UNIVERSIDADE DE AVEIRO

DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

42078 - Informação e Codificação

Eduardo Fernandes 98512

João Ferreira 103625

Pedro Monteiro 97484

Lab Work nº2

2023

Contents

List of Code Blocks

1 Exercise 1

1.1 Exercise Description

This exercise involves creating a program using the OpenCV library to manipulate an image at the pixel level being the primary objective to extract a specific color channel from an image.

Color images are typically composed of three primary color channels: red, green, and blue

1.2 Code

1.2.1 extract_color.cpp

The program starts by asking the user for the necessary arguments, and then checks whether it can load the image using *cv::imread*.

Code Block 1: Extract Color Start Code

```
if (argc != 4)
{ // check for the right number of arguments
    std::cout << "Usage: "<< argv[0] << " <image_path> <output_path>
       <channel_number>\n";
    return -1;
}

cv::Mat image = cv::imread(argv[1], cv::IMREAD_COLOR); // read image
   file
if (image.empty())
{
    std::cout << "Could not read the image: " << argv[1] << std::endl
       ;
    return -1;
}
```

Then the program translates the user-specified color channel, designated as 0 for Red, 1 for Green, and 2 for Blue, into the corresponding channel in the RGB color model.

To align this with OpenCV's default BGR (Blue, Green, Red) format, a switch statement remaps the user's input from the RGB to the BGR format.

Code Block 2: Extract Color Channel

```
int channel = std::stoi(argv[3]); // get channel from command line

int bgrChannel; // convert to R (0) G (1) B(2) (instead of opencv
   default BGR)
switch (channel)
{
case 0:
    bgrChannel = 2;
    break; // Red
case 1:
    bgrChannel = 1;
```

```
    break; // Green
case 2:
    bgrChannel = 0;
    break; // Blue
default:
    std::cout << "Invalid channel number. It should be 0 (Red), 1 (
       Green), or 2 (Blue).\n";
    return -1;
}
```

Finally, the program creates a single-channel image from the specified color channel of the input image and saves the newly processed image.

Code Block 3: Extract Color Save Image

```
// create a single channel image with the same size as the input
   image
cv::Mat singleChannel(image.rows, image.cols, CV_8UC1);

for (int y = 0; y < image.rows; ++y)
{ // iterate over each pixel
    for (int x = 0; x < image.cols; ++x)
    {
        uchar color = image.at<cv::Vec3b>(y, x)[bgrChannel]; //
           extract specific channel value
        singleChannel.at<uchar>(y, x) = color;             // write
           the color to the single channel image
    }
}

// save the single channel image
cv::imwrite(argv[2], singleChannel);

return 0;
```

1.3 Usage

To execute the program, should be provided three arguments: the path of the image to process, the desired name for the output image, and the specific color channel to retain, red (R), green (G), or blue (B).

```
$ ./extract_color <image_path> <output_path> <channel_number>
```

1.4 Results

The resulting images, as shown in Figure 1, illustrate the distinct components of the original image's color information. The Channel 0, Channel 1, and Channel 2 images highlight the variations in color intensity across the Red, Green, and Blue channels, respectively. These grayscale images are particularly useful for analyzing the color composition and presence of specific color intensities within the original image.

The original image (Figure 1a) shows a vibrant sunflower with rich yellow petals and a dark brown center against a clear blue sky.

The Channel 0 image (Figure 1b) represents the red component of the image. In the grayscale translation, areas that were originally red appear lighter, indicating a higher intensity in the red channel. Given the lack of strong red tones in the primary image, this channel appears relatively uniform, with only subtle variations in intensity. The central part of the sunflower, which contains some reddish-brown hues, shows a moderate level of brightness, distinguishing it from the petals and background.

The Channel 1 image (Figure 1c) captures the green component. The leaves and stem of the sunflower, abundant in green, are noticeably bright in this channel. The petals, though yellow in the original image, reflect some green light and thus exhibit a moderate brightness here. In contrast, the sky, which lacks green, is rendered as a darker backdrop.

The Channel 2 image (1d) illustrates the blue component. The original image's blue sky is vividly represented here, appearing as the lightest part of the grayscale image, indicating its dominance in the blue channel. The yellow petals show as dark gray because yellow is the complement of blue and contains little to no blue light.

