Universidade de Aveiro

Information and Coding

Lab Work nr.2

André Clérigo (98485), João Amaral (98373), Pedro Rocha (98256)

Departamento de Eletrónica, Telecomunicações e Informática

December 4th, 2022

Contents

List of Figures

Chapter 1

Exercise 1

1.1 Code

We started by creating a new image with all pixels set to zero, then we calculate the number of channels, rows and columns to check if the image is continuous.

After that we loop through the rows and columns of the original image and copy each pixel value to the new image matrix created.

```
Mat img = imread(argv[1]);
Mat new_image = Mat::zeros(img.size(), img.type());
int channels = img.channels();
int nRows = img.rows;
int nCols = img.cols * channels;

if (img.isContinuous()){
    nCols *= nRows;
    nRows = 1;
}

uchar* pixel;
for (int i = 0; i < nRows; i++){
    pixel = img.ptr<uchar>(i);
    for (int j = 0; j < nCols; j++){
        new_image.at<uchar>(i,j) = pixel[j];
    }
}

imwrite(argv[2], new_image);
```

1.2 Usage

The usage of the program is done following this pattern:

```
./cp_image <input file> <output file> [view]
```

Where the input file is an image file (tested with .ppm).

1.3 Results

Testing image_files/lena.ppm and using the "view" argument, we got the following result:

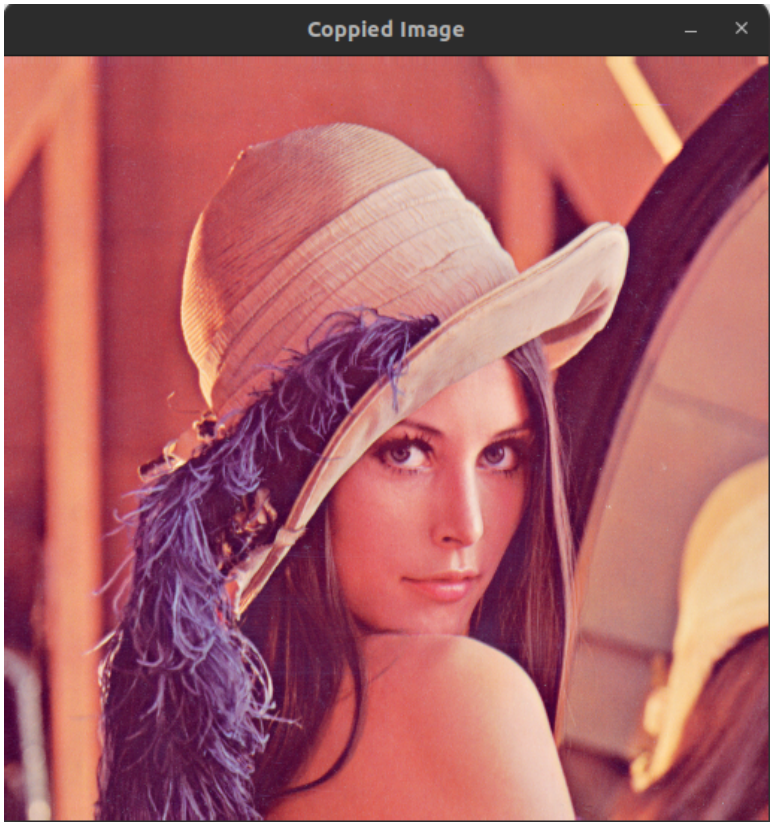

Figure 1.1: Copy of lena.ppm

Chapter 2

Exercise 2

2.1 a)

2.1.1 Code

We start by calculating the number of channels, rows and columns to check if the image is continuous. After that, we loop through the rows and columns of the image and change each pixel value to: 255 - current value.

```
Mat img = imread(argv[1]);
int nRows = img.rows;
int nCols = img.cols * img.channels();

if (img.isContinuous()) {
    nCols *= img.rows;
    nRows = 1;
}

for (int i = 0; i < nRows; i++) {
    for (int j = 0; j < nCols; j++) {
        img.at<uchar>(i, j) = 255 - img.at<uchar>(i, j);
    }
}

imwrite(argv[2], img);
```

2.1.2 Usage

The usage of the program is done following this pattern:

```
./negative_image <input file> <output file> [view]
```

Where the input file is an image file (tested with .ppm).

2.1.3 Results

Testing image_files/monarch.ppm and using the "view" argument, we got the following result:

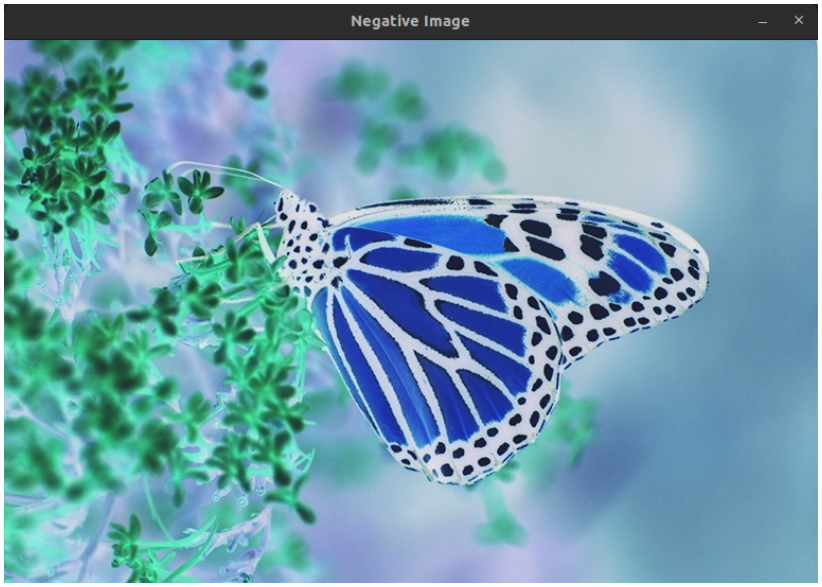

Figure 2.1: Negative image of monarch.ppm

2.2 b)

2.2.1 Code

We started by creating a new matrix called mirror, after that we set the axis to mirror. Finally, we iterate through the original image pixels and copy it's value to the corresponding position on the mirrored matrix.

```
1  Mat img = imread(argv[1]);
2  Mat mirror = Mat::zeros(img.size(), img.type());
3  int direction = 0;
4  int nRows = img.rows;
5  int nCols = img.cols;
6
7  if (string(argv[3]) == "h") {
8      direction = 1;
9  } else if (string(argv[3]) == "v") {
10     direction = 0;
11 } ...
12
13 for (int i = 0; i < nRows; i++) {
14     for (int j = 0; j < nCols; j++) {
15         if (direction) {
16             mirror.at<Vec3b>(nRows-1-i, j) = img.at<Vec3b>(i,j);
17         } else {
18             mirror.at<Vec3b>(i, nCols-1-j) = img.at<Vec3b>(i,j);
19         }
20     }
21 }
22 imwrite(argv[2], mirror);
```

2.2.2 Usage

The usage of the program is done following this pattern:

```
./mirror_image <input file> <output file> <h | v> [view]
```

Where the input file is an image file (tested with .ppm).

2.2.3 Results

Testing image_files/airplane.ppm, choosing the horizontal axis, and using the "view" argument, we got the following result:

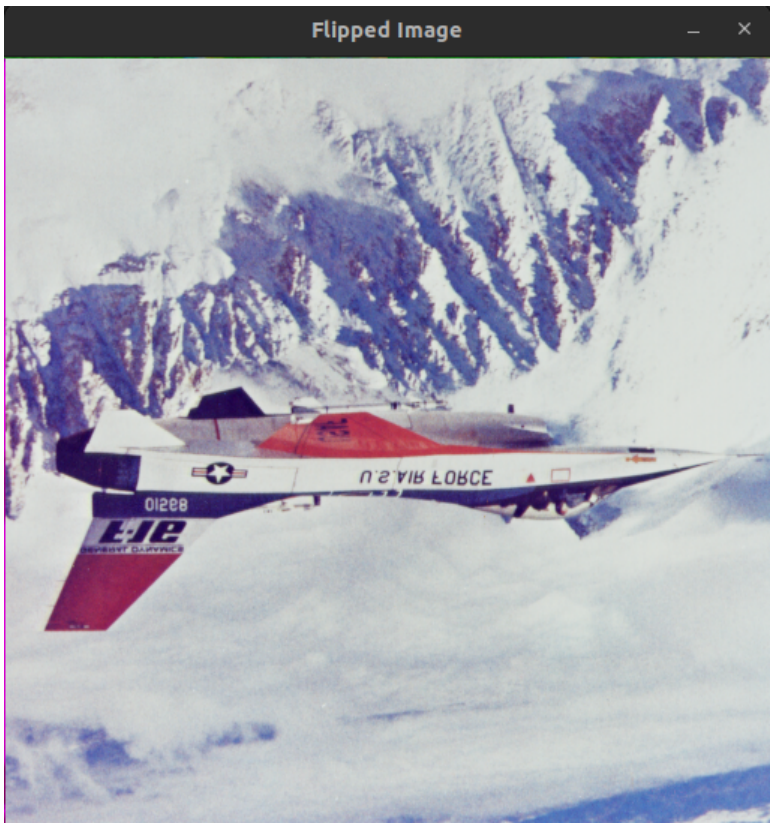

Figure 2.2: Mirrored image on the horizontal axis of airplane.ppm

Testing image_files/airplane.ppm, choosing the vertical axis, and using the "view" argument, we got the following result:

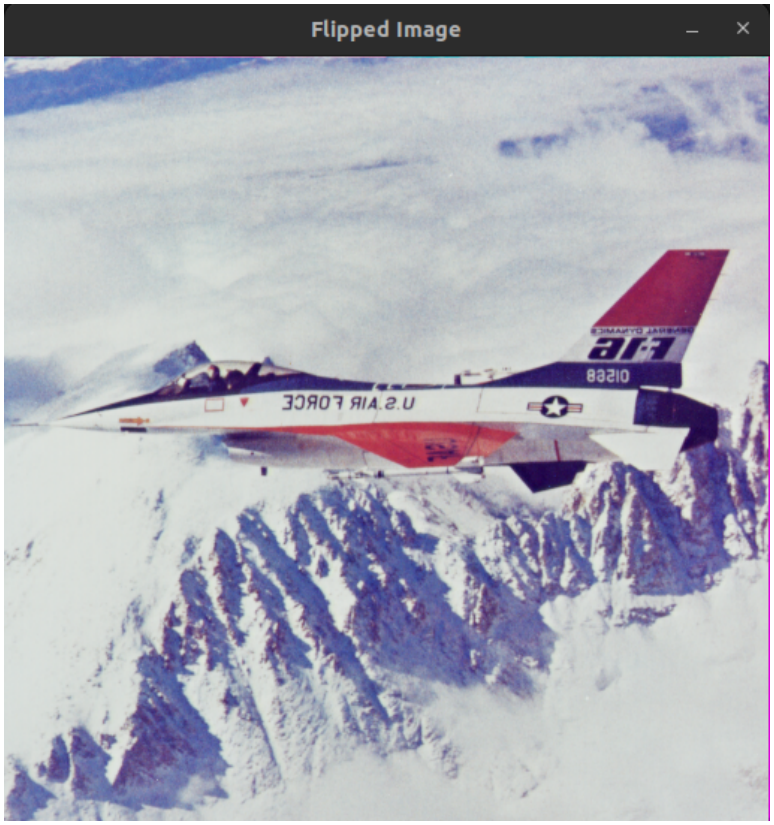

Figure 2.3: Mirrored image on the vertical axis of airplane.ppm

2.3 c)

