Information and Coding

Universidade de Aveiro

Sebastian D. González, Catarina Raposo Barroqueiro, Ana Raquel Ângera Conceição

Information and Coding

Dept. de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

sebastian.duque@ua.pt(103690), cbarroqueiro@ua.pt(103895),
arconceicao@ua.pt(98582)

December 3, 2023

Contents

List of Figures

Glossário

RGB Red Green Blue.

Chapter 1

Introduction

This lab work involves several tasks related to information and coding, including image and audio file manipulation using the OpenCV library.

These tasks involve a combination of image and audio processing, as well as coding and decoding using Golomb coding techniques. The report will play a crucial role in presenting the findings and comparing the results with existing codecs.

The project is available at the following link: `https://github.com/ElSebasdg/IC/tree/main/trab2`.

Chapter 2

Exercise 1

Using the **OpenCV** library, we had to implement a program that extracts a color channel from an image, reading it and writing pixel by pixel, creating a single channel image with the result. The file names and channel number are passed as command line arguments to the program.

2.1 Usage Instructions

To use the program correctly, follow these steps:

1. To compile the ‘color_channel.cpp‘ you just need to do ‘make‘ inside the build directory:

```
make

```

2. Finally to execute the following command ‘./color_channel‘ followed by image path and the channel number:

```
./color_channel ../horacio 1

```

   Use 0 for Blue channel, 1 for Green channel and 2 for Red.

   All the images will be stored inside the build directory.

2.2 Code

```cpp
int main(int argc, char** argv){
    if (argc != 3) {
        printf("Usage: ./color_channel <Image_Path> <Channel_Number>\n");
        return -1;
    }
    char* inputImagePath = argv[1];
    int channelNumber = atoi(argv[2]);
    Mat image = imread(inputImagePath, 1);
    if (!image.data) {
        printf("Image not found\n");
        return -1;
    }
    if (channelNumber < 0 || channelNumber > 2) {
        printf("Invalid channel number.\nUse 0 for Blue \n1 for Green\n2 for Red.\n");
        return -1;
    }
    Mat extractedChannel(image.rows, image.cols, CV_8UC1);
    // Iterate over pixels and copy the selected channel
    for (int i = 0; i < image.rows; ++i) {
        for (int j = 0; j < image.cols; ++j) {
            extractedChannel.at<uchar>(i, j) = image.at<Vec3b>(i, j)[channelNumber];
        }
    }
    string channelName;
    switch (channelNumber) {
        case 0:
            channelName = "Blue";
            break;
        case 1:
            channelName = "Green";
            break;
        default:
            channelName = "Red";
            break;
    }
    string outputFileName = getOutputFileName(inputImagePath, channelName);
    imwrite(outputFileName, extractedChannel);
    namedWindow("Extracted Channel", WINDOW_AUTOSIZE);
    imshow("Extracted Channel", extractedChannel);
    waitKey(0);
    return 0;
}
```

Listing 2.1: color_channel.cpp

As said before, the code uses **OpenCV** library and it uses functions like `imread` to load the input image, at to access pixel values, `imwrite` to save the output image, and `namedWindow` and `imshow` to display the image of the extracted channel. It iterates over each pixel of the input image and copies the selected color channel to the single-channel image.

The code also includes a helper function `getOutputFileName` to generate the output file name based on the input image path and the color channel name.

```
string getOutputFileName(const string& inputImagePath, const
     string& channelName) {
    size_t dotPos = inputImagePath.find_last_of(".");
    if (dotPos != string::npos) {
        return "extracted_" + channelName + inputImagePath.
    substr(dotPos);
    } else {
        printf("Unable to determine input image format.
    Saving as extracted_channel.png.\n");
        return "extracted_" + channelName + ".png";
    }
}
```

Listing 2.2: getOutputFileName function

2.3 Results

The expected outputs include the successful extraction and display of the specified color channel from the input image depending on the color channel extracted.

Figure 2.1: Extracted Blue channel image

(a) Extracted Green channel (b) Extracted Red channel

Figure 2.2: Extracted Green and Red channel images

As it can be seen the three possible outputs are in a grayscale appearance. This a natural outcome of extracting a single color channel from a color image and storing it in a single-channel image. When displayed, **OpenCV** interprets it as a grayscale image because it has only one channel. The lack of color channels means that each pixel is displayed with a single intensity value, creating a grayscale appearance.

The absence of color channels in the displayed and saved images makes them visually equivalent to grayscale images.

Chapter 3

Exercise 2

In this exercise we implemented a program without utilizing existing **OpenCV** functions to perform the following operations on an image:

a) Generate the negative version of the image

b) Create mirrored versions of the image: (a) horizontally, (b) vertically

c) Rotate the image by a multiple of 90 degrees

d) Adjust the intensity values of the image to increase (more light) or decrease (less light).