(a) Original Image

(b) Channel 0

(c) Channel 1

(d) Channel 2

Figure 1: Get Single Channel Color Image from Original Image

2 Exercise 2

2.1 Exercise Description

This exercise involves, without relying on existing functions from the OpenCV library, to perform several image processing operations. Furthermore, OpenCV was only used to display the images and, as previously mentioned, it was not used to implement the functionalities.

- Creating a Negative Image;
- Mirroring an Image;
- Rotating an Image by Multiples of 90°;
- Adjusting Light Intensity;

2.2 Code

2.2.1 image_processing.h

Central to this header file is the Pixel structure, which represents an individual pixel with three color channels: red, green, and blue. This structure forms the basis for manipulating image data at the pixel level.

Following the pixel definition, the file outlines various function prototypes. The *readImage* function reads an image file into a 2D vector of Pixel objects, while *writeImage* does the reverse, writing this pixel data back to a file. For image alterations, the *createNegative* function generates an inverted color version of the image, and *adjustBrightness* modifies the brightness level. The file also includes functions for image transformations: *rotateImage* to rotate the image by multiples of 90 degrees, and *mirrorHorizontal* and *mirrorVertical* for creating mirrored versions of the image along horizontal and vertical axes, respectively.

Code Block 4: Image Processing Header File

```
#ifndef IMAGE_PROCESSING_H
#define IMAGE_PROCESSING_H

#include <vector>
#include <string>

struct Pixel { // define Pixel structure to represent individual pixels of an image
    unsigned char r, g, b;
};

// function prototype for reading an image file and converting it into a 2D vector of Pixels
std::vector<std::vector<Pixel>> readImage(const std::string &file);

// function prototype for writing a 2D vector of Pixels to an image file
void writeImage(const std::string &file, const std::vector<std::vector<Pixel>> &data);

```

```
// function prototype to get the negative of the image
std::vector<std::vector<Pixel>> createNegative(const std::vector<std
    ::vector<Pixel>> &data);

// function prototype to adjust the brightness of an image
void adjustBrightness(std::vector<std::vector<Pixel>> &data, int
    adjustment);

// function prototype for rotating an image by a specified angle
std::vector<std::vector<Pixel>> rotateImage(const std::vector<std::
    vector<Pixel>>& data, int angle);

// function prototype for creating a horizontal mirror of an image
std::vector<std::vector<Pixel>> mirrorHorizontal(const std::vector<
    std::vector<Pixel>> &data);

// function prototype for creating a vertical mirror of an image
std::vector<std::vector<Pixel>> mirrorVertical(const std::vector<std
    ::vector<Pixel>> &data);

#endif
```

2.2.2 negative_image.cpp

This program focuses on reading an image file, creating its negative version by inverting the colors, and then saving this modified image to a new file.

The essence of the program lies in its three core functions. Firstly, *readImage* employs OpenCV's functionality to load an image from a file and transforms it into a 2D vector of Pixel structures.

Code Block 5: Read Image Function

```
// read an image file and convert it into a 2D vector of Pixels
std::vector<std::vector<Pixel>> readImage(const std::string &file) {
    cv::Mat img = cv::imread(file, cv::IMREAD_COLOR); // Read the
        image using OpenCV

    if (img.empty()) { // check if image loading was successful
        throw std::runtime_error("Error in loading the image");
    }

    int height = img.rows;
    int width = img.cols;
    std::vector<std::vector<Pixel>> data(height, std::vector<Pixel>(
        width));

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            cv::Vec3b color = img.at<cv::Vec3b>(cv::Point(x,y));
            data[y][x] = {color[2], color[1], color[0]}; // Note:
                OpenCV uses BGR format
        }
```

```
    }

    return data;
}
```

The *writeImage* function performs the reverse operation. It takes a 2D vector of Pixel structures and converts it into an OpenCV image, which is then saved to a file.

Code Block 6: Write Image Function

```
// function to write a 2D vector of Pixels to an image file
void writeImage(const std::string &file, const std::vector<std::
   vector<Pixel>> &data) {
    int height = data.size();
    int width = data[0].size();
    cv::Mat img(height, width, CV_8UC3); // OpenCV image

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            img.at<cv::Vec3b>(cv::Point(x,y)) = cv::Vec3b(data[y][x].
               b, data[y][x].g, data[y][x].r); // Note: OpenCV uses
               BGR format
        }
    }

    cv::imwrite(file, img); // Write the image using OpenCV
}
```

The *createNegative* function showcases basic pixel manipulation by inverting the color of each pixel. This function iterates over each pixel, subtracting its value from the maximum possible (255), creating a negative image.

