University of Aveiro

DETI

Information and Coding
Project nº3

Gonçalo Silva (103244) goncalolsilva@ua.pt
Samuel Teixeira (103325) steixeira@ua.pt
Tiago Alves (104110) tiagojba9@ua.pt

January 7, 2023

Index

List of Figures

Chapter 1

Introduction

The report here conducted follows the structure of the work guide. Part I approaches exercises 1-2. Part II handles exercise 3. Part III describes exercises 4. This project was developed using the GitHub platform for version control, it can be accessed at `https://github.com/detiuaveiro/ic_gts`

Chapter 2

Part I

2.1 Exercise 1

2.1.1 Encoder And Decoder

Our "image_encoder.cpp" *Options class* followed the same structure as the previous done for "audio_encoder.cpp". Below we can see all the configurable parameters required.

```
// configurable parameters
namespace Options {
string fileName = "images/airplane_gray.pgm";
string encodedName = "encodedImage";
size_t blockSize = 64;
int m = -1;  // automatic
bool lossy = false;
PREDICTOR_TYPE predictor = AUTOMATIC;
APPROACH approach = SIGN_MAGNITUDE;
}
```

Listing 2.1: Encode Value Interleaving function

The only parameters that are now needed, are the names of the input and output files, which will be checked later if they are the correct type, Portable Gray Map (PGM). The other ones stay the same as before even though it is important to notice that since we will be dealing with images that might have different size from audios, a correct block size must be chosen to obtain better encoding.

We have also created a PGM Header structure to store the width, height and maximum gray level present on the header of the PGM file.

To start our process we needed to be ensure a PGM file is received. To deal with this situation we created an extra class, named *grayscale_ converter*. This class's .cpp file is shown below.

```
#include <grayscale_converter.h>

GrayscaleConverter::GrayscaleConverter(std::string fileName){
```

```
    this->file.open(fileName, std::fstream::in | std::fstream::
    binary);
    this->img = imread(fileName);
}

bool GrayscaleConverter::IsGrayScale(){
      //For grayscale, pixels must have the same value
    for (int i = 0; i < img.rows; i++)
    {
        for (int j = 0; j < img.cols; j++)
        {
            Vec3b pixel = img.at<Vec3b>(i, j);
            if((pixel[0] != pixel[1]) || (pixel[1] != pixel[2]))
    return false;
        }

    }
    return true;
}

void GrayscaleConverter::ColorScaleToGrayScale() {
        //Perform Color Scale change
        if (IsGrayScale()) {
            cout << "Image is already in greyscale\n";
            return;
        }

        cvtColor(img, img, COLOR_BGR2GRAY);
}

void GrayscaleConverter::SaveToFile(std::string outputFileName) {
        imwrite(outputFileName, img);
        cout << "Grayscale image saved to: " << outputFileName <<
    endl;
    }
```

Listing 2.2: Encode Value Interleaving function

The 'IsGrayScale()' is designed to identify whether an image is grayscale. Since grayscale images are characterized by uniform BGR values, this function checks the if the intensity of the values for each pixel, represented by a 3-vector byte, is identical. If the image retrieved is not in gray scale, 'ColorScaleToGrayScale()' function will convert the image from BGR scale to gray scale.

On the main code, the checking of the correct image format and possible conversion is done using the following code.

```
GrayscaleConverter grayscaleConverter(Options::fileName);
if (!grayscaleConverter.IsGrayScale()) {
    grayscaleConverter.ColorScaleToGrayScale();
    // get first letters of filename and replace extension
    string outputFileName = Options::fileName.substr(0, Options::
    fileName.find_last_of(".")) + ".pgm";
    grayscaleConverter.SaveToFile(outputFileName);
    std::cout << "New grayscale image saved to: " << outputFileName
     << endl;
```

```
}
```