2.3.1 Code

We started by checking if the angle to rotate is a multiple of 90 degrees, after that we set the correct image dimensions according to the rotation value, then we iterate through the original image and copy the pixel value to the corresponding pixel in the new matrix.

```
1  Mat img = imread(argv[1]);
2  int degrees_to_rotate = atoi(argv[3]);
3  int rotatedRows = degrees_to_rotate % 180 ? img.size().width :
       img.size().height;
4  int rotatedCols = degrees_to_rotate % 180 ? img.size().height :
       img.size().width;
5  Mat rotated_image = Mat::zeros(rotatedRows, rotatedCols, img.
       type());
6  int nRows = img.rows;
7  int nCols = img.cols;
8
9  for (int i = 0; i < nRows; i++) {
10     for (int j = 0; j < nCols; j++) {
```

```
11          if (degrees_to_rotate % 360 == 0) {
12              rotated_image.at<Vec3b>(i, j) = img.at<Vec3b>(i, j);
13          else if (degrees_to_rotate % 360 == 270) {
14              rotated_image.at<Vec3b>(j, nRows - 1 - i) = img.at<
    Vec3b>(i, j);
15          } else if (degrees_to_rotate % 360 == 180) {
16              rotated_image.at<Vec3b>(nRows - 1 - i, nCols - 1 - j
    ) = img.at<Vec3b>(i, j);
17          } else if (degrees_to_rotate % 360 == 90) {
18              rotated_image.at<Vec3b>(nCols - 1 - j, i) = img.at<
    Vec3b>(i, j);
19          }
20      }
21  }
22  imwrite(argv[2], rotated_image);
```

2.3.2 Usage

The usage of the program is done following this pattern:

```
1 ./rotate_image <input file> <output file> <degrees_to_rotate> [
    view]
```

Where the input file is an image file (tested with .ppm).

2.3.3 Results

By testing with the image inside image_files/airplane.ppm, 90 degrees of rotation and the view argument we got the following result:

Figure 2.4: Rotated image 180 degrees of image arial.ppm

Figure 2.5: Rotated image 90 degrees of image arial.ppm

2.4 d)

2.4.1 Code

We started by checking if the brightness values is between -255 and 255, after that we sum the desired value in each pixel channel.

```
Mat img = imread(argv[1]);
int b = 0;
b = stoi(argv[3]);
Mat output = Mat::zeros(img.size(), img.type());
int nRows = img.rows;
int nCols = img.cols;

for (int i = 0; i < nRows; i++) {
    for (int j = 0; j < nCols; j++) {
        for (int c = 0; c < img.channels(); c++)
            output.at<Vec3b>(i, j)[c] = saturate_cast<uchar>(img
    .at<Vec3b>(i, j)[c] + b);
    }
}
imwrite(argv[2], output);
```

2.4.2 Usage

The usage of the program is done following this pattern:

```
./bright_image <input file> <output file> <num> [view]
```

Where the input file is an image file (tested with .ppm).

2.4.3 Results

By testing with the image inside image_files/bike3.ppm, 50 of pixel value and the view argument we got the following result:

Figure 2.6: Brightness increased of image bike3.ppm

By testing with the image inside image_files/bike3.ppm, -50 of pixel value and the view argument we got the following result:

Figure 2.7: Brightness decreased of image bike3.ppm

Chapter 3

Exercise 3

3.1 Contextualization

Golomb coding is a data compression method using several data compression codes. Alphabets following a geometric distribution will have a Golomb code as an optimal prefix code, making Golomb coding highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than large values. With that in mind and with the purpose of representing numbers with this coding, the procedures are: making a whole division of the number you want to represent, N, by M (the divisor that's adaptable, or not, depending on the implementation), we'll get a quotient and a remainder.
The quotient, q, will be represented in a q-length string of bits set as 0, in unary coding, followed by a bit set as 1 to separate the quotient from the remainder.
This remainder, r, will be represented in truncated binary coding, using $b = \lfloor (log2(m)) \rfloor$.

If $r < 2^{b+1} - m$, code r in binary representation using b bits.

If $r \geq 2^{b+1} - m$, code the number $r + 2^{b+1} - M$ in binary representation using b + 1 bits.

3.2 Code

After several improvements on our implementation, we ended up with the solution explained bellow.

3.2.1 Private part of the class

Firstly, we've got the private section of our class that includes several functions, being the following the most relevant of this part.

calculateBits()

This function has the purpose of calculating the maximum number of bits of the remainder given by the following theoretical expression:

$$b = \lceil (log2(m)) \rceil$$

And, also, calculate the number of values in the remainders table that have the minimum of bits (given by subtracting 1 bit to the maximum bits):

$$NumberOfValuesWithMinBits = 2^k - m$$

with k being the maximum number of bits;

```
void calculateBits(int m) {
    if (m != 0){
        max_bits = ceil(log2(m));    // log2(m) with ceiling to
    integer
        min_bits = max_bits - 1;     // max_bits - 1
        n_values_with_min_bits = pow(2, max_bits) - m; // (2^
    max_bits) - m
    } else {
        max_bits = 0;
        min_bits = 0;
        n_values_with_min_bits = 0;
    }
}
```

3.2.2 Public part of the class

Secondly, when addressing the public part of the class, we can detach the 3 main functions: encode, decode and decodeMultiple.
The encoding process gets the bits calculated for the given m and, like previously explained in the contextualization, the values of maximum and minimum bits for the remainder and the number of values of remainders with the minimum bits. If the m isn't zero, calculate the whole division of the number that's pretended to be represented by m and keep the remainder, concatenating to the result bit string the number of 0 bits equal to the quotient (unary code) and a 1 bit at the end to separate the quotient from the remainder. If the m is zero, this first step is skipped and it's concatenated only the bit set as 1, since you can't divide numbers by zero.
When it comes to the remainder encoding, if the m isn't 1: the remainder is expressible with the minimum bits and it is the literal value (in binary code), it just needs to be converted to bits, if the remainder has a value smaller than the number of values with minimum bits; otherwise - which means it has a higher value - it's expressed with the maximum bits and it's the sum of the remainder and the number of values with minimum bits (binary code). If the m is 1, the remainder will always be zero, therefore it

only concatenates a 0 bit.
Finally, it adds a 0 bit or a 1 bit at the end of the bit string in order to identify if it is a positive or negative number, respectively.

encode()

```
std::string encode(int num, int m){
    calculateBits(m);
    std::string result = "";
    int quotient = 0;
    int remainder = 0;
    // if m isn't 0, calculate the quotient and the remainder
    with golomb coding
    if (m != 0){
        quotient = abs(num) / m;
        remainder = abs(num) % m;
    }
    // concatenate the quotient in unary code
    for (int i = 0; i < quotient; i++)
        result += "0";

    // use a bit (1) to represent the end of the quotient and
    the begginning of the remainder
    result += "1";

    // if m is 1, the remainder is 0, otherwise calculate the
    remainder in binary code
    if (m != 1){
        if (remainder < n_values_with_min_bits) {
            result += remaindersBitString(remainder, min_bits);
        } else {
            result += remaindersBitString(remainder +
    n_values_with_min_bits, max_bits);
        }
    }else result += "0";

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

    return result;
}
```

Moreover, the decoding process is subdivided in two types of decoding, one when the this process is done with m as a fixed value or a dynamic value updated every blocksize.
When it comes to the decoding process using just one m value for all the file, we use the function below. This function starts, like the encode(), to calculate the value of maximum and minimum bits and the number of values with minimum bits of the remainder, with the given m.
Iterating thought the encoded string, we can count the number of zeros that

will represent the quotient (because it was in unary code) and then we skip the 1 bit separating the quotient from the remainder. After this, we'll have a variable (j) to index the remainder bits, besides the index variable of the all encoded string (i).
Iterating through the part of the remainder, if the m isn't 1 (or 0, but it's forced as 1 to avoid divisions by 0), the bit will be concatenated to a temporary string while the index of the remainder doesn't exceed the minimum bits. If the integer value of the remainder, that was in binary code, is smaller than number of values with minimum bits, it's the remainder itself, however, it that doesn't verify, the remainder is given by the difference of the remainder obtained and the number of values with minimum bits. Otherwise, when m is 1, the remainder is 0 and there's no need for calculations. Lastly, after the remainder is obtained, we verify if the string has a 0 or a 1 at the end to get the signal of the number (positive or negative, respectively).

decode()

```
//decode function for a fixed M value
std::vector<int> decode(std::string encoded_string, int m) {
    calculateBits(m);
    std::vector<int> result;
    // bit position in the encoded string
    int i = 0;

    // loop through the encoded string
    while((long unsigned int) i < encoded_string.length()) {
        int quotient = 0;
        // count the number of 0s in the encoded string to get the quotient (written in unary code)
        while (encoded_string[i] == '0') {
            quotient++;
            // next bit
            i++;
        }
        // skip a bit (the 1 in the unary code, that represents the end of the quotient)
        i++;
        // initialize the remainder
        int remainder = 0;
        // counter for the number of bits read from the encoded string in the remainder part (binary code)
        int j = 0;
        // temporary string to store the remainder
        std::string tmp = "";

        // if m is 1, the remainder is 0
        // besides that if m is 0, it's forced as 1 (to avoid division by 0)
        if (m != 1){
            // while the number of bits of the remainder doesn't
```

```
     reach min_bits
30           while (j < min_bits) {
31               // add the bit to the temporary string (to get
    the remainder)
32               tmp += encoded_string[i];
33               // next bit
34               i++;
35               // next bit of the remainder part (binary code)
36               j++;
37           }
38
39           // convert the temporary string to integer to get
    the remainder
40           int res1 = bitStringToInt(tmp);
41
42           // if the remainder has a value that is greater than
     the number of values with min_bits (which corresponds to the
     max value with min_bits)
43           // the next bit must be read as part of the
    remainder
44           // if the value is smaller, the remainder is the
    value read so far
45           if (res1 < n_values_with_min_bits) {
46               remainder = res1;
47           } else {
48               tmp += encoded_string[i];
49               i++;
50               // convert the temporary string to integer to
    get the remainder
51               remainder = bitStringToInt(tmp) -
    n_values_with_min_bits;
52           }
53       } else {
54           remainder = 0;
55           i++;
56       }
57
58       // result value without sign
59       int res = quotient * m + remainder;
60
61       // if the encoded string has a 1 in the end of the
    remainder, the result is negative, otherwise it's positive
62       if (encoded_string[i] == '1')
63           result.push_back(-(res));
64       else
65           result.push_back(res);
66       i++;
67   }
68   return result;
69 }
```