Code Block 7: Create Negative Function

```
std::vector<std::vector<Pixel>> createNegative(const std::vector<std
   ::vector<Pixel>> &data) {
    std::vector<std::vector<Pixel>> result = data; // get a copy of
       the image

    // iterate over each pixel and invert the colors by subtracting
       the current value from 255
    for (auto &row : result) {
        for (auto &pixel : row) {
            pixel.r = 255 - pixel.r; // red component
            pixel.g = 255 - pixel.g; // green component
            pixel.b = 255 - pixel.b; // blue component
        }
    }

    return result;
}
```

2.2.3 Usage

The program only needs 2 arguments, the original image and the output image.

```
$ ./negative_image <input_file> <output_file>
```

2.2.4 Results

Figure 2 displays the original image, a photograph of a sunflower characterized by its rich yellow petals, a deep brown core, and a green stem, all set against a backdrop of a clear blue sky.

Conversely, Figure 3 presents the negative of the aforementioned image. This visual effect is achieved by inverting the color values across the image. The inversion process transforms the yellow petals into a dramatic blue, the once brown core becomes tinged with pale azure, and the sky adopts a warm golden hue. The green stem now exhibits shades of magenta, completing a spectrum that is diametrically opposed to the original.

Figure 2: Original Image

Figure 3: Negative Image

2.2.5 mirror_image.cpp

The code is designed to read an image, create both horizontal and vertical mirrored versions of the image, and then write these modified images to new files.

The process begins with the function *readImage*, which takes a file path as input and uses OpenCV's cv::imread function to load the image into a matrix structure (cv::Mat).z

The *writeImage* function performs the inverse operation. It takes a two-dimensional vector of Pixel structures and constructs a new OpenCV matrix. The function ensures that the data is converted back into BGR format before calling cv::imwrite to save the image to a file.

Additionally, two functions, *mirrorHorizontal* and *mirrorVertical*, are responsible for creating the mirrored versions of the image.

The *mirrorHorizontal* function creates a copy of the image data and then iterates over half of each row, swapping pixels from one side to the other, effectively flipping the image horizontally.

Code Block 8: Mirror Horizontal Function

```
1 // create a horizontal mirror of an image
2 std::vector<std::vector<Pixel>> mirrorHorizontal(const std::vector<
     std::vector<Pixel>> &data) {
3     std::vector<std::vector<Pixel>> result = data; // copy of the
        original image data
```

```
    int width = result[0].size();
    int height = result.size();

    // iterate through each row and swap pixels horizontally
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width / 2; ++x) { // width / 2 ensures
           that each pixel is only swapped once
            // without width / 2, reaching the midpoint of the image,
                starts swapping the pixels back
            // to their original positions.
            std::swap(result[y][x], result[y][width - x - 1]); //
               swap pixels for horizontal mirroring
        }
    }

    return result;
}
```

Similarly, the *mirrorVertical* function flips the image vertically, swapping pixels in each column.

Code Block 9: Mirror Vertical Function

```
// create a vertical mirror of an image
std::vector<std::vector<Pixel>> mirrorVertical(const std::vector<std
   ::vector<Pixel>> &data) {
    std::vector<std::vector<Pixel>> result = data; // copy of the
       original image data
    int width = result[0].size();
    int height = result.size();

    // iterate through each row and swap pixels vertically
    for (int x = 0; x < width; ++x) {
        for (int y = 0; y < height / 2; ++y) { // height / 2 ensures
           that each pixel is only swapped once
            std::swap(result[y][x], result[height - y - 1][x]); //
               swap pixels for vertical mirroring
        }
    }

    return result;
}
```

2.2.6 Usage

To execute his program is necessary the path to the original image, and the names of the inverted images, horizontally and vertically.

```
$ ./mirror_image <input_file> <output_file_horizontal> <
   output_file_vertical>"
```

2.2.7 Results

The original image, as shown in Figure 4, displays a single sunflower with a bright yellow appearance against a clear blue sky. The subsequent images, Figure 5 and Figure 6, demonstrate the outcomes of the horizontal and vertical mirroring processes, respectively.