Listing 2.3: Encode Value Interleaving function

By now we are ready to store the header parameters of the PGM image. These are the image type, normally 'P5' for PGM images, width, height and max gray level. We use the following function, inside the "image_encoder.cpp", to obtain these parameters.

```
PGMHeader readPGMHeader(const string& filename) {
    ifstream file(filename, ios::binary);
    PGMHeader header;

    if (!file.is_open()) {
        cerr << "Error: Could not open the file " << filename << endl;
        // You might want to handle the error appropriately
        cerr << "Exiting..." << endl;
        exit(1);
    }

    // Read the PGM header
    getline(file, header.type); // Read the first line, should be "P5" for a binary PGM

    // Read comments and discard them
    string line;
    while (getline(file, line) && line[0] == '#');

    // Read width, height, and max gray level
    istringstream dimensions(line);
    dimensions >> header.width >> header.height;

    getline(file, line);
    istringstream maxGrayLevel(line);
    maxGrayLevel >> header.maxGrayLevel;

    file.close();
    return header;
}
```

Listing 2.4: Encode Value Interleaving function

The parameters are stored in a PGMHeader structure that contains space these 4 parameters. After this, parameters are passed to the 'file' structure present on 'image_coded.h', and will be encoded alongside other important parameters as the header of the encoded file.

Now we need to obtain the number of blocks that will be analysed individually during the encoding. This number depends on the width and height, that will be divided by the block size. We also make sure the padding is correct by adding an extra block within the width or the height one of these does not match block size completely after the division.

```
int numBlocksWidth = f.width / Options::blockSize;
int numBlocksHeight = f.height / Options::blockSize;

```

```
if (f.width % Options::blockSize != 0)
    ++numBlocksWidth;  // Add another column of blocks not fully
    occupied

if (f.height % Options::blockSize != 0)
    ++numBlocksHeight;  // Add another row of blocks not fully
    occupied
```

Listing 2.5: Encode Value Interleaving function

We can finally create the gEncoder object using *GEncoder class* and encode the file header. After this we create a Mat only with Y (brightness) values by just reading the PGM file. This file only has the Y values because of it's gray scale nature and encode this frame. To encode this one we used the same image codec, that will be used in exercise two, since instead of having multiple frames we only have to process one.

For the decoder the structure was pretty simple. On our main we only had to create a Golomb Decoder object for the encoded file, read the file header, and finally decode the only frame, since we are dealing with images. The decoded image, represented as a Mat, can now be written on the output file.

```
// Create Golomb Encoder class
GDecoder gDecoder(Options::encodedName);

int nBlocks = gDecoder.read_file_header();

File f = gDecoder.get_file();
print_processing_information(f, nBlocks);

std::cout << "Decoding file with " << unsigned(f.nFrames) << "
    Frames..."
          << endl;

Mat decodedImg = gDecoder.decode_frame(0);

// Write decoded samples to file
imwrite(Options::imageName, decodedImg);
```

2.1.2 Results

Since our image_codec was already adapted for videos, we leave the testing for the next exercises.

2.2 Exercise 2

2.2.1 Modifications or Insertions in the code

The following bash script was created to address the challenge of handling large y4m videos. Its purpose is to retrieve a video from the website and save it on a new folder named movies, ensuring the local processing of this video.

```bash
#!/bin/bash

if [ ! -d "movies" ]; then
  mkdir movies
fi

cd movies

if [ ! -f "sintel_trailer_2k_480p24.y4m" ]; then
  wget https://media.xiph.org/video/derf/y4m/
    sintel_trailer_2k_480p24.y4m
else
  echo "File already exists"
  exit 1
fi
```

2.2.2 Frame Namespace

We created a namespace for handling frames.

```
#pragma once

#include <stdio.h>
#include <string.h>
#include <fstream>
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

namespace Frame {
uint8_t get_pixel(Mat& image, int pixelIndex);

std::vector<cv::Mat> get_blocks(Mat& image, int blockSize);

Mat compose_blocks(std::vector<cv::Mat> blocks, int blockSize, int
    rows,
                   int cols);

std::vector<uint8_t> mat_to_linear_vector(Mat& image);

Mat linear_vector_to_mat(std::vector<uint8_t> data, int rows, int
    cols);

/*! Accept an image only with Y field and display it */
void display_image(Mat& image);

```

```
};  // namespace Frame
```

This namespace contains functions for getting a pixel and blocks from a Mat File, a function to rebuild a frame using blocks, as well as conversions between Mat and vector of 8 bit unsigned integer and a display function.