Finally, the decodeMultiple() function follows the idea of the decode() function, having in mind that it has several m values that should be reevaluated (calling calculateBits() function again) every time the indicated block-

size is reached, resetting its bit counter (count).

decodeMultiple()

```
//decode function for dynamic Ms (changing Ms)
std::vector<int> decodeMultiple(std::string encoded_string, std
    ::vector<int> m_vector, int block_size) {
    std::vector<int> result;
    // bit position in the encoded string
    int i = 0;
    // m index on the vector for the current block
    int m_i = 0;
    // blocksize bits counter
    int count = 0;
    calculateBits(m_vector[m_i]);
    while((long unsigned int) i < encoded_string.length()) {
            int quotient = 0;
            // count the number of 0s in the encoded string to
    get the quotient (written in unary code)
            while (encoded_string[i] == '0') {
                quotient++;
                // next bit
                i++;
            }
            // skip a bit (the 1 in the unary code, that
    represents the end of the quotient)
            i++;
            int remainder = 0;
            // counter for the number of bits read from the
    encoded string in the remainder part (binary code)
            int j = 0;
            std::string tmp = "";

            // if m is 1, the remainder is 0;
            // besides that if m is 0, it's forced as 1 (to
    avoid division by 0)
            if (m_vector[m_i] != 1){
                while (j < min_bits) {
                    tmp += encoded_string[i];
                    // next bit
                    i++;
                    // next bit of the remainder part (binary
    code)
                    j++;
                }

                // convert the temporary string to integer to
    get the remainder
                int res1 = bitStringToInt(tmp);

                // if the remainder has a value that is greater
    than the number of values with min_bits (which corresponds to
     the max value with min_bits)
```

```
41                  // the next bit must be read as part of the
    remainder
42                  // if the value is smaller, the remainder is the
     value read so far
43                  if (res1 < n_values_with_min_bits) {
44                      remainder = res1;
45                  } else {
46                      tmp += encoded_string[i];
47                      // next bit
48                      i++;
49                      remainder = bitStringToInt(tmp) -
    n_values_with_min_bits;
50                  }
51              } else {
52                  remainder = 0;
53                  // next bit
54                  i++;
55              }
56
57              // result value without sign
58              int res = quotient * m_vector[m_i] + remainder;
59
60              // if the encoded string has a 1 in the end of the
    remainder, the result is negative, otherwise it's positive
61              if (encoded_string[i] == '1')
62                  result.push_back(-(res));
63              else
64                  result.push_back(res);
65
66              // next bit
67              i++;
68              count++;
69              // if the block size is reached, the m index is
    incremented
70              if (count == block_size) {
71                  // next m index
72                  m_i++;
73                  // reset blocksize bits counter
74                  count = 0;
75                  calculateBits(m_vector[m_i]);
76              }
77          }
78      return result;
79  }
```

Chapter 4

Exercise 4 & 5

The *lossess audio codec* implemented based on Golomb coding of the prediction residuals is based on the temporal and channel prediction of the values of the samples in a given audio file. The developed program accepts either encodings with a fixed m value or encodings where this m value is adaptive and calculated optimally based on a *blocksize*, which is also an input argument, only if the keyword "auto" is present when the program is called.

4.1 Code : Encoder

Our implementation starts by doing the necessary padding to the audio file so that the adaptive m calculation can be done based on a specific *blocksize*, as well as the separation of the audio to two vectors (each channel), in case the audio file is stereo. However, one of the optional program arguments is the quantization factor (in our case, the number of bits to be discarded) and therefore the quantization of the original audio is done before all the process of calculating the prediction residuals is done.

```
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);

//do quantization if q is passed in as argument
if (quantization)
    for (long unsigned int i = 0; i < samples.size(); i++)
        samples[i] = samples[i] >> q;
```

```
17
18 if (nChannels > 1) {
19     for (long unsigned int i = 0; i < samples.size()/2; i++) {
20         left_samples[i] = samples[i * nChannels];
21         right_samples[i] = samples[i * nChannels + 1];
22     }
23 }
```

After that, the audio is analysed sample by sample and the next value is predicted using the following expression:

$$x_n = 3x_{n-1} - 3x_{n-2} + x_{n-3}$$

```
1 //function that predicts the next value in the sequence based on
      3 previous values
2 auto predict = [](int a, int b, int c) {
3     //3*a - 3*b + c
4     return 3*a - 3*b + c;
5 };
```

The predicted value is then subtracted from the actual value of the sample, which theoretically gives us a smaller value to encode (the residual) using Golomb coding, which results in a smaller representation, compressing the size of the resulting binary file .
The calculation of each value of m adapted to the respective block is done at the same time as the residuals of the predictions are calculated. From block to block of samples, the calculation of the optimal m for the previous block is made by initially averaging the absolute values of the samples in the last block:

$$u = \frac{sum(abs(samples))}{blocksize}$$

```
1
2 vector<int> m_vector;
3 vector<int> valuesToBeEncoded;
4 if (nChannels < 2) {
5     for(long unsigned int i = 0; i < samples.size(); i++) {
6         if (i >= 3) {
7             int difference = samples[i] - predict(samples[i-1],
    samples[i-2], samples[i-3]);
8             valuesToBeEncoded.push_back(difference);
9         } else {
10            valuesToBeEncoded.push_back(samples[i]);
11        }
12    }
13 } else {
14    for (long unsigned int i = 0; i < left_samples.size(); i++)
    {
15        if (i >= 3) {
16            int difference = left_samples[i] - predict(
    left_samples[i-1], left_samples[i-2], left_samples[i-3]);
17            valuesToBeEncoded.push_back(difference);
```

```
18          } else {
19              valuesToBeEncoded.push_back(left_samples[i]);
20          }
21          //calculate m every bs samples
22          if (i % bs == 0 && i != 0) {
23              int sum = 0;
24              for (long unsigned int j = i-bs; j < i; j++) {
25                  sum += abs(valuesToBeEncoded[j]);
26              }
27              int u = round(sum/bs);
28              m = calc_m(u);
29              if (m < 1) m = 1;
30              m_vector.push_back(m);
31          }
32          if (i == left_samples.size() - 1) {
33              int sum = 0;
34              for (long unsigned int j = i - (i % bs); j < i; j++)
    {
35                  sum += abs(valuesToBeEncoded[j]);
36              }
37              int u = round(sum/(i % bs));
38              m = calc_m(u);
39              if (m < 1) m = 1;
40              m_vector.push_back(m);
41          }
42      }
43
44      for(long unsigned int i = 0; i < right_samples.size(); i++){
45          if (i >= 3) {
46              int difference = right_samples[i] - predict(
    right_samples[i-1], right_samples[i-2], right_samples[i-3]);
47              valuesToBeEncoded.push_back(difference);
48          }
49
50          else valuesToBeEncoded.push_back(right_samples[i]);
51
52          //calculate m every bs samples
53          if (i % bs == 0 && i != 0) {
54              int sum = 0;
55              for (long unsigned int j = i-bs; j < i; j++) {
56                  sum += abs(valuesToBeEncoded[j]);
57              }
58              int u = round(sum/bs);
59              m = calc_m(u);
60              if (m < 1) m = 1;
61              m_vector.push_back(m);
62          }
63          if (i == left_samples.size() - 1) {
64              int sum = 0;
65              for (long unsigned int j = i - (i % bs); j < i; j++)
    {
66                  sum += abs(valuesToBeEncoded[j]);
67              }
68              int u = round(sum/(i % bs));
```

```
69              m = calc_m(u);
70              if (m < 1) m = 1;
71              m_vector.push_back(m);
72          }
73      }
74  }
```

From this value, we calculate the optimal m for this given block by doing:

$$\alpha = \frac{u}{1+u}$$

$$m = -\frac{1}{log(\alpha)}$$

Where all m values are stored in an vector for further encoding and decoding.

```
1  //function to calculate m based on u
2      auto calc_m = [](int u) {
3          //u = alpha / 1 - alpha
4          //m = - (1 / log(alpha))
5          return (int) - (1/log((double) u / (1 + u)));
6      };
```

After saving the residuals of the predictions and the optimal values of m, all these residuals are encoded using the respective m of that block of values using the Golomb class created earlier, where all the encoding is stored in a string that only contains 0s and 1s. For consistency reasons, a padding with 0s is done in that string so that its size is a multiple of 8, which corresponds to a byte, given that the minimum unit of writing in a file is a byte.

```
1  string encodedString = "";
2  Golomb g;
3
4  if (!autoMode){
5      for(long unsigned int i = 0; i < valuesToBeEncoded.size() ;
       i++)
6          encodedString += g.encode(valuesToBeEncoded[i], og);
7  } else{
8      int m_index = 0;
9      for (long unsigned int i = 0; i < valuesToBeEncoded.size();
       i++) {
10         if (i % bs == 0 && i != 0) m_index++;
11         encodedString += g.encode(valuesToBeEncoded[i], m_vector
       [m_index]);
12     }
13 }
14
15 vector<int> bits;
16 vector<int> encoded_bits;
17
18 for(long unsigned int i = 0; i < encodedString.length(); i++)
19     encoded_bits.push_back(encodedString[i] - '0');
20
```

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

Writing to a binary file is done using the BitStream class developed earlier. In order to allow the decoding to be carried out by another program, it is necessary to write a header in the file that contains all the necessary information to proceed with the inverse encoding process. For this, the number of channels of the original audio file, the padding done, the number of frames, the *blocksize*, the number of zeros added to the end of the encoded string, the size of the vector containing the textitm, the m values to decode each block and, finally, the string resulting from the Golomb encoding are all written to the binary file.

```
// writing number of channels, the padding, the quantization
    factor, the number of frames, the block size and the the
    number of zeros added to the end of the encoded string (...)

BitStream bitStream(output, "w");

for(int i = NUMBER_NECESSARY(15 or 31); i >= 0; i--)
    bits.push_back((information >> i) & 1);

if (!autoMode){
    m_vector.clear();
    m_vector.push_back(og);
}

for (int i = 15; i >= 0; i--)
    bits.push_back((m_vector.size() >> i) & 1);

for (long unsigned int i = 0; i < m_vector.size(); i++) {
    for (int j = 15; j >= 0; j--) {
        bits.push_back((m_vector[i] >> j) & 1);
    }
}

for (long unsigned int i = 0; i < encoded_bits.size(); i++)
    bits.push_back(encoded_bits[i]);
...

bitStream.writeBits(bits);
bitStream.close();
```

