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)

January 7, 2024

Contents

List of Figures

Chapter 1

Introduction

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

These tasks involve a combination of image and video 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/trab3`.

Chapter 2

Exercise 1

For the first exercise, he had to implement a lossless image codec for grayscale images, based on Golomb coding of the prediction residuals, similar to the audio codec done previously.

2.1 Usage Instructions

To use the image 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

```

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

```
./image_encoder ../bartolomeu.jpg encoded-bart.bin

```

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

```
./image_decoder encoded-bart.bin decoded_bart.jpg

```

   This command decodes the encoded file `encoded_bart.bin` and generates the decoded image file `decoded_bart.jpg`.

2.2 Code

2.2.1 Image Encoder (`image_encoder.cpp`)

We created a program that encodes image residuals by picking the best 'm' value for each line of pixels. Our software uses the non-linear predictor JPEG-LS:

$$\hat{x} = \begin{cases} min(a,b) & \text{if c} \geq \max(\text{a, b}) \\ max(a,b) & \text{if c} \leq \min(\text{a, b}) \\ a+b-c & \text{otherwise} \end{cases}$$

For a better prediction of the pixel values, the image is loaded in grayscale mode. Grayscale mode means the image will be read and stored as a single-channel image, where each pixel represents the intensity or brightness of the corresponding location in the image.

```
    // read the image in grayscale
    Mat img = imread(argv[1], IMREAD_GRAYSCALE);
```

Listing 2.1: Loading the image in image_encoder.cpp

With the image in grayscale we can now calculate the residuals of the predictions for each pixel. We skip the first pixel of the image (top-left corner), we do not use prediction. And for the rest of the first line, the prediction is based on the value of the pixel to its left. This prediction is achieved simply by subtracting the current pixel's value from the previous one. For the pixels in the first column of each row (excluding the very first row), we predict their values based solely on the pixel directly above them. When predicting the values for all other pixels in the image, we consider a broader context. We use the values of the neighboring pixels: the one to the left, the one above, and the one diagonally up and to the left. Similar to the approach taken in the audio codec, we determine the optimal 'm' value based on the analysis of the last line of pixels processed.

```
vector<int> m_vector;
vector<int> valuesToBeEncoded;

int pixel_index = 0;

// 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 (i,j)
        int pixel = img.at<uchar>(i,j);

        //if its the first pixel of the image, do not use
    prediction
```

```
13          if (i == 0 && j == 0) {
14              valuesToBeEncoded.push_back(pixel);
15              pixel_index++;
16          } else if(i==0){
17              //if the pixel is in the first line of the image
    , use only the pixel to its left
18              int error = pixel - img.at<uchar>(i,j-1);
19              valuesToBeEncoded.push_back(error);
20              pixel_index++;
21          }else if(j==0){
22              //if the pixel is the first of the line, use
    only the pixel above
23              int error = pixel - img.at<uchar>(i-1,j);
24              valuesToBeEncoded.push_back(error);
25              pixel_index++;
26          } else {
27              //for all other pixels, use 3 pixels (to the
    left, above and to the top left)
28              int error = pixel - int(predict(img.at<uchar>(i,
    j-1), img.at<uchar>(i-1,j), img.at<uchar>(i-1,j-1)));
29              valuesToBeEncoded.push_back(error);
30              pixel_index++;
31          }
32
33          if(pixel_index % img.cols == 0 && pixel_index != 0)
    {
34              int sum = 0;
35
36              for(int j = pixel_index - img.cols; j <
    pixel_index; j++) {
37                  sum += abs(valuesToBeEncoded[j]);
38              }
39              int p = round(sum / img.cols);
40
41              int m = calc_m(p);
42
43              if (m < 1) m = 1;
44
45              m_vector.push_back(m);
46          }
47      }
48  }
```

Listing 2.2: Pixel prediction in image_encoder.cpp

Once we've computed all the prediction residuals, we call the encode function, this function is responsible for encoding these residual values while considering the current 'm' value specific to the block being processed at that moment.