2.2.3 Predictor class

The Predictor class manages prediction algorithms designed for image data. It is designed to enhance image compression efficiency through the implementation of 8 distinct predictor types, each contributing to the overall objective of minimizing occupied space while preserving image quality.

```
enum PREDICTOR_TYPE {
    AUTOMATIC,
    JPEG1,
    JPEG2,
    JPEG3,
    JPEG4,
    JPEG5,
    JPEG6,
    JPEG7,
    JPEG_LS  // JPEG-LS is a non-linear predictor
}

std::string get_type_string(PREDICTOR_TYPE type);

class Predictor {
   private:
    int predict_jpeg_1(int a);
    int predict_jpeg_2(int b);
    int predict_jpeg_3(int c);
    int predict_jpeg_4(int a, int b, int c);
    int predict_jpeg_5(int a, int b, int c);
    int predict_jpeg_6(int a, int b, int c);
    int predict_jpeg_7(int a, int b);
    int predict_jpeg_LS(int a, int b, int c);

    double calculate_entropy(PREDICTOR_TYPE type, Mat& frame);

   public:
    Predictor();
    ~Predictor();

    /*!
        Pass a set of frames/block and return the best predictor to be used
            (the one that resulted in less occupied space)
    */
    PREDICTOR_TYPE benchmark(Mat& frame);

    /*!
        Predict the next sample based on the type of the predictor and the
            previous samples
    */
```

```
42      int predict(PREDICTOR_TYPE type, Mat& frame, int idX, int idY);
43
44      bool check_type(PREDICTOR_TYPE type);
45  };
```

Listing 2.6: Encode Value Interleaving function

The implemented predictor types, named JPEG1 to JPEG_LS, draw inspiration from established compression techniques commonly used in image processing. Each predictor type is designed to predict the next pixel value in the image based on the values of the preceding pixels.

The first 7 predictors are linear predictors. They use the values of three previous pixels, a, b and c. Blocks can be thinked of as bidimensional matrixes. In these matrixes, a is the value of the pixel in the same row and previous column, b is in the same column but previous row and finally c is both in the previous column and row.

The following code details calculation of the results for the first 7 predictors:

```
1  int Predictor::predict_jpeg_1(int a) {
2      return a;
3  }
4
5  int Predictor::predict_jpeg_2(int b) {
6      return b;
7  }
8
9  int Predictor::predict_jpeg_3(int c) {
10     return c;
11 }
12
13 int Predictor::predict_jpeg_4(int a, int b, int c) {
14     return a + b - c;
15 }
16
17 int Predictor::predict_jpeg_5(int a, int b, int c) {
18     return a + (b - c) / 2;
19 }
20
21 int Predictor::predict_jpeg_6(int a, int b, int c) {
22     return b + (a - c) / 2;
23 }
24
25 int Predictor::predict_jpeg_7(int a, int b) {
26     return (a + b) / 2;
27 }
```

Listing 2.7: Encode Value Interleaving function

The last and final predictor is a nonlinear one, used in JPEG-LS. It is represented by the following equation:

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

And here is it's code:

```
int Predictor::predict_jpeg_LS(int a, int b, int c) {
    if (c >= max(a, b))
        return min(a, b);
    else if (c <= min(a, b))
        return max(a, b);
    else
        return a + b - c;
}
```

Determining the best predictor can be accomplished by using the `benchmark` function which iterates through all the predictor types to find the one that yields the lowest entropy, that will improve our program's efficiency. Using the `calculate_entropy` function, we can compute the entropy based on the given predictor type and type Mat frame. The probability of a predictor determining the correct value is given by P, with i representing the cycle of the various Predictors:

$$\mathrm{P}_i = \frac{\text{number of correct predictions}}{\text{total predictions}}$$