3.1 Usage Instructions

1. To compile the ‘ex2.cpp‘ you just need to do ‘make‘ inside the build directory:

```
make

```

2. Finally to execute the following command ‘./ex2‘ followed by image path, rotation_angle and light intensity value.

```
./ex2 ../horacio.png 90 150

```

The rotation angle must be multiple of 90 degrees and the light intensity value must be between 0 and 255

All the images produced are stored inside the build directory

3.2 Code

3.2.1 ex2 main

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

    if (argc != 4) {
        cout << "Usage: " << argv[0] << " <image_path> <rotation_angle> <intensity_value>" << endl;
        return -1;
    }

    Mat originalImage = imread(argv[1], IMREAD_COLOR);

    if (!originalImage.data) {
        cout << "No image data." << endl;
        return -1;
    }

    int rotationAngle = stoi(argv[2]);
    if (rotationAngle % 90 != 0) {
        cout << "Rotation angle must be a multiple of 90 degrees." << endl;
        return -1;
    }

    int intensityValue = stoi(argv[3]);
    if (intensityValue < 0 || intensityValue > 255) {
        cout << "The intensity value must be between 0 and 255." << endl;
        return -1;
    }

    // a)
    Mat negativeImage = createNegativeImage(originalImage);
    saveImage(negativeImage, "negative_image.jpg");

    // b)
    Mat horizontallyMirroredImage = createMirroredImage(originalImage, true);
    Mat verticallyMirroredImage = createMirroredImage(originalImage, false);

    saveImage(horizontallyMirroredImage, "horizontally_mirrored_image.jpg");
    saveImage(verticallyMirroredImage, "vertically_mirrored_image.jpg");


```

```
    // c)
    rotateImage(originalImage, rotationAngle);

    // d)
    Mat lightenedImage, darkenedImage;

    adjustIntensity(originalImage, intensityValue,
    lightenedImage);
    adjustIntensity(originalImage, -intensityValue,
    darkenedImage);

    string lightenedOutputFileName = "lightened.jpg";
    string darkenedOutputFileName = "darkened.jpg";

    imwrite(lightenedOutputFileName, lightenedImage);
    imwrite(darkenedOutputFileName, darkenedImage);

    waitKey(0);
    destroyAllWindows();
    return 0;
}
```

Listing 3.1: ex2 main condensed

The program works by calling the required functions, in the main program, for each point.

The main program is divided in points, each one corresponding to a letter of the operations that the program performs(explained in the beginning of the chapter 3).

3.2.2 ex2 a)

To generate the negative version of the image we use the function createNegativeImage. 3.2

```
Mat createNegativeImage(const Mat& original) {
    Mat negativeImage = original.clone();
    for (int y = 0; y < original.rows; ++y) {
        for (int x = 0; x < original.cols; ++x) {
            Vec3b& pixel = negativeImage.at<Vec3b>(y, x);

            // Invert RGB values
            pixel[0] = 255 - pixel[0]; // Blue
            pixel[1] = 255 - pixel[1]; // Green
            pixel[2] = 255 - pixel[2]; // Red
        }
    }
    return negativeImage;
}
```

Listing 3.2: ex2 createNegativeImage

This function creates a deep copy of the source image matrix and, for each pixel, obtains a reference to the pixels color values. It then inverts each channel value by subtracting the current value from 255.

3.2.3 ex2 b)

To generate the mirrored versions of the image we use the function createMirroredImage.

```
Mat createMirroredImage(const Mat& original, bool
    horizontally) {

    Mat mirroredImage = Mat::zeros(original.size(), original
    .type());
    for (int y = 0; y < original.rows; ++y) {
        for (int x = 0; x < original.cols; ++x) {
            int newX, newY;

            if (horizontally) {
                newX = original.cols - x - 1;
                newY = y;
            } else { // vertically
                newX = x;
                newY = original.rows - y - 1;
            }

            mirroredImage.at<Vec3b>(newY, newX) = original.
    at<Vec3b>(y, x);
        }
    }

    return mirroredImage;

}
```

Listing 3.3: ex2 createMirroredImage

The function begins by creating a new Mat named mirroredImage with the same size and type as the original image, initialized to all zeros and then, for each pixel:

- If **horizontally** is true, it mirrors the image horizontally by calculating the new X-coordinate as original.cols - x - 1 while keeping the Y-coordinate the same (newY = y)
- Else, it mirrors the image vertically by calculating the new Y-coordinate as original.rows - y - 1 while keeping the X-coordinate the same (newX = x).

3.2.4 ex2 c)

To generate the images rotated by a multiple of 90 degrees we use the function rotateImage.

Before actually calling the function itself, we check in the main 3.1 function if the rotation angle provided as a command line argument is a multiple of 90 degrees. If not, it prints an error message and exits the program with a return value of -1.

```
void rotateImage(const Mat& original, int rotationAngle) {
    int rows = original.rows;
    int cols = original.cols;
    Mat rotatedImage(rows, cols, original.type());switch (rotationAngle) {
        case 90:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(x, rows - y - 1) = original.at<Vec3b>(y, x);
                }
            }
            break;
        case 180:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(rows - y - 1, cols - x - 1) = original.at<Vec3b>(y, x);
                }
            }
            break;
        case 270:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(cols - x - 1, y) = original.at<Vec3b>(y, x);
                }
            }
            break;
        case -90:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(cols - x - 1, y) = original.at<Vec3b>(y, x);
                }
            }
            break;
        case -180:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
```

