = -200)

The program successfully demonstrated the flexibility to adjust the brightness levels of images, showcasing the impact of different scaling factors on the overall intensity. The images with increased intensity appear brighter, while those with decreased intensity exhibit a darker appearance. This manual adjustment provides a hands-on experience with the basic principles of image processing, offering insights into the control of image brightness.

2.7 Exercise 3

2.7.1 Source Code

```
#include <fstream>
#include <vector>
#include <string>
#include <iostream>
#include <bitset>
#include <cmath>

class Golomb {
    private:
        int minimo;
        int maximo;
        int valores;

        void bit_calc(int m) {
            if (m != 0){
                maximo = ceil(log2(m));
                minimo = maximo - 1;
                valores = pow(2, maximo) - m;
            } else {
                maximo = 0;
                minimo = 0;
                valores = 0;
            }
        }

        std::string string_resto(int n, int rep){
            std::string result = "";
            for (int i = 0; i < rep; i++) {
                result = std::to_string(n % 2) + result;
                n /= 2;
            }

            return result;
        }

        int string_to_int(std::string bit_string) {
            int result = 0;
            for (long unsigned int i = 0; i < bit_string.length();
                i++)
                result = result * 2 + (bit_string[i] - '0');

            return result;
        }

    public:
        Golomb() { }

        std::vector<int> decode(std::string string_enc, int m) {
            bit_calc(m);
            std::vector<int> result;
            int i = 0;

            while((long unsigned int) i < string_enc.length()) {
```

```
                int quociente = 0;
                while (string_enc[i] == '0') {
                    quociente++;
                    i++;
                }
                i++;
                int resto = 0;
                int j = 0;
                std::string tmp = "";
                if (m != 1){
                    while (j < minimo) {
                        tmp += string_enc[i];
                        i++;
                        j++;
                    }

                    int aux_resto = string_to_int(tmp);
                    if (aux_resto < valores) {
                        resto = aux_resto;
                    } else {
                        tmp += string_enc[i];
                        i++;
                        resto = string_to_int(tmp) - valores;
                    }
                } else {
                    resto = 0;
                    i++;
                }

                int res = quociente * m + resto;
                if (string_enc[i] == '1') {
                    result.push_back(-(res));
                } else {
                    result.push_back(res);
                }

                i++;
            }

            return result;
        }

        std::vector<int> decode_m(std::string string_enc, std::
            vector<int> m_vector, int block_size) {
            std::vector<int> result;
            int i = 0;
            int m_i = 0;
            int count = 0;
            bit_calc(m_vector[m_i]);
            while((long unsigned int) i < string_enc.length()) {
                    int quociente = 0;
                    while (string_enc[i] == '0') {
                        quociente++;
                        i++;
                    }
                    i++;
                    int resto = 0;
```

```
                    int j = 0;
                    std::string tmp = "";
                    if (m_vector[m_i] != 1){
                        while (j < minimo) {
                            tmp += string_enc[i];
                            i++;
                            j++;
                        }

                        int aux_resto = string_to_int(tmp);
                        if (aux_resto < valores) {
                            resto = aux_resto;
                        } else {
                            tmp += string_enc[i];
                            i++;
                            resto = string_to_int(tmp) - valores;
                        }
                    } else {
                        resto = 0;
                        i++;
                    }
                    int res = quociente * m_vector[m_i] + resto;
                    if (string_enc[i] == '1') {
                        result.push_back(-(res));
                    } else {
                        result.push_back(res);
                    }
                    i++;
                    count++;

                    if (count == block_size) {
                        m_i++;
                        count = 0;
                        bit_calc(m_vector[m_i]);
                    }
                }
            return result;
        }

        std::string encode(int n, int m){
            bit_calc(m);
            std::string result = "";
            int quociente = 0;
            int resto = 0;
            if (m != 0){
                quociente = abs(n) / m;
                resto = abs(n) % m;
            }
            for (int i = 0; i < quociente; i++) {
                result += "0";
            }
            result += "1";
            if (m != 1){
                if (resto < valores) {
                    result += string_resto(resto, minimo);
                } else {
                    result += string_resto(resto + valores, maximo)
```

```
                        ;
166                 }
167             }else{
168                 result += "0";
169             }
170             n < 0 ? result += "1" : result += "0";
171
172
173             return result;
174         }
175 };
176 }
```

The *Golomb* class encapsulates Golomb coding functionality. The *decode* and *decode_m* methods implement the decoding process, converting *Golomb*-encoded binary strings into vectors of integers. The *encode* method performs *Golomb* encoding on an integer, producing a *Golomb*-encoded binary string.