With the Entropy can be calculated using the Shannon's entropy formula:

$$\text{Entropy} = -\sum_i \mathrm{P}_i \cdot \log_2(\mathrm{P}_i)$$

```
PREDICTOR_TYPE Predictor::benchmark(Mat& frame) {
    double min_entropy = numeric_limits<double>::max();
    PREDICTOR_TYPE best_predictor = AUTOMATIC;

    array<double, 8> entropies = {
        calculate_entropy(JPEG1, frame), calculate_entropy(JPEG2,
    frame),
        calculate_entropy(JPEG3, frame), calculate_entropy(JPEG4,
    frame),
        calculate_entropy(JPEG5, frame), calculate_entropy(JPEG6,
    frame),
        calculate_entropy(JPEG7, frame), calculate_entropy(JPEG_LS,
     frame)};

    for (size_t i = 0; i < entropies.size(); ++i) {
        if (entropies[i] < min_entropy) {
            min_entropy = entropies[i];
            best_predictor = static_cast<PREDICTOR_TYPE>(i);
        }
    }

    return best_predictor;
}
```

Listing 2.8: benchmark function

```
double Predictor::calculate_entropy(PREDICTOR_TYPE type, Mat& frame
    ) {
```

```
2
3       if (!check_type(type) || type == AUTOMATIC) {
4           cerr << "Error: Unknown/Invalid Predictor " << unsigned(
        type)
5                << " encountered while calculating entropy" << endl;
6           exit(2);
7       }
8
9       int total_predictions = 0;
10      std::vector<int> predictions;
11
12      // Assuming frame is a single channel grayscale image
13      for (int i = 0; i < frame.rows; ++i) {
14          for (int j = 0; j < frame.cols; ++j) {
15              int prediction = predict(type, frame, i, j);
16              predictions.push_back(prediction);
17              total_predictions++;
18          }
19      }
20
21      // Calculate probability distribution and entropy
22      double entropy = 0.0;
23      for (int value : predictions) {
24          double probability =
25              count(predictions.begin(), predictions.end(), value) /
26              static_cast<double>(total_predictions);
27          entropy -= probability * log2(probability);
28      }
29
30      return entropy;
31  }
```

Listing 2.9: calculate_entropy function

2.2.4 Movie Class

We also created a class called Movie with very important features to dismantle an y4m file into frames and rebuilding one using frames. For representing the y4m's header parameters, a struct called HeaderParameters had to be created, where each of the most relevant parameters, such as height, width, chroma, fps, among others were stored. Also, information about the file itself such as the number of frames and the size of each frame (Y plan only in our case) is stored in there.

Some of the most important functions are get_header_parameters and read_frame, writemovie_header and write_movie_frame.

```
1   if (!stream.is_open()) {
2           cerr << "Error: movie file is not open" << std::endl;
3       }
4
5       // check appropriate format
6
7       stream.seekg(0, std::ios::end);
```

```
    headerParameters.fileSize = stream.tellg();
    stream.seekg(0, std::ios::beg);

    char ch;
    string header = "";
    // Attention that '\n' after Frame get's deleted
    while (stream.get(ch) && !check_contains_frame(header)) {
        header += ch;
    }

    //cout << "Header: " << header << endl;

    // Takes only space separated C++ strings.
    stringstream headerStream(header);
    string parameter;
    // Extract word from the stream (remove whitespaces)
    int count = 0;
    while (headerStream >> parameter) {
        if (count == 0)
            headerParameters.format = parameter;
        else if (count == 1)
            headerParameters.chroma = parameter;
        else if (count == 2)
            headerParameters.width = std::stoi(parameter.substr(1));  // Skip 'W'
        else if (count == 3)
            headerParameters.height = std::stoi(parameter.substr(1));  // Skip 'H'
        else if (count == 4) {
            size_t pos = parameter.find(':');
            if (pos != std::string::npos) {
                int frameRateNumerator = std::stoi(parameter.substr(1, pos - 1));  // before ':'
                int frameRateDenominator = std::stoi(parameter.substr(pos + 1));   // after ':'
                headerParameters.fps = static_cast<int>(frameRateNumerator) / frameRateDenominator;
            }
        } else if (count == 5)
            headerParameters.interlace = parameter;
        else if (count == 6)
            headerParameters.aspectRatio = parameter;

        count++;
    }

    headerParameters.frameSize =
        headerParameters.width * headerParameters.height * 1.5;

    // Count header, but remove the inclusion of Frame keyword
    int headerLength = header.length() - 6;

    // Excluding header size
    int64_t frameFileSize = headerParameters.fileSize - headerLength;

    // Include Frame keyword
```

```
    int frameS = headerParameters.frameSize + 6;
    headerParameters.numberFrames = static_cast<int>(frameFileSize
    / frameS);

    return headerParameters;
```