```
                    rotatedImage.at<Vec3b>(rows - y - 1,
cols - x - 1) = original.at<Vec3b>(y, x);
                }
            }
            break;
        case -270:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(x, rows - y - 1)
= original.at<Vec3b>(y, x);
                }
            }
            break;
        default:
            cout << "Invalid rotation angle." << endl;
            return;
    }
    switch (rotationAngle) {
        case 90:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(x, rows - y - 1)
= original.at<Vec3b>(y, x);
                }
            }
            break;
        case 180:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(rows - y - 1,
cols - x - 1) = original.at<Vec3b>(y, x);
                }
            }
            break;
        case 270:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(cols - x - 1, y)
= original.at<Vec3b>(y, x);
                }
            }
            break;
        case -90:
            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < cols; ++x) {
                    rotatedImage.at<Vec3b>(cols - x - 1, y)
= original.at<Vec3b>(y, x);
                }
            }
            break;
```

```
80          case -180:
81              for (int y = 0; y < rows; ++y) {
82                  for (int x = 0; x < cols; ++x) {
83                      rotatedImage.at<Vec3b>(rows - y - 1,
    cols - x - 1) = original.at<Vec3b>(y, x);
84                  }
85              }
86              break;
87          case -270:
88              for (int y = 0; y < rows; ++y) {
89                  for (int x = 0; x < cols; ++x) {
90                      rotatedImage.at<Vec3b>(x, rows - y - 1)
    = original.at<Vec3b>(y, x);
91                  }
92              }
93              break;
94          default:
95              cout << "Invalid rotation angle." << endl;
96              return;
97      }
98      string outputFileName = "rotated_" + to_string(
    rotationAngle) + ".jpg";
99      imwrite(outputFileName, rotatedImage);
100     cout << "Rotated image saved as: " << outputFileName <<
    endl;
101 }
```

Listing 3.4: ex2 rotateImage

The function begins by obtaining the number of rows and columns of the original image. Then, it creates a new Mat object (rotatedImage) with the same dimensions and type as the original image and uses a switch statement to handle different rotation angles (90, 180, 270, -90, -180, -270). For each case, nested loops iterate through each pixel of the original image and assign the corresponding pixel value to the rotated image.

3.2.5 ex2 d)

To generate the images with increased (more light) and decreased (less light) of the intensity values of the image we implemented the function adjustIntensity:

```
void adjustIntensity(const Mat& original, int intensityValue
    , Mat& result) {
    result = Mat::zeros(original.size(), original.type());
    for (int y = 0; y < original.rows; ++y) {
        for (int x = 0; x < original.cols; ++x) {
            for (int c = 0; c < original.channels(); ++c) {
                int newValue = original.at<Vec3b>(y, x)[c] +
     intensityValue;
                result.at<Vec3b>(y, x)[c] = saturate_cast<
    uchar>(newValue);
            }
        }
    }
}
```

Listing 3.5: ex2 adjustIntensity

On our approach, we decided that the user, using the command line the light intensity. Before proceeding, with the image processing operations, the input, is validated to ensure that the intensity value is within the acceptable range(between 0 and 255). If the intensity value is not valid, the program exits early, indicating an error condition.

3.3 Results

This is the image we used to perform our image operations:

Figure 3.1: Original image

3.3.1 ex2 a)

This is the image we obtained after inverting the RGB values of each pixel. That is why we obtain this expected "negative" colour:

Figure 3.2: Original image

3.3.2 ex2 b)

These were the images we get after using the createMirroredImage 3.3.

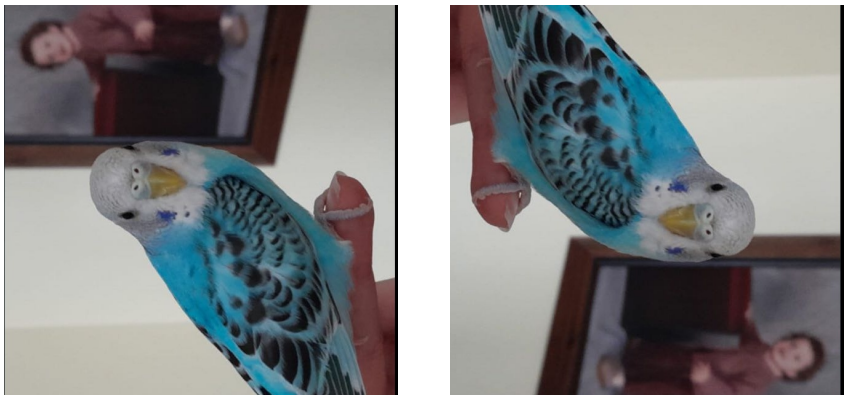
(a) Vertically mirrored image (b) Horizontally mirrored image.

Figure 3.3: Mirrored images

3.3.3 ex2 c)

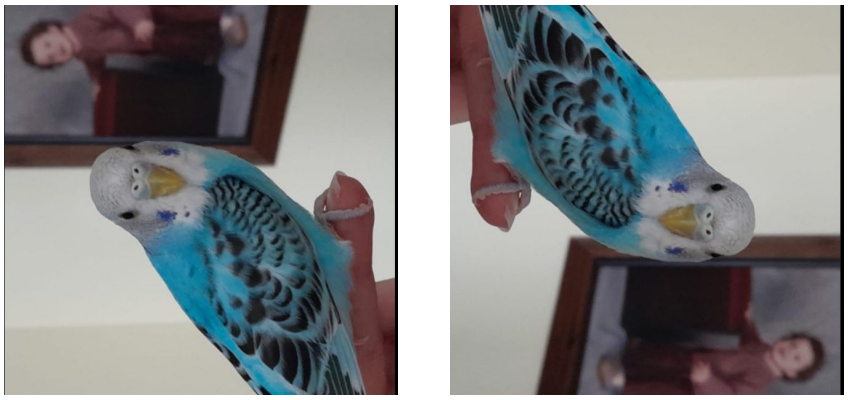
These were the images we obtained after doing the disered rotation angle.

The rotation is performed by iterating through each pixel in the original image and placing it in a new position in the rotated image based on the specified rotation angle.



(a) Rotated -90 degrees (b) Rotated 90 degrees

Figure 3.4: Rotation 90 and -90 degrees



(a) Rotated 270 degrees (b) Rotated -270 degrees

Figure 3.5: Rotation 270 and -270 degrees

(a) Rotated -180 degrees (b) Rotated 180 degrees

Figure 3.6: Rotation 180 and -180 degrees

We can conclude that a rotation of 270 degrees is equivalent to a rotation of -90 degrees. This is due to the geometric symmetry of rotations. Pixel manipulation for both cases involves mapping each pixel at position (y, x) in the original image to the rotated position (cols - x - 1, y). Visually, this rotation is commonly described as a "90 degrees clockwise rotation" because it represents a 90-degree rotation in the opposite (clockwise) direction.

3.3.4 ex2 d)

The program always produces two different images, one lightened and the other darkened according with the input in the command line. For each channel of the pixel at position (y, x) in the original image, the intensity is adjusted by adding the specified intensity value (intensityValue) obtaining the pretended images .

(a) Lightened image (b) Darkened image

Figure 3.7: Lightened and darkened image with input 120

Chapter 4

Exercise 3

In exercise 3, we developed a C++ class named `Golomb` for Golomb coding. This class encapsulates an entropy encoding technique designed to, efficiently, encode and decode integers accommodating both non-negative and negative numbers.

4.1 Code

The implemented C++ class for `Golomb` coding efficiently encodes and decodes integers using a specified divisor 'm', allowing adaptation to diverse probability distributions. This coding technique partitions integers into a quotient 'q' and a remainder 'r', aiming to represent 'q' as a sequence of 'q' consecutive false bits followed by a true bit. Then the remainder, r, is represented in truncated binary coding, using b:

$$b = \lceil log(m) \rceil$$

We define the *stopper* as $2^b - m$.
If $r < stopper$, code 'r' in binary representation using b-1 bits.
If $r \geq stopper$, code $r + 2^b - m$ in binary representation using b bits.

In the private part of our code, we created a method to calculate b and *stopper* based on the value of 'm'.