4.2 Code : Decoder

The decoder, in general, implements the inverse behavior of the encoder. In a first phase, the header is read and the values necessary for the implemen-

tation of the inverse algorithm are extracted:

```
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_nFrames = 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);
```

After reading the header, all values m are read in binary form and, consequently, converted to the respective integer, and stored in a vector to be used later to decode the residuals.

```
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);
}
```

In the end, the last use of the BitStream class is to read the rest of the information that is in the file into a string, which are the residual values encoded with Golomb, and the respective truncation value of the padding made (extra zeros).

```
//total size of the file - the size of the header and the size
    of the m vector
int total = bs.getFileSize() - (16 + 2*m_size);
long totalBits = total*8;
vector<int> v_encoded = bs.readBits(totalBits);
//convert vector<int> of bits to string of bits
string encoded = "";
for(long unsigned int  i = 0; i < v_encoded.size(); i++) {
    encoded += to_string(v_encoded[i]);
}

//discard the last num_zeros bits
encoded = encoded.substr(0, encoded.size() - num_zeros);
```

The decoding process boils down to just invoking the *decode* or *decodeMultiple* function of the Golomb class, depending on whether the vector of m has only 1 element (the encoding was done with a fixed m), or more than 1 element ("auto" encoding using an optimal m for each blocksize samples).

```
//decode looping through the v_m vector
Golomb g;
vector<int> decoded;
if (m_size == 1)
    decoded = g.decode(encoded, m_vector[0]);
else
```

```
    decoded = g.decodeMultiple(encoded, m_vector, blockSize);
```

Having the vector of integers already decoded, which in this case are the residuals, it is only necessary to make the inverse of the prediction made in the encoder, where the real values are based on the 3 previous values, and the calculation of the real value becomes the addition between the residual and the predicted value. The process is almost identical both for audios with 1 or two audio channels, where in stereo the calculation of the real values is done in each channel and, in the end, merged into a single vector with both channels.

```
if (nChannels < 2) {
    //mono
    for (long unsigned int i = 0; i < decoded.size(); i++) {
        if (i >= 3) {
            int difference = decoded[i] + predict(samplesVector[i-1], samplesVector[i-2], samplesVector[i-3]);
            samplesVector.push_back(difference);
        }
        else samplesVector.push_back(decoded[i]);
    }
} else {
    //stereo
    for (int i = 0; i < nFrames; i++) {
        if (i >= 3) {
            int difference = decoded[i] + predict(samplesVector[i-1], samplesVector[i-2], samplesVector[i-3]);
            samplesVector.push_back(difference);
        }
        else samplesVector.push_back(decoded[i]);
    }

    for (int i = nFrames; i < decoded.size(); i++) {
        if ((int) i >= nFrames + 3) {
            int difference = decoded[i] + predict(samplesVector[i-1], samplesVector[i-2], samplesVector[i-3]);
            samplesVector.push_back(difference);
        }
        else samplesVector.push_back(decoded[i]);
    }

    //merge the two channels into one vector
    vector<short> merged;
    //the first channel is the first nFrames samples
    vector<short> firstChannel(samplesVector.begin(), samplesVector.begin() + nFrames);
    //the second channel is the last nFrames samples
    vector<short> secondChannel(samplesVector.begin() + nFrames, samplesVector.end());
    for(int i = 0; i < nFrames; i++) {
        merged.push_back(firstChannel[i]);
        merged.push_back(secondChannel[i]);
    }
```

```
38      samplesVector = merged;
39  }
```

Finally, we check whether quantization was used in the encoding, evaluating the value of the quantization factor extracted from the header. In there is quantization, the replacement of the "discarded" bits can be done in a more optimal way, reducing the error produced by cutting these bits. This reduction is done by inserting half of the cut value, instead of inserting all the bits to 0. In other words, if 4 bits are cut from the original value, instead of adding these 4 bits as 0's (0000), we set the first one to 1 and the rest to 0 (1000), which results in a value approximately closer to the original value before being quantized.
The padding made in the encoder are also removed taking into account the *blocksize*. The final audio file is written and the program is finished.

```
1  //quantize the samples
2  if (q != 0) {
3      if(q != 1){
4          for (long unsigned int i = 0; i < samplesVector.size();
       i++) {
5              //shift the sample to the left by 1 bit
6              samplesVector[i] = samplesVector[i] << 1;
7              //change that last bit to 1
8              samplesVector[i] = samplesVector[i] | 1;
9              // //shif the sample to the left by q-1 bits
10             samplesVector[i] = samplesVector[i] << (q-1);
11         }
12     } else {
13         for (long unsigned int i = 0; i < samplesVector.size();
       i++) {
14             //shift the sample to the left by 1 bit
15             samplesVector[i] = samplesVector[i] << 1;
16         }
17     }
18 }
19 //remove the last int padding values
20 samplesVector = vector<short>(samplesVector.begin(),
       samplesVector.end() - padding);
21 //write to wav file the samples vector
22 sfhOut.write(samplesVector.data(), samplesVector.size());
23 bs.close();
```

4.3 Usage

The usage of the program is done following this pattern:

```
1  ./golomb_encoder <input file> <output file> <m | bs> [auto] [q]
```

Where the input file is an audio file (tested with .wav), the output file is a binary file (without extension or .txt), the m and blocksize (bs) are integers, auto is literally the string "auto" and q is an integer between 1 and 15.

There are several options on how to use the program, being mandatory that the input and output files and one more argument is passed, and those options are: indicating the m and use it as a fixed value, therefore the arguments of bs and/or auto can't be passed and quantization (q) is optional; or indicating the blocksize, therefore the m isn't considered as an argument and it's mandatory that the flag "auto" is passed and quantization (q) is optional.

And then, to decode the output file of the encoder:

```
./golomb_decoder <input file> <output file>
```

Where the input file is the binary file used before and the output file is an audio file (tested with .wav).

4.4 Results

Executing automate.py we test the encoder and decoder with different parameters and analyze the results by constructing a few tables and deriving plots from the results. We tested lossless and lossy encode, with fixed M values or automatic M, calculated every block size, to see the differences.

Audio size (MB)		
sample04.wav	sample06.wav	sample01.wav
2.35	4.24	5.18

Table 4.1: Audio files size in Megabytes

M fixed values			
sample04.wav			
M value	Time (ms)	Out size (MB)	Compress. ratio (%)
128	400	1.46	38.1
256	435	1.51	35.7
512	456	1.63	31.0
1024	494	1.77	25.0
2048	515	1.92	18.6
sample06.wav			
M value	Time (ms)	Out size (MB)	Compress. ratio (%)
128	684	2.39	43.7
256	755	2.65	37.5
512	799	2.91	31.2
1024	864	3.18	24.9
2048	933	3.45	18.6
sample01.wav			
M value	Time (ms)	Out size (MB)	Compress. ratio (%)
128	1121	4.27	17.6
256	992	3.84	25.8
512	1012	3.80	26.7
1024	1044	3.95	23.6
2048	1191	4.22	18.5

Table 4.2: Lossless encoder time and compression performance for fixed M

From these results we can derive Figure 4.1 and Figure 4.2 which plots the time needed to run the encoder in order of M value and the compression ratio (relative to the output file size) in order of M value.

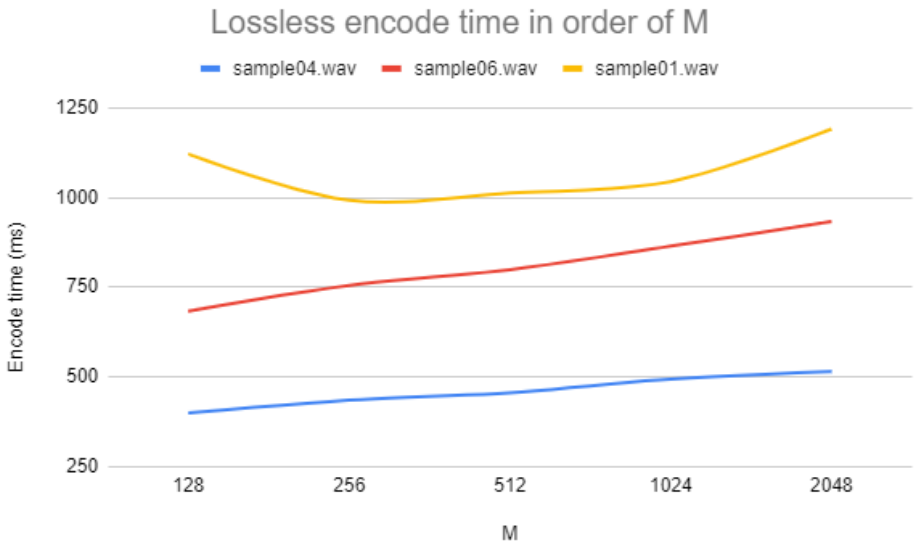

Figure 4.1: Lossless encoder time in order of M

Analyzing the figure above we can clearly see that, firstly, the encode time is directly dependant of the input file size, this is to be expected as the bigger the file, the more frames the encoder needs to iterate through. Secondly we can see that there is not a best M value for every file, on sample04.wav and sample06.wav the best M value is 128, but for sample01.wav the best M value is 256.

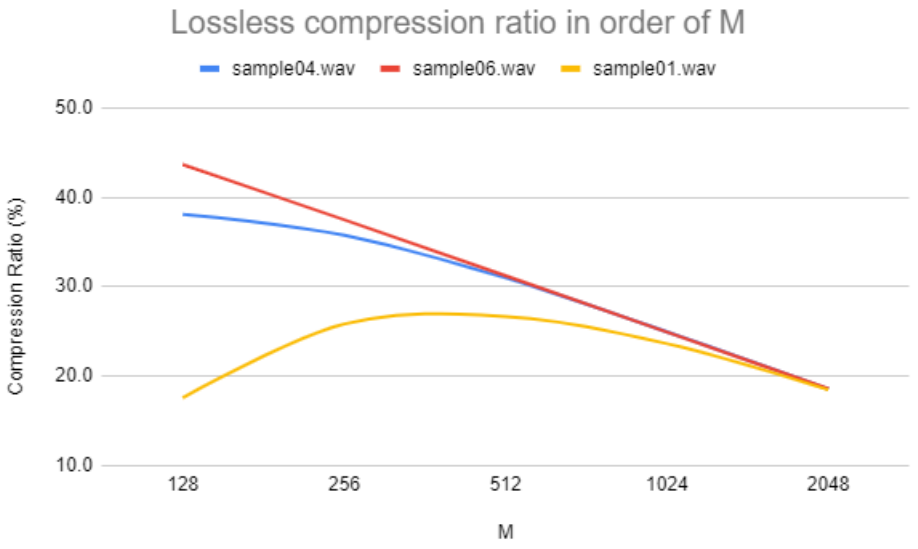

Figure 4.2: Lossless encoder compression in order of M

Analyzing the lossless compression ratio in order of M we see a similar pattern as in the previous observation. Firstly, we see that the input file size is not related to the compression ratio, as the sample06.wav size is greater than sample04.wav and smaller than sample01.wav, but the best compression rate is achieved on sample06.wav. Secondly, we see again that there is not best M value, because for sample06.wav and sample04.wav the best M value is 128, but for sample01.wav the best M value is 512.