```
string encodedString = "";

Golomb g;

//size = rows * cols
int size = img.rows * img.cols;

int m_index = 0;
for (int i = 0; i < size; i++) {
    if (i % img.cols == 0 && i != 0) m_index++;
    //cout << "values to be encoded: " << valuesToBeEncoded[
    i] << endl;

    encodedString += g.encode(valuesToBeEncoded[i], m_vector
    [m_index]);
}
```

Listing 2.3: Encoding frame values in intra_encoder.cpp

At the end of the process, we save the data into a binary file using the BitStream class. Initially, we write important information into a header to ensure proper decoding in another program. This header includes details like the image dimensions (rows and columns), size of the encoded string, size of the 'm' vector, the values within this vector, and finally the encoded string itself. This additional information helps in differentiating and decoding the data accurately when needed later on.

```
BitStream bitStream(output, true);

bitStream.writeNBits(img.cols, 16); // Write image width (16
     bits)
bitStream.writeNBits(img.rows, 16); // Write image height
    (16 bits)
bitStream.writeNBits(encodedString.size(), 32); // Write
    encodedString size (32 bits)
bitStream.writeNBits(m_vector.size(), 16); // Write m_vector
     size (16 bits)


// Write m_vector values (16 bits each)
for (size_t i = 0; i < m_vector.size(); i++) {
    bitStream.writeNBits(m_vector[i], 16);
}

//contains all the image's encoded values
vector<int> encoded_bits;

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

```
19
20
21 // Write the encoded bits
22 for (size_t i = 0; i < encoded_bits.size(); i++) {
23     bitStream.writeBit(encoded_bits[i]);
24 }
25
26
27
28 bitStream.close();
```

Listing 2.4: Saving data in binary file in image_encoder.cpp

2.2.2 Image Decoder (`image_decoder.cpp`)

The decoder operates by reading the binary file created by the encoder and reversing the entire encoding process using predictions and Golomb codes. Initially, the decoder reads the header of the file to extract and compute the relevant values needed for the decoding process. With the 'm' vector and encoded string already established, the next step involves the decoding process using the Golomb class, which returns the residuals in a vector of integers.

```
1  BitStream bs (argv[1], false);
2
3  int imgwidth = bs.readNBits(16);
4  int imgheight = bs.readNBits(16);
5  int enc = bs.readNBits(32);
6  int msize = bs.readNBits(16);
7
8  vector<int> m_vector;
9  for (int i = 0; i < msize; i++) {
10     int m = bs.readNBits(16); // Read 16 bits for m
11
12     m_vector.push_back(m);
13     //cout << "m:" << m << endl;
14 }
15
16 vector<int> encoded_values;
17 for (int i = 0; i < enc; i++) {
18     encoded_values.push_back(bs.readBit());
19 }
20
21 string encodedString = "";
22 for(long unsigned int i = 0; i < encoded_values.size(); i++)
       {
23     encodedString += to_string(encoded_values[i]);
24 }
25
```

```
Golomb g;

vector<int> decoded = g.decode(encodedString, m_vector,
    imgwidth);
```

Listing 2.5: Decoding image values in image_decoder.cpp

After this, we can perform the reverse operation of predictions. Again, it is important to take into account the pixel we are processing, as the predictions vary depending on whether the pixel is in the first row or column. Finally with the matrix calculated, we write it into a new image file.

```
//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 at
    the current pixel index and add it to the image
            new_image.at<uchar>(j, i) = uchar(decoded[
    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 pixel = new_image.at<uchar>(i, j-1) +
    decoded[pixel_idx];
            new_image.at<uchar>(i, j) = uchar(pixel);
            pixel_idx++;
        } else if (j == 0) {
            //if its the first pixel of the line, use only
    the pixel above
            int pixel = new_image.at<uchar>(i-1, j) +
    decoded[pixel_idx];
            new_image.at<uchar>(i, j) = uchar(pixel);
            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 pixel = int(predict(new_image.at<uchar>(i, j
    -1), new_image.at<uchar>(i-1, j), new_image.at<uchar>(i
    -1, j-1))) + decoded[pixel_idx];
            new_image.at<uchar>(i, j) = uchar(pixel);
            pixel_idx++;
        }
    }
}

//save the image
```

```
29 imwrite(output, new_image);
```

Listing 2.6: Reversing predictions in image_decoder.cpp

2.3 Results

```
sebastian  main -  ./image_encoder ../bartolomeu.jpg bart.bin
Time: 96.712 ms
```

Figure 2.1: Image Encoder

```
sebastian  main -  ./image_decoder bart.bin reconstructed-bart.jpg
Execution time: 97.364 ms
```

Figure 2.2: Image Decoder

Figure 2.3: Reconstructed-Bart

Figure 2.4: Original-Bart

The results show that the implemented lossless image codec using Golomb coding of prediction residuals, based on the JPEG-LS non-linear predictor, successfully encodes and decodes grayscale images. The images "Original-Bart" and "Reconstructed-Bart" demonstrate a visually indistinguishable resemblance, indicating that the codec effectively preserves image quality without introducing noticeable artifacts.