The *Golomb* parameters (minimum, maximum, and values) are calculated using the *bit_calc* method based on the specified parameter *m*. The private methods *string_resto* and *string_to_int* assist in binary string manipulation and conversion.

This class provides a modular and reusable implementation of *Golomb* encoding and decoding.

2.7.2 *Golomb* class

Golomb coding stands out as a data compression technique employing various compression codes. It particularly excels when dealing with alphabets that follow a geometric distribution, as *Golomb* coding serves as an optimal prefix code in such scenarios. This makes *Golomb* coding particularly well-suited for situations where smaller values in the input stream are significantly more probable than larger values.

To represent numbers using *Golomb* coding, the following procedures are employed: Firstly, a division of the desired number, N, by M takes place, where M is the adaptable divisor (or a fixed value, depending on the implementation). This division yields both a quotient and a remainder.

The quotient, denoted as 'q,' is then represented as a string of bits with a length of q, all set as 0, in unary coding. This is followed by a bit set to 1, serving as a separator between the quotient and the remainder.

As for the remainder, r, it is represented in truncated binary coding, utilizing b bits where $b = \lfloor \log_2(m) \rfloor$. The coding process for r follows two distinct cases:

- If $r < 2^{b+1} - M$, 'r' is encoded in binary representation using b bits.
- If $r \geq 2^{b+1} - M$, the number $(r + 2^{b+1} - M)$ is encoded in binary representation using $(b + 1)$ bits.

These procedures encapsulate the essence of *Golomb* coding, providing an efficient means to represent numbers with a focus on optimizing compression for datasets with a geometric distribution of values.

We have some functions that we considered key to our class:

1. ***bit_ calc(m)***: calculates the minimum and maximum bit lengths required for *Golomb* coding based on the provided 'm' value and computes the number of values represented by the minimum bit length.

2. ***string_ resto(num, nbitsrepresentation)***: converts an integer to a string of bits with a specified number of bits for representation.

3. ***string_ to_ int(bitstring)***: Converts a binary string to an integer

4. ***decode(encodedstring, m)***: Decodes an encoded string using Golomb coding with a fixed 'm' value and utilizes unary and binary representations for quotient and remainder.

5. ***decode_ m(encodedstring, mvector, blocksize)***: Decodes an encoded string with dynamic 'm' values based on a vector and block size and supports flexibility in adapting 'm' values during decoding.

6. ***encode(num, m)***: Encodes an integer using Golomb coding with a specified 'm' value, employs unary and binary representations for quotient and remainder and handles the sign of the integer based on the chosen approach.

2.8 Exercises 4 and 5

Important note: Please be advised that the file 'bitStream.h' utilized in this implementation has been extracted from a prior project for the purpose of facilitating certain functionalities in the current context. This header file, responsible for handling bitstream operations, was originally employed in another project and has been integrated into the present work to enhance specific aspects of the coding implementation. It is important to acknowledge the origin of this file for transparency and attribution.

2.8.1 Source Code

Decoder

```
#include "golomb.h"
#include <iostream>
#include <string>
#include <sndfile.hh>
#include "bitStream.h"

using namespace std;

constexpr size_t FRAMES_BUFFER_SIZE = 65536;

int main(int argc, char **argv) {

    auto predict = [](int a, int b, int c) {
        return 3 * a - 3 * b + c;
    };

    if (argc < 3) {
        cerr << "Usage: " << argv[0] << " <input file> <output file
            >\n";
        return 1;
    }

    int rate = 44100;
    BitStream bs(argv[1], "r");
    vector<int> v_channels = bs.readBits(16);
    vector<int> v_padding = bs.readBits(16);
    vector<int> v_q = bs.readBits(16);
    vector<int> v_frames = bs.readBits(32);
    vector<int> v_blockSize = bs.readBits(16);
    vector<int> v_num_zeros = bs.readBits(16);
    vector<int> v_m_size = bs.readBits(16);

    int nChannels = 0;
    for (long unsigned int i = 0; i < v_channels.size(); i++)
        nChannels += v_channels[i] * pow(2, v_channels.size() - i -
             1);

    int padding = 0;
    for (long unsigned int i = 0; i < v_padding.size(); i++)
        padding += v_padding[i] * pow(2, v_padding.size() - i - 1);