```
void getBits(int m) {
    if (m != 0){
        b = ceil(log2(m));
        // stopper: (2^b) - m
        stopper = pow(2, b) - m;
    } else {
        b = 0;
        stopper = 0;
    }
```

```
}
```

Listing 4.1: golomb.h

Our encode function starts by calculating the quotient (q) and remainder (r). If 'm' is not zero, it divides the number to be represented (num) by 'm' to obtain 'q' and 'r'. It constructs the unary code by appending 'q' number of '0's to the encoded string followed by a '1' to separate the quotient from the remainder. If 'm' is zero, it only appends a '1' since division by zero is not possible.

Then we call the getBits(m) function to calculate the number of bits 'b' required for encoding and the stopper value based on the provided 'm', and, if 'm' value is not one, we encode the remainder 'r' based on the logic explained before: If 'r' is within the representable range with minimum bits ($r < stopper$), encode 'r' in binary. Otherwise, if 'r' exceeds the representable range, encode the sum of the remainder and the number of values with minimum bits ($r + stopper$) using the maximum bits (b).

In the end, we add the sign bit ('1' for negative or '0' for positive), based on the sign of the original number 'num', and return the Golomb-encoded string representing the that number.

```
string encode(int num, int m){

    int q = 0;
    int r = 0;

    if (m != 0){
        q = abs(num) / m;
        r = abs(num) % m;
    }

    string encodedString = "";

   // Apply Golomb encoding
    for (int i = 0; i < q; i++) {
        encodedString += "0";
    }
    // use a bit (1) to represent the end of the quotient
   and the begginning of the r
    encodedString += "1";

    //int b = ceil(log2(m));
    //int stopper = pow(2, b) - m;
    getBits(m);


    if (m != 1) {
        std::string result = "";
        if (r < stopper) {
            int binary = r;
```

```
            for (int i = 0; i <= b - 2; i++) {
                //encodedString += ((binary & (1 << i)) != 0) ? '1' : '0';
                result = to_string(binary % 2) + result;
                binary /= 2;

            }
            encodedString += result;
        } else {
            int binary = r + stopper;   // r + (2^b)-m
            for (int i = 0; i <= b - 1; i++) {
                //encodedString += ((binary & (1 << i)) != 0) ? '1' : '0';
                result = to_string(binary % 2) + result;
                binary /= 2;
            }
            encodedString += result;
        }
    } else {
        encodedString += '0';
    }