The execution times for both the image encoder and decoder are reasonable, with the encoder taking approximately 96.712 milliseconds and the decoder taking around 97.364 milliseconds. These times suggest that the implemented codec performs efficiently in terms of processing speed.

In conclusion, the developed codec demonstrates its capability to compress and decompress grayscale images using Golomb coding techniques and prediction residuals. The preserved image quality and the efficient execution times highlight the effectiveness of the implemented solution for lossless image compression in this context.

Chapter 3

Exercise 2

For the second exercise, he had to adapt the image codec in order to encode videos as simple sequences of images.

3.1 Usage Instructions

To use the video 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

```

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

```
./intra_encoder ../akiyo_cif.y4m encoded_akiyo.bin

```

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

```
./intra_decoder encoded_akiyo.bin decoded_akiyo.y4m

```

   This command decodes the encoded file `encoded_akiyo.bin` and generates the decoded image file `decoded_akiyo.y4m`.

4. Using VCL media player, to play the reconstructed video file, use the following command:

```
vcl decoded_akiyo.y4m

```

3.2 Code

3.2.1 Video Encoder (`intra_encoder.cpp`)

We began by creating a new function named parseY4MHeader. This function helps extract the video details from the video file we're working with, like color space, interlace, aspect ratio, frame rate, frame number and dimensions.

```
//video parameters needed for codec
int width, height, colorspace;
string interlace;
vector<int> pixelAspectRatio(2);
vector<int> frameRate(2);
int num_frames = 0;

void parseY4MHeader(const string& filename) {
    ifstream file;
    file.open(filename, ios::in | ios::binary);

    // Check if the file opened successfully
    if (!file.is_open()) {
        std::cerr << "Error: Couldn't open the file " <<
    filename << std::endl;
        return;
    }
    string line;
    getline(file, line);

    sscanf(line.c_str(), "YUV4MPEG2 W%d H%d F%d:%d", &width,
     &height, &frameRate[0], &frameRate[1]);
    // check if 'C' is in the line
    if (strchr(line.c_str(), 'C') == NULL) {
        colorspace = 420;
    } else {
        // extract the color space after the 'C'
        colorspace = stoi(line.substr(line.find('C') + 1));
    }
    // check if 'I' is in the line
    if (strchr(line.c_str(), 'I') == NULL) {
        interlace = "p";
    } else {
        // extract the interlace after the 'I'
        interlace = line.substr(line.find('I') + 1, 1);
    }
    // check if 'A' is in the line
    if (strchr(line.c_str(), 'A') == NULL) {
        pixelAspectRatio[0] = 1;
        pixelAspectRatio[1] = 1;
    } else {
```

```
40          // extract the aspect ratio after the 'A'
41          string aspect_ratio = line.substr(line.find('A') +
    1);
42          // extract the first number
43          pixelAspectRatio[0] = stoi(aspect_ratio.substr(0,
    aspect_ratio.find(':')));
44          // extract the second number
45          pixelAspectRatio[1] = stoi(aspect_ratio.substr(
    aspect_ratio.find(':') + 1));
46      }
47
48      if (line.find("FRAME") != string::npos) {
49          num_frames++;
50      }
51
52      while (getline(file, line)) {
53            if (line.find("FRAME") != std::string::npos) {
54                num_frames++;
55            }
56      }
57
58  }
```

Listing 3.1: parseY4MHeader function in intra_encoder.cpp

Our intra-frame encoder operates by processing one frame at a time. It identifies the marker "FRAME" to recognize each frame and then proceeds to organize its information into the appropriate vectors. The sizes of these vectors are adjusted according to the color space used in the video.

The Y vector has always the size equal to $Height \times Width$. For the 4:2:0 color space, the U and V vectors have size $Height \times Width/4$, for the 4:2:2 color space, the U and V vectors have size $Height \times Width/2$ and for the 4:4:4 color space, the U and V vectors have the same size as Y, $Height \times Width$.

Because the goal of this exercise is to develop a codec that functions exclusively with grayscale, after extracting the Y, U and V values from the frames, we disregard the U and V values. These U and V values contain color information for the video, and only the luminance (Y) values are retained for further processing or analysis.

The prediction of the next values is done in a similar way as we did when it was just an image. We skip the first pixel of the image. For the rest of the first line, the prediction is based on the value of the pixel to its left. For the pixels in the first column of each row, we predict their values based solely on the pixel directly above them. For the rest of the pixels in the image, we use the values of the neighboring pixels: the one to the left, the one above, and the one diagonally up and to the left.

The prediction, like the previous exercise, is made using the nonlinear predictor of the JPEG-LS, and the result of these calculations are the residual values, stored in the valuesToBeEncoded vector. Same as we did for the images, we

determine the optimal 'm' value based on the analysis of the last line of pixels processed.