Figure 5 presents the horizontally mirrored image. In this transformation, the sunflower is flipped along its vertical axis. This effect is achieved by swapping pixels on the left side of the image with their corresponding pixels on the right side. As a result, the left side of the original sunflower now appears on the right and vice versa, creating a symmetrical image that mimics the reflection one would see on the surface of water.

Figure 6 shows the vertically mirrored image. Here, the mirroring is applied along the horizontal axis, effectively inverting the sunflower's position as if reflected by a horizontal surface below the original image. The top of the sunflower is now at the bottom, and the stem appears to be directed upwards

Figure 4: Original Image

Figure 5: Horizontally Mirroed Image

Figure 6: Vertically Mirroed Image

2.2.8 rotate_image.cpp

The program's primary functionality is to read an image from a file, rotate it by an angle multiple of 90, and then write the rotated image back to a file.

The core image manipulation is handled by the *rotateImage* function. This function takes the 2D vector of Pixel structures and an angle as inputs.

The angle is first normalized to be one of the four possible values: 0, 90, 180, or 270 degrees. The function then proceeds to rotate the image accordingly, throwing an error if the

provided angle is not a multiple of 90, ensuring that only valid rotations are performed. The rotation is achieved by repositioning pixels in the 2D vector to their new locations corresponding to the specified rotation angle.

Code Block 10: Rotate Image Function

```
std::vector<std::vector<Pixel>> rotateImage(const std::vector<std::
   vector<Pixel>>& data, int angle) {
    angle = (angle % 360 + 360) % 360; // normalize the angle to one
       of 0, 90, 180, 270 degrees

    if (angle % 90 != 0) { // check if the angle is a multiple of 90
        throw std::invalid_argument("Rotation angle must be a
           multiple of 90 degrees");
    }

    int height = data.size();
    int width = data[0].size();
    std::vector<std::vector<Pixel>> result;

    switch (angle) {
        case 90: // rotate clockwise by 90 degrees
            result.resize(width, std::vector<Pixel>(height));
            for (int y = 0; y < height; ++y) {
                for (int x = 0; x < width; ++x) {
                    result[x][height - 1 - y] = data[y][x];
                }
            }
            break;

        case 180: // rotate by 180 degrees
            result.resize(height, std::vector<Pixel>(width));
            for (int y = 0; y < height; ++y) {
                for (int x = 0; x < width; ++x) {
                    result[height - 1 - y][width - 1 - x] = data[y][x
                       ];
                }
            }
            break;

        case 270: // rotate by 270 degrees
            result.resize(width, std::vector<Pixel>(height));
            for (int y = 0; y < height; ++y) {
                for (int x = 0; x < width; ++x) {
                    result[width - 1 - x][y] = data[y][x];
                }
            }
            break;

        default:
            return data; // for 0 degrees or any multiple of 360
               degrees, return the original image
```

```
42      }
43
44      return result;
45  }
```

2.2.9 Usage

```
$ ./rotate_image <input_file> <output_file> <angle>
```

2.2.10 Results

The original image 7a works as the baseline for comparison.

The second image 7b shows the original image after being rotated by 180 degrees. This rotation creates a mirror image along both the horizontal and vertical axes. The sunflower appears upside down but remains otherwise unaltered, retaining its colors and details. This demonstrates the effect of a half-turn rotation where the top and bottom, as well as the left and right sides of the original image, have been swapped.

The third image 7c illustrates the original image rotated by 90 degrees. In this rotation, the image's orientation changes from portrait to landscape. The sunflower, which was previously upright, is now lying on its side, indicating a quarter-turn clockwise rotation. This transformation swaps the image's dimensions, with the width becoming the height and vice versa, which can lead to changes in the perceived aspect ratio if the original image was not square.

Lastly, image 7d presents the original image after a 270-degree rotation. This rotation is the equivalent of a 90-degree counter-clockwise turn or a 180-degree rotation followed by an additional 90-degree clockwise rotation. The sunflower is seen as if it has been turned to the left from its original position, with its left side now at the bottom. Similar to the 90-degree rotation, this also changes the image's orientation from portrait to landscape or the reverse, affecting the aspect ratio if the image is rectangular.