```

```
    int q = 0;
    for (long unsigned int i = 0; i < v_q.size(); i++)
        q += v_q[i] * pow(2, v_q.size() - i - 1);

    int frames = 0;
    for (long unsigned int i = 0; i < v_frames.size(); i++)
        frames += v_frames[i] * pow(2, v_frames.size() - i - 1);

    SndfileHandle sfhOut{argv[argc - 1], SFM_WRITE, SF_FORMAT_WAV |
         SF_FORMAT_PCM_16, nChannels, rate};

    int m_size = 0;
    for (long unsigned int i = 0; i < v_m_size.size(); i++)
    {
        m_size += v_m_size[i] * pow(2, v_m_size.size() - i - 1);
    }

    int blockSize = 0;
    for (long unsigned int i = 0; i < v_blockSize.size(); i++)
    {
        blockSize += v_blockSize[i] * pow(2, v_blockSize.size() - i
             - 1);
    }

    int num_zeros = 0;
    for (long unsigned int i = 0; i < v_num_zeros.size(); i++)
    {
        num_zeros += v_num_zeros[i] * pow(2, v_num_zeros.size() - i
             - 1);
    }

    vector<int> m_vector;
    for (int i = 0; i < m_size; i++)
    {
        vector<int> v_m_i = bs.readBits(16);
        int m_i = 0;
        for (long unsigned int j = 0; j < v_m_i.size(); j++)
        {
            m_i += v_m_i[j] * pow(2, v_m_i.size() - j - 1);
        }
        m_vector.push_back(m_i);
    }

    int total = bs.getFileSize() - (16 + 2 * m_size);
    long totalBits = total * 8;
    vector<int> v_encoded = bs.readBits(totalBits);
    string encoded = "";
    for (long unsigned int i = 0; i < v_encoded.size(); i++)
    {
        encoded += to_string(v_encoded[i]);
    }

    encoded = encoded.substr(0, encoded.size() - num_zeros);

    Golomb g;
    vector<int> decoded;
    if (m_size == 1)
```

```
94          decoded = g.decode(encoded, m_vector[0]);
95      else
96          decoded = g.decode_m(encoded, m_vector, blockSize);
97
98      vector<short> samplesVector;
99
100     if (nChannels < 2)
101     {
102         for (long unsigned int i = 0; i < decoded.size(); i++)
103         {
104             if (i >= 3)
105             {
106                 int difference = decoded[i] + predict(samplesVector
                        [i - 1], samplesVector[i - 2], samplesVector[i
                        - 3]);
107                 samplesVector.push_back(difference);
108             }
109             else
110                 samplesVector.push_back(decoded[i]);
111         }
112     }
113     else
114     {
115         for (int i = 0; i < frames; i++)
116         {
117             if (i >= 3)
118             {
119                 int difference = decoded[i] + predict(samplesVector
                        [i - 1], samplesVector[i - 2], samplesVector[i
                        - 3]);
120                 samplesVector.push_back(difference);
121             }
122             else
123                 samplesVector.push_back(decoded[i]);
124         }
125
126         for (long unsigned int i = frames; i < decoded.size(); i++)
127         {
128             if ((int)i >= frames + 3)
129             {
130                 int difference = decoded[i] + predict(samplesVector
                        [i - 1], samplesVector[i - 2], samplesVector[i
                        - 3]);
131                 samplesVector.push_back(difference);
132             }
133             else
134                 samplesVector.push_back(decoded[i]);
135         }
136         vector<short> merged;
137         vector<short> firstChannel(samplesVector.begin(),
                samplesVector.begin() + frames);
138         vector<short> secondChannel(samplesVector.begin() + frames,
                 samplesVector.end());
139         for (int i = 0; i < frames; i++)
140         {
141             merged.push_back(firstChannel[i]);
142             merged.push_back(secondChannel[i]);
```

```
143         }
144         samplesVector = merged;
145     }
146
147     if (q != 0)
148     {
149         if (q != 1)
150         {
151             for (long unsigned int i = 0; i < samplesVector.size();
                     i++)
152             {
153                 samplesVector[i] = samplesVector[i] << 1;
154                 samplesVector[i] = samplesVector[i] | 1;
155                 samplesVector[i] = samplesVector[i] << (q - 1);
156             }
157         }
158         else
159         {
160             for (long unsigned int i = 0; i < samplesVector.size();
                     i++)
161             {
162                 samplesVector[i] = samplesVector[i] << 1;
163             }
164         }
165     }
166     samplesVector = vector<short>(samplesVector.begin(),
            samplesVector.end() - padding);
167     sfhOut.write(samplesVector.data(), samplesVector.size());
168     bs.close();
169
170     cout << "File has been stored in the 'opencv-bin' directory."
            << endl;
171
172     return 0;
173 }
174 }
```

