Universidade de Aveiro

Information and Coding

Lab Work nr.3b

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

Departamento de Eletrónica, Telecomunicações e Informática

January 6th, 2023

Contents

List of Figures

Chapter 1

Exercise 1

1.1 Code

We started by creating a class called YUV4MPEGReader.h to get information from the video file we are about to process, mainly extracting the color space, aspect ratio, frame rate and dimensions of the frames.

```
YUV4MPEG2Reader(const std::string& file_name)
    : file_(file_name, std::ios::in | std::ios::binary) {
        std::string line;
        std::getline(file_, line);

        sscanf(line.c_str(), "YUV4MPEG2 W%d H%d F%d:%d", &width_
    , &height_, &frame_rate_1_, &frame_rate_2_);

        if (strchr(line.c_str(), 'C') == NULL)
            color_space_ = "420";
        else
            color_space_ = line.substr(line.find('C') + 1);

        if (strchr(line.c_str(), 'I') == NULL)
            interlace_ = "p";
        else
            interlace_ = line.substr(line.find('I') + 1, 1);

        if (strchr(line.c_str(), 'A') == NULL) {
            aspect_ratio_1_ = 1;
            aspect_ratio_2_ = 1;
        } else {
            std::string aspect_ratio = line.substr(line.find('A'
    ) + 1);
            aspect_ratio_1_ = std::stoi(aspect_ratio.substr(0,
    aspect_ratio.find(':')));
            aspect_ratio_2_ = std::stoi(aspect_ratio.substr(
    aspect_ratio.find(':') + 1));
        }
    }
```

Another thing we did was changing our previously created bitStream.h to increase the performance when writing bits to a file by changing thebitArrayToByte2 function.

```
char bitArrayToByte2(std::vector<int> bitArray){
    char byte = 0;
    for (int i = 0; i < 8; i++) {
        byte += bitArray[i] << (7-i);
    }
    return byte;
}
```

Our **intra-frame** encoder works by reading each frame at a time - by finding the word *"FRAME"* and putting it's info into the correct vectors with the correct sizes (depending on the color space used by the video) -, doing the necessary computations and writting an encoded file at the end with all the necessary information to recreate the video without losing any piece of information.

The last input argument is the block size, this will be used to calculate the optimal value of m, for each *blockSize* residuals.

In the case that the color space is 4:2:0, the size of the frame of the Y channel is $Height + Width$, the U and V are $Height/2 * Width/2$. In the 4:2:2 color space, the frame size for Y channel is $Height + Width$, the U and V are $Height + Width/2$, and for the 4:4:4 color space, Y, U and V frame have the same size - $Height * Width$.

```
while (!feof(input)) {
    fgets(line, 100, input);
    //read the Y data  (Height x Width)
    for(int i = 0; i < width * height; i++){
        Y[i] = fgetc(input);
        if(Y[i] < 0) {
            numFrames--;
            finish = true;
            break;
        }
    }
    ...

    //Create Mat objects for Y, U, and V
    ...

    //copy the Y, U, and V data into the Mat objects for 420,
    422 and 444 Color Space
    ...
```

The prediction of the next values is done in a similar way as if it was just an image. For each of the channels, the predicted values in the first row are only taking in consideration the value on the left. On the first column, they are only based on the values above. Finally, all the others positional possibilities are using the value on the left, above and diagonally up and to the

left. The prediction is made using the nonlinear predictor of the JPEG-LS mode, showed bellow.

$$prediction = \begin{cases} min(A,B), if C >= max(A,B) \\ max(A,B), if C <= min(A,B) \\ A - B + C \end{cases}$$

The result of these calculations are the residual values, each one stored in a vector (*Yresiduals, Cbresiduals e Crresiduals*), and they will be used to calculate the optimal m value for each *blockSize* residual, as well as the respective encoding using Golomb codes.

```
    //go pixel by pixel through the Y, U, and V Mat objects to
    make predictions
    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
//if its the 1st pixel of the image, do not use prediction
            if (i == 0 && j == 0) {
                Yresiduals.push_back(Y);
                Cbresiduals.push_back(U);
                Crresiduals.push_back(V);
            } else if (i==0) {
//if its the 1st line, use only the previous pixel (left)
                Yresiduals.push_back(Y -YMat.at<uchar>(i, j-1));
                ...
            } else if(j==0){
//if its the 1st pixel of the line, use only the pixel above
                Yresiduals.push_back(Y -YMat.at<uchar>(i-1, j));
                ...
            } else {
//otherwise, use the prediction function
                Yresiduals.push_back(Y - predict(YMat.at<uchar>(
    i, j-1), YMat.at<uchar>(i-1, j), YMat.at<uchar>(i-1, j-1)));
                ...
            }
        }
    }
    // Padding to vectores
    if (Yresiduals.size() % blockSize != 0) {
        int padding = blockSize - (Yresiduals.size() % blockSize
    );
        for (int i = 0; i < padding; i++) Yresiduals.push_back
    (0);
    }
    if (Cbresiduals.size() % blockSize != 0) {
        int padding = blockSize - (Cbresiduals.size() %
    blockSize);
        for (int i = 0; i < padding; i++) {
```

```
            Cbresiduals.push_back(0);
            Crresiduals.push_back(0);
        }
    }
    // M Vector Calculation (same as in the previous project)
    ...

    for (long unsigned int i = 0; i < Yresiduals.size(); i++) {
        if (i % blockSize == 0 and i != 0) {
            Ym.push_back(Ym_vector[m_index]);
            m_index++;
        }
        Yencoded += g.encode(Yresiduals[i], Ym_vector[m_index]);
        if (i == Yresiduals.size() - 1) Ym.push_back(Ym_vector[
    m_index]);
    }
    m_index = 0;
    // Cr and Cb optimal M calculation
    ...

    // Create and fill the Ym_vector, Cbm_vector, Crm_vector,
    Yresiduals, Cbresiduals and Crresiduals
    ...
}
...
// write in the header the width, height, no. of frames, color
    space, aspect ratio, frame rate, interlace vector, block size
    , the enconder Y, Cb, Cr, Ym vector size, Cbm vector size,
    Crm vector size and the last three vector values
...
```

After encoding the residuals, the binary file is created. We start by writing an header that contains the necessary information to reconstruct the video file, such as the height, width, number of frames per second, aspect ratio, etc. Besides that, we also write the optimal values of m, as well as all the residuals to reconstruct the frames without losing any information.

On the other end, the decoder starts by reading the header in the encoded file. This header contains all the necessary information to reconstruct the video file without losing information. After that, we write an header in the decoded file for a *.y4m* file and then, the word *"FRAME"* - which indicates the beginning of writing the Y, U and V values for each frame.

Then we read the values of m, as well as the YUV for each vector. After having these two values, they will be used to decode the residual values using the Golomb class.

```
    //write the header on the output file
    //write the keyword "FRAME"
    ...

    //Reading M values
    vector<int> Ym;
```

```
    for(int i = 0; i < Ym_size; i++){
        vector<int> v_Ym = bs.readBits(8);
        int Ym_i = 0;
        for (long unsigned int j = 0; j < v_Ym.size(); j++) Ym_i
    += v_Ym[j] * pow(2, v_Ym.size() - j - 1);
        Ym.push_back(Ym_i);
    }

    vector<int> Cbm;
    for(int i = 0; i < Cbm_size; i++){
        vector<int> v_Cbm = bs.readBits(8);
        int Cbm_i = 0;
        for (long unsigned int j = 0; j < v_Cbm.size(); j++)
    Cbm_i += v_Cbm[j] * pow(2, v_Cbm.size() - j - 1);
        Cbm.push_back(Cbm_i);
    }

    vector<int> Crm;
    for(int i = 0; i < Crm_size; i++){
        vector<int> v_Crm = bs.readBits(8);
        int Crm_i = 0;
        for (long unsigned int j = 0; j < v_Crm.size(); j++)
    Crm_i += v_Crm[j] * pow(2, v_Crm.size() - j - 1);
        Crm.push_back(Crm_i);
    }

    //Reading YUV values
    ...

    string Yencodedstring = "";
    for(long unsigned int i = 0; i < Ybits.size(); i++)
    Yencodedstring += Ybits[i] + '0';
    string Cbencodedstring = "";
    string Crencodedstring = "";
    for(long unsigned int i = 0; i < Cbbits.size(); i++)
        Cbencodedstring += Cbbits[i] + '0';
    for(long unsigned int i = 0; i < Crbits.size(); i++)
        Crencodedstring += Crbits[i] + '0';

    //DECODE YUV VALUES
    Golomb g;
    vector<int> Ydecoded = g.decodeMultiple(Yencodedstring, Ym,
    bs_size);
    vector<int> Cbdecoded = g.decodeMultiple(Cbencodedstring,
    Cbm, bs_size);
    vector<int> Crdecoded = g.decodeMultiple(Crencodedstring,
    Crm, bs_size);

    Mat YMat = Mat(height, width, CV_8UC1);
    Mat UMat;
    Mat VMat;

    if (color_space == 420) {
        UMat = Mat(height/2, width/2, CV_8UC1);
```

```
        VMat = Mat(height/2, width/2, CV_8UC1);
    } else if (color_space == 422) {
        UMat = Mat(height, width/2, CV_8UC1);
        VMat = Mat(height, width/2, CV_8UC1);
    } else if (color_space == 444) {
        UMat = Mat(height, width, CV_8UC1);
        VMat = Mat(height, width, CV_8UC1);
    }
```