This function reads the firsts line of the file and gets the parameters that are relevant for the class. After that, it calculates the number of frames and the bytes per each frame.

```
Mat Movie::read_frame(std::fstream& stream) {
    if (!stream.is_open()) {
        std::cerr << "Error: movie file is not open" << std::endl;
        return cv::Mat();
    }

    // Check for the end of file
    if (stream.eof()) {
        std::cerr << "Reached end of file while reading frame" <<
    std::endl;
        return cv::Mat();  // Return an empty Mat object or handle
    appropriately
    }

    // Allocate memory to read a frame, but only Y plane
    int dataToRead =(int) ((double) headerParameters.frameSize/1.5)
    ;
    char* frameData = new char[dataToRead+1];
    frameData[dataToRead] = '\0'; // add termination character

    // Read the frame data
    stream.read(frameData, dataToRead);

    // Ignore the remaining planes and "FRAME\n" tag at the
    beginning of the next frame
    int offset = 20;
    stream.ignore(headerParameters.frameSize - dataToRead + offset,
     '\n');

    // Convert the frame data to a Mat object with just the Y field
    cv::Mat frame(headerParameters.height, headerParameters.width,
    CV_8UC1, frameData);

    delete[] frameData;  // Free memory

    return frame;
}
```

This function reads the Y plane only, ignoring the next planes, and returns it as a Mat object.

```
void Movie::write_movie_header(std::fstream& stream) {
    if (!stream.is_open()) {
        std::cerr << "Error: movie file is not open" << std::endl;
        return;
```

```
    }

    // Write Y4M header information to the file
    stream << headerParameters.format << " ";
    stream << headerParameters.chroma << " ";
    stream << "W" << headerParameters.width << " ";
    stream << "H" << headerParameters.height << " ";
    stream << "F" << headerParameters.fps << ":1 ";
    stream << headerParameters.interlace << " ";
    stream << headerParameters.aspectRatio << std::endl;
}
```

This function writes the header into the new reconstructed file

```
void Movie::write_movie_frame(std::fstream& stream, cv::Mat& frame)
     {
    if (!stream.is_open()) {
        std::cerr << "Error: movie file is not open" << std::endl;
        return;
    }

    // Write the frame data to the file in Y4M format
    stream << "FRAME" << std::endl;

    if (frame.channels() != 1) {
        std::cerr << "Error: Expected grayscale frame format (Y
    plane only)" << std::endl;
        return;
    }

    int yWidth = frame.cols;
    int yHeight = frame.rows;
    int uvWidth = yWidth / 2;  // U and V planes have half the
    width and height of Y plane
    int uvHeight = yHeight / 2;

    // Prepare U and V planes (filled with 127, as mentioned)
    cv::Mat uPlane = cv::Mat::ones(uvHeight, uvWidth, CV_8UC1) *
    127;
    cv::Mat vPlane = cv::Mat::ones(uvHeight, uvWidth, CV_8UC1) *
    127;

    // Write Y plane
    for (int i = 0; i < yHeight; ++i) {
        stream.write(reinterpret_cast<char*>(frame.ptr(i)), yWidth)
    ;
    }

    // Write U and V planes
    for (int i = 0; i < uvHeight; ++i) {
        stream.write(reinterpret_cast<char*>(uPlane.ptr(i)),
    uvWidth);
    }
    for (int i = 0; i < uvHeight; ++i) {
        stream.write(reinterpret_cast<char*>(vPlane.ptr(i)),
    uvWidth);
    }
```

```
}
```

This function writes an Y frame into a reconstructed movie file, as well as the U and V plans filled with the mentioned values.