This code is part of a program that processes audio data, specifically performing *Golomb* decoding and signal prediction.

The ‘main‘ function begins by checking if there are enough command-line arguments provided. It expects at least three arguments: the input file, the output file, and optional parameters. Several parameters are then read from a binary input file using the *BitStream* class, including channel information, padding, the quantization parameter (q), the number of frames, block size, the number of zero bits, and *Golomb* parameters. Values are calculated from the read parameters, such as the number of channels (*nChannels*), padding, quantization (q), frames, block size, and others. An output audio file is created using the *SndfileHandle* class with specified parameters. *Golomb* parameters and encoded audio data are read from the input file, the *Golomb*-encoded data is decoded, and a prediction algorithm is applied. The decoded data is processed based on the number of channels, and quantization is applied. Sample values are adjusted based on the quantization parameter (q). Finally, the processed audio data is then written to the output file.

In summary, the program is designed to decode *Golomb*-encoded audio data, apply a prediction algorithm, and perform other signal processing operations before storing the result in a new audio file. The specific functionality relies on *Golomb* decoding and the parameters read from the input file.

Encoder

```
#include "golomb.h"
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <cstring>
#include <fftw3.h>
#include <sndfile.hh>
#include "bitStream.h"

using namespace std;

constexpr size_t FRAMES_BUFFER_SIZE = 65536;

auto previsao = [](int a, int b, int c) {
    return 3 * a - 3 * b + c;
};

auto calc_m = [](int u) {
    return static_cast<int>(-1 / log(static_cast<double>(u) / (1 +
        u)));
};

int main(int argc, char *argv[]) {

    if (argc < 4 || argc > 6) {
        cerr << "Usage: " << argv[0] << " <input file> <output file
            > <m | bs> [auto] [q] \n";
        return 1;
    }

    int q = 0;
    bool modo = false;
    bool quant = false;

    if (argc >= 5) {
        if (strcmp(argv[4], "auto") == 0) {
            modo = true;
        } else {
            q = atoi(argv[4]);
            quant = true;
        }
    }

    if (argc == 6) {
        q = atoi(argv[5]);
        quant = true;
    }

    if (q > 16 || q < 0) {
        cerr << "[q] must be between 1 and 15\n";
```

```
        return 1;
    }

    SndfileHandle sfhIn{argv[1]};
    if (sfhIn.error()) {
        cerr << "Error: invalid input file\n";
        return 1;
    }

    if ((sfhIn.format() & SF_FORMAT_TYPEMASK) != SF_FORMAT_WAV) {
        cerr << "Error: file is not in WAV format\n";
        return 1;
    }

    if ((sfhIn.format() & SF_FORMAT_SUBMASK) != SF_FORMAT_PCM_16) {
        cerr << "Error: file is not in PCM_16 format\n";
        return 1;
    }

    string output = argv[2];
    short m = atoi(argv[3]);

    short bs = m;
    short og = m;

    size_t nFrames{static_cast<size_t>(sfhIn.frames())};
    size_t nChannels{static_cast<size_t>(sfhIn.channels())};
    vector<short> samples(nChannels * nFrames);
    sfhIn.readf(samples.data(), nFrames);
    size_t nBlocks{static_cast<size_t>(ceil(static_cast<double>(
        nFrames) / bs))};

    samples.resize(nBlocks * bs * nChannels);

    int padding = samples.size() - nFrames * nChannels;

    vector<short> left_samples(samples.size() / 2);
    vector<short> right_samples(samples.size() / 2);

    if (quant) {
        for (long unsigned int i = 0; i < samples.size(); i++) {
            samples[i] = samples[i] >> q;
        }
    }

    if (nChannels > 1) {
        for (long unsigned int i = 0; i < samples.size() / 2; i++)
            {
            left_samples[i] = samples[i * nChannels];
            right_samples[i] = samples[i * nChannels + 1];
        }
    }

    vector<int> m_vector;
    vector<int> valores;

    if (nChannels < 2) {
```

```
105         for (long unsigned int i = 0; i < samples.size(); i++) {
106             if (i >= 3) {
107                 int dif = samples[i] - previsao(samples[i - 1],
                        samples[i - 2], samples[i - 3]);
108                 valores.push_back(dif);
109             } else {
110                 valores.push_back(samples[i]);
111             }
112         }
113     } else {
114         for (long unsigned int i = 0; i < left_samples.size(); i++)