(a) Original Image

(b) Image Rotated 180º

(c) Image Rotated 90º

(d) Image Rotated 270º

Figure 7: Get Single Channel Color Image from Original Image

2.2.11 brightness.cpp

The *readImage* and *writeImage* functions in the program work as supporting operations for the *adjustBrightness* function and follow the previously defined functions for reading and writing image data.

The *adjustBrightness* function is a crucial part of the program, designed to modify the luminance of each pixel in the image. This function directly alters the intensity of the red, green, and blue (RGB) components of each pixel uniformly by a specified adjustment value, which can be positive or negative.

When the *adjustBrightness* function is called, it iterates through every row of pixels and then through every pixel within that row. For each pixel, the function increases or decreases the value of each color component (RGB) by the adjustment parameter provided. If the adjustment value is positive, the function brightens the image; if negative, it darkens the image.

Code Block 11: Adjust Brightness Image Function

```
1 // function to adjust the brightness of an image
```

```
void adjustBrightness(std::vector<std::vector<Pixel>> &data, int
    adjustment) {
     for (auto &row : data) { // iterate over each row
         for (auto &pixel : row) { // iterate over each pixel
             // adjust the red, green, and blue components
             pixel.r = std::min(std::max(pixel.r + adjustment, 0),
                255); // increase or decrease the red component
             pixel.g = std::min(std::max(pixel.g + adjustment, 0),
                255); // increase or decrease the green component
             pixel.b = std::min(std::max(pixel.b + adjustment, 0),
                255); // // increase or decrease the blue component
         }
     }
}
```

2.2.12 Usage

This program similarly needs the input image and the name of the output image, and also the value for adjusting the brightness.

```
$ ./brightness <input_file> <output_file> <brightness_adjustment>
```

2.2.13 Results

The original image, 8, is subjected to different brightness adjustments, both positive or negative values.

Figure 8: Original Image

Figure 9 illustrates the effects of these adjustments in four parts: 9a and 9b show the image after applying a brightness adjustment of +100 and -100, respectively.

In 9a, the brightness increase results in a much lighter image where the details of the sunflower's petals and the background become more washed out due to the added intensity.

Conversely, 9b presents a darker version of the same image; the decrease in brightness makes the image appear slightly underexposed, with the details of the sunflower and background less discernible.

In 9c, the image becomes overexposed, with most of the details lost to the high levels of brightness; the sunflower petals and center merge with the background in a high-key effect.

In contrast, 9d shows an image that is significantly darkened, where the sunflower is barely visible, illustrating a near-complete loss of detail.

(a) brightness_adjustment = 100

(b) brightness_adjustment = -100

(c) brightness_adjustment = 200

(d) brightness_adjustment = -200

Figure 9: Images with Different Adjustment Values for Brightness

3 Exercise 3

3.1 Exercise Description

This program involves the implementation of a Golomb coding class. Golomb coding is a lossless data compression method, well-suited for geometrically distributed data, and is particularly effective for encoding sequences of integers where smaller values are more frequent. Having this in mind, this class should have some key functionalities:

- **Encoding function**: The class should include a function to encode integers into a sequence of bits. This function is the centerpiece of the Golomb coding process, converting input integers into their corresponding compressed binary representations.
- **Decoding function**: The class should also feature a function to decode sequences of bits back into integers. This function reverses the encoding process, ensuring the retrieval of original data from its compressed form.
- **Adaptive Parameter 'm'**: The class needs to incorporate a parameter 'm', which plays a critical role in adapting the Golomb codes to suit different probability distributions.
- **Handling Negative Numbers**: This class should have the ability to handle negative integers. The user can choose, based on their specific requirements, which of the following approaches to choose. They are the sign and magnitude approach and positive/negative value interleaving.

3.2 Code

3.2.1 golomb_coder.h

The class has a private section that includes several functions. The *intToBinary* function converts an integer *n* to its binary representation, fitting it into a specified number of bits, and the *binaryToInt* one transforms a binary string back into its integer form. This section also declares the fundamental Golomb encoding and decoding algorithms, *encodeGolomb* and *decodeGolomb*.