After all the readings are done, alongside with the decoding process we do the inverse prediction process of the values, which is done in order to obtain the original video frames. The inverse process varies according to which pixel is being used, as it's done on the encoding side. It is important to note that the inverse prediction process varies slightly depending on the color space in which we are working on, as there are U and V pixels that exist in certain positions of the frame or not depending on the color space.

```
//undo the predictions
int pixel_idx = 0;
int pixel_idx2 = 0;
for (int n = 0; n < num_frames; n++) {
    YMat = Mat(height, width, CV_8UC1);
    if (color_space == 420) {
        UMat = Mat(height/2, width/2, CV_8UC1);
        VMat = Mat(height/2, width/2, CV_8UC1);
    } else if (color_space == 422) {
        UMat = Mat(height, width/2, CV_8UC1);
        VMat = Mat(height, width/2, CV_8UC1);
    } else if (color_space == 444) {
        UMat = Mat(height, width, CV_8UC1);
        VMat = Mat(height, width, CV_8UC1);
    }

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i == 0 && j == 0) {
                YMat.at<uchar>(i, j) = Ydecoded[pixel_idx];
                UMat.at<uchar>(i,j) = Cbdecoded[pixel_idx2];
                VMat.at<uchar>(i,j) = Crdecoded[pixel_idx2];
                pixel_idx++;
                pixel_idx2++;
            } else if (i == 0) {
                YMat.at<uchar>(i, j) = Ydecoded[pixel_idx] +
    YMat.at<uchar>(i, j-1);
                if (color_space == 420 || color_space == 422) {
                    if (j < (width/2)) {
                        UMat.at<uchar>(i,j) = Cbdecoded[
    pixel_idx2] + UMat.at<uchar>(i, j-1);
                        VMat.at<uchar>(i,j) = Crdecoded[
    pixel_idx2] + VMat.at<uchar>(i, j-1);
                        pixel_idx2++;
                    }
                } else if (color_space == 444) {
```

```
34                      UMat.at<uchar>(i,j) = Cbdecoded[pixel_idx2]
    + UMat.at<uchar>(i, j-1);
35                      VMat.at<uchar>(i,j) = Crdecoded[pixel_idx2]
    + VMat.at<uchar>(i, j-1);
36                      pixel_idx2++;
37                  }
38                  pixel_idx++;
39              } else if (j == 0) {
40                  YMat.at<uchar>(i, j) = Ydecoded[pixel_idx] +
    YMat.at<uchar>(i-1, j);
41                  if (color_space == 420) {
42                      if (i < (height/2)) {
43                          UMat.at<uchar>(i,j) = Cbdecoded[
    pixel_idx2] + UMat.at<uchar>(i-1, j);
44                          VMat.at<uchar>(i,j) = Crdecoded[
    pixel_idx2] + VMat.at<uchar>(i-1, j);
45                          pixel_idx2++;
46                      }
47                  } else if(color_space == 422 || color_space ==
    444){
48                      UMat.at<uchar>(i,j) = Cbdecoded[pixel_idx2]
    + UMat.at<uchar>(i-1, j);
49                      VMat.at<uchar>(i,j) = Crdecoded[pixel_idx2]
    + VMat.at<uchar>(i-1, j);
50                      pixel_idx2++;
51                  }
52                  pixel_idx++;
53              } else {
54                  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));
55                  if(color_space == 420){
56                      if(i < (height/2) && j < (width/2)){
57                          UMat.at<uchar>(i,j) = Cbdecoded[
    pixel_idx2] + predict(UMat.at<uchar>(i, j-1), UMat.at<uchar>(
    i-1, j), UMat.at<uchar>(i-1, j-1));
58                          VMat.at<uchar>(i,j) = Crdecoded[
    pixel_idx2] + predict(VMat.at<uchar>(i, j-1), VMat.at<uchar>(
    i-1, j), VMat.at<uchar>(i-1, j-1));
59                          pixel_idx2++;
60                      }
61                  } else if(color_space == 422){
62                      if(j < (width/2)){
63                          UMat.at<uchar>(i,j) = Cbdecoded[
    pixel_idx2] + predict(UMat.at<uchar>(i, j-1), UMat.at<uchar>(
    i-1, j), UMat.at<uchar>(i-1, j-1));
64                          VMat.at<uchar>(i,j) = Crdecoded[
    pixel_idx2] + predict(VMat.at<uchar>(i, j-1), VMat.at<uchar>(
    i-1, j), VMat.at<uchar>(i-1, j-1));
65                          pixel_idx2++;
66                      }
67                  } else if(color_space == 444){
68                      UMat.at<uchar>(i,j) = Cbdecoded[pixel_idx2]
    + predict(UMat.at<uchar>(i, j-1), UMat.at<uchar>(i-1, j),
```

```
    UMat.at<uchar>(i-1, j-1));
                    VMat.at<uchar>(i,j) = Crdecoded[pixel_idx2]
    + predict(VMat.at<uchar>(i, j-1), VMat.at<uchar>(i-1, j),
    VMat.at<uchar>(i-1, j-1));
                    pixel_idx2++;
                }
                pixel_idx++;
            }
        }
    }
```

After reverting all the predictions and obtaining the frames of the original video, the values are written on the decoded file in the correct order (first Y, than U and V), with the keyword *"FRAME"* being written after the values.

```
//convert the matrix back to a vector
vector<int> Y_vector;
vector<int> Cb_vector;
vector<int> Cr_vector;
for(int i = 0; i < height; i++){
    for(int j = 0; j < width; j++){
        Y_vector.push_back(YMat.at<uchar>(i, j));
    }
    if(color_space == 420){
        if (i < height/2 && i < width/2) {
            for(int j = 0; j < width/2; j++){
                Cb_vector.push_back(UMat.at<uchar>(i, j));
                Cr_vector.push_back(VMat.at<uchar>(i, j));
            }
        }
    } else if(color_space == 422){
        if (i < width/2) {
            for(int j = 0; j < width/2; j++){
                Cb_vector.push_back(UMat.at<uchar>(i, j));
                Cr_vector.push_back(VMat.at<uchar>(i, j));
            }
        }
    } else if(color_space == 444){
        for(int j = 0; j < width; j++){
            Cb_vector.push_back(UMat.at<uchar>(i, j));
            Cr_vector.push_back(VMat.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));
}

//write the Cb_vector to the file
```

```
40 for(long unsigned int i = 0; i < Cb_vector.size(); i++){
41     //convert the int to a byte
42     char byte = (char)Cb_vector[i];
43     //write the byte to the file
44     out.write(&byte, sizeof(byte));
45 }
46
47 //write the Cr_vector to the file
48 for(long unsigned int i = 0; i < Cr_vector.size(); i++){
49     //convert the int to a byte
50     char byte = (char)Cr_vector[i];
51     //write the byte to the file
52     out.write(&byte, sizeof(byte));
53 }
54 if(n < num_frames - 1) out << "FRAME" << endl;
```

1.2 Usage

The usage of the program is done following this pattern:

```
1 ./intraframe_encoder <intput_file> <output_file> <bs>
```

Where the input file is a video file (tested with .y4m), the output file is a binary file (without extension or .txt) and the bs is an integer.

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

```
1 ./intraframe_decoder <intput_file> <output_file>
```

Where the input file is the binary file used before and the output file is a video file (tested with .y4m).

1.3 Results

Video file size (MB)		
garden_sif.y4m	students_qcif.y4m	akyio_cif.y4m
14.57	38.29	45.62

Table 1.1: Video files size in Megabytes

garden_sif.y4m			
Block Size	Time (ms)	Out size (MB)	Compress. ratio (%)
16	2381	11.51	21.0
32	2356	11.14	23.5
64	2515	11.01	24.5
128	2452	10.96	24.8
256	2587	10.99	24.6
352	2510	10.96	24.8
students_qcif.y4m			
16	5591	25.06	34.6
32	5894	24.18	36.9
64	5936	23.64	38.3
128	5774	23.55	38.5
176	5677	23.45	38.8
akyio_cif.y4m			
16	5153	25.99	43.0
32	5261	24.71	45.8
64	5080	24.08	47.2
128	4945	23.79	47.9
176	5263	23.56	48.4
352	6590	23.58	48.3

Table 1.2: Intra-frame encode time and compression ratio

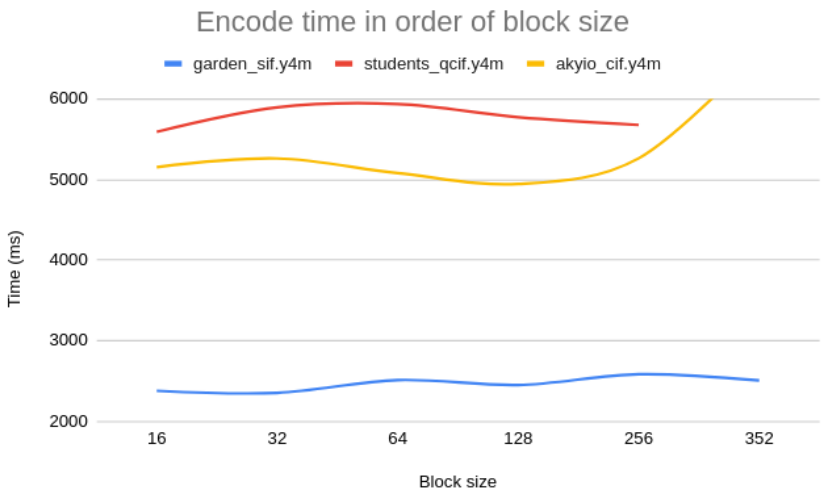

Figure 1.1: Encode time performance vs block size

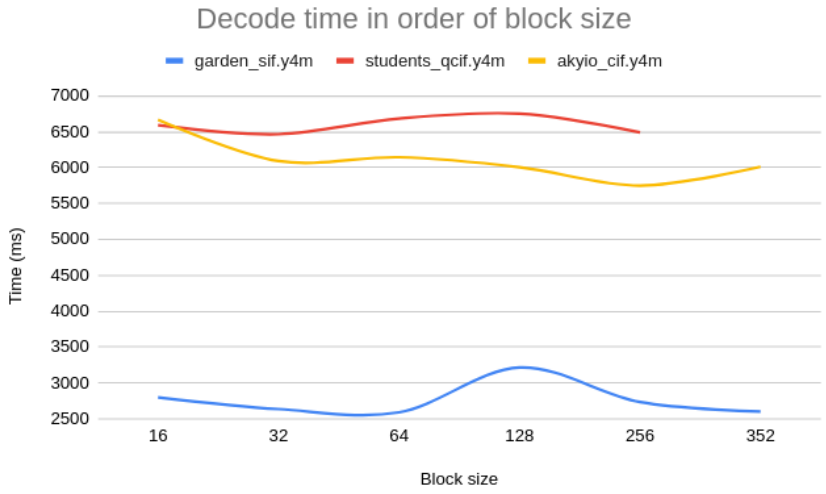

Figure 1.2: Decode time performance vs block size

garden_sif.y4m	
Block Size	Time (ms)
16	2800
32	2635
64	2590
128	3214
256	2733
352	2602
students_qcif.y4m	
16	6591
32	6467
64	6685
128	6752
176	6491
akyio_cif.y4m	
16	6669
32	6093
64	6146
128	6006
176	5751
352	6012

Table 1.3: Intra-frame decode time ratio

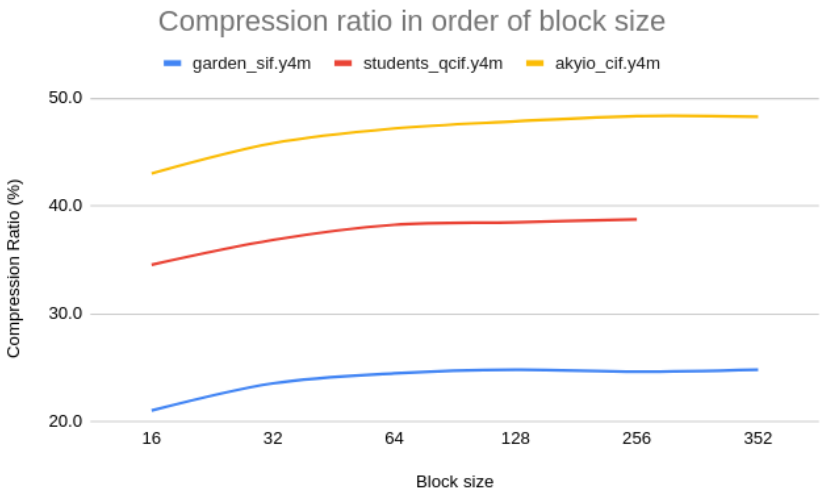

Figure 1.3: Compression ratio vs block size

A trend we can clearly observe is that, generally, as the video file size increases, so does the time to encode and decode it, this is expected because a bigger file has more frames, thus the iteration process will be longer. However, the increase on the time is not linear, we see that students_qcif is smaller than akyio_cif but it takes more time to encode and decode. This can happen because either students_qcif contains more complex or less repetitive information, which will lead to more complex calculations, or akyio_cif has very low complexity/heavy repetitive frames.

We can see that as the block size increases, we achieve a greater compression ratio, noting that the best block size is the one equal to the width. In general, smaller block sizes may result in more accurate representations of the original data, but may also result in larger sizes due to the increased number of residuals that need to be quantized and encoded. Larger block sizes may result in coarser representations of the data, but may also result in smaller file sizes due to the reduced number of residuals.

We also noted that as the video file size increases, the compression ratio also increases, this is to be expected because larger videos tend to contain more redundancy that can be exploited by the compression algorithm, leading to more efficient compression. It is also worth to bear in mind that the compression ratio will depend on the video's content and may also vary across different types of video formats which weren't tested in this project and report.