    // if the number is negative, add a 1 at the end of the remainder to indicate the sign
    num < 0 ? encodedString += "1" : encodedString += "0";


    return encodedString;
}
```

Listing 4.2: Encode function in golomb.h

Our decode function is responsible for decoding a Golomb-encoded string back into the original numbers based on the provided parameters. This decoding process involves handling multiple m values from the provided m_vector. The function iterates through the Golomb-encoded string, decoding the values based on different 'm' parameters. When the blockSize, specified in the arguments, is reached, it marks the end of a block of encoded values. After this, it increments the index to move to the next m value and resets the block counter (count). Then, the getBits() function is called again to recalculate the b and stopper values for the updated m value. This ensures that the decoding process adapts to different m values, recalculating the encoding parameters as needed at each block to accurately decode the Golomb-encoded string.

For the actual decoding, our decode function, similarly to the encode, starts by calling the getBits(m) method to calculate the number of bits 'b' required for encoding and the stopper value based on the first m value in the m_vector, counts consecutive '0's in the encoded string, representing the quotient (encoded in unary code), and then skips the '1' bit, separating the quotient from the remainder. If the 'm' value is not 1 (or forced as 1 to avoid divisions by 0 when

m is 0), it iterates through the remainder part. While the index of the remainder doesn't exceed the minimum bits, it concatenates the bits to a temporary string, to form the remainder (in binary). It then calculates the actual remainder based on whether the this binary value of the remainder is smaller than the *stopper*. If so, the remainder itself is obtained; otherwise, it's computed as the difference between the obtained remainder and the *stopper*.

$$(res = r + 2^b - m) \equiv (r = res - (2^b - m)) \equiv (r = res - stopper)$$

When 'm' is 1, the remainder is always 0. In the end, after getting the value of the remainder, it checks for the sign bit at the end to determine the sign of the number ('0' for positive, '1' for negative).

This function essentially reverses the encoding steps, extracting the quotient and remainder parts, and recalculating the remainders as needed to reconstruct the original values from the Golomb-encoded string.

```
vector<int> decode(string encodedString, vector<int>
    m_vector, int blockSize) {
    vector<int> decodedValues;
    int i = 0, m_i = 0, count = 0;    //current index,
    current m index, blockSize counter
    getBits(m_vector[m_i]);
    while(i < encodedString.length()) {
            int q = 0;
            while (encodedString[i] == '0') { //get the
    count of quotient ( unary code )
                q++;
                i++;    // next bit
            }

            i++;  //skip the '1'
            int r = 0;

            string aux = "";

            if (m_vector[m_i] != 1){      // m!=1

                for (int j = 0; j < b-1; j++){  //get the
    count of remainder ( binary code )
                    aux += encodedString[i];
                    i++;
                }

                int res = 0;
                for (int i = 0; i < aux.length(); i++)
                  res = res * 2 + (aux[i] - '0');   //
    convert the remainder to int


                if (res < stopper) {
```

```
30                      r = res;    // binary codeword of b-1
    bits
31                  } else {
32                      aux += encodedString[i];
33                      i++;    // next bit -> binary codeword
    of b bits
34
35                      int res = 0;
36                      for (int i = 0; i < aux.length(); i++)
37                        res = res * 2 + (aux[i] - '0'); //
    convert the remainder to int
38
39                      r = res - stopper;    //res = r + (2^b)-
    m
40                  }
41              } else {                        // m = 1
42                  r = 0;
43                  i++;    // next bit
44              }
45              // result value without sign
46              int result = q * m_vector[m_i] + r;
47
48              // if encoded string has a 1 in the end of the
    remainder, the result is negative, otherwise it's
    positive
49              if (encodedString[i] == '1') {
50                  result = -result;
51              }
52
53              decodedValues.push_back(result);
54              i++;    // next bit
55
56              //blockSize--;
57              //if (blockSize == 0) {
58              count++;
59              if (count == blockSize) {
60                  m_i++;    // next m
61                  // reset blocksize
62                  count = 0;
63                  //blockSize = blockSize;
64                  getBits(m_vector[m_i]);
65              }
66          }
67      return decodedValues;
68  }
```

Listing 4.3: Decode function in golomb.h

Chapter 5

Exercises 4 & 5

Exercise 4 focuses on the implementation of a lossless audio codec based on Golomb coding of prediction residuals. The main goal is to develop a codec capable of handling both mono and stereo audio files. The codec incorporates temporal prediction for mono audio and extends to inter-channel prediction for stereo audio. The efficiency of Golomb coding depends on a parameter, m, which can be either fixed and directly specified to the encoder or, preferably, be adaptively and optimally determined during encoding/decoding.

Exercise 5 enhances the codec by including an option for lossy audio coding. This feature is controlled by an average bitrate target provided by the user, which the encoder should meet. This exercise adds the capability for controlled lossy compression while maintaining flexibility in bitrate settings.

5.1 Usage Instructions

To use the audio codec program, follow these steps:

1. Compile the encoder and decoder programs by navigating to the build directory and running the following commands:

```
cd path/to/your/build/directory
make

```

   Ensure that you have the necessary dependencies installed, including the FFTW library and the SNDFILE library.

2. To execute the audio encoder, use the following command:

```
./audio_encoder ../sample.wav encoded-sample.bin
 -bs <block_size> -br <bitrate>

```

Replace `<block_size>` with the desired block size and `<bitrate>` with the desired average bitrate target. This command encodes the input audio file (`sample.wav`) using the specified block size and average bitrate target, producing the encoded output file (`encoded-sample.bin`). The number of channels in the input audio file is automatically detected.

3. To execute the audio decoder, use the following command:

```
./audio_decoder encoded-sample.bin decoded_sample.wav

```

This command decodes the encoded file `encoded-sample.bin` and generates the decoded audio file `decoded-sample.wav`.

5.2 Code

5.2.1 Audio Encoder (`audio_encoder.cpp`)

The audio encoder program implements a lossless audio codec based on Golomb coding of prediction residuals. Key components and the approach include:

- **Prediction and Golomb Parameter Calculation:**
  - Utilizes `predict(a, b, c)` for sample estimation ($x_n = 3 \cdot x_{n-1} - 3 \cdot x_{n-2} + x_{n-3}$).
  - Calculates Golomb parameter (m) using `calc_m(p, bitrateTarget)` based on average absolute prediction error (p), where `bitrateTarget` is an optional parameter representing the average bitrate target for lossy audio coding. If not provided, the default bitrate value obtained from the command line is used.

```
    // function to calculate m based on p and bitrate
    target
    auto calc_m = [](int p, double bitrateTarget = 0.0)
    {
        if (bitrateTarget == 0.0) {
            // If bitrate target is not provided, use
    the default value obtained from the command line.
            bitrateTarget = br;
        }

        // Convert bitrate target to an alpha factor
        double alpha = bitrateTarget / (1.0 +
    bitrateTarget);

        // Calculate m based on p and alpha
```

```
        return static_cast<int>(-1.0 / log(static_cast<double>(p) / (1.0 + static_cast<double>(p) + alpha)));
    };

    // prediction function
    auto predict = [](int a, int b, int c) {
        // x(n) = 3*x(n-1) - 3*x(n-2) + x(n-3)
        return 3*a - 3*b + c;
    };

```

Listing 5.1: Prediction and Golomb Parameter Calculation Functions

- **Command-Line Arguments:**
  - Accepts input audio file (`<input file>`), output encoded file (`<output file>`), block size (`-bs`), and bitrate (`-br`).
  - Default values provided for block size (1024) and bitrate (0).

```
// default values
size_t bs{1024};
int br{0};

// check for user-specified block size and bitrate
for (int n = 1; n < argc; n++)
    if (string(argv[n]) == "-bs") {
        bs = atoi(argv[n + 1]);
        break;
    }

for (int n = 1; n < argc; n++)
    if (string(argv[n]) == "-br") {
        br = atof(argv[n + 1]);
        break;
    }

```

Listing 5.2: Command-Line Argument Handling

- **Input File Validation:**
  - Validates input audio file in WAV format and PCM_16 format.

```
SndfileHandle sfhIn { argv[1] };

// validate input file
if(sfhIn.error()) {
    cerr << "Error: invalid input file\n";
    return 1;
```

```
}

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

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

```

Listing 5.3: Input File Validation

- **Processing Blocks:**
  - Processes input audio in user-specified (or default 1024) block size.
  - Displays error if input is too short for processing.

```
// start a timer
clock_t start = clock();

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

// Ensure nFrames is a multiple of bs
nFrames = (nFrames / bs) * bs;

if (nFrames < bs) {
    cerr << "Error: Input file is too short for
    processing\n";
    return 1;
}

// ...
// Sample reading and zero padding code
// ...

// end the timer
clock_t end = clock();
double elapsed_secs = double(end - start) /
    CLOCKS_PER_SEC;
elapsed_secs = elapsed_secs * 1000;
cout << "Time: " << elapsed_secs << " ms" << endl;

```

Listing 5.4: Processing Blocks

- **Stereo and Mono Handling:**
  - For stereo audio, separates samples into left and right channels, and calculates the predictions for each sample of each channel.
  - For mono audio, calculates the prediction for each sample using the prediction function.

```
    vector<short> left(samples.size() / 2);
    vector<short> right(samples.size() / 2);
    if (nChannels > 1) {    //if stereo
        for (int i = 0; i < samples.size()/2; i++) {
            left[i] = samples[i * nChannels];
            right[i] = samples[i * nChannels + 1];
        } // get samples of each channel
    }
    vector<int> m_vector;
    vector<int> valuesToBeEncoded;
if (nChannels < 2) {  //mono audio
        for(int i = 0; i < samples.size(); i++) {
            if (i >= 3) {
                int error = samples[i] - predict(samples
   [i-1], samples[i-2], samples[i-3]);    //x(n) = 3*x(
   n-1) - 3*x(n-2) + x(n-3)
                valuesToBeEncoded.push_back(error);
            } else {
                valuesToBeEncoded.push_back(samples[i]);
            }
        }
    } else {    //stereo audio
        std::vector<std::vector<short>> channels = {left
   , right}; // Store both channels
        cout << "Number of channels: " << channels.size
   () << endl;
        for (int channel = 0; channel < channels.size();
    ++channel) {   // first for one channel, then for
   the other
            for (int i = 0; i < channels[channel].size()
   ; ++i) {
                if (i >= 3) {
                    int error = channels[channel][i] -
   predict(channels[channel][i-1], channels[channel][i
   -2], channels[channel][i-3]);   //x(n) = 3*x(n-1) -
   3*x(n-2) + x(n-3)
                    valuesToBeEncoded.push_back(error);
                } else {
                    valuesToBeEncoded.push_back(channels
   [channel][i]);
                }

```

Listing 5.5: Stereo and Mono Handling

- **Golomb Parameter Adaptation:**
  - Calculates Golomb parameter (m) every bs samples, where bs is the block size.
  - Determines Golomb parameter based on average absolute prediction error (p) for each block.