With these results we conclude that the best way to encode a file is using an automatic M value calculated on the fly and appropriate to the file in question, the automatic M will be calculated every block size samples given.

Fixed block size, auto M			
sample04.wav			
Block size	Time (ms)	Out size (MB)	Compress. ratio (%)
128	404	1.46	38.0
512	400	1.45	38.5
1024	406	1.45	38.6
2048	412	1.45	38.5
16384	413	1.45	38.5
sample06.wav			
Block size	Time (ms)	Out size (MB)	Compress. ratio (%)
128	472	1.65	60.9
512	459	1.64	61.3
1024	478	1.64	61.3
2048	469	1.64	61.3
16384	463	1.65	61.1
sample01.wav			
Block size	Time (ms)	Out size (MB)	Compress. ratio (%)
128	1007	3.79	26.8
512	990	3.76	27.4
1024	1008	3.75	27.5
2048	1004	3.75	27.5
16384	1002	3.78	27.0

Table 4.3: Lossless encoder time and compression performance for fixed block size and auto M

From these results we can derive Figure 4.3 and Figure 4.4 which plots the time needed to run the encoder in order of block size and the compression ratio (relative to the output file size) in order of block size.

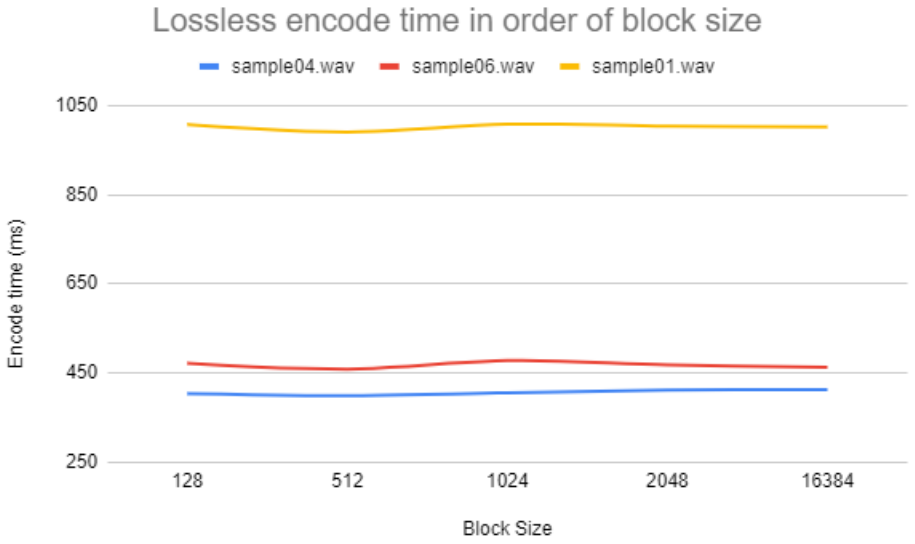

Figure 4.3: Lossless encoder time in order of block size

Analyzing the figure above we can clearly see that, firstly, the encode time is directly dependant of the input file size, this is to be expected as the bigger the file, the more frames the encoder needs to iterate through. All the results are very similar bu we can see that using a block size of 1024 or 2048 should be a good value to use to encode.

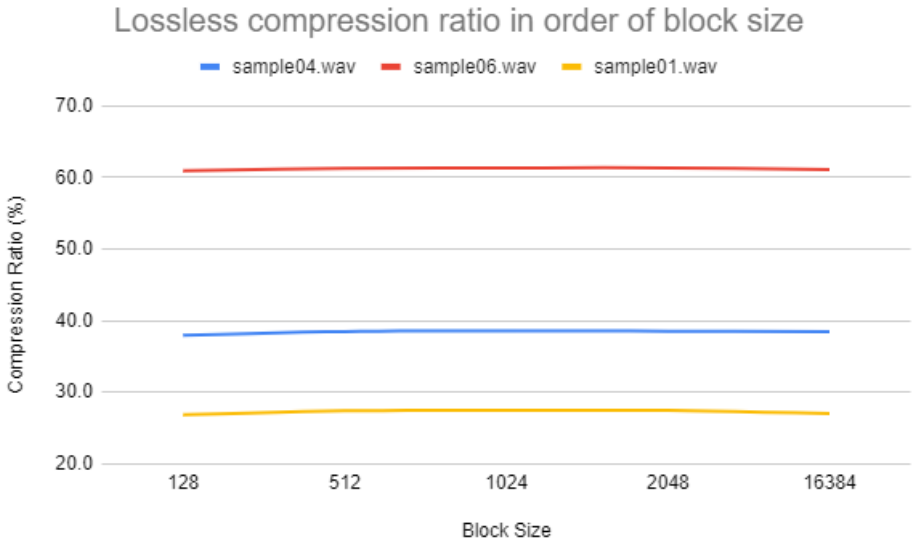

Figure 4.4: Lossless encoder compression in order of block size

Analyzing the lossless compression ratio in order of block size we see a similar pattern as in the previous observation. Firstly, we see that the input file size is not related to the compression ratio, as the sample06.wav size is greater than sample04.wav and smaller than sample01.wav, but the best compression rate is achieved on sample06.wav. Secondly, we see again that the block size value doesn't have a very big impact, nevertheless, having 1% smaller compression ratio for a file in the order of Megabytes will translate in a bigger output file in the order of the tens of kilobytes. We determine again that a block size value of 1024 or 2048 should is a good value to use to encode.

Knowing that those block size values are good options we will now test the lossy encoder with a fixed block size value of 2048.

Fixed block size, auto M			
sample04.wav			
Quantified bits	Time (ms)	Out size (MB)	Compress. ratio (%)
2	335	1.15	51.0
4	252	0.87	62.9
6	190	0.64	72.8
8	164	0.62	73.8
sample06.wav			
Quantified bits	Time (ms)	Out size (MB)	Compress. ratio (%)
2	351	1.17	72.4
4	301	1.12	73.5
6	282	1.07	74.7
8	248	1.00	76.5
sample01.wav			
Quantified bits	Time (ms)	Out size (MB)	Compress. ratio (%)
2	861	3.11	40.0
4	713	2.47	52.4
6	509	1.85	64.3
8	406	1.55	70.0

Table 4.4: Lossy encoder compression rate performance

With the first two columns we can derive Figure 4.5 and Figure 4.4 which plots the time needed to run the encoder in order of block size and the compression ratio (relative to the output file size) in order of block size.

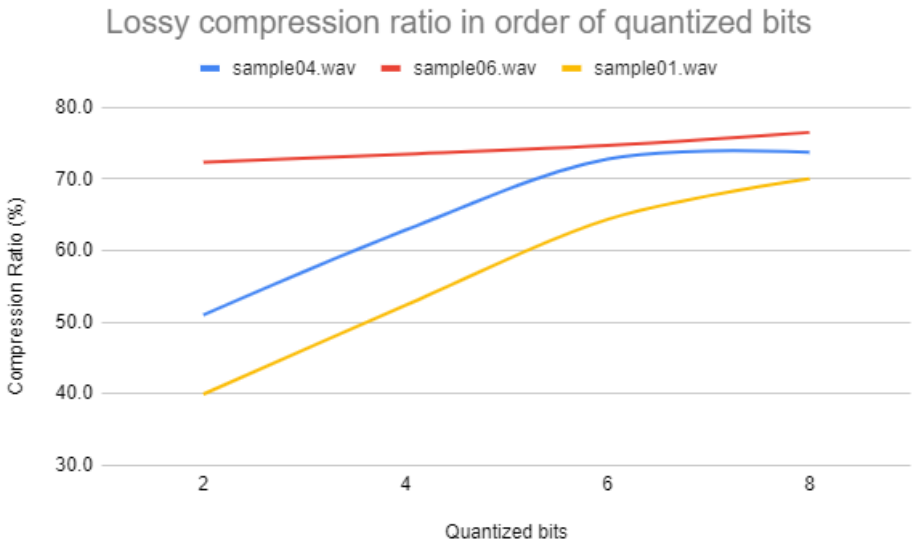

Figure 4.5: Lossy encoder compression in order of quantified bits

Analyzing the figure above we can clearly see that, firstly, the more bits removed the greater, this is expected as the more bits removed, less information is left and the less space needed to represent that information. Secondly we see again that the input file size is not related to the compression ratio, as the sample06.wav size is greater than sample04.wav and smaller than sample01.wav, but the best compression rate is achieved on sample06.wav, this is probably because this audio contains a lot of moments where it's silent, thus the values needed to be encoded are the value 0. We will later compare these results against the decoded sample and see what was the effect on the SNR value.

One thing that we have glanced over, it is strange to see that sample06.wav has a very large compression ration when compared with the other samples, this type of behaviour can occur when a audio has a lot of silence or uniform sounds may be more compressible than audio with a lot of complex or varied sounds.

M fixed values		Fixed block size, auto M	
sample04.wav			
M value	Time (ms)	Block size	Time (ms)
128	400	128	404
256	435	512	400
512	456	1024	406
1024	494	2048	412
2048	515	16384	413
sample06.wav			
128	684	128	472
256	755	512	459
512	799	1024	478
1024	864	2048	469
2048	933	16384	463
sample01.wav			
128	1121	128	1007
256	992	512	990
512	1012	1024	1008
1024	1044	2048	1004
2048	1191	16384	1002

Table 4.5: Decoder time performance

From these results we can derive Figure 4.6 and Figure 4.7 which plots the time needed to run the decoder in order of M value and in order of block size.