Chapter 2

Exercise 2

2.1 Code

Similar to the **intra-frame** encoder, we start by verifying all the user inputs. After that, the encoder works by reading each frame at a time - by finding the word "FRAME" and putting it's info into the correct vectors with the correct sizes -, having in mind that this **inter-frame** encoder only supports the 4:2:0 color space.

```
1  //reading frames
2  while(!feof(input)){
3      numFrames++;
4      //read the frame line
5      fgets(line, 100, input);
6      //read the Y data  (Height x Width)
7      for(int i = 0; i < width * height; i++){
8          Y[i] = fgetc(input);
9          if(Y[i] < 0) {
10             numFrames--;
11             finish = true;
12             break;
13         }
14     }
15     if (finish) break;
16     //U AND V DATA
17     for(int i = 0; i < width * height / 4; i++) U[i] = fgetc(
       input); //read the U data (Height/2 x Width/2)
18     for(int i = 0; i < width * height / 4; i++) V[i] = fgetc(
       input); //read the V data (Height/2 x Width/2)
19     (...)
```

The initial part of reading the input file and gathering all the information to the YUV vectors is the same as the **intra-frame** encoder, however in this program we have two distinct ways to predict future values. Given to the fact that one of the input variables is the *Keyframe period*, every *keyframe* frames encoded, we encode the next frame using **intra-frame** prediction (we only use **intra-frame** prediction for the *keyframes*). This works well

for several reasons, namely because **inter-frame** prediction uses another frame to predict the values of the current frame and this can create very good results, specially in videos without much movement. That other frame is the *keyframe* and this method gives the user flexibility to choose for how long frames are encoded with the same *keyframe* before going to another *keyframe*.

To do that, we save the *keyframes* into matrices to be used by the inter-frame prediction algorithm.

```
(...)
//KEYFRAME SAVING
//if its the first frame, set it as the keyframe and copy the data into the keyframe Mat objects
if(frameIndex==0){
    keyYmat = YMat.clone();
    keyUmat = UMat.clone();
    keyVmat = VMat.clone();
} //else if the current keyFrame is a multiple of keyFramePeriod, set the current frame as the keyframe and copy the data into the keyframe Mat objects
else if(frameIndex % keyFramePeriod == 0){
    keyYmat = YMat.clone();
    keyUmat = UMat.clone();
    keyVmat = VMat.clone();
}
//PREDICTION
//INTRA-FRAME PREDICTION (keyFrame)
//if its the first frame, or if the current frame is a keyframe, use intra-frame prediction
if(frameIndex==0 || (frameIndex % keyFramePeriod==0)){
    //go pixel by pixel through the Y, U, and V Mat objects to make predictions
    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
            int Y = YMat.at<uchar>(i, j);
            int U = UMat.at<uchar>(i, j);
            int V = VMat.at<uchar>(i, j);
            int Yerror = 0;
            int Uerror = 0;
            int Verror = 0;
            //if its the first pixel of the image, do not use prediction
            if (i == 0 && j == 0) {
                Yerror = Y;
                Uerror = U;
                Verror = V;
                Yresiduals.push_back(Yerror);
                Cbresiduals.push_back(Uerror);
                Crresiduals.push_back(Verror);
            } else if(i==0){
            (...)
```

For all the other frames, we use inter-frame prediction. Inter-frame prediction is a technique used in video encoding to reduce the amount of data that needs to be transmitted or stored. It works by taking advantage of the fact that many frames in a video will have some similarity to nearby frames.

The idea is to use information from a reference frame, called the keyframe, to predict the content of other frames, called prediction frames. The prediction is done by dividing the frames into small blocks and searching for the best matching block in the keyframe. The algorithm takes into account two other variables passed as inputs of the program, the *blockSize* and the *searchArea*.

The search for the best matching block works by dividing the current frame in squares of *blockSize* per *blockSize* pixels and search the given block in the *keyframe* within a specific *searchArea* (the area to search is *searchArea* pixels to the left, right, top and bottom of the position where the current block is, keeping in mind the boundaries of the frame). Our codec supports any values for the blockSize because we do the necessary padding to the frame.

```
//INTER-FRAME PREDICTION (non-keyFrame, block by block)
//motion compensation (block by block) with the keyframe
//go block by block through the Y, U, and V Mat objects to make predictions
//for each block calculate the motion vector

//calculate the padding, check if width and height are divisible by block size
int padded_width = width;
int padded_height = height;
if (width % blockSize != 0) {
    padded_width = width + (blockSize - (width % blockSize));
}
if (height % blockSize != 0) {
    padded_height = height + (blockSize - (height % blockSize));
}

//keyFrame Mat is saved in keyYmat, keyUmat, keyVmat

//first thing to do is to create a Mat object for all of the frame, combining the Y, U and V Mat objects
Mat frame = Mat(padded_height, padded_width, CV_8UC3);
Mat keyFrameMat = Mat(padded_height, padded_width, CV_8UC3);
for (int i = 0; i < padded_height; i++){
    for (int j = 0; j < padded_width; j++){
        int half_i = i/2;
        int half_j = j/2;
        frame.at<Vec3b>(i, j)[0] = YMat.at<uchar>(i, j);
        frame.at<Vec3b>(i, j)[1] = UMat.at<uchar>(half_i, half_j);
        frame.at<Vec3b>(i, j)[2] = VMat.at<uchar>(half_i, half_j);
        keyFrameMat.at<Vec3b>(i, j)[0] = keyYmat.at<uchar>(i, j)
```

```
    ;
29          keyFrameMat.at<Vec3b>(i, j)[1] = keyUmat.at<uchar>(
    half_i, half_j);
30          keyFrameMat.at<Vec3b>(i, j)[2] = keyVmat.at<uchar>(
    half_i, half_j);
31      }
32  }
```

The criteria to choose if the best block has been found is to do the sum of absolute differences (SAD) with the block from the prediction frame and the reference frame. The SAD is calculated by summing the absolute differences between corresponding pixels in the two blocks. When we find the best block, we calculate the motion vectors by subtracting the current position of the block with the position of the best block in the *keyframe*.

```
1  //current Y, Cb and Cr block
2              Mat frameBlock = Mat(blockSize, blockSize, CV_8UC3);
3
4              int num_blocks_width = padded_width/blockSize;
    //number of blocks on the width of the whole frame
5              int num_blocks_height = padded_height/blockSize;
    //number of blocks on the height of the whole frame
6              // cout << num_blocks_height << " " <<
    num_blocks_width << endl;
7              int block_index = 0;     //index of the current block
8
9              //motion vector calculation
10             for (int bw = 0; bw < num_blocks_width; bw++){
11                 for(int bh = 0; bh < num_blocks_height; bh++){
12                     //copy the pixels of the current frame
13                     for (int i = 0; i < blockSize; i++){
14                         for (int j = 0; j < blockSize; j++){
15                             frameBlock.at<Vec3b>(i, j)[0] =
    frame.at<Vec3b>(bh*blockSize + i, bw*blockSize + j)[0];
16                             frameBlock.at<Vec3b>(i, j)[1] =
    frame.at<Vec3b>(bh*blockSize + i, bw*blockSize + j)[1];
17                             frameBlock.at<Vec3b>(i, j)[2] =
    frame.at<Vec3b>(bh*blockSize + i, bw*blockSize + j)[2];
18                         }
19                     }
20
21                     //calculate the motion vector
22                     int motionVectorX = 0;
23                     int motionVectorY = 0;
24
25                     //the area to search is the position of the
    block in the keyframe + searchDistance (in both directions -
    left, right, up and down)
26                     //searchDistance is the number of pixels to
    search in each direction
27                     //the first block is bw = 0, bh = 0, so the
    search area is from (0,0) to (blockSize + searchDistance,
    blockSize + searchDistance)
```

```
28                  //the last block is bw = num_blocks_width -
    1, bh = num_blocks_height - 1, so the search area is from (
    width - blockSize - searchDistance, height - blockSize -
    searchDistance) to (width - blockSize, height - blockSize)
29                  int searchAreaTopX = bw*blockSize -
    searchDistance;
30                  int searchAreaTopY = bh*blockSize -
    searchDistance;
31                  int searchAreaBottomX = bw*blockSize +
    blockSize + searchDistance;
32                  int searchAreaBottomY = bh*blockSize +
    blockSize + searchDistance;
33                  if(searchAreaTopX < 0) searchAreaTopX = 0;
34                  if(searchAreaTopY < 0) searchAreaTopY = 0;
35                  if(searchAreaBottomX > width)
    searchAreaBottomX = width;
36                  if(searchAreaBottomY > height)
    searchAreaBottomY = height;
37                  //calculate the sum of the absolute
    differences between the current block and the blocks in the
    search area
38                  int minSum = 1000000000;
39                  for (int i = bh*blockSize - searchDistance;
    i <= bh*blockSize + searchDistance; i++){
40                      for (int j = bw*blockSize -
    searchDistance; j <= bw*blockSize + searchDistance; j++){
41                          if(i < 0 || j < 0 || i + blockSize >
     padded_height || j + blockSize > padded_width) continue;
42                          Mat ref_block = keyFrameMat(Rect(j,
    i, blockSize, blockSize));
43                          int sum = 0;
44                          for (int k = 0; k < blockSize; k++){
45                              for (int l = 0; l < blockSize; l
    ++){
46                                  sum += abs(frameBlock.at<
    uchar>(k,l) - ref_block.at<uchar>(k,l));
47                              }
48                          }
49                          if (sum < minSum){
50                              minSum = sum;
51                              motionVectorX = j - bw*blockSize
    ;
52                              motionVectorY = i - bh*blockSize
    ;
53                          }
54                      }
55                  }
56                  //save the motion vector
57                  motionVectorXs.push_back(motionVectorX);
58                  motionVectorYs.push_back(motionVectorY);
59                  (...)
```

After finding the motion vector for each block, we save those vectors in two global variables (*motionVectorXs and motionVectorYs* to be used in the

decoder. Now, all we need to do is predict the next value by subtracting the current value of the pixel with the value of the pixel in the position given by the motion vector (keeping in mind that we do this to all the 3 channels of the frame):

```
                (...)
                //predict the current block using the motion vector
                for (int i = 0; i < blockSize; i++){ //height
                    for (int j = 0; j < blockSize; j++){ //width
                        int errorY = frame.at<Vec3b>(bh*blockSize + i, bw*blockSize + j)[0] - keyFrameMat.at<Vec3b>(bh*blockSize + i + motionVectorY, bw*blockSize + j + motionVectorX)[0];
                        int errorCb = frame.at<Vec3b>(bh*blockSize + i, bw*blockSize + j)[1] - keyFrameMat.at<Vec3b>(bh*blockSize + i + motionVectorY, bw*blockSize + j + motionVectorX)[1];
                        int errorCr = frame.at<Vec3b>(bh*blockSize + i, bw*blockSize + j)[2] - keyFrameMat.at<Vec3b>(bh*blockSize + i + motionVectorY, bw*blockSize + j + motionVectorX)[2];
                        Yresiduals.push_back(errorY);
                        //only save the Cb and Cr residuals every 2 pixels
                        if (i % 2 == 0 and j % 2 == 0){
                            Cbresiduals.push_back(errorCb);
                            Crresiduals.push_back(errorCr);
                        }
                    }
                }
```

At the end, we have the 3 vectors containing the residuals of all the predicted frames, either using **inter-frame** or **intra-frame** prediction. Now, doing everything similarly to the **intra-frame** encoder, we calculate the best values for the m quotient using the Golomb coding. Then, we encode those values with those optimal m and save all the necessary information to the binary file, starting by the header that contains all the information to recreate the original video file, as well as all the motion vectors, m values and all the residuals.