```
// Calculate 'm' every bs samples
if (i % bs == 0 && i != 0) {
    int sum = 0;
    for (int j = i - bs; j < i; ++j) {
        sum += abs(valuesToBeEncoded[j]);
    }
    int p = round(sum / bs);
    int m = calc_m(p);
    if (m < 1) m = 1;
    m_vector.push_back(m);
}
if (i == channels[channel].size() - 1) {
    int sum = 0;
    for (int j = i - (i % bs); j < i; ++j) {
        sum += abs(valuesToBeEncoded[j]);
    }
    int p = round(sum / (i % bs));
    int m = calc_m(p);
    if (m < 1) m = 1;
    m_vector.push_back(m);
}
```

Listing 5.6: Golomb Parameter Adaptation

- **Encoding with Golomb Coding:**
  - Encodes prediction errors using Golomb coding with the calculated Golomb parameter.
  - Constructs the encoded string for further processing.

```
    string encodedString = "";
    Golomb g;


    int m_index = 0;
    for (int i = 0; i < valuesToBeEncoded.size(); i++) {
        if (i % bs == 0 && i != 0) m_index++;
        encodedString += g.encode(valuesToBeEncoded[i],
    m_vector[m_index]);
    }


```

```
12      BitStream bitStream(output, 1);
13
14      bitStream.writeNBits(sfhIn.channels(), 16); // Write
         the number of channels (16 bits)
15      bitStream.writeNBits(samples.size()/2, 32); // Write
         the number of frames (32 bits)
16      bitStream.writeNBits(bs, 16); // Write the block
        size (16 bits)
17      bitStream.writeNBits(encodedString.size(), 32); //
        Write encodedString size (32 bits)
18      bitStream.writeNBits(m_vector.size(), 16); // Write
        m_vector size (16 bits)
19
20
21      // Write m_vector values converted to binary (16
        bits each)
22      for (int i = 0; i < m_vector.size(); i++) {
23          bitStream.writeNBits(m_vector[i], 16);
24      }
25
26      vector<int> encoded_bits;
27
28      //the next bits will be the encoded string
29      for(int i = 0; i < encodedString.length(); i++)
30          encoded_bits.push_back(encodedString[i] - '0');
31
32
33      //the next bits are the encoded bits
34      for (int i = 0; i < encoded_bits.size(); i++)
35          bitStream.writeBit(encoded_bits[i]);
36
37
38      bitStream.close();
39
```

Listing 5.7: Encoding with Golomb Coding

- **Output:**
  - Writes encoded data, including audio information (number of channels and frames) and Golomb parameters, to the output binary file.
  - Provides timing information for execution time.

```
1       cout << "Number of channels: " << channels.size() <<
         endl;
2   // ...
3       cout << "Time: " << elapsed_secs << " ms" << endl;
4   // ...
5
```

Listing 5.8: Output Writing and Timing Information

5.2.2 Audio Decoder (`audio_decoder.cpp`)

The audio decoder decodes Golomb-encoded audio data produced by the audio encoder. Key components and the approach include:

- **Prediction Function:**
  - Uses `predict(a, b, c)` for sample estimation ($x_n = 3 \cdot x_{n-1} - 3 \cdot x_{n-2} + x_{n-3}$).

```
auto predict = [](int a, int b, int c) {
    // x(n) = 3*x(n-1) - 3*x(n-2) + x(n-3)
    return 3 * a - 3 * b + c;
};
```

Listing 5.9: Prediction Function

- **Command-Line Arguments:**
  - Accepts input encoded file (`<input file>`) and output decoded file (`<output file>`).
  - Reads additional parameters (number of channels, frames, block size, encoded data size, Golomb parameter vector size) from the encoded file.

```
int sampleRate = 44100;
BitStream bs(argv[1], 0);

int nChannels = bs.readNBits(16);
int nFrames = bs.readNBits(32);
int blockSize = bs.readNBits(16);
int enc = bs.readNBits(32);
int m_size = bs.readNBits(16);

vector<int> m_vector;
for (int i = 0; i < m_size; i++) {
    int m = bs.readNBits(16); // Read 16 bits for m
    m_vector.push_back(m);
}
```

Listing 5.10: Initialization and Golomb Parameter Reading

- **Initialization:**
  - Reads Golomb parameters (m) from the encoded file and stores them in the vector (`m_vector`).
  - Initializes the output WAV file with the specified number of channels and sample rate.

- **Decoding Golomb-Encoded Data:**
  - Reads Golomb-encoded data from the input file and converts it to a binary string (`encodedString`).
  - Applies the Golomb decoding function to obtain decoded values using Golomb parameters and block size.