                {
115             if (i >= 3) {
116                 int dif = left_samples[i] - previsao(left_samples[i
                        - 1], left_samples[i - 2], left_samples[i -
                        3]);
117                 valores.push_back(dif);
118             } else {
119                 valores.push_back(left_samples[i]);
120             }
121             if (i % bs == 0 && i != 0) {
122                 int sum = 0;
123                 for (long unsigned int j = i - bs; j < i; j++) {
124                     sum += abs(valores[j]);
125                 }
126                 int u = round(sum / bs);
127                 m = calc_m(u);
128                 if (m < 1) m = 1;
129                 m_vector.push_back(m);
130             }
131             if (i == left_samples.size() - 1) {
132                 int sum = 0;
133                 for (long unsigned int j = i - (i % bs); j < i; j
                        ++) {
134                     sum += abs(valores[j]);
135                 }
136                 int u = round(sum / (i % bs));
137                 m = calc_m(u);
138                 if (m < 1) m = 1;
139                 m_vector.push_back(m);
140             }
141         }
142 
143         for (long unsigned int i = 0; i < left_samples.size(); i++)
                {
144             if (i >= 3) {
145                 int dif = right_samples[i] - previsao(right_samples
                        [i - 1], right_samples[i - 2], right_samples[i
                        - 3]);
146                 valores.push_back(dif);
147             } else {
148                 valores.push_back(right_samples[i]);
149             }
150             if (i % bs == 0 && i != 0) {
151                 int sum = 0;
152                 for (long unsigned int j = i - bs; j < i; j++) {
153                     sum += abs(valores[j]);
```

```
                }
                int u = round(sum / bs);
                m = calc_m(u);
                if (m < 1) m = 1;
                m_vector.push_back(m);
            }
            if (i == left_samples.size() - 1) {
                int sum = 0;
                for (long unsigned int j = i - (i % bs); j < i; j
                    ++) {
                    sum += abs(valores[j]);
                }
                int u = round(sum / (i % bs));
                m = calc_m(u);
                if (m < 1) m = 1;
                m_vector.push_back(m);
            }
        }
    }

    string string_enc = "";
    Golomb g;

    if (!modo) {
        for (long unsigned int i = 0; i < valores.size(); i++)
            string_enc += g.encode(valores[i], og);
    } else {
        int m_index = 0;
        for (long unsigned int i = 0; i < valores.size(); i++) {
            if (i % bs == 0 && i != 0) m_index++;
            string_enc += g.encode(valores[i], m_vector[m_index]);
        }
    }

    BitStream bitStream(output, "w");
    vector<int> bits;
    vector<int> encoded_bits;
    for (long unsigned int i = 0; i < string_enc.length(); i++)
        encoded_bits.push_back(string_enc[i] - '0');

    int count_zeros = 0;
    while (encoded_bits.size() % 8 != 0) {
        encoded_bits.push_back(0);
        count_zeros++;
    }

    for (int i = 15; i >= 0; i--) {
        bits.push_back((sfhIn.channels() >> i) & 1);
    }

    for (int i = 15; i >= 0; i--) {
        bits.push_back((padding >> i) & 1);
    }

    for (int i = 15; i >= 0; i--) {
        bits.push_back((q >> i) & 1);
    }
```

```
210
211     for (int i = 31; i >= 0; i--) {
212         bits.push_back((samples.size() / 2 >> i) & 1);
213     }
214
215     for (int i = 15; i >= 0; i--) {
216         bits.push_back((bs >> i) & 1);
217     }
218
219     for (int i = 15; i >= 0; i--) {
220         bits.push_back((count_zeros >> i) & 1);
221     }
222
223     if (!modo) {
224         m_vector.clear();
225         m_vector.push_back(og);
226     }
227     for (int i = 15; i >= 0; i--) {
228         bits.push_back((m_vector.size() >> i) & 1);
229     }
230
231     for (long unsigned int i = 0; i < m_vector.size(); i++) {
232         for (int j = 15; j >= 0; j--) {
233             bits.push_back((m_vector[i] >> j) & 1);
234         }
235     }
236
237     for (long unsigned int i = 0; i < encoded_bits.size(); i++)
238         bits.push_back(encoded_bits[i]);
239
240     bitStream.writeBits(bits);
241     bitStream.close();
242
243     cout << "File has been stored in the 'opencv-bin' directory."
         << endl;
244
245     return 0;
246 }
```