The decoder computes the original frames by doing the inverse of the encoding process. In a similar way to the **intra-frame** decoder, we start by reading all the information from the header, as well as all the values of the motion vectors, optimal m values and reading and decoding the residuals stored in the binary file. At the end, we have a video file equal to the original one.

After reading the header, we need to make sure we open a new writable file and proceed to write the correct header, as well as the keyword *"FRAME"* to indicate that the following information stored corresponds to the values

of the Y, U and V channels.

```
//The output file is a YUV4MPEG2 file
//write the header
ofstream out(output_file, ios::out | ios::binary);

//write the header
out << "YUV4MPEG2 W" << width << " H" << height << " F" <<
    frame_rate_1 << ":" << frame_rate_2 << " I" << interlace << "
     A" << aspect_ratio_1 << ":" << aspect_ratio_2 << endl;

//write to the file FRAME
out << "FRAME" << endl;
```

What we have now is the number of frames of the video file, all the residuals and the values of the motion vectors to apply to each block of each frame. Similarly to the encoder, we go through all the frames and detect if it's either a *keyframe* or not, and do the relative predictions (**inter-frame** or **intra-frame**). If it's a *keyframe*, we decode by reversing the algorithm used in the **intra-frame** encoder, and we save that frame to be used in the **inter-frame** decoding process:

```
//FOR EACH FRAME
for (int n = 0; n < num_frames; n++) {

    //PREDICTION
    //INTRA-FRAME PREDICTION (keyFrame)
    //if its the first frame, or if the current frame is a
    keyframe, do not use inter frame prediction
    if(n == 0 || (n % keyFramePeriod == 0)){
        UMat = Mat(height/2, width/2, CV_8UC1);
        VMat = Mat(height/2, width/2, CV_8UC1);

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (i == 0 && j == 0) {
                    YMat.at<uchar>(i,j) = Ydecoded[pixel_idx];
                    UMat.at<uchar>(i,j) = Cbdecoded[pixel_idx2];
                    VMat.at<uchar>(i,j) = Crdecoded[pixel_idx2];
                    pixel_idx2++;
                } else if (i == 0) {
                    YMat.at<uchar>(i, j) = Ydecoded[pixel_idx] +
     YMat.at<uchar>(i, j-1);
                    if (j < (width/2)) {
                        UMat.at<uchar>(i,j) = Cbdecoded[
    pixel_idx2] + UMat.at<uchar>(i, j-1);
                        VMat.at<uchar>(i,j) = Crdecoded[
    pixel_idx2] + VMat.at<uchar>(i, j-1);
                        pixel_idx2++;
                    }
                } else if (j == 0) {
                    YMat.at<uchar>(i, j) = Ydecoded[pixel_idx] +
     YMat.at<uchar>(i-1, j);
                    if (i < (height/2)) {
```

```
                        UMat.at<uchar>(i,j) = Cbdecoded[
    pixel_idx2] + UMat.at<uchar>(i-1, j);
                        VMat.at<uchar>(i,j) = Crdecoded[
    pixel_idx2] + VMat.at<uchar>(i-1, j);
                        pixel_idx2++;
                    }
                } else {
                    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));
                    if(i < (height/2) && j < (width/2)){
                        UMat.at<uchar>(i,j) = Cbdecoded[
    pixel_idx2] + predict(UMat.at<uchar>(i, j-1), UMat.at<uchar>(
    i-1, j), UMat.at<uchar>(i-1, j-1));
                        VMat.at<uchar>(i,j) = Crdecoded[
    pixel_idx2] + predict(VMat.at<uchar>(i, j-1), VMat.at<uchar>(
    i-1, j), VMat.at<uchar>(i-1, j-1));
                        pixel_idx2++;
                    }
                }
                pixel_idx++;
            }
        }
        //KEYFRAME SAVING
        keyYmat = YMat.clone();
        keyUmat = UMat.clone();
        keyVmat = VMat.clone();
```

Now, every time we get back an original frame, we save it to the output file after converting the matrix back to a vector (for each channel of the frame):

```
        //convert the matrix back to a vector
        vector<int> Y_vector;
        vector<int> Cb_vector;
        vector<int> Cr_vector;
        for(int i = 0; i < height; i++){
            for(int j = 0; j < width; j++){
                Y_vector.push_back(YMat.at<uchar>(i, j));
            }
            if(color_space == 420){
                if (i < height/2 && i < width/2) {
                    for(int j = 0; j < width/2; j++){
                        Cb_vector.push_back(UMat.at<uchar>(i, j)
    );
                        Cr_vector.push_back(VMat.at<uchar>(i, j)
    );
                    }
                }
            } else if(color_space == 422){
                if (i < width/2) {
                    for(int j = 0; j < width/2; j++){
                        Cb_vector.push_back(UMat.at<uchar>(i, j)
    );
```

```
                        Cr_vector.push_back(VMat.at<uchar>(i, j)
    );
                    }
                }
            } else if(color_space == 444){
                for(int j = 0; j < width; j++){
                    Cb_vector.push_back(UMat.at<uchar>(i, j));
                    Cr_vector.push_back(VMat.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));
        }

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

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

For the frames that require inter-frame prediction, the process to get back the original pixel values is to revert the process done by the encoder, considering the fact that now, we have the residuals, the motion vectors and the *keyframes* decoded:

```

    //INTER-FRAME PREDICTION (non-keyFrame, block by block)
    //motion compensation (block by block) with the keyframe
    //go block by block through the Y, U, and V Mat object to
    make predictions
    YMat = Mat(padded_height, padded_width, CV_8UC1);
    UMat = Mat(padded_height/2, padded_width/2, CV_8UC1);
    VMat = Mat(padded_height/2, padded_width/2, CV_8UC1);

    //FRAME CONSTRUCTION
    Mat keyFrameMat = Mat(padded_height, padded_width, CV_8UC3);
    //colorspace 420
    for (int i = 0; i < padded_height; i++){
```

```
13          for (int j = 0; j < padded_width; j++){
14              int half_i = i/2;
15              int half_j = j/2;
16              keyFrameMat.at<Vec3b>(i, j)[0] = keyYmat.at<uchar>(i
    , j);
17              keyFrameMat.at<Vec3b>(i, j)[1] = keyUmat.at<uchar>(
    half_i, half_j);
18              keyFrameMat.at<Vec3b>(i, j)[2] = keyVmat.at<uchar>(
    half_i, half_j);
19          }
20      }
21
22      int num_blocks_width = padded_width/blockSize;      //number
    of blocks on the width of the whole frame
23      int num_blocks_height = padded_height/blockSize;    //number
    of blocks on the height of the whole frame
24      for (int bw = 0; bw < num_blocks_width; bw++){
25          for(int bh = 0; bh < num_blocks_height; bh++){
26              for (int i = 0; i < blockSize; i++){
27                  for (int j = 0; j < blockSize; j++){
28                      //Getting the current block (predicted)
29                      YMat.at<uchar>(bh*blockSize + i, bw*
    blockSize + j) = keyFrameMat.at<Vec3b>(bh*blockSize + i +
    motionVectorYs[motionY_idx], bw*blockSize + j +
    motionVectorXs[motionX_idx])[0] + Ydecoded[pixel_idx];
30                      if(n==7){
31                          if ((bh*blockSize + i + motionVectorYs[
    motionY_idx]) > padded_height || (bh*blockSize + i +
    motionVectorYs[motionY_idx]) < 0){
32                              cerr << "Error: Motion vector[X] out
     of bounds [0-" << padded_height << "]: " << (bh*blockSize +
    i + motionVectorYs[motionY_idx]) << endl;
33                              return -1;
34                          }
35                          if ((bw*blockSize + j + motionVectorXs[
    motionX_idx]) > padded_width || (bw*blockSize + j +
    motionVectorXs[motionX_idx]) < 0){
36                              cerr << "Error: Motion vector[Y] out
     of bounds [0-" << padded_width << "]: " << (bw*blockSize + j
     + motionVectorXs[motionX_idx]) << endl;
37                              return -1;
38                          }
39                      }
40                      pixel_idx++;
41                      if (i % 2 == 0 and j % 2 == 0){
42                          UMat.at<uchar>((bh*blockSize + i)/2, (bw
    *blockSize + j)/2) = keyFrameMat.at<Vec3b>(bh*blockSize + i +
     motionVectorYs[motionY_idx], bw*blockSize + j +
    motionVectorXs[motionX_idx])[1] + Cbdecoded[pixel_idx2];
43                          VMat.at<uchar>((bh*blockSize + i)/2, (bw
    *blockSize + j)/2) = keyFrameMat.at<Vec3b>(bh*blockSize + i +
     motionVectorYs[motionY_idx], bw*blockSize + j +
    motionVectorXs[motionX_idx])[2] + Crdecoded[pixel_idx2];
44                          pixel_idx2++;
```

```
                    }

                }
            }
            motionY_idx++;
            motionX_idx++;
        }
```

At the end of the **inter-frame** decoding, we also convert the values of the matrix back to a vector and save each channel into the output file in the correct order. When the program ends, the result is a video file exactly the same as the original one, even thought it went through a lossless codec that compressed the file considerably.

2.2 Usage

The usage of the program is done following this pattern:

```
./interframe_encoder <intput_file> <output_file> <bs> <search
    area> <key-frame period>
```

Where the input file is a video file (tested with .y4m), the output file is a binary file (without extension or .txt) and the blocksize (bs), search area and key-frame period are integers.

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