2.2.5 Video Encoder

The video encoder is used for encoding the frames that come from an y4m file. It firstly gets the parameters from the file, then computes the number of blocks per frame and finally it reads frames and encodes them by blocks.

```
int main(int argc, char* argv[]) {
    int ret = process_arguments(argc, argv);
    if (ret < 0)
        return 1;
    else if (ret > 0)
        return 0;

    clock_t startTime = clock();

    // TODO - Check .pgm or .y4m file

    fstream movieStream;
    movieStream.open(Options::fileName, std::fstream::in | std::
    fstream::binary);

    if (!movieStream.is_open()) {
        std::cerr << "Error: Could not open file " << Options::
    fileName << std::endl;
        return -1;
    }

    Movie movieClass = Movie();
    movieClass.get_header_parameters(movieStream);

    // Create Golomb Encoder class
    GEncoder gEncoder(Options::encodedName, Options::m, Options::
    predictor);

    File f;
    f.type = Y4M;
    f.blockSize = Options::blockSize;
    f.nFrames = movieClass.get_number_frames();
    f.chroma = C420jpeg;  // movieClass.getChroma(); Change this in
     the future
    f.width = movieClass.get_width();
    f.height = movieClass.get_height();
    f.fps = (uint8_t)movieClass.get_fps();
    f.approach = Options::approach;
    f.lossy = Options::lossy;

    int numBlocksWidth = f.width / Options::blockSize;
    int numBlocksHeight = f.height / Options::blockSize;

    if (f.width % Options::blockSize != 0)
```

```
        ++numBlocksWidth;  // Add another column of blocks not fully occupied

    if (f.height % Options::blockSize != 0)
        ++numBlocksHeight;  // Add another row of blocks not fully occupied

    size_t nBlocksPerFrame = static_cast<size_t>(int(numBlocksWidth * numBlocksHeight));

    Options::nFrames = f.nFrames;

    print_processing_information(nBlocksPerFrame);

    gEncoder.encode_file_header(f, nBlocksPerFrame, Options::intraFramePeriodicity);

    std::cout << "Video processing starting..." << std::endl;

    Mat mat = Mat();
    int frameCounter = 0;
    while (true) {
        std::cout << " - Processing frame: " << std::setw(4) << frameCounter << "/" << std::setw(4)
                  << Options::nFrames << "\r" << std::flush;
        Mat frame = movieClass.read_frame(movieStream);
        if (frame.size() == mat.size())
            break;
        //Frame::display_image(frame);
        gEncoder.encode_frame(frame, frameCounter);
        frameCounter++;
    }

    std::cout << "Video processing finished. All good!\n" << std::endl;

    movieStream.close();

    clock_t endTime = clock();
    std::cout << "Program took " << std::fixed << std::setprecision(2)
              << (double(endTime - startTime) / CLOCKS_PER_SEC)
              << " seconds to run. Video compressed to " << Options::encodedName << std::endl;

    return 0;
}
```

2.2.6 Video Decoder

The video decoder decodes the header, along with the frames and constructs a new y4m file.

```
int main(int argc, char* argv[]) {
    int ret = process_arguments(argc, argv);
    if (ret < 0)
```

```
        return 1;
    else if (ret > 0)
        return 0;

    clock_t startTime = clock();

    // Create Golomb Encoder class
    GDecoder gDecoder(Options::encodedName);

    int nBlocks = gDecoder.read_file_header();

    File f = gDecoder.get_file();
    print_processing_information(f, nBlocks);

    fstream movieStream;
    movieStream.open(Options::movieName, std::fstream::out | std::
    fstream::binary);

    if (!movieStream.is_open()) {
        std::cerr << "Error: Could not open file " << Options::
    movieName << std::endl;
        return -1;
    }

    HeaderParameters movieParams;
    movieParams.format = "YUV4MPEG2";
    movieParams.chroma = "C420jpeg";
    movieParams.width = f.width;
    movieParams.height = f.height;
    movieParams.fps = f.fps;
    movieParams.interlace = "Ip";
    movieParams.aspectRatio = "A1:1";
    movieParams.numberFrames = f.nFrames;

    Movie movieClass = Movie();
    movieClass.set_headerParameters(movieParams);
    movieClass.write_movie_header(movieStream);

    std::cout << "Video Decoding starting..." << endl;

    for (int fId = 1; fId <= (int)f.nFrames; fId++) {
        Mat decodedFrame = gDecoder.decode_frame(fId);
        movieClass.write_movie_frame(movieStream, decodedFrame);
        //Frame::display_image(decodedFrame);
    }

    movieStream.close();

    std::cout << "All Frames read and decoded. All good!\n" << endl
    ;

    clock_t endTime = clock();
    std::cout << "Program took " << (double(endTime - startTime) /
    CLOCKS_PER_SEC) * 1000
              << " ms to run. Movie decompressed to " << Options::
    movieName << std::endl;
```

```
56      return 0;
57  }
```