```
vector<int> encoded_values;
for (int i = 0; i < enc; i++) {
    encoded_values.push_back(bs.readBit());
}
string encodedString = "";
for (int i = 0; i < encoded_values.size(); i++) {
    encodedString += to_string(encoded_values[i]);
}
Golomb g;
vector<int> decoded = g.decode(encodedString, m_vector,
    blockSize);
```

Listing 5.11: Decoding Golomb-Encoded Data

- **Mono and Stereo Reconstruction:**
  - For mono audio, uses decoded values to reconstruct samples based on the prediction function.
  - For stereo audio, performs separate reconstruction for left and right channels, considering the block size and Golomb parameters.

```
vector<short> samplesVector;
    if (nChannels < 2) {   //mono
        for ( int i = 0; i < decoded.size(); i++) {
            if (i >= 3) {
                //create a new sample, using the 3 previous
    samples
                int sample = decoded[i] + predict(
    samplesVector[i-1], samplesVector[i-2], samplesVector[i
    -3]);
                samplesVector.push_back(sample);
            }
            else samplesVector.push_back(decoded[i]);
        }
    } else {    // stereo
        for (int i = 0; i < nFrames; i++) {   //left channel
            if (i >= 3) {
                //create a new sample, using the 3 previous
    samples
                int sample = decoded[i] + predict(
    samplesVector[i-1], samplesVector[i-2], samplesVector[i
    -3]);
```

```
16                  samplesVector.push_back(sample);
17              }
18              //create a new sample with the decoded value at
    the current sample index
19              else samplesVector.push_back(decoded[i]);
20          }
21          for ( int i = nFrames; i < decoded.size(); i++) {
    //right channel
22              if (i >= nFrames + 3) {
23                  //create a new sample, using the 3 previous
    samples
24                  int sample = decoded[i] + predict(
    samplesVector[i-1], samplesVector[i-2], samplesVector[i
    -3]);
25                  samplesVector.push_back(sample);
26              }
27              //create a new sample with the decoded value at
    the current sample index
28              else samplesVector.push_back(decoded[i]);
29          }
30          vector<short> merged;
31          vector<short> firstChannel(samplesVector.begin(),
    samplesVector.begin() + nFrames); // first nFrames
    samples -> left channel
32          vector<short> secondChannel(samplesVector.begin() +
    nFrames, samplesVector.end()); //second nFrames samples
    -> right channel
33          for(int i = 0; i < nFrames; i++) {
34              merged.push_back(firstChannel[i]);
35              merged.push_back(secondChannel[i]);
36          }
37          samplesVector = merged;
38      }
```

Listing 5.12: Mono and Stereo Reconstruction

- **WAV File Output:**
  - Writes the reconstructed samples to the output WAV file.
- **Timing Information:**
  - Provides timing information for the decoding process.

5.3 Results

The developed audio codec was tested with a stereo audio file (`sample.wav`) using various parameters for encoding. The following commands were used for encoding:

```
./audio_encoder ../sample.wav encoded_sample_512_256.bin -bs 512 -br 256
./audio_encoder ../sample.wav encoded_sample_2048_750.bin -bs 2048 -br 750
./audio_encoder ../sample.wav encoded_sample_4096_1000.bin -bs 4096 -br 1000
```

The obtained results for encoding with different block sizes and bitrates are as follows:

- **Encoding Time (512, 256):** Approximately 1365.42 milliseconds.
- **Encoding Time (2048, 750):** Approximately 1316.96 milliseconds.
- **Encoding Time (4096, 1000):** Approximately 1367.38 milliseconds.
- **Number of Channels:** The encoder consistently detected 2 channels in the input audio file, confirming its ability to handle stereo audio.
- **Decoding Time (512, 256):** Approximately 1274.25 milliseconds.
- **Decoding Time (2048, 750):** Approximately 1272.45 milliseconds.
- **Decoding Time (4096, 1000):** Approximately 1292.27 milliseconds.
- **Quality Assessment:** The decoded audio files (`sample_512_256.wav`, `sample_2048_750.wav`, `sample_4096_1000.wav`) were compared with the original sample (`sample.wav`). The quality of the decoded audio is observed to be similar to the original for all configurations, indicating that the codec maintains audio fidelity during the encoding and decoding process.
- **Audio Quality and Bitrate:** Higher bitrates generally result in better audio quality, while lower bitrates may lead to a perceptible decrease in quality. The user should carefully choose the bitrate based on the desired trade-off between file size and audio fidelity.
- **Conclusions:** The choice of block size and bitrate has an impact on the encoding and decoding times. As the block size increases, the encoding time tends to increase, but it might result in better compression efficiency. Similarly, adjusting the bitrate target influences both encoding and decoding times.

In conclusion, the developed audio codec demonstrates the ability to encode and decode stereo audio files with satisfactory performance across different block sizes and bitrates. The quality assessment suggests that the codec preserves the audio content, making it suitable for lossless audio compression applications.