This code is a program for compressing and encoding audio data using *Golomb* coding.

The main starts by checking the number of command-line arguments and providing usage instructions if they are incorrect. It parses command-line arguments to determine the input and output file names, the value of m, the optional "auto"mode, and the quantization parameter q. It checks the validity of the input file format, ensuring it is a WAV file with PCM16 format. Reads audio samples from the input file, applies quantization if specified, and separates samples into left and right channels if the audio is stereo. Computes *Golomb* parameters based on the input samples and performs *Golomb* encoding. Writes *Golomb*-encoded data to a binary file using the *bitStream* class. Finally, outputs *Golomb* parameters and the *Golomb*-encoded data to the output file.

The program performs *Golomb* encoding on audio data, with the option to use an adaptive *Golomb* parameter (m) based on the observed sample values. The encoded data is then written to an output file. The program also handles

various command-line options for configuring the encoding process.

2.8.2 Results

The Golomb-based lossless audio codec successfully encodes mono and stereo audio files. The encoder reads input audio files, predicts the next values using temporal and inter-channel predictions, and encodes the prediction residuals using *Golomb* coding. The encoded data is written to an output file in the WAV format.

The decoder reads the encoded *Golomb* data, performs *Golomb* decoding based on the provided parameters, and reconstructs the prediction residuals. The predicted values are then added to the prediction residuals to reconstruct the original audio samples. Decoded audio data is written to an output file in the WAV format.

The code has various parameters such as the number of channels, padding, quantization factor q, the number of frames, block size, and Golomb coding parameters m. The Golomb parameters are adaptively determined during encoding and transmitted in the header for decoding.

In relation to efficiency levels, the *Golomb* code effectively compresses audio data by encoding prediction residuals, leveraging *Golomb* coding's efficiency for representing non-negative integers.

In terms of flexibility, the code is versatile, supporting both mono and stereo audio files. It incorporates temporal prediction for mono audio and extends to inter-channel prediction for stereo audio, providing adaptability to different audio scenarios.

Finally, the adaptive determination of *Golomb* parameters during encoding allows for optimal encoding efficiency based on the characteristics of the input audio. The encoded parameters facilitate accurate decoding without requiring explicit transmission.

2.8.3 Conclusions

Examining the lossless compression ratios concerning the parameter 'M,' we observe a recurring pattern mirroring our previous observations. Initially, it's notable that the size of the input files, such as sample06.wav, does not directly correlate with the achieved compression ratio. Despite sample06.wav being larger than sample04.wav and smaller than sample01.wav, the optimal compression rate is achieved with sample06.wav.

Moreover, we reiterate that there is no universally optimal 'M' value. For both sample06.wav and sample04.wav, the most effective 'M' value is 128. However, intriguingly, for sample01.wav, the optimal 'M' value deviates and is determined to be 512.

This variability in the optimal 'M' values underscores the influence of specific characteristics within each audio sample on the compression process. As such, the effectiveness of Golomb coding, dependent on the chosen 'M' value,

appears to be intricately tied to the inherent properties of the audio data being compressed.

Capítulo 3

Repository and Contributions

If there is any doubts about any part of the code that was put into this report, here is the github repository that has every part of the code that we have worked on and every file:

https://github.com/ruicampos2/IC_Project2

All the compilation methods are inserted in the Readme in the main page of the repository. The contributions of the group work to this project are the following:

Bernardo Marçal (103236) - 33%
Ricardo Machado (102737) - 33%
Rui Campos (103709) - 33%