2.2.7 Usage

To compile the code, navigate to the $class_3$ folder and execute the command:

```
1  make
```

Then navigate to the *bin* folder and launch the program using the following command:

```
1  ./video_encoder [OPTIONS]
```

To check the available options, use the -h option, like:

```
1  ./video_encoder -h
```

This command will generate the following text in the console:

```
1  Usage: %s [OPTIONS]
2    OPTIONS:
3    -h, --help         --- print this help
4    -i, --input        --- set image/video file name (default: ../
       movies/sintel_trailer_2k_480p24.y4m)
5    -o, --output       --- set encoded file name (default:
       encodedMovie)
6    -b, --blockSize    --- set block size (default: 1024)
7    -l, --lossy x       --- set lossy compression with x quantization
       (default: 1)
8    -m, --modulus      --- set m number (default: automatic
       calculation)
9    -p, --predict      --- set predictor [0,7] (default: JPEG1)
10   -f, --framePeriod --- set intra frame periodicity (default: 10)
11   -a, --approach     --- set approach [0,1] (default: SIGN_MAGNITUDE
       )
```

Listing 2.10: video_encoder arguments

As present in the help menu, to set the input and output file names, use flags $-i$, with valid *.y4m*, or $-o$ files. The $-b$ flag is used to define the size of each block, $-l$ needs to be selected in this case to choose lossy compression and the the number of quantization levels, $-m$ for the golomb m, $-p$ to set a specific predictor, $-f$ to choose the periodicity of $Intra-frames$ and $-a$ for the golomb approach.

After that, we need to decode the binary file that resulted from the encoding. For that, we use:

```
1  ./vide_decoder [OPTIONS]
```

To check the available options, use the -h option, like:

```
1  ./vide_decoder -h
```

This command will generate the following text in the console:

```
Usage: %s [OPTIONS]
  OPTIONS:
  -h, --help         --- print this help
  -i, --input        --- set encoded file name (default:
    encodedSample)
  -o, --output       --- set decoded music name (default:
    decodedMusic.wav)
```

Listing 2.11: video_decoder arguments

As present in the help menu, to set the input and output file names, use flags $-i$, with valid *.bin* or $default$ file type, $-o$ with valid $.y4m$ file name.

2.2.8 Results

Executing our Video Encoder, we manage to reduce the size of a file with 770.5 Mb to around 423.3 Mb, still with lossless encoding, this was achieved by using a fixed predictor, $-p$ 7 which corresponds to the **JPEG-LS**, automatic determined m parameter and 10 fixed $Intra - Frames$, our program takes 141.61 seconds to encode and 80.01 seconds to decode

Chapter 3

Part III

3.1 Exercise 4

3.1.1 Video compare program

To help us assessing the quality of lossy compressed video sequences, we developed a "video_cmp.cpp" program, that compares two videos in terms of the peak signal to noise ratio (PSNR).

We start by creating two movie objects, one for the original file and the other one for the reconstructed one. After this we obtain the width and height of both and make sure they have the same dimensions in order to be compared. We also check if they have the same number of frames.