```
./interframe_decoder <intput_file> <output_file>
```

Where the input file is the binary file used before and the output file is a video file (tested with .y4m).

2.3 Results

We started by testing various combinations and ended up narrowing it down to testing block sizes of 16, 32, 64 and 128, search area values of 4, 8, 16 and 32 and key-frame period values of 3, 5, 10 and 15. A lot more configurations were tested but these were the ones that gave better results. From these tests we saw that the best results of compression for garden_sif.y4m and students_qcif.y4m were achieved when using a block size of 16, and for akyio_cif.y4m the best result was achieved using a block size of 32.

The following tables show the results for the best block size configuration for each video file, the rest of the results can be viewed on the .xlsx file in the project's repository.

KF: Key-Frame Period, SA: Search Area, BS: Block Size

garden_sif.y4m					
KF	SA	BS	Time (ms)	size (MB)	Comp. ratio (%)
3	4	16	2471	10.99	24.6
3	8	16	2673	10.82	25.8
3	16	16	3365	10.80	25.9
3	32	16	6044	10.80	25.9
5	4	16	2600.	11.44	21.5
5	8	16	2789	11.10	23.9
5	16	16	3565	10.92	25.1
5	32	16	6521	10.90	25.2
10	4	16	2737	12.21	16.2
10	8	16	2930	11.76	19.3
10	16	16	3939	11.39	21.8
10	32	16	7510	11.21	23.1
15	4	16	2944	12.59	13.6
15	8	16	2981	12.22	16.2
15	16	16	3997	11.74	19.4
15	32	16	7629	11.48	21.2
students_qcif.y4m					
3	4	16	4989	21.97	42.6
3	8	16	5384	21.97	42.6
3	16	16	7088	21.97	42.6
3	32	16	12563	21.97	42.6
5	4	16	4886	21.78	43.1
5	8	16	5698	21.79	43.1
5	16	16	7647	21.79	43.1
5	32	16	14378	21.79	43.1
10	4	16	5143	22.05	42.4
10	8	16	5989	22.06	42.4
10	16	16	7985	22.06	42.4
10	32	16	15243	22.07	42.4
15	4	16	5009.	22.39	41.5
15	8	16	5801	22.39	41.5
15	16	16	8582	22.40	41.5
15	32	16	15688	22.40	41.5

akyio_cif.y4m					
KF	SA	BS	Time (ms)	Out size (MB)	Comp. ratio (%)
3	4	32	4438	22.16	51.4
3	8	32	4793	22.16	51.4
3	16	32	5767	22.16	51.4
3	32	32	9633	22.16	51.4
5	4	32	4522	21.85	52.1
5	8	32	4775	21.85	52.1
5	16	32	5802	21.85	52.1
5	32	32	10032	21.85	52.1
10	4	32	4468	21.80	52.2
10	8	32	4728	21.78	52.3
10	16	32	6088	21.78	52.3
10	32	32	10772	21.78	52.3
15	4	32	4432	21.98	51.8
15	8	32	4843	21.95	51.9
15	16	32	6037	21.94	51.9
15	32	32	11322	21.94	51.9

Table 2.1: Inter-frame encode time and compression ratio

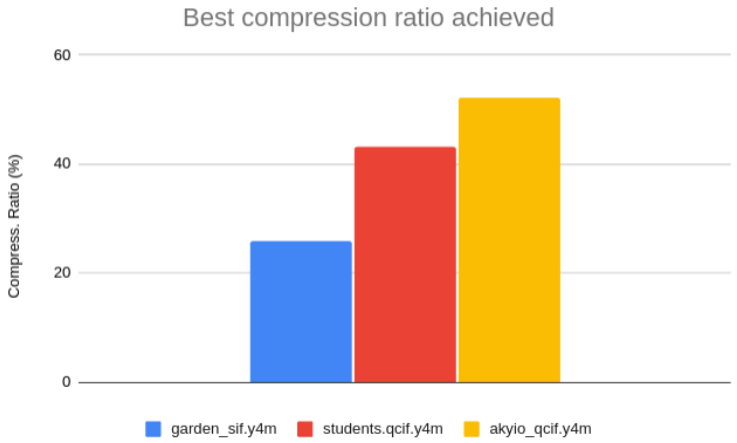

Figure 2.1: Compression ratio for best configuration

Starting by analyzing the compression ratio results, the first thing we see on Figure 2.1 is a clear relation between compression ratio and file size: as the size increases so does the best compression ratio achieved, which was a trend already observed in the **intra-frame** encoder and the conclusions on that section of the report are also be applied here.

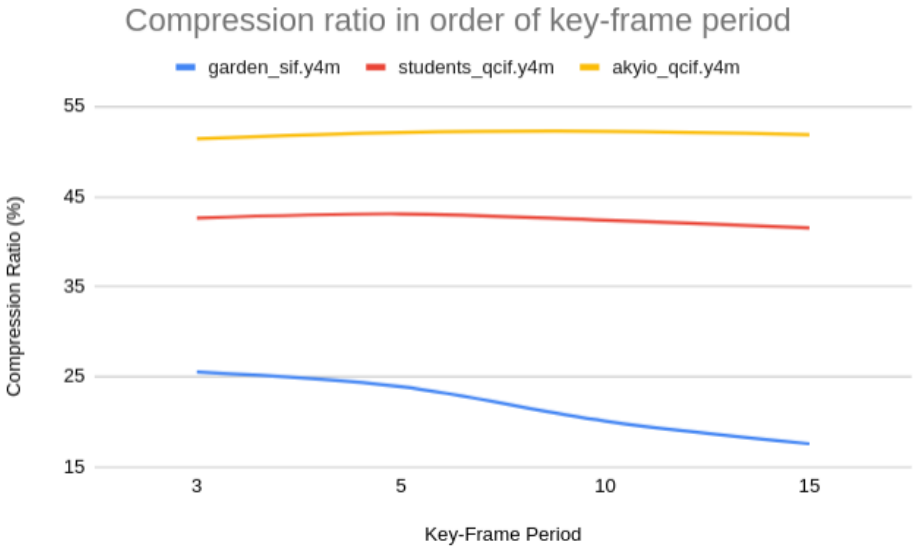

Figure 2.2: Compression ratio vs key-frame period

On Figure 2.2 we can see that for garden_sif.y4m it is very obvious that as the key-frame period increases, the compression ratio gets lower, for students_qcif.y4m and akyio_cif.y4m the best key-frame periods are 5 and 10 respectively, noting that for both videos the values 5 and 10 achieve very similar results. These results can be explained when viewing the content of the video. We can clearly see that garden is more complex (due to the constant movement of the camera) when comparing it to students and akyio.

When taking that in consideration, we can see that the results got are expected, for a video with constant movement (garden_sif.y4m) it is expected that when comparing a frame with another that is 10 or 15 frames old, it is a lot harder to find similar blocks and therefore a worse compression ratio is achieved; When the content of the video has less movement and static objects like a background wall (students_qcif.y4m and akyio_cif.y4m), it is expected to find zones of the image that are similar when comparing with 5, 10 or 15 frames behind.

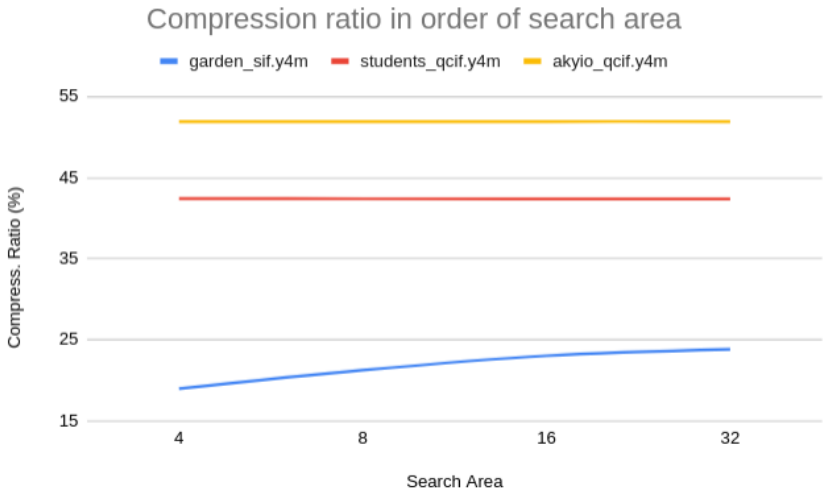

Figure 2.3: Compression ratio vs search area

Similar conclusions are taken when observing Figure 2.3, for images with less movement (students_qcif.y4m and akyio_cif.y4m), search area doesn't affect compression ratio much, because the pixels haven't shifted much from the previous frame. However, for garden_sif.y4m, the search area is very important, because the video has much more movement, therefore a better compression ratio is achieved when having a greater search area.

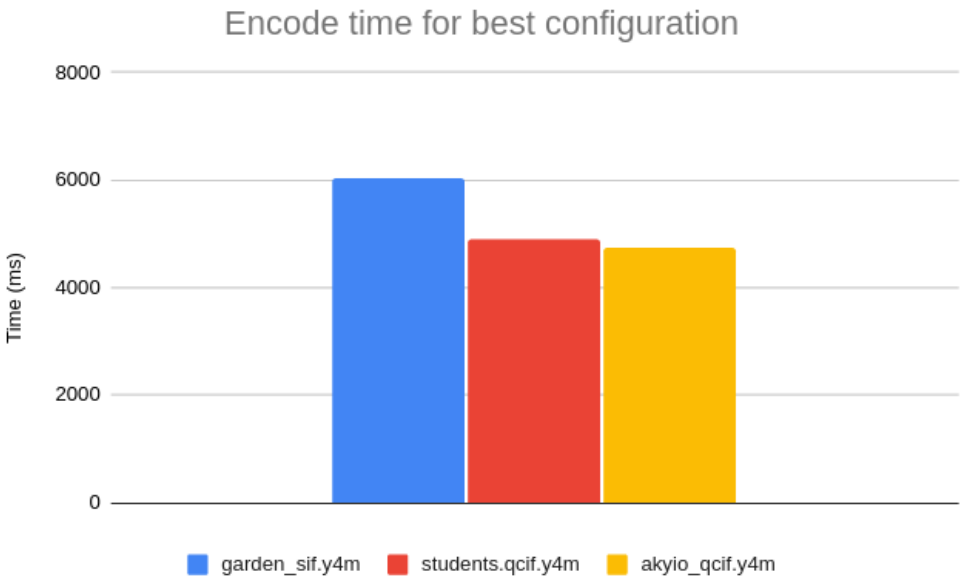

Figure 2.4: Encode time performance for best configuration

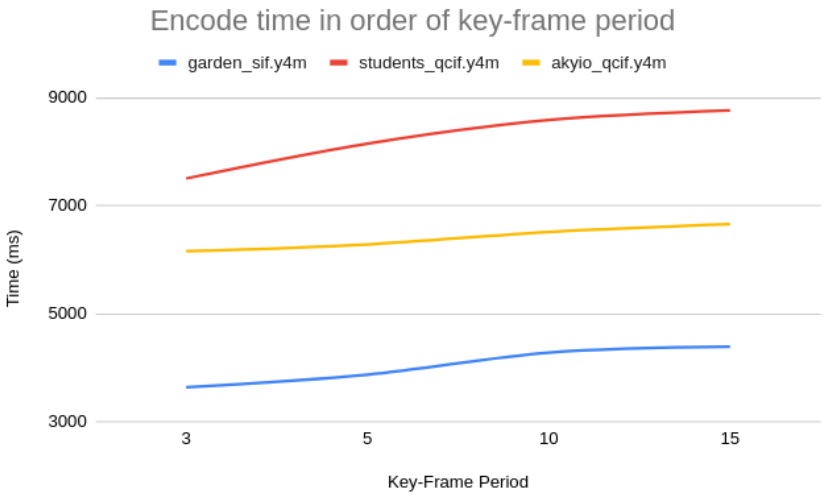

Figure 2.5: Encode time performance vs key-frame period

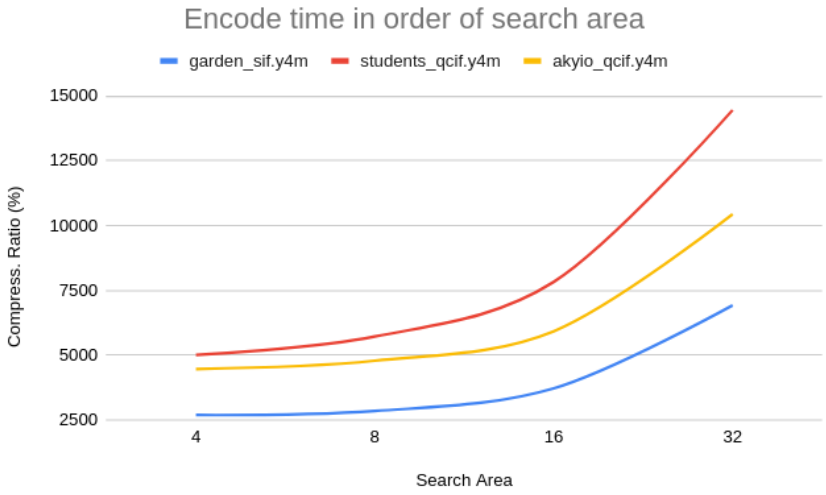

Figure 2.6: Encode time performance vs search area

On Figure 2.4 we see that as the file size increases, the time to encode decreases, even thought that to fully prove this conclusion, we would need to have a bigger sample of video files, since the expected behaviour would be that bigger files would take more time to encode because there are more frame to encode. We think that, in this case, this behaviour happens because as the file size increases the complexity of the video decreases, meaning that akyio_qcif as less complexity and movement than students_qcif.y4m and so on. Our assumption is that a more complex video will take more processing

time to find a good matching block and therefore it will take more time, and that is why we are observing this behaviour.

For Figure 2.5 and Figure 2.3 the results are expected. In both cases, we see that, as the key-frame period or search area increases, so does the time to encode.

To explain the results got on Figure 2.5 we need to establish that **intra-frame** encoding is faster than **inter-frame** encoding, which happens because in **inter-frame** encoding, in addition to predicting the values, there is also the calculation of the motion vectors. Knowing that a long key-frame period means that **inter-frame** encoding is done more times and that **inter-frame** encoding is slower than **intra-frame** we can see why a longer key-frame period is reflected in an increase of encoding time.

For Figure 2.6 the results are expected because a larger search area requires more processing time to search for the best matching block, while a smaller search area requires less processing time as it has less pixels to go through.

garden_sif.y4m				students_qcif.y4m	akyio_cif.y4m			
KF	SA	BS	Time (ms)	Time (ms)	KF	SA	BS	Time (ms)
3	4	16	2687	5419	3	4	32	5102
3	8	16	2516	5460	3	8	32	5292
3	16	16	2596	5451	3	16	32	5471
3	32	16	2606	5209	3	32	32	5289
5	4	16	2730	5389	5	4	32	5233
5	8	16	2575	5257	5	8	32	5052
5	16	16	2577	5421	5	16	32	5103
5	32	16	2551	5345	5	32	32	5128
10	4	16	2796	5256	10	4	32	5096
10	8	16	2823	5201	10	8	32	5050
10	16	16	2748	5619	10	16	32	5133
10	32	16	2689	5224	10	32	32	5167
15	4	16	3013	5357	15	4	32	5193
15	8	16	2805	5286	15	8	32	5069
15	16	16	2706	5291	15	16	32	5092
15	32	16	2640	5283	15	32	32	5239

Table 2.2: Inter-frame decode time

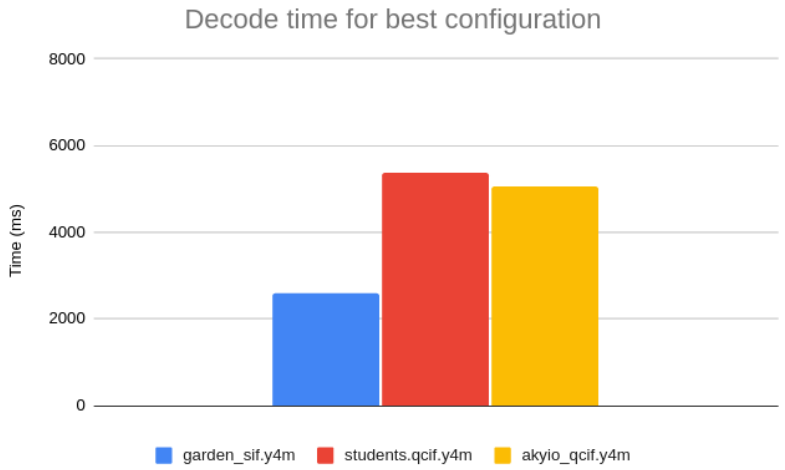

Figure 2.7: Decode time performance for best configuration

On Figure 2.7 we can see that, as expected, the decode time takes longer for bigger files, which is to be expected because a bigger video file will have more information to decode. The slight difference between akyio_cif.y4m and students_qcif.y4m can be explained by the following: even though akyio is a bigger video file, the encoded file of akyio has the same size as students best encoded file. However, when asking ourselves: why does the students encoded file take more time to decode if it has the same size as akyio? This could be explained by the compression ratio achieved, since students achieves a worse compression ratio than akyio we can deduce that it must have more complex calculations and bigger vectors, therefore taking a little bit more time to decode.

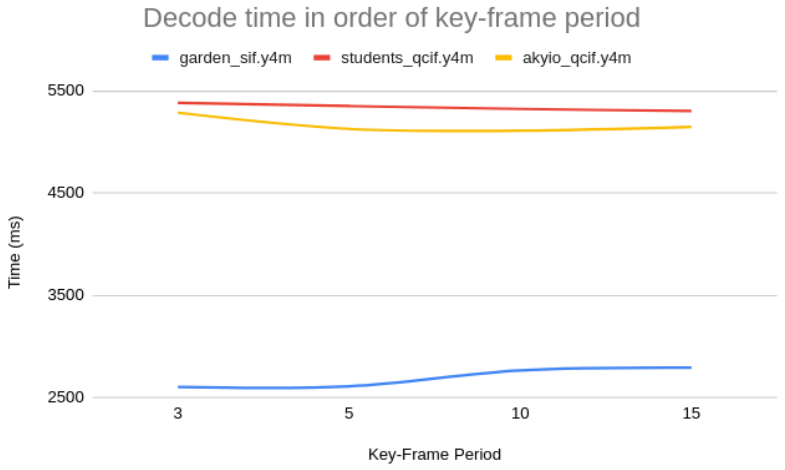

Figure 2.8: Decode time performance vs key-frame period

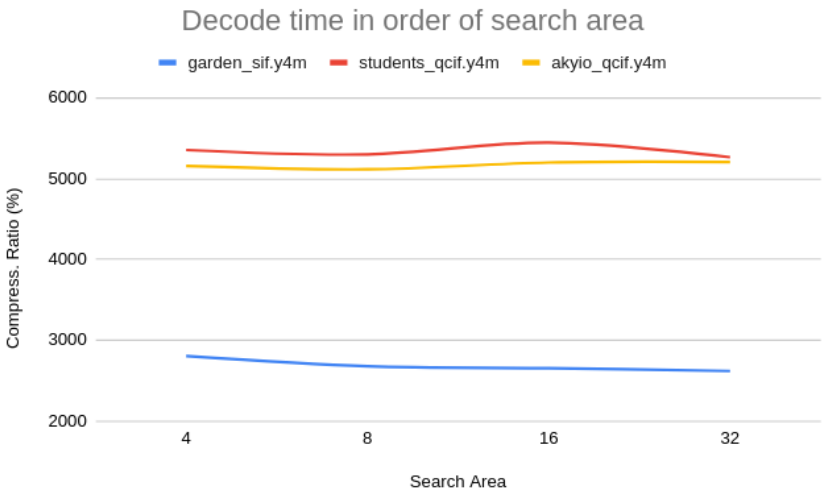

Figure 2.9: Decode time performance vs search area

On Figure 2.8 and Figure 2.9 we can see that the key-frame period values and search area values do not have a great effect on the decode time, which is to be expected because these parameters are only useful to find the best and most similar block on the encoding process. On the decoding process the decoder already knows where the best block is located, therefore, these parameters do not affect the decoding time.

Chapter 3

Exercise 3

3.1 Code

This exercise consists on expanding our codec to support simple quantization of the residuals, resulting in a lossy output file. To do this, all we did for both encoders (**inter-frame** and **intra-frame**) was quantize the residuals after they were calculated, but before they were encoded.

Intra-frame lossy encoder:

```
//go pixel by pixel through the Y, U, and V Mat objects to make predictions
(...)
// Quantization of residuals
for (long unsigned int i = 0; i < Yresiduals.size(); i++) {
    Yresiduals[i] = Yresiduals[i] >> quantization;
}
(...)
```

Inter-frame lossy encoder:

```
//Quantization and encoding
Golomb g;
int m_index = 0;
bool keyFrame = false;
for (long unsigned int i = 0; i < Yresiduals.size(); i++) {
    if (i % blockSize == 0 and i != 0) {
        Ym.push_back(Ym_vector[m_index]);
        m_index++;
    }
    //only quantize the residuals that are not in the keyframe
    if((frameIndex !=0) && (frameIndex % keyFramePeriod != 0)){
        Yresiduals[i] = Yresiduals[i] >> quantization;
    }
    Yencoded += g.encode(Yresiduals[i], Ym_vector[m_index]);
    if (i == Yresiduals.size() - 1) {
        Ym.push_back(Ym_vector[m_index]);
    }
}
```

Here, on both encoders, we only quantize the channel that contains information about the luminosity of the pixels. We ran several tests to compare if it was worth to quantize all the 3 channels of the pixel (Y, U and V) and losing information not only on the luminance part of the pixel but also on the chrominance, or just cutting information from the luminance. What we found out was that the compression ratio was almost the same between removing information from the 3 channels or just the luminance channel.

This gave us flexibility to cut more bits without compromising too much quality on the output file because the colors of the decoded video are the same as the original one, not losing any piece of information on the *keyframe*.

On the **inter-frame** lossy encoder we went a step further and only quantized the information of the non-keyframe frames of the video. We chose this approach because on the **inter-frame** encoding algorithm the residuals are calculated based on a reference frame, which can result in propagation of the error introduced by removing little bits of information. To counter this, we made sure that the *keyframes* of the video are original and have all the information, however the following frames have less information to reconstruct the original frame, and when we do the reverse operation to get back the correct values of the frame, instead of summing two values with error (the residual and the *keyframe* pixel), only one of those values has errors - the residual values.

Overall, this approach, by just removing information from the luminance channel and also only the non-keyframe frames (when using the **inter-frame** encoding), allowed us to achieve higher compression ratios but keeping the video's content with better quality, rather than just cutting information of all the frames on all the channels.

Both decoders work in the same way by trying to get the original value back, minimizing the overall error introduced by cutting bits representing information.
Intra-frame lossy decoder:

```
//undo the quantization of YUV decoded values back to the
    original size
for (long unsigned int i = 0; i < Ydecoded.size(); i++){
    if(quantization != 1){
        Ydecoded[i] = Ydecoded[i] << 1;
        Ydecoded[i] = Ydecoded[i] | 1;
        Ydecoded[i] = Ydecoded[i] << (quantization - 1);
    }else{
        Ydecoded[i] = Ydecoded[i] << 1;
    }
}
```

Inter-frame lossy decoder:

```
//undo the quantization of YUV decoded values back to the
    original size
```

```
int frameIndex = 0;
int total = padded_height*padded_width;

for (int i = 0; i < Ydecoded.size(); i++){
    if (i % total == 0 and i != 0) {
            frameIndex++;
    }
    //Only undo quantization if it is not a keyframe
    if((frameIndex !=0) && (frameIndex % keyFramePeriod != 0)){
        if(quantization != 1){

            Ydecoded[i] = Ydecoded[i] << 1;
            Ydecoded[i] = Ydecoded[i] | 1;
            Ydecoded[i] = Ydecoded[i] << (quantization - 1);
        }else{
            Ydecoded[i] = Ydecoded[i] << 1;
        }
    }
}
```

When the quantization value is different from 1, we can reduce the error in a more optimal way. This reduction is done by inserting half of the cut value, instead of inserting all the bits to 0, which results in a value approximately closer to the original value before being quantized.

However, if the quantization factor is 1, it means that just 1 bit of information was discarded and we recover that bit as 0.

To compare the original video to the one generated by the lossy codec, we developed a new program called *video_cmp.cpp* that compares two video sequences in terms of peak signal to noise ratio (PSNR) given by:

$$PSNR = 10\log_{10}\frac{A^2}{e^2}$$

Where A is the maximum value of the signal, in this case 255, and e^2 is the mean squared error between the reconstructed frame rf and the original frame f:

$$e^2 = \frac{1}{NM}\sum_{r=1}^{N}\sum_{c=1}^{M}[f(r,c) - rf(r,c)]^2$$

And N and M denote, respectively, the number of rows and columns of the video.

The program takes in two video files, checks for compatibility of those videos and outputs the *PSNR* values at the end, for each channel, as well as an overall *PSNR* value.