```
int pixel_index = 0;

//go pixel by pixel through the Y Mat object to make predictions
for(int i = 0; i < height; i++){
    for(int j = 0; j < width; j++){
        int Y = YMat.at<uchar>(i, j);
        //if its the first pixel of the image, do not use prediction
        if (i == 0 && j == 0) {
            valuesToBeEncoded.push_back(Y);
            pixel_index++;
        } else if(i==0){
            //if the pixel is in the first line of the image, use only the pixel to its left
            valuesToBeEncoded.push_back(Y - YMat.at<uchar>(i, j-1));
            pixel_index++;
        } else if(j==0){
            //if the pixel is the first of the line, use only the pixel above
            valuesToBeEncoded.push_back(Y - YMat.at<uchar>(i-1, j));
            pixel_index++;
        } else {
            //for all other pixels, use 3 pixels (to the left, above and to the top left)
            valuesToBeEncoded.push_back(Y - predict(YMat.at<uchar>(i, j-1), YMat.at<uchar>(i-1, j), YMat.at<uchar>(i-1, j-1)));
            pixel_index++;
        }

        //m_vector calculation
        if (pixel_index % width == 0 and pixel_index != 0) {
            int sum = 0;
            for (int j = pixel_index - width; j < pixel_index; j++) {
                sum += abs(valuesToBeEncoded[j]);
            }

            int p = round(sum / width);

            int m = calc_m(p);

            if (m < 1) m = 1;
```

```
37
38              m_vector.push_back(m);
39          }
40      }
41  }
```

Listing 3.2: Pixel prediction for each frame in intra_encoder.cpp

Once we've computed all the prediction residuals for that frame, we call the encode function, encoding these residual values while considering the current 'm' value specific to the block being processed at that moment. We add each encoded frame to the encoded_bits vector.

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

Listing 3.3: Encoding image values in image_encoder.cpp

At the end of the process, we save the data into a binary file using the BitStream class. Initially, we write all important information into a header to ensure proper decoding in another program. This information will be crutial in decoding the data accurately when needed later on and reconstructing the y4m file.

```
1  BitStream bitStream(output, true);
2
3  bitStream.writeNBits(width, 16); // Write image width (16
       bits)
4  bitStream.writeNBits(height, 16); // Write image height (16
       bits)
```

```
bitStream.writeNBits(numFrames, 16); // Write the number of
    frames (16 bits)
bitStream.writeNBits(pixelAspectRatio[0], 16); // Write the
    first number in the pixel aspect ratio (16 bits)
bitStream.writeNBits(pixelAspectRatio[1], 16); // Write the
    second number in the pixel aspect ratio (16 bits)
bitStream.writeNBits(frameRate[0], 16); // Write the first
    number in the frame rate (16 bits)
bitStream.writeNBits(frameRate[1], 16); // Write the first
    number in the frame rate (16 bits)
bitStream.writeString(interlace); // Write interlace (string
     ->of just one char)
bitStream.writeNBits(blockSize, 16); // Write the block size
    (16 bits)
bitStream.writeNBits(encoded_bits.size(), 32); // Write the
    encoded_bits.size() (32 bits)
bitStream.writeNBits(Ym.size(), 32); // Write the Ym.size()
    (32 bits)


// Write Ym values (8 bits each)
for (size_t i = 0; i < Ym.size(); i++) {
    bitStream.writeNBits(Ym[i], 8);
}

// Write the encoded bits
for (size_t i = 0; i < encoded_bits.size(); i++) {
    bitStream.writeBit(encoded_bits[i]);
}

bitStream.close();
```

Listing 3.4: Saving data in binary file in intra_encoder.cpp

3.2.2 Video Decoder (`intra_decoder.cpp`)

The decoder starts by reading the header in the encoded file. This header contains essential information crucial for reconstructing the video file accurately. After that, it writes a header in the decoded file, specifically formatted for a .y4m file. This header is specifically modified to achieve the grayscale objective, with the color space now altered to "mono".