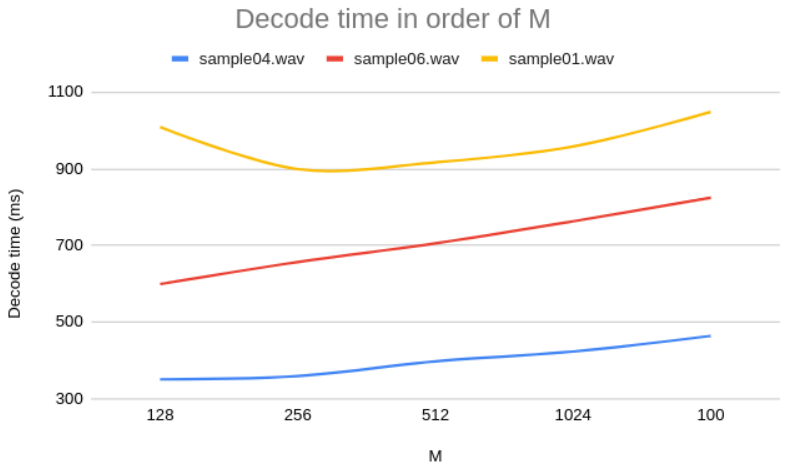

Figure 4.6: Decoder time in order of M

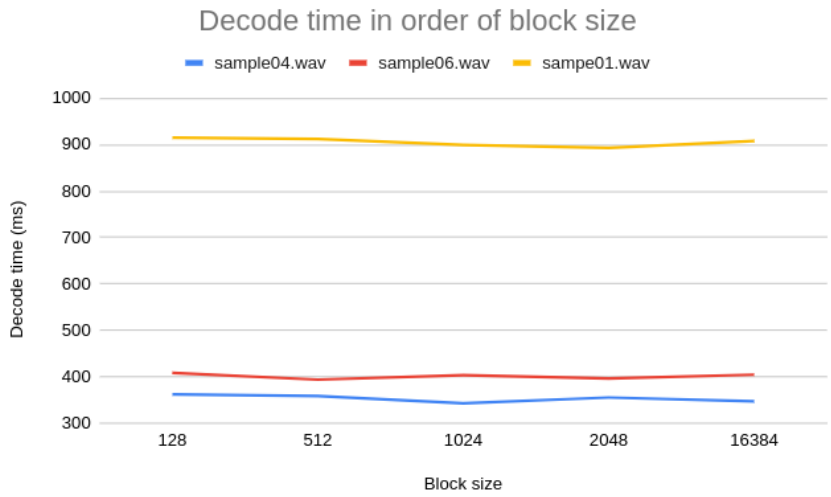

Figure 4.7: Decoder time in order of BS

The results got for the decoder time performance are very similar to the ones got in Figure 4.1 and in Figure 4.3, therefore we get similar conclusions, the only thing to note is that the decode process is a little bit faster than the encode process.

Block size of 2048, auto M			
sample04.wav			
Quantization bits	Time (ms)	SNR (dB)	SNR wav_quant (dB)
2	282	72.4	68.7
4	210	60.8	55.3
6	160	48.8	42.9
8	163	36.5	30.7
sample06.wav			
2	295	61.5	57.8
4	266	49.9	44.3
6	257	37.8	31.9
8	230	25.8	19.8
sample01.wav			
2	757	70.1	66.4
4	624	58.5	52.9
6	444	46.5	40.6
8	366	34.5	28.5

Table 4.6: Signal-to-noise ratio for the decoder and using previous project wav_quant

From these results we can derive Figure 4.8, Figure 4.9 and Figure 4.10 which plots the SNR in order of removed bits, the SNR in order of removed bits using wav_quant from the previous project (SNR wav_quant) and the SNR vs SNR (SNR wav_quant). Keep in mind that the SNR calculation was done using wav_cmp from the previous project.

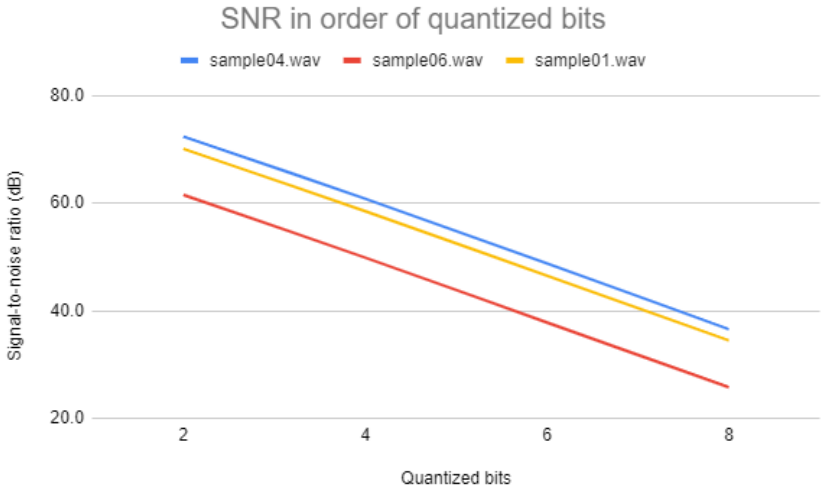

Figure 4.8: Decoder time in order of BS

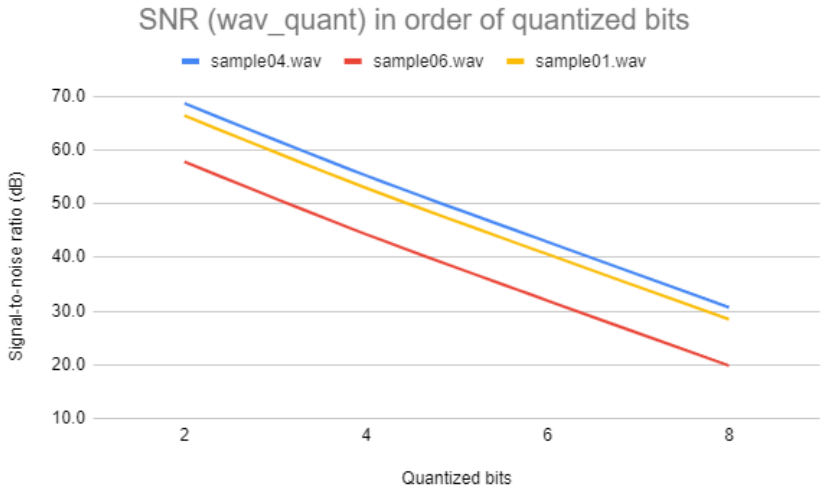

Figure 4.9: Decoder time in order of BS

From Figure 4.8 and in Figure 4.9 we can see that as the bits removed increase the worst is the SNR from the output file, this is to be expected, the explanation and testing of this effect was studied in the previous project.

Comparing both approaches in the figure below we can see that, for all cases, the SNR value got from our process in higher when compared with the same quantization done with the previous project approach. We that in mind we can safely say that, the quantization change made in our encode process and explained before, is clearly a better solution than the approach done in the previous project setting the removed bits to zero.

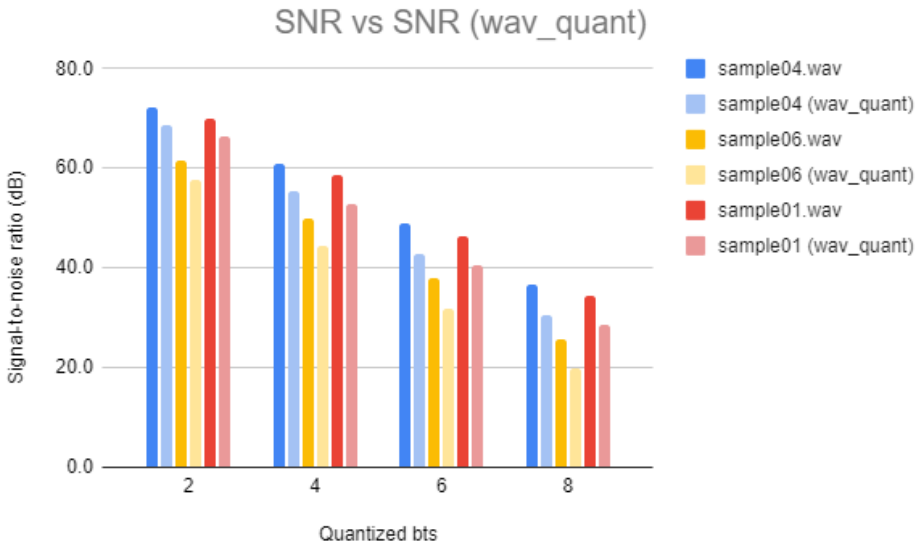

Figure 4.10: Decoder time in order of BS

Another conclusion to take in consideration is that, when comparing the results got in Figure 4.10 and Figure 4.5, we can see that a quantification of 6 bits is a good compromise between sound quality and compression ratio, this because the SNR values are roughly around 40dB (which still provides a good sound quality) and the compression ratio for this quantification value is above 60% for all audio samples.

Chapter 5

Exercise 6

Exercise 6 consists of implementing a lossless image codec, based on Golomb coding of the prediction residuals, similar to the audio codec.

5.1 Code : Encoder

We developed a program that encodes the residuals with the value of m optimally determined for each block, where, in our case, each block is each line of pixels. In addition, we implemented the various linear prediction modes, 1 to 7, (lossless mode of JPEG) and the non-linear predictor of JPEG-LS, mode 8, where the choice of prediction mode is passed as an optional argument (in the case of absence, the non-linear prediction mode was chosen as default).
For a better prediction of the pixel values, the image was transformed from RGB representation to YCbCr, which allows to have smaller residual values and, consequently, a smaller compressed value.

```
auto rgb2ycbcr = [](Mat img) {
    // convert to YCbCr
    Mat ycbcr;
    cvtColor(img, ycbcr, COLOR_BGR2YCrCb);
    return ycbcr;
};

...

//convert the image to YCbCr
img = rgb2ycbcr(img);
```

Having already calculated the matrix of values in YCbCr, we proceed to calculate the residuals of the predictions, which varies taking into account not only the mode, but also the pixel that we are currently analyzing. Our implementation does not predict the first pixel of the image (first row and column). However, for the rest of the first line, the prediction is made based

on the previous pixel, just doing a subtraction of the current one with the previous one. Then, for each of the following lines, the first pixel (i.e. the first column of the image) is predicted taking into account only the pixel above. For all other pixels, we use the previous, above and diagonally up and left pixel to predict the current one (depending on the mode, of course).

```
// for each pixel in the image
for (int i = 0; i < img.rows; i++) {
    for (int j = 0; j < img.cols; j++) {
        // get the pixel value
        int Y = img.at<Vec3b>(i,j)[0];
        int Cb = img.at<Vec3b>(i,j)[1];
        int Cr = img.at<Vec3b>(i,j)[2];

        //if its the first pixel of the image, do not use prediction
        if (i == 0 && j == 0) {
            YvaluesToBeEncoded.push_back(Y);
            CbvaluesToBeEncoded.push_back(Cb);
            CrvaluesToBeEncoded.push_back(Cr);
            pixel_index++;
        } else if(i==0){
            //if its the first line of the image, use only the previous pixel (to the left)
            int Yerror = Y - img.at<Vec3b>(i,j-1)[0];
            int Cberror = Cb - img.at<Vec3b>(i,j-1)[1];
            int Crerror = Cr - img.at<Vec3b>(i,j-1)[2];
            YvaluesToBeEncoded.push_back(Yerror);
            CbvaluesToBeEncoded.push_back(Cberror);
            CrvaluesToBeEncoded.push_back(Crerror);
            pixel_index++;
        }else if(j==0){
            //if its the first pixel of the line, use only the pixel above
            int Yerror = Y - img.at<Vec3b>(i-1,j)[0];
            int Cberror = Cb - img.at<Vec3b>(i-1,j)[1];
            int Crerror = Cr - img.at<Vec3b>(i-1,j)[2];
            YvaluesToBeEncoded.push_back(Yerror);
            CbvaluesToBeEncoded.push_back(Cberror);
            CrvaluesToBeEncoded.push_back(Crerror);
            pixel_index++;
        } else {
            //if its not the first pixel of the image nor the first line, use the 3 pixels to the left, above and to the left top
            int Yerror = Y - int(predict(img.at<Vec3b>(i,j-1)[0], img.at<Vec3b>(i-1,j)[0], img.at<Vec3b>(i-1,j-1)[0], mode));
            int Cberror = Cb - int(predict(img.at<Vec3b>(i,j-1)[1], img.at<Vec3b>(i-1,j)[1], img.at<Vec3b>(i-1,j-1)[1], mode));
            int Crerror = Cr - int(predict(img.at<Vec3b>(i,j-1)[2], img.at<Vec3b>(i-1,j)[2], img.at<Vec3b>(i-1,j-1)[2], mode
```

```
    ));
38              YvaluesToBeEncoded.push_back(Yerror);
39              CbvaluesToBeEncoded.push_back(Cberror);
40              CrvaluesToBeEncoded.push_back(Crerror);
41              pixel_index++;
42          }
43          ...
```