The public section has the *GolombCoder(int m)* constructor, which initializes the class with a parameter *m*, crucial for determining the efficiency of the Golomb coding based on the data's probability distribution. Also declares the *encode* and *decode* functions that leverage the internal *encodeGolomb*/*decodeGolomb* functions and applies additional logic for the chosen negative number representation.

Code Block 12: Golomb Class

```
#ifndef GOLOMB_CODER_H
#define GOLOMB_CODER_H

#include <string>

using namespace std;

class GolombCoder {
private:
    int m;

    string intToBinary(int n, int bits);
    int binaryToInt(const string &binary);
    string encodeGolomb(int num);
    int decodeGolomb(const string& golomb);

public:
    GolombCoder(int m);

    string encode(int num, const string& mode);
    int decode(const string& encoded, const string& mode);
};

#endif
```

3.2.2 golomb_coder.cpp

The *intToBinary* function starts by initializing a string binary of length bits, filled with '0's. The conversion process involves iterating through the binary string in reverse order. For each position i in the string, the function calculates the corresponding bit by taking the remainder of the integer n. If the remainder is 1, the bit at position i is set to '1', otherwise it remains '0'. After processing each bit, the integer n is divided by 2, effectively shifting it right by one-bit position for the next iteration.

The *binaryToInt* function performs the reverse operation, converting a binary string back into its integer equivalent. It initializes an integer *num* to 0, which accumulates the value as the function processes each bit in the binary string. The function iterates over each character in the binary string. For every bit, it multiplies *num* by 2 (a left shift in binary terms) and then adds the numeric value of the bit (either 1 or 0) to *num*.

Code Block 13: intToBinary and binaryToInt implementations

```
string GolombCoder::intToBinary(int n, int bits) {
    string binary = string(bits, '0');
    for (int i = bits - 1; i >= 0; --i) {
        binary[i] = (n % 2) ? '1' : '0';
        n /= 2;
    }
    return binary;
}

int GolombCoder::binaryToInt(const string &binary) {
    int num = 0;
    for (char bit : binary) {
        num = num * 2 + (bit - '0');
    }
    return num;
}
```

The *encodeGolomb* function performs the encoding of an integer into its Golomb code representation. The process begins by dividing this number by the class's parameter m, yielding two components: the quotient q and the remainder r. The quotient q represents the number of uninterrupted '1's in the Golomb code, followed by a '0', meaning the end of the quotient part.

To handle the remainder r, the function first determines the required number of bits to represent r based on m. This is calculated using *ceil(log2(m))*, ensuring that the binary representation of r accommodates the largest possible value within m. The function then converts r into its binary representation using *intToBinary(r, bits)* and it concatenates the quotient part with the remainder part.

The *decodeGolomb* function performs the reverse operation: it decodes a Golomb-coded string back into its original integer form. The function initializes two variables, q (the quotient) and *index*, to traverse the Golomb code.

The decoding process starts by counting the number of consecutive '1's in the Golomb string, which represents the quotient q. Upon encountering a '0', the function knows that the quotient part has ended, and it proceeds to the remainder part. The index is incremented to skip the '0' and to start processing the remainder. The remainder part requires calculating the number of bits (*rBitsLength*) that encode the remainder, which is derived from the class parameter m. The function then extracts the binary string representing the remainder and uses *binaryToInt* to convert it back into an integer r.

In the end, the function calculates the original integer using the formula $q * m + r$, combining the quotient and remainder.

Code Block 14: encodeGolomb and decodeGolomb implementations

```
string GolombCoder::encodeGolomb(int num) {
    int q = num / m;
    int r = num % m;
    int bits = ceil(log2(m));
    string qBits = string(q, '1') + "0";
    string rBits = intToBinary(r, bits);

    return qBits + rBits;
}

int GolombCoder::decodeGolomb(const string& golomb) {
    int q = 0;
    int index = 0;

    // Count the number of '1's until '0' is encountered
    while (golomb[index] == '1') {
        q++;
        index++;
    }
    // Skip the '0'
    index++;

    // Calculate the number of bits for the remainder
    int b = ceil(log2(m));
    int rBitsLength = b - ((1 << b) - m);
    string rBits = golomb.substr(index, rBitsLength);

    int r = binaryToInt(rBits);
    return q * m + r;
}
```

The *encode* and *decode* functions extend the basic functionality of Golomb coding by introducing the ability to handle negative numbers. This is achieved through two distinct modes: ”sign-magnitude” and ”interleaving”. Depending on the user’s choice, the program performs different actions that in the end, will result in the same results.