```
//write the header
out << "YUV4MPEG2 W" << width << " H" << height << " F" <<
    frameRate[0] << ":" << frameRate[1] << " I" << interlace
    << " A" << pixelAspectRatio[0] << ":" << pixelAspectRatio
    [1] << " Cmono" << endl;

//write to the file FRAME
```

```
out << "FRAME" << endl;
```

Listing 3.5: Writing the y4m header in intra_decoder.cpp

Following this, it writes the word "FRAME" to mark the beginning of a frame. Then we read the values of m, and encoded string. After having these two values, they will be used to decode the residual values using the Golomb class.

```
vector<int> Ym;
for (int i = 0; i < Ym_size; i++) {
    int m = bs.readNBits(8); // Read 8 bits for m

    Ym.push_back(m);
}

//read encoded Y values
vector<int> encoded_bits;
for (int i = 0; i < encoded_bits_size; i++) {
    encoded_bits.push_back(bs.readBit());
}

string encodedString = "";
for(long unsigned int i = 0; i < encoded_bits.size(); i++) {
    encodedString += to_string(encoded_bits[i]);
}

//decode Y values
Golomb g;
vector<int> Ydecoded = g.decode(encodedString, Ym, bs_size);
```

Listing 3.6: Decoding video values in intra_decoder.cpp

Once all the readings and decoding procedures are completed, we start with the inverse prediction process to retrieve the original video frames. This inverse prediction procedure adapts according to the specific pixel being processed, mirroring the encoding process's behavior, just like we did in the image codec.

```
//undo the predictions
int pixel_idx = 0;
for (int n = 0; n < num_frames; n++) {
    YMat = Mat(height, width, CV_8UC1);

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i == 0 && j == 0) {
                //create a new pixel with the decoded values
    at the current pixel index and add it to the image
                YMat.at<uchar>(i, j) = Ydecoded[pixel_idx];
                pixel_idx++;
```

```
            } else if (i == 0) {
                //if its the first line of the image, use only the previous pixel (to the left)
                YMat.at<uchar>(i, j) = Ydecoded[pixel_idx] + YMat.at<uchar>(i, j-1);
                pixel_idx++;
            } else if (j == 0) {
                //if its the first pixel of the line, use only the pixel above
                YMat.at<uchar>(i, j) = Ydecoded[pixel_idx] + YMat.at<uchar>(i-1, j);
                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
                YMat.at<uchar>(i, j) = Ydecoded[pixel_idx] + predict(YMat.at<uchar>(i, j-1), YMat.at<uchar>(i-1, j), YMat.at<uchar>(i-1, j-1));
                pixel_idx++;
            }
        }
    }
```

Listing 3.7: Reversing predictions in intra_decoder.cpp

After reversing all the predictions and retrieving the original video frames, the obtained values are orderly written onto the decoded file. After all these Y values have been written, the keyword "FRAME" is written to the file, indicating the end of one frame and the start of the next frame's values in the correct sequential order.

```
//convert the matrix back to a vector
    vector<int> Y_vector;
    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
            Y_vector.push_back(YMat.at<uchar>(i, j));
        }
    }