In a similar way to the audio codec, the optimal m is calculated based on the last line of pixels traversed. However, each m is calculated for each of the pixel channels (Y, Cb, Cr) and stored in the respective vectors.

```
1           ...
2           if(pixel_index % bs == 0 && pixel_index != 0) {
3               int Ysum = 0;
4               int Cbsum = 0;
5               int Crsum = 0;
6               for(int j = pixel_index - bs; j < pixel_index; j++)
    {
7                   Ysum += abs(YvaluesToBeEncoded[j]);
8                   Cbsum += abs(CbvaluesToBeEncoded[j]);
9                   Crsum += abs(CrvaluesToBeEncoded[j]);
10              }
11              int Yu = round(Ysum / bs);
12              int Cbu = round(Cbsum / bs);
13              int Cru = round(Crsum / bs);
14              int Ym = calc_m(Yu);
15              int Cbm = calc_m(Cbu);
16              int Crm = calc_m(Cru);
17              if (Ym < 1) Ym = 1;
18              if (Cbm < 1) Cbm = 1;
19              if (Crm < 1) Crm = 1;
20              Ym_vector.push_back(Ym);
21              Cbm_vector.push_back(Cbm);
22              Crm_vector.push_back(Crm);
23          }
24      }
25  }
```

```
1   auto calc_m = [](int u) {
2       //u = alpha / 1 - alpha
3       //m = - (1 / log(alpha))
4       return (int) - (1/log((double) u / (1 + u)));
5   };
```

After calculating all the residuals, we call the function that encodes these values taking into account the current m, depending on the block being encoded at the given moment.

```
1   int m_index = 0;
2   for (int i = 0; i < size; i++) {
3       if (i % bs == 0 && i != 0) m_index++;
4       YencodedString += g.encode(YvaluesToBeEncoded[i], Ym_vector[
    m_index]);
```

```
    CbencodedString += g.encode(CbvaluesToBeEncoded[i],
    Cbm_vector[m_index]);
    CrencodedString += g.encode(CrvaluesToBeEncoded[i],
    Crm_vector[m_index]);
}
```

At the end we write to a binary file, using the BitStream class already developed. Initially, the necessary values are written into a header for the decoding to be done correctly in another program. These values are the image type, the mode, the number of rows and columns, the size of each encoded string (*YencodedString.size()*, *CbencodedString.size()* and *CrencodedString.size()*, the size of each vector of m (it's the same value for all 3 vectors) and the vectors themselves, and finally each of the encoded strings, along with their sizes for future distinction .

```
BitStream bitStream(output, "w");
vector<int> bits;

//the first 16 bits are the image type
for (int i = 31; i >= 0; i--)
    bits.push_back((type >> i) & 1);

...

//contains all the bits from the values to be encoded vectors (Y, Cb, Cr)
vector<int> encoded_bits;

for (long unsigned int i = 0; i < YencodedString.length(); i++)
    encoded_bits.push_back(YencodedString[i] - '0');

for (long unsigned int i = 0; i < CbencodedString.length(); i++)
    encoded_bits.push_back(CbencodedString[i] - '0');

for (long unsigned int i = 0; i < CrencodedString.length(); i++)
    encoded_bits.push_back(CrencodedString[i] - '0');

//the next bits are the encoded bits
for (long unsigned int i = 0; i < encoded_bits.size(); i++)
    bits.push_back(encoded_bits[i]);

//write the bits to the file
bitStream.writeBits(bits);
bitStream.close();
```

5.2 Code : Decoder

The decoder relies on reading the binary file generated by the encoder and reversing the entire encoding process using predictions and Golomb codes. The first step is to read the header of that file and calculate the extracted values.

```
BitStream bs (argv[1], "r");
vector<int> v_imgtype = bs.readBits(32);
vector<int> v_mode = bs.readBits(16);
vector<int> v_imgwidth = bs.readBits(16);
vector<int> v_imgheight = bs.readBits(16);
vector<int> v_bs = bs.readBits(16);
vector<int> v_encY = bs.readBits(32);
vector<int> v_encCb = bs.readBits(32);
vector<int> v_encCr = bs.readBits(32);
vector<int> v_msize = bs.readBits(16);
```

After interpreting the header, the next part of the binary file contains each of the m vectors associated with each of the image channels and, therefore, all these values are read and converted to the respective integers. Furthermore, the strings containing all the values encoded with Golomg are read, finally resulting in 3 vectors with the optimal m for each channel and 3 strings, each of them with the residuals encoded in the encoder.

```
vector<int> Ym_vector;
for (int i = 0; i < msize; i++) {
    vector<int> v_Ym = bs.readBits(16);
    int Ym = 0;
    for(long unsigned int i = 0; i < v_Ym.size(); i++) {
        Ym += v_Ym[i] * pow(2, v_Ym.size() - i - 1);
    }
    Ym_vector.push_back(Ym);
}

vector<int> Cbm_vector;
for(int i = 0; i < msize; i++) {
    vector<int> v_Cbm = bs.readBits(16);
    int Cbm = 0;
    for(long unsigned int i = 0; i < v_Cbm.size(); i++) {
        Cbm += v_Cbm[i] * pow(2, v_Cbm.size() - i - 1);
    }
    Cbm_vector.push_back(Cbm);
}

vector<int> Crm_vector;
for(int i = 0; i < msize; i++) {
    vector<int> v_Crm = bs.readBits(16);
    int Crm = 0;
    for(long unsigned int i = 0; i < v_Crm.size(); i++) {
        Crm += v_Crm[i] * pow(2, v_Crm.size() - i - 1);
    }
    Crm_vector.push_back(Crm);
}

vector<int> Y_values = bs.readBits(encY);
vector<int> Cb_values = bs.readBits(encCb);
vector<int> Cr_values = bs.readBits(encCr);

string YencodedString = "";
for(long unsigned int i = 0; i < Y_values.size(); i++) {
```

```
    YencodedString += to_string(Y_values[i]);
}
string CbencodedString = "";
for(long unsigned int i = 0; i < Cb_values.size(); i++) {
    CbencodedString += to_string(Cb_values[i]);
}
string CrencodedString = "";
for(long unsigned int i = 0; i < Cr_values.size(); i++) {
    CrencodedString += to_string(Cr_values[i]);
}
```

Having these vectors and string already determined, we move on to the decoding using the Golomg class, which returns the residuals in a vector of integers.

```
Golomb g;
vector<int> Ydecoded;
vector<int> Cbdecoded;
vector<int> Crdecoded;

Ydecoded = g.decodeMultiple(YencodedString, Ym_vector, imgwidth);
Cbdecoded = g.decodeMultiple(CbencodedString, Cbm_vector, imgwidth);
Crdecoded = g.decodeMultiple(CrencodedString, Crm_vector, imgwidth);
```

Finally, the last thing to do is the reverse process of the predictions, depending on the selected mode, taking into account the values of the residuals and the previous predictions made, to calculate the actual values. Again, it is important to take into account the pixel we are analyzing, as the predictions vary depending on whether the pixel is in the first row or column.

```
//undo the predictions
int pixel_idx = 0;
for (int i = 0; i < imgheight; i++) {
    for (int j = 0; j < imgwidth; j++) {
        if (i == 0 && j == 0) {
            //create a new pixel with the decoded values of Y, Cb and Cr at the current pixel index and add it to the image without RGB conversion
            new_image.at<Vec3b>(j, i) = Vec3b(Ydecoded[pixel_idx], Cbdecoded[pixel_idx], Crdecoded[pixel_idx]);
            pixel_idx++;
        } else if (i == 0) {
            //if its the first line of the image, use only the previous pixel (to the left)
            int Y = new_image.at<Vec3b>(i, j-1)[0] + Ydecoded[pixel_idx];
            int Cb = new_image.at<Vec3b>(i, j-1)[1] + Cbdecoded[pixel_idx];
            int Cr = new_image.at<Vec3b>(i, j-1)[2] + Crdecoded[pixel_idx];
            new_image.at<Vec3b>(i, j) = Vec3b(Y, Cb, Cr);
```

```
            pixel_idx++;
        } else if (j == 0) {
            //if its the first pixel of the line, use only the pixel above
            int Y = new_image.at<Vec3b>(i-1, j)[0] + Ydecoded[pixel_idx];
            int Cb = new_image.at<Vec3b>(i-1, j)[1] + Cbdecoded[pixel_idx];
            int Cr = new_image.at<Vec3b>(i-1, j)[2] + Crdecoded[pixel_idx];
            new_image.at<Vec3b>(i, j) = Vec3b(Y, Cb, Cr);
            pixel_idx++;
        } else {
            //if its not the first pixel of the image nor the first line, use the 3 pixels to the left, above and to the left top
            int Y = int(predict(new_image.at<Vec3b>(i, j-1)[0], new_image.at<Vec3b>(i-1, j)[0], new_image.at<Vec3b>(i-1, j-1)[0], mode)) + Ydecoded[pixel_idx];
            int Cb = int(predict(new_image.at<Vec3b>(i, j-1)[1], new_image.at<Vec3b>(i-1, j)[1], new_image.at<Vec3b>(i-1, j-1)[1], mode)) + Cbdecoded[pixel_idx];
            int Cr = int(predict(new_image.at<Vec3b>(i, j-1)[2], new_image.at<Vec3b>(i-1, j)[2], new_image.at<Vec3b>(i-1, j-1)[2], mode)) + Crdecoded[pixel_idx];
            new_image.at<Vec3b>(i, j) = Vec3b(Y, Cb, Cr);
            pixel_idx++;
        }
    }
}
```

With the matrix already correctly calculated, we make the conversion from YCbCr to RGB and then write this matrix into a *ppm* file.

```
//convert the image from YCbCr to RGB
new_image = ycbcr2rgb(new_image);

//save the image
imwrite(output, new_image);
```

5.3 Usage

The usage of the program is done following this pattern:

```
./image_encoder <intput_file> <output_file> [mode]
```

Where the input file is an image file (tested with .ppm), the output file is a binary file (without extension or .txt) and the mode is an integer between 1 and 8, that, if absent, the default value of prediction mode is 8 (non-linear).