```
 // loop through the frames of the two videos
while(!feof(input1)){
    //VIDEO 1
    //read the frame line
    fgets(line, 100, input1);
    //read the Y data  (Height x Width)
```

```
    for(int i = 0; i < width1 * height1; i++){
        int value = fgetc(input1);
        Y1[i] = value;
        if(value < 0){
            finish = true;
            break;
        }
    }
    if(finish) break;
    for(int i = 0; i < width1 * height1 / 4; i++) U1[i] = fgetc(input1); //read the U data (Height/2 x Width/2)
    for(int i = 0; i < width1 * height1 / 4; i++) V1[i] = fgetc(input1); //read the V data (Height/2 x Width/2)

    //VIDEO 2
    //read the frame line
    fgets(line, 100, input2);
    //read the Y data  (Height x Width)
    for(int i = 0; i < width2 * height2; i++){
        int value = fgetc(input2);
        Y2[i] = value;
        if(value < 0){
            finish = true;
            break;
        }
    }
    if(finish) break;
    for(int i = 0; i < width2 * height2 / 4; i++) U2[i] = fgetc(input2); //read the U data (Height/2 x Width/2)
    for(int i = 0; i < width2 * height2 / 4; i++) V2[i] = fgetc(input2); //read the V data (Height/2 x Width/2)
```

After reading the information from both files into specific vectors, we convert those vectors into matrices and perform the calculation of e^2 for the 3 channels in that specific frame, as well as the *PSNR* value of that frame.

```
(...)
        //psnr calculation
        //Y
        double psnr;
        long int sum = 0;
        for(int i = 0; i < height1; i++){
            for(int j = 0; j < width1; j++){
                // cout << int(YMat1.at<uchar>(i, j)) << " " << int(YMat2.at<uchar>(i, j)) <<  " = " << pow(YMat1.at<uchar>(i, j) - YMat2.at<uchar>(i, j), 2) << endl;
                sum += pow(YMat1.at<uchar>(i, j) - YMat2.at<uchar>(i, j), 2);


            }
        }
        double e2 = (double)sum / (width1 * height1);
        psnr = 10 * log10(255 * 255 / e2);
```

```
        Ypsnr_values.push_back(psnr);

        //U
        sum = 0;
        for(int i = 0; i < height1/2; i++){
            for(int j = 0; j < width1/2; j++){
                sum += pow(UMat1.at<uchar>(i, j) - UMat2.at<uchar>(i, j), 2);
            }
        }
        e2 = sum / (width1/2 * height1/2);
        psnr = 10 * log10(255 * 255 / e2);
        Upsnr_values.push_back(psnr);

        //V
        sum = 0;
        for(int i = 0; i < height1/2; i++){
            for(int j = 0; j < width1/2; j++){
                sum += pow(VMat1.at<uchar>(i, j) - VMat2.at<uchar>(i, j), 2);
            }
        }
        e2 = sum / (width1/2 * height1/2);
        psnr = 10 * log10(255 * 255 / e2);
        Vpsnr_values.push_back(psnr);
```

At the end, we have 3 vectors containing the *PSNR* values of each frame for each channel of the pixel, so we get the average of all the frames and output the correct information. We also calculate the average of values of the 3 channels and get an overall *PSNR* value that takes into account all the frames and all the channels of the video:

```
//calculate the lowest psnr for each component
double Ypsnr = 0;
double Upsnr = 0;
double Vpsnr = 0;

for(int i = 0; i < Ypsnr_values.size(); i++){
    //if Ypnsr is infinite, Ypsnr is 100
    if(Ypsnr_values[i] == numeric_limits<double>::infinity())
    Ypsnr_values[i] = 100;
    Ypsnr += Ypsnr_values[i];
    if(Upsnr_values[i] == numeric_limits<double>::infinity())
    Upsnr_values[i] = 100;
    Upsnr += Upsnr_values[i];
    if(Vpsnr_values[i] == numeric_limits<double>::infinity())
    Vpsnr_values[i] = 100;
    Vpsnr += Vpsnr_values[i];
}

Ypsnr /= Ypsnr_values.size();
Upsnr /= Upsnr_values.size();
Vpsnr /= Vpsnr_values.size();

```

```
20 //print the results
21 if (Ypsnr == 100 ) cout << "YPSNR: " << "inf" << endl;
22 else cout << "YPSNR: " << Ypsnr << endl;
23
24 if (Upsnr == 100 ) cout << "UPSNR: " << "inf" << endl;
25 else cout << "UPSNR: " << Upsnr << endl;
26
27 if (Vpsnr == 100 ) cout << "VPSNR: " << "inf" << endl;
28 else cout << "VPSNR: " << Vpsnr << endl;
29
30 //average over all components
31 double psnr = (Ypsnr + Upsnr + Vpsnr) / 3;
32 if (psnr == 100 ) cout << "PSNR: " << "inf" << endl;
33 else cout << "PSNR: " << psnr << endl;
```

3.2 Usage

The usage of the **intra-frame** lossy codec is done following this pattern:

```
1 ./lossy_intra_encoder <input file> <output file> <block size> <
    quantization>
```

```
1 ./lossy_intra_decoder <input file> <output file>
```

Where the only difference to the lossless one is the last argument of the encoder, quantization, that is an integer regarding how many bits are going to be cut from the residuals.

The usage of the **inter-frame** lossy codec is done following this pattern:

```
1 ./lossy_inter_encoder <input file> <output file> <block size> <
    search area> <key-frame period> <quantization>
```

```
1 ./lossy_inter_decoder <input file> <output file>
```

Where the only difference to the lossless one is the last argument of the encoder, quantization, that is an integer regarding how many bits are going to be cut from the residuals.

To use the video compare program:

```
1 ./video_cmp <original input file> <reconstructed input file>
```

Where both input files are video files (tested with .y4m), one is the original, and the other is the reconstructed one.

3.3 Results

When testing Lossy **Intra-frame** this is what we got:

garden_sif.y4m				
Bits removed	Time (ms)	Out Size (MB)	PSNR	Comp. Ratio (%)
1	2260	9.79	70.3	32.8
2	2080	8.89	71.4	39.0
3	1814	8.27	71.4	43.2
4	1651	8.05	71.6	44.7
5	1569	7.55	71.4	48.2
6	1534	7.30	70.9	49.9
students_qcif.y4m				
1	4631	21.16	70.8	44.7
2	4198	20.80	69.1	45.7
3	3955	19.45	68.8	49.2
4	3824	18.55	68.7	51.5
5	3811	18.22	68.8	52.4
6	3861	18.13	69.0	52.6
akyio_cif.y4m				
1	4186	22.84	70.1	49.9
2	4012	21.15	68.2	53.6
3	3955	20.51	68.1	55.1
4	3938	20.25	68.1	55.6
5	3922	20.16	68.2	55.8
6	3898	20.14	68.4	55.9

Table 3.1: Lossy Intra-frame encoder performance

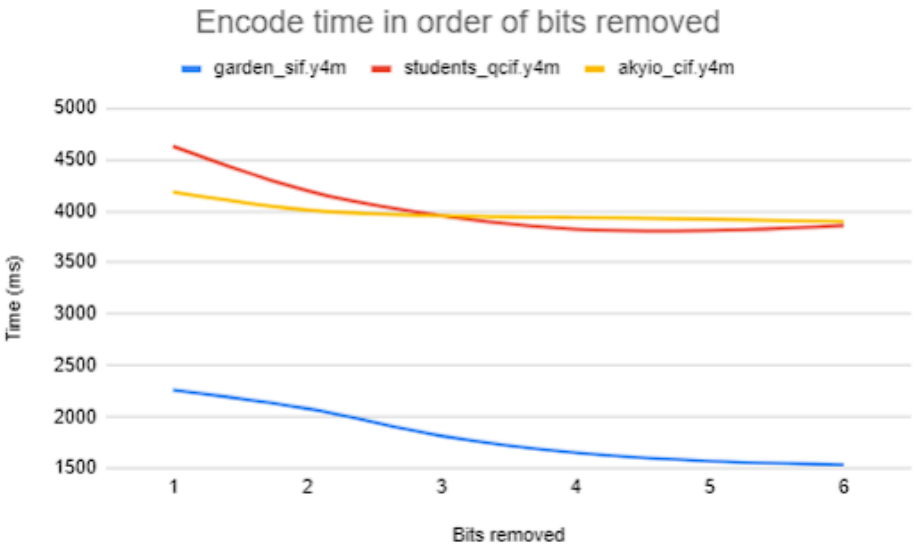

Figure 3.1: Lossy Intra-frame Encode time performance vs bits removed

Analyzing Figure 3.1 we see that the encoding time decreases as we increase the number of quantized bits, even though in the encoding process the same number of frames are processed regardless of the number of quantized bits. However, the amount of data that needs to be processed can still vary depending on the number of quantized bits.

Quantizing bits results in a lower resolution of the residuals, which means that there is less information. As a result, there will be fewer non-zero residuals after quantization, and fewer bits will be needed to represent the residuals. This can result in a decrease on the amount of data that needs to be processed during the encoding process, which may lead to a decrease in the encode time.

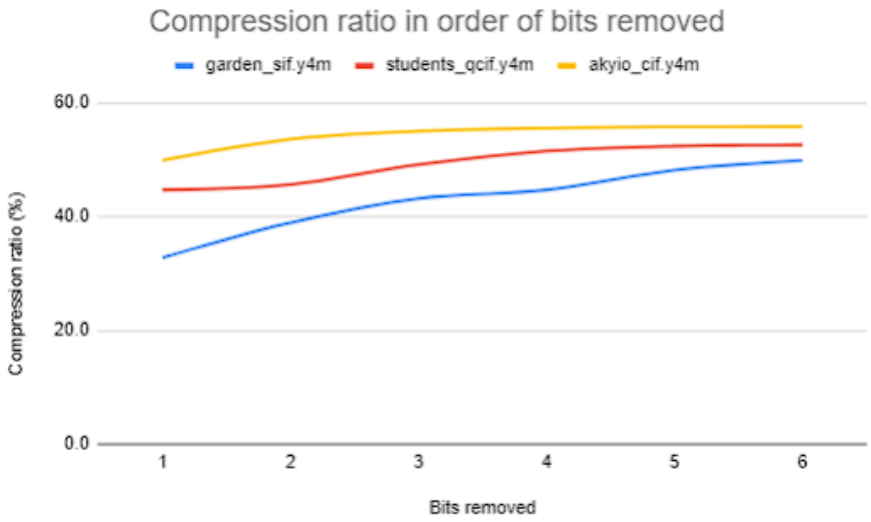

Figure 3.2: Lossy Intra-frame Compression ratio vs bits removed

When viewing Figure 3.2 we see that the compression ratio achieved increases as the number of bits removed do so, too. This is expected because, in general, quantizing more bits leads to a reduction in the precision of the data and a corresponding decrease in the file size. As a result, the compression ratio may decrease in a linear fashion as the number of quantized bits increases.

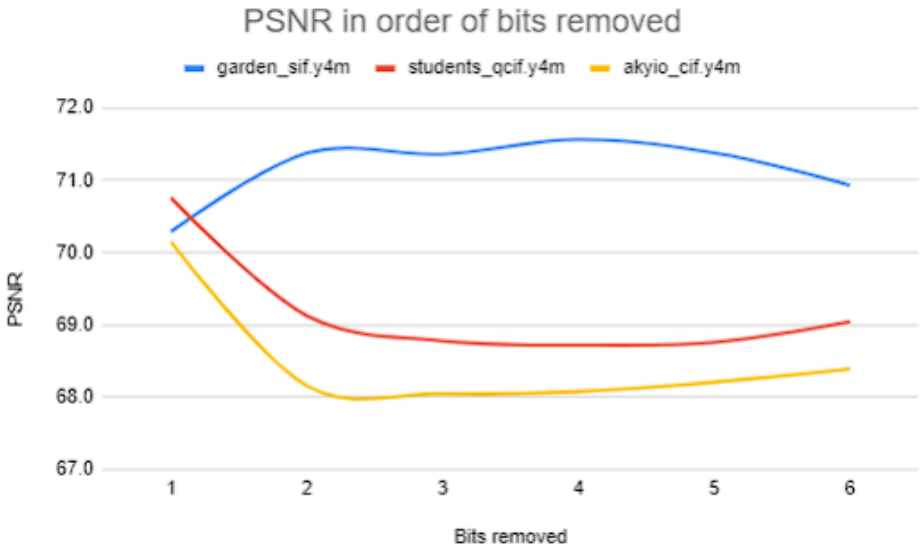

Figure 3.3: Lossy Intra-frame PSNR vs bits removed

The PSNR value sees a downward trend until a certain point, having in mind the number of bits removed. However, when the values that are quantized (the residuals) are smaller, removing, e.g., either 3 or 4 bits, is the same. This happens because if the quantized residual with 3 bits removed is 0, then 4 bits is 0 as well. It happens a lot because most of the residuals are very small integers that can become 0 with just 3 or 4 bits removed, which ultimately means that the PSNR value doesn't change that much after the removal of some bits.

garden_sif.y4m	
Bits removed	Time (ms)
1	2346
2	2142
3	2052
4	1920
5	1847
students_qcif.y4m	
6	1815
1	4970
2	4978
3	4535
4	4536
5	4550
6	4549
akyio_cif.y4m	
1	5298
2	4995
3	4896
4	4864
5	4845
6	4852

Table 3.2: Lossy Intra-frame decode time

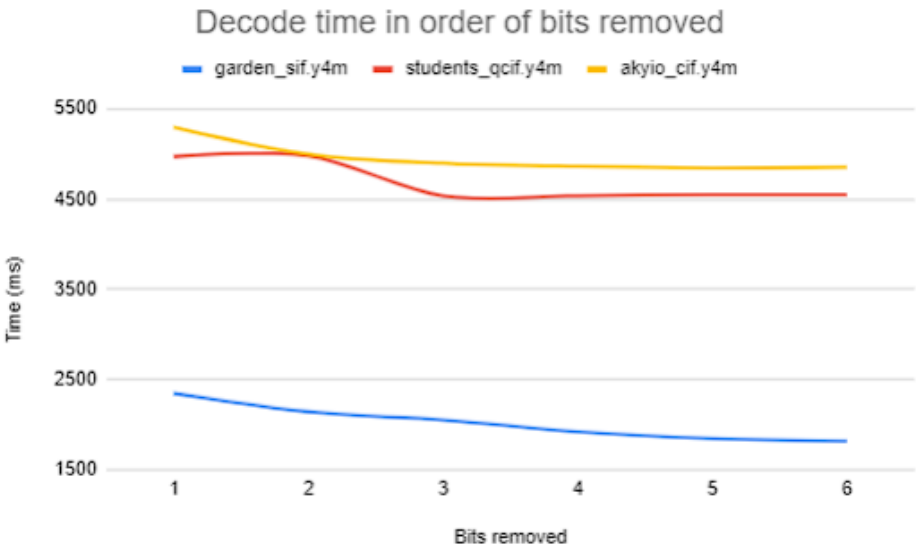

Figure 3.4: Lossy Intra-frame Decode time performance vs bits removed

Basically the same is observed here as we saw on Figure 3.1 and the explanation falls in the same category. Quantizing more bits results in a lower resolution of the data, which means that there is less information contained in the encoded image.

When testing Lossy Inter-frame this is what we got:

garden_sif.y4m				
Bits removed	Time (ms)	Out Size (MB)	PSNR	Comp. Ratio (%)
1	6178	10.36	89.2	28.9
2	6107	10.16	88.2	30.3
3	6174	10.07	86.8	30.9
4	6059	10.03	85.4	31.2
5	6145	10.02	83.9	31.3
6	6160	10.01	82.4	31.3
students_qcif.y4m				
1	7546	20.78	87.1	45.7
2	7511	20.47	85.3	46.5
3	7520	20.38	83.5	46.8
4	7460	20.35	81.7	46.9
5	7268	20.34	79.9	46.9
6	7509	20.34	78.3	46.9
akyio_cif.y4m				
1	4859	20.98	86.6	54.0
2	4720	20.67	83.0	54.7
3	4755	20.57	81.1	54.9
4	4659	20.54	79.2	55.0
5	4684	20.53	77.3	55.0
6	4886	20.52	75.5	55.0

Table 3.3: Lossy Inter-frame encoder performance

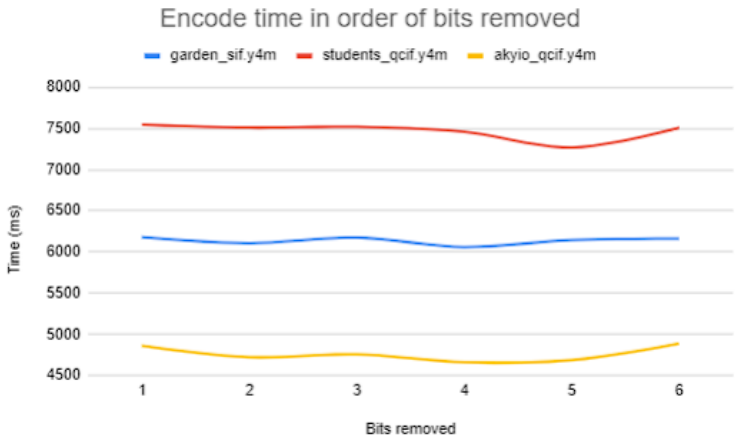

Figure 3.5: Lossy Inter-frame Encode time performance vs bits removed

Analyzing Figure 3.5 we can see that the amount of bits removed doesn't have an impact on the encoding time, which happens because the process of comparing the current frame to the reference frame and generating the residuals is typically the most computationally intensive part of the encoding process (note that every encode time for lossy **inter-frame** takes more time than **intra-frame**). The quantization step, on the other hand, is typically a relatively simple operation that can be performed quickly.

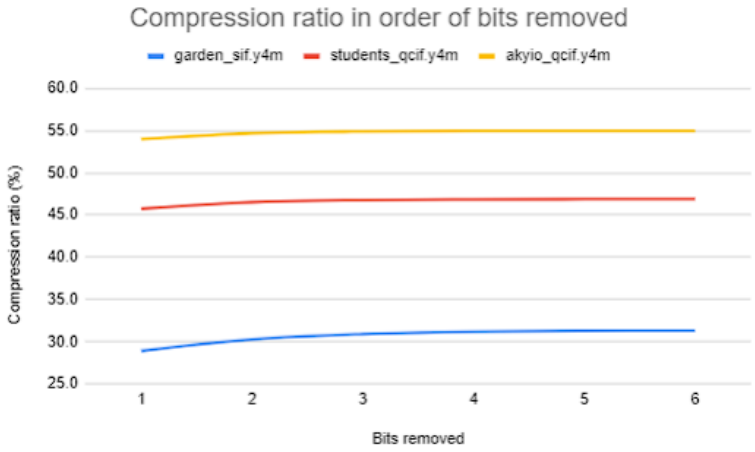

Figure 3.6: Lossy Inter-frame Compression ratio vs bits removed

When viewing the results that study the effect of quantization in the compression ratio achieved we see that it achieved increases logarithmically. After a couple of bits removed, the compression ratio doesn't increase by much. It turns out that there's not much more information to quantize when the values that we quantize with fewer bits already have the value 0. What happens is that the residuals with higher values, which are in much smaller quantity, end up being the only ones affected when we cut more bits, taking into account that smaller residual values were already at 0 with a less bits cut. This trend is not very similar to the trend seen with **intra-frame** encoding, mostly because the algorithm to predict the values is better in **inter-frame**, resulting in residuals with smaller values, which are greatly affected by the quantization.

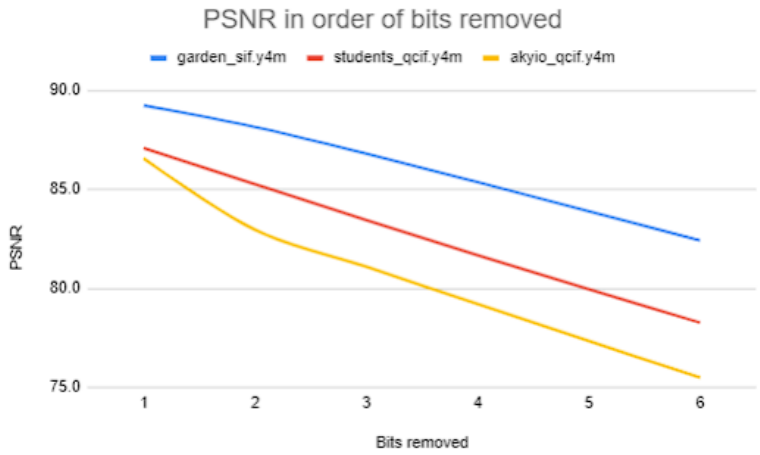

Figure 3.7: Lossy Inter-frame PSNR vs bits removed

The clear trend seen on Figure 3.7 is an expected effect, as the quantization of bits increases, the PSNR obtained when comparing it with the original video decreases linearly.

Another thing to explain is the difference in the PSNR values between files. Explained by the fact that, if the original frames in file1 contain more detail than the frames in file2, the PSNR file1 may be higher than the PSNR file2 even if the same number of bits are quantized. This is because the quality of the reconstructed frames may be more sensitive to the loss of information in the original frames with more detail.

garden_sif.y4m	
Bits removed	Time (ms)
1	2688
2	2606
3	2593
4	2582
5	2540
6	2538
students_qcif.y4m	
1	5237
2	5197
3	5234
4	5275
5	5243
6	5174
akyio_cif.y4m	
1	5188
2	5154
3	5118
4	5209
5	5120
6	5126

Table 3.4: Lossy Inter-frame decode time

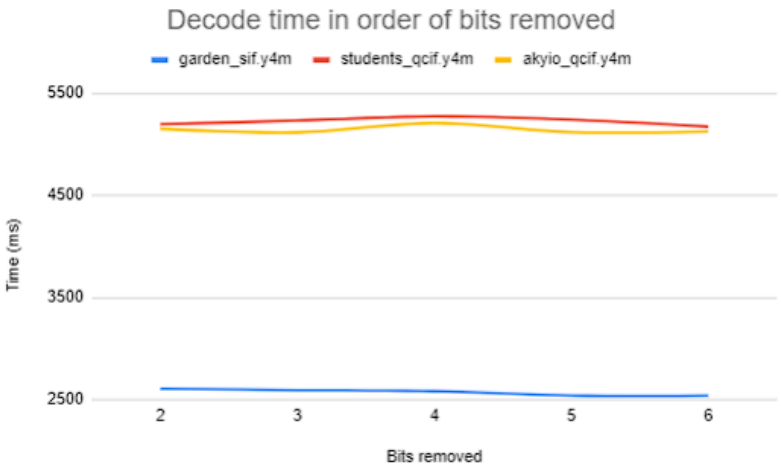

Figure 3.8: Decode time performance vs bits removed

The effect seen in the decoding time performance follows a similar pattern as the encoding time effect seen on Figure 3.5, therefore the explanation of this effect is similar. The decoding process does not change whether 2 or 6 bits were removed. The decoder has to do the same process and perform similar calculations, with the detail that the only thing that changes is the number of bits used to represent the residuals.

Chapter 4

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 on the .xlsx files.

NOTE: All graphics showing akyio_qcif.y4m should be regarded as akyio_cif.y4m.