The *encode* function’s purpose is to encode an integer *num* into a Golomb code, with additional handling for negative numbers based on the specified mode. In the *sign-magnitude* mode, the function first determines if *num* is negative. If it is, a sign bit ’1’ is added to the Golomb code of the absolute value of *num*. If *num* is positive, a ’0’ is used as the sign bit. The *interleaving* mode alters the representation of negative numbers by interleaving them with positive numbers. If *num* is non-negative, it is doubled. If *num* is negative, it is transformed into a positive odd number using the formula *-2 * num - 1*. In both modes, the *encodeGolomb* function is applied to the transformed *num*.

The decode function reverses the encoding process, reconstructing the original integer from its Golomb code and considering the mode of negative number handling. In the *sign-magnitude* mode, begins by checking the first bit of the encoded string to determine the sign of the number. It then decodes the remainder of the string, excluding the sign bit, using *decodeGolomb*. The decoded number is negated if the sign bit indicates a negative value. On the *interleaving* mode,

the function first decodes the entire Golomb string. It then determines the sign and original value of the number based on its parity. Even decoded numbers are divided by 2 to retrieve the original positive value, while odd numbers are processed to recover the original negative value using the reverse of the encoding formula.

Code Block 15: encode and decode implementations

```
string GolombCoder::encode(int num, const string& mode) {
    bool isNegative = num < 0;
    string encoded;
    if (mode == "sign-magnitude") {
        string signBit = isNegative ? "1" : "0";
        num = abs(num);
        encoded = signBit + encodeGolomb(num);
    } else if (mode == "interleaving") {
        num = (num >= 0) ? (2 * num) : (-2 * num - 1);
        encoded = encodeGolomb(num);
    }
    return encoded;
}

int GolombCoder::decode(const string& encoded, const string& mode) {
    int num = 0;
    if (mode == "sign-magnitude") {
        bool isNegative = encoded[0] == '1';
        num = decodeGolomb(encoded.substr(1)); // Decode Golomb part,
            excluding the sign bit
        return isNegative ? -num : num;
    } else if (mode == "interleaving") {
        num = decodeGolomb(encoded);
        if (num % 2 == 0) { // Even numbers are positive
            return num / 2;
        } else { // Odd numbers are negative
            return -((num + 1) / 2);
        }
    }
    return num; // Fallback, should not occur
}
```

3.2.3 Main section

This program has the *main* function that was developed to test the Golomb class functionalities.

The function begins by validating the command-line arguments. It checks if exactly three arguments are provided: the program name, the number to be encoded and decoded, and the mode of operation (either ”sign-magnitude” or ”interleaving”). If the argument count does not match this requirement, the program displays a usage message and exits with an error status. In addition, some validations are carried out on the values entered by the user.

An instance of *GolombCoder* is created with a predefined parameter m, in this case, $m = 2$. Then, the provided number is encoded using the selected mode. The encoded string is then printed to showcase the result of the Golomb encoding process. Subsequently, the encoded

string is decoded back into an integer using the same mode.

Code Block 16: main function

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

    if(argc != 3) {
        cerr << "Usage: " << argv[0] << " <
           number_to_encode_and_decode> <sign-magnitude |
           interleaving>\n";
        return 1;
    }

    int number = 0;
    try
    {
        number = stoi(argv[argc-2]);
    }
    catch(const exception& e)
    {
        cerr << "Error: invalid number" << '\n';
        return 1;
    }

    string mode { argv[argc-1] };
    if (mode != "sign-magnitude" && mode != "interleaving")
    {
        cerr << "Error: invalid mode" << '\n';
        return 1;
    }

    GolombCoder golomb(2); // Example with m = 2

    // Example with interleaving
    string encoded = golomb.encode(number, mode); // Encode -5 using
       interleaving

    cout << "Original: " << number << endl;
    cout << "Encoded: " << encoded << endl;
    int decoded = golomb.decode(encoded, mode); // Decode using
       interleaving
    cout << "Decoded: " << decoded << endl;
    return 0;
}
```

3.3 Usage

To run the program, it is necessary to provide the number to encode and decode and the mode that will be used on the negative values:

```
$ ./golomb_coder <number_to_encode_and_decode> <sign-magnitude |
   interleaving>$
```

3.4 Results

Below are two examples of the execution of the program with the numbers 5 and -7 for both modes. The original number, the encoded string, and the decoded number are displayed.