And then, to decode the output file of the encoder:

```
./image_decoder <intput_file> <output_file>
```

Where the input file is the binary file used before and the output file is an image file (tested with .ppm).

5.4 Results

Executing imageauto.py we test the encoder and decoder and analyze the results by constructing a few tables and deriving plots from the results.

Image size (MB)			
house.ppm	anemone.ppm	arial.ppm	bike3.ppm
0.20	1.02	1.09	2.15

Table 5.1: Image files size in Megabytes

house.ppm			
Mode	Time (ms)	Out size (MB)	Compress. ratio (%)
1	31	0.11	44.5
2	31	0.11	42.9
3	32	0.12	40.2
4	30	0.11	45.0
5	29	0.11	45.6
6	30	0.11	44.9
7	30	0.11	44.5
8	30	0.11	45.2
anemone.ppm			
1	177	0.64	37.2
2	186	0.64	37.1
3	199	0.69	32.0
4	184	0.62	39.1
5	181	0.61	40.5
6	172	0.61	40.6
7	175	0.61	40.2
8	181	0.59	42.1

arial.ppm			
1	184	0.66	39.2
2	183	0.67	38.3
3	185	0.68	37.5
4	230	0.71	34.4
5	212	0.68	37.9
6	192	0.68	37.6
7	187	0.65	40.5
8	194	0.66	38.9
bike3.ppm			
Mode	Time (ms)	Out size (MB)	Compress. ratio (%)
1	407	1.42	34.1
2	412	1.42	34.0
3	426	1.50	30.2
4	420	1.48	31.5
5	411	1.41	34.6
6	415	1.41	34.6
7	421	1.37	36.5
8	415	1.38	35.8

Table 5.2: Image encoder time and compression performance

From these results we also constructed a Table 5.3 with the average compression ratio for all image on each mode.

Average results across images	
Mode	Compress. Ratio (%)
1	38.7
2	38.1
3	35.0
4	37.5
5	39.6
6	39.4
7	40.4
8	40.5

Table 5.3: Average compression ratio results

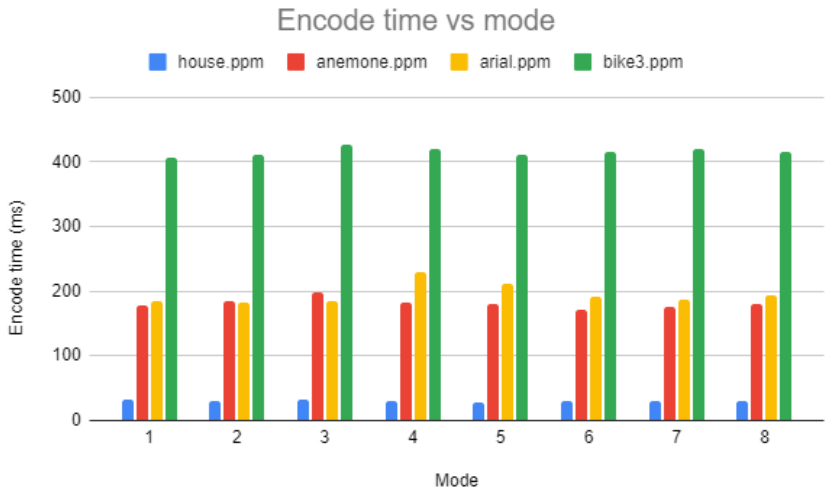

Figure 5.1: Encode time performance vs mode

From the results got from Table 5.2 we can plot the encode time vs mode Figure 5.1. Analyzing this graph we can clearly see that regardless of the mode used, as the image file size increases, the time needed to encode increases as well, this is to be expected because a bigger image has more pixels, therefore the encoder needs more time to iterate through the pixels.

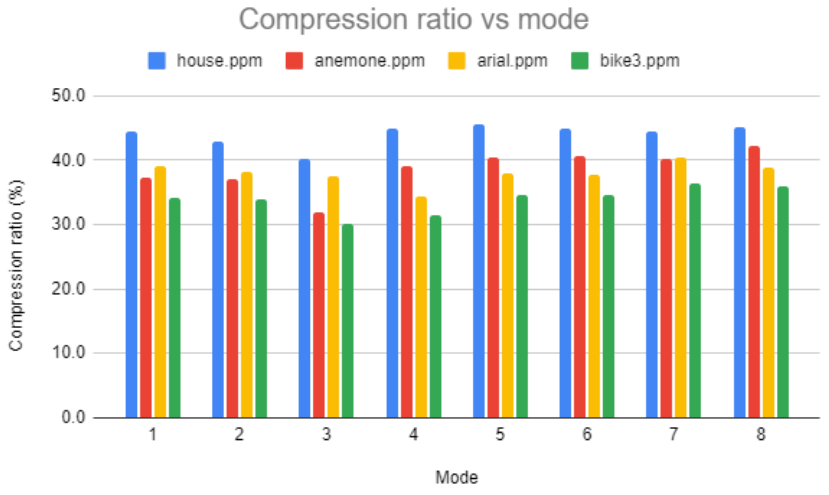

Figure 5.2: Encode compression ratio vs mode

We can also derive Figure 5.2, which shows a curious trend, as the file size increases the compression ratio decreases this could be because, as the size of the image increases, the amount of statistical information that can

be gleaned from the data also increases. This means that the amount of redundancy in the data also increases, which can lead to a decrease in the overall compression ratio. Additionally, larger images are often more complex and may contain more information that cannot be easily compressed, which can also lead to a decrease in the compression ratio.

In summary, the compression ratio decreases as the size of the image increases because of the increased amount of statistical information and complexity in the data.

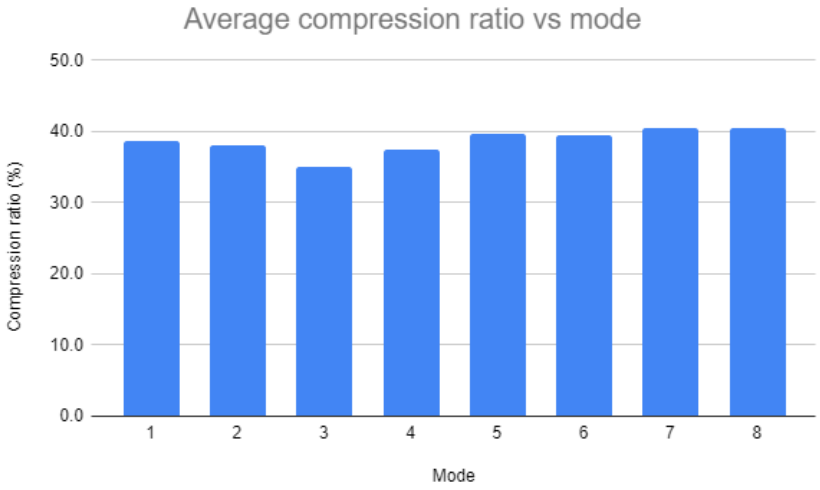

Figure 5.3: Average encode compression vs mode

With the results got on Table 5.3 we can see what is the average compression rate for each mode, with these results, we can safely say that mode 7 and 8 best parameters to encode most of the images used. Mode 8 may achieve a better compression ratio because it uses a non-linear prediction method. This means that it uses a more complex prediction algorithm that takes into account the relative values of the previous elements. This can result in a more accurate prediction of the next element in the sequence, which can in turn lead to better compression. In some cases, mode 7 may achieve a similar or even better compression ratio than mode 8 because it uses a simpler prediction method. This means that it may be less sensitive to changes in the image data, and may be able to more consistently predict the values of the next elements in the sequence. This can result in better compression, particularly when the image data contains regular patterns or is relatively smooth.

hosue.ppm		anemone.ppm		arial.ppm		bike3.ppm	
Mode	Time (ms)	Mode	Time (ms)	Mode	Time (ms)	Mode	Time (ms)
1	162	1	177	1	184	1	407
2	153	2	186	2	183	2	412
3	155	3	199	3	185	3	426
1	162	1	177	1	184	1	407
4	169	4	184	4	230	4	420
5	157	5	181	5	212	5	411
6	180	6	172	6	192	6	415
7	197	7	175	7	187	7	421
8	166	8	181	8	194	8	415

Table 5.4: Decoder time performance results

From the results got from Table 5.4 we can plot the decode time vs mode Figure 5.4. Analyzing this graph we will get the same conclusion that we got previously when analyzing Figure 5.1.

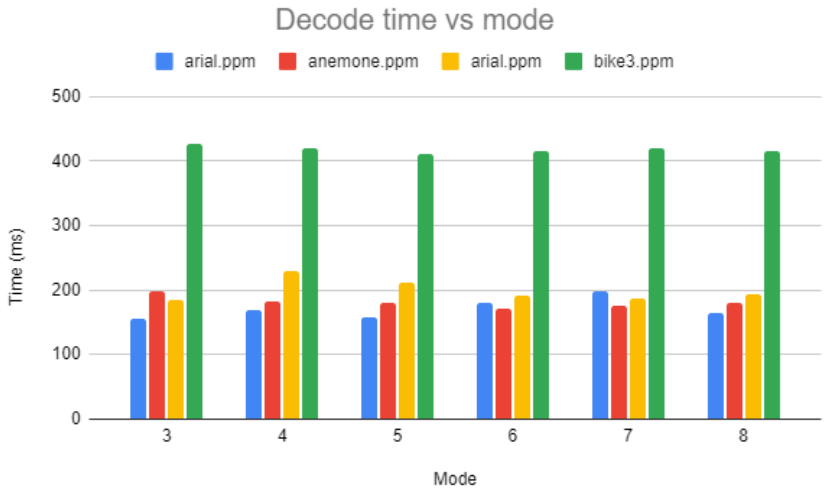

Figure 5.4: Decode time vs mode

Chapter 6

General Information

The contributions between each member of the group were equal.
The project's repository can be viewed here: `https://github.com/PedroRocha9/IC` Every table and plot shown in this report can be found in project's repository (audio-results.xlsx and image-results.xlsx).