    //write the Y_vector to the file
    for(long unsigned int i = 0; i < Y_vector.size(); i++){
        //convert the int to a byte
        char byte = (char)Y_vector[i];
        //write the byte to the file
        out.write(&byte, sizeof(byte));
    }

    if(n < num_frames - 1) out << "FRAME" << endl;
```

Listing 3.8: Writing frame values onto the decoded file in intra_decoder.cpp

3.3 Results

Figure 3.1: Intra Encoder

Figure 3.2: Intra Decoder

Table 3.1: Video Compression Results

Video File Name	**Initial Size (MB)**	**Encoding Time (ms)**	**Decoding Time (ms)**	**Final Size (MB)**	**Compression Ratio (%)**
sign_irene_cif.y4m	82,1	22167.9	23515.2	54,7	33.4
ak1yo_cif.y4m	45.6	10852.7	11815.6	30.4	33.3
bus_cif.y4m	22.8	7244.23	7256.88	15.2	33.3

The video compression results for `sign_irene_cif.y4m` `ak1yo_cif.y4m` and `bus_cif.y4m` are presented in Table3.1. Firstly, the size of the `sign_irene_cif.y4m` video was 82.1 MB. The encoding time was 22167.9 ms, and the decoding time was 23515.2 ms. The final size of the video after compression was 54,7 MB, resulting in a compression ratio of 33.4%.

The initial size of the `ak1yo_cif.y4m` video was 45.6 MB. The encoding time was 10852.7 ms, and the decoding time was 11815.6 ms. The final size of the video after compression was 30.4 MB, resulting in a compression ratio of 33.3%.

Finally for the `bus_cif.y4m` video, the initial size was 22.8 MB. The encoding time was 7244.23 ms, and the decoding time was 7256.88 ms. The final size of the video after compression was 15.2 MB, resulting in a compression ratio of 33.3%.

These results provide a more accurate representation of the compression achieved for both videos. The compression ratio of 33% indicates a reduction in the size of the compressed videos compared to the original sizes. Importantly, it is noteworthy that the quality of the compressed videos remains the same as the original. The lossless nature of the codec ensures that there is no degradation in video quality during the compression process.

The following graph shows the relation between initial size and Encoding time:

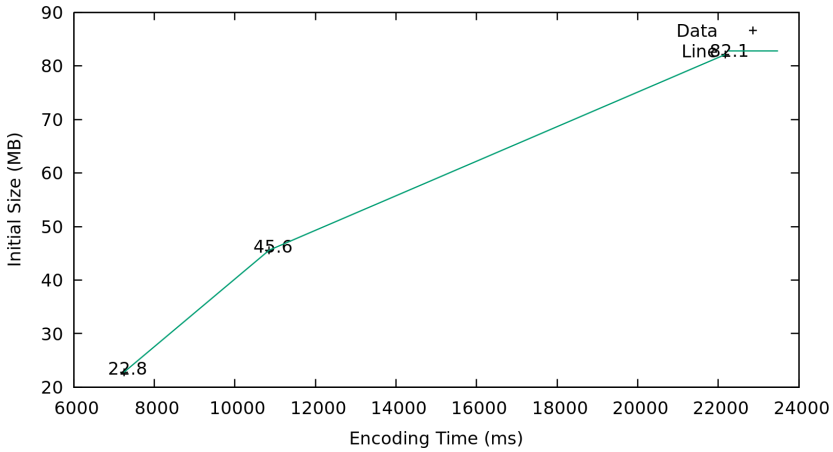

Figure 3.3: Initial size and Encoding time

The following graph shows the relation between Final size and Decoding time:

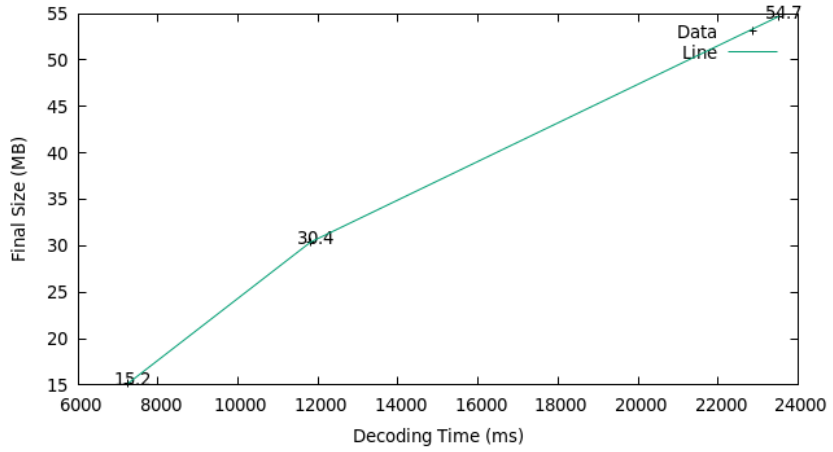

Figure 3.4: Final size and Decoding time