```
~/Universidade/5Ano/IC/ic/trab2/bin main !2 ❯ ./golomb_coder 5 sign-magnitude
Original: 5
Encoded: 01101
Decoded: 5
~/Universidade/5Ano/IC/ic/trab2/bin main !2 ❯ ./golomb_coder 5 interleaving
Original: 5
Encoded: 1111100
Decoded: 5
~/Universidade/5Ano/IC/ic/trab2/bin main !2 ❯ ./golomb_coder -7 sign-magnitude
Original: -7
Encoded: 111101
Decoded: -7
~/Universidade/5Ano/IC/ic/trab2/bin main !2 ❯ ./golomb_coder -7 interleaving
Original: -7
Encoded: 11111101
Decoded: -7
```

Figure 10: Some results for testing Golomb Class

4 Exercise 4

4.1 Exercise Description

In this exercise, we are asked to implement a codec using the previously implemented Golomb class. The second implementation was used to compress and decompress audio.

4.2 Lossless audio codec

The audio encoding function initially takes a .wav file to encode, a parameter x (x > 0), which updates the value of m every x values, a parameter y (x >= y > 0), which uses the previous y values to calculate the new m, a parameter for the order (2 or 3) that defines how the prediction of values is calculated, and finally, a name for the resulting compressed file.

The first step begins with reading the .wav file. After this, a header is created, containing all the necessary information to perform the encoding process. For the encoding process, the first (order * number of audio channels) values keep their original values and are mapped. For the remaining values, errors are calculated by subtracting the predicted value from the actual value. Next, these errors are mapped to a positive even number for when the error < 0 or to a positive odd number when the error >= 0. Finally, all values are encoded and written to a file.

While the error calculation process is ongoing, the value of m is altered every x values, where y values of the previous mapped errors are used to calculate the average and consequently the new value of m.

As for the audio decoding function, this function receives an encoded file and the name of the file that will be created with the decoded audio. Initially, the function reads the header, which will contain all the information to perform the decoding of the file. Consequently, the decoding process performs the operations of the encoding process but in reverse order.

4.2.1 Implementation Details

- Error Calculation:
  - First, it is necessary to determine which order to apply for predicting the value, and we have two options:
    - Order 3 Formula:

      .

      $$X_n = 3 \cdot X_{n-1} - 3 \cdot X_{n-2} + X_{n-3} \tag{1}$$

    - Order 2 Formula

      .

      $$X_n = 2 \cdot X_{n-1} - X_{n-2} \tag{2}$$

  - Mapping
    - If Err < 0 => Err = Err * (-2), otherwise Err = (Err + 1) * 2
  - Dynamic m calculation
    - The parameter x in the program will be inversely proportional to the frequency of dynamic calculation of m. In other words, if x is a high value, the frequency at which we will alter the value of m is lower than if we use a smaller value of x.

* The parameter y in the program defines the range of values used to calculate the average of values that will be used to compute the new m. In general, the higher the value of y, the larger the sample space, and consequently, the better the 'quality' of the newly calculated m.

– Formula to calculate m:

$$m = -\frac{1}{\log_2\left(\frac{\text{medium}}{\text{medium+1}}\right)} \tag{3}$$

4.3 Usage

```
$ ./golomb_codec encode ../src/sample.wav output.txt

$ ./golomb_codec decode ../bin/output.txt outputSample.wav
```

In addition to these commands, it is also possible to configure the values of order, x, and y:

```
$ ./golomb_codec encode ../src/sample.wav output.txt -order
   <[2,3]> -x <value> -y <value>
```

4.4 Results

In this section, there will be no results given the fact that we are unable to generate the files correctly. Despite being able to encode and decode the file, we cannot obtain any results. After several unsuccessful attempts, we decided to only leave the code and the explanation of its implementation (as previously mentioned) and what we intended to achieve.

5 Workload and General Information

The repository for the project is ⟨https://github.com/pedromonteiro01/ic⟩.
Each team member made an equitable contribution to the project's resolution.