Then we start iterating through the frames, and obtaining the Mat objects for each one. Since those Mat objects are obtained using the read_frame they already contain only the Y values of the YUV scale. These values are then stored in two vectors, one for each file.

```
Mat originalFrame = originalMovie.read_frame(originalFile);
Mat reconstructedFrame = reconstructedMovie.read_frame(
    reconstructedFile);

if (originalFrame.size() == mat.size() || reconstructedFrame.size()
     == mat.size()){
    break;
}

// Get the Y values from the original frame
for (int i = 0; i < originalFrame.rows; i++){
    for (int j = 0; j < originalFrame.cols; j++){
        originalYValues.push_back(originalFrame.at<uint8_t>(i, j));
    }
}

// Get the Y values from the reconstructed frame
for (int i = 0; i < reconstructedFrame.rows; i++){
    for (int j = 0; j < reconstructedFrame.cols; j++){
```

```
        reconstructedYValues.push_back(reconstructedFrame.at<uint8_t>(i, j));
    }
}
```

After having stored all the Y values, we can calculate PSNR, given by the following formula:

$$\text{PSNR} = 10 \cdot \log_{10}\left(\frac{A^2}{e^2}\right) \tag{3.1}$$

where A is the maximum value of the signal (typically 255), e2 is the mean squared error between the reconstructed frame, $\tilde{f}$, and the original frame, f

$$e^2 = \frac{1}{NM}\sum_{r=1}^{N}\sum_{c=1}^{M}[f(r,c) - \tilde{f}(r,c)]^2 \tag{3.2}$$

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

The calculation of the PSNR in the code is done as below:

```
// Calculate PSNR
        double e2 = 0; // Mean squared error
        for (int i = 0; i < originalYValues.size(); i++){
            e2 += pow((originalYValues[i] - reconstructedYValues[i]), 2);
        }
        e2 /= originalYValues.size();

        double psnr = 10 * log10(pow(255, 2) / e2);

        if (psnr != INFINITY){
            // Update peak PSNR
            if (psnr > peakPSNR){
                peakPSNR = psnr;
            }

            // Update average PSNR
            avgPSNR += psnr;
        }
```

After reading all the frames we are able to obtain both the peakPSNR and avgPSNR (dividing the sum by the number of frames).

In the *video_encoder* program, we created a new function to quantize a given value and called that function, when doing prediction and after obtaining the error.

```
uint8_t GEncoder::quantize_error(uint8_t cell) {
    cell >>= fileStruct.quantizationLevels;
    return cell;
}
```

Listing 3.1: quantize_error function

```
Block GEncoder::process_block(Mat& block, int blockId, int nBlocks,
     PREDICTOR_TYPE pred,
                                FrameSegment& frame) {

    Block encodedBlock;

    // If it's an inter-frame, no need to test data
    if (frame.type == I) {
        encodedBlock.type = I;
        encodedBlock.data = Frame::mat_to_linear_vector(block);
        return encodedBlock;
    }

    // Predict the data and create a new predicted object
    cv::Mat predictBlock = cv::Mat::zeros(block.size(), block.type
    ());
    for (int i = 0; i < block.rows; ++i) {
        for (int j = 0; j < block.cols; ++j) {
            int prediction = predictorClass.predict(pred, block, i,
     j);
            int error = block.at<uint8_t>(i, j) - prediction;
            if (fileStruct.lossy)
                error = quantize_error(error);
            predictBlock.at<uint8_t>(i, j) = static_cast<uint8_t>(
    error);
        }
    }

    // Use golomb to test the best frame type to encode and save
    the result
    std::vector<uint8_t> blockVector = Frame::mat_to_linear_vector(
    block);
    std::vector<uint8_t> predictBlockVector = Frame::
    mat_to_linear_vector(predictBlock);
    int bitsIntra = golomb.get_bits_needed(blockVector);
    int bitsPredict = golomb.get_bits_needed(predictBlockVector);
    if (bitsIntra < bitsPredict) {
        encodedBlock.type = I;
        encodedBlock.data = blockVector;
    } else {
        encodedBlock.type = P;
        encodedBlock.data = predictBlockVector;
    }

    return encodedBlock;
}
```

Listing 3.2: Changes in process_block function

3.1.2 Usage

To execute this program, follow the advice from exercise 2, but use the program argument $-l1$, with 1 being the number of quantization levels to discard.

3.1.3 Results

As we can see by Figure 3.1, the quantization is applied in blocks, which results in the frame appearing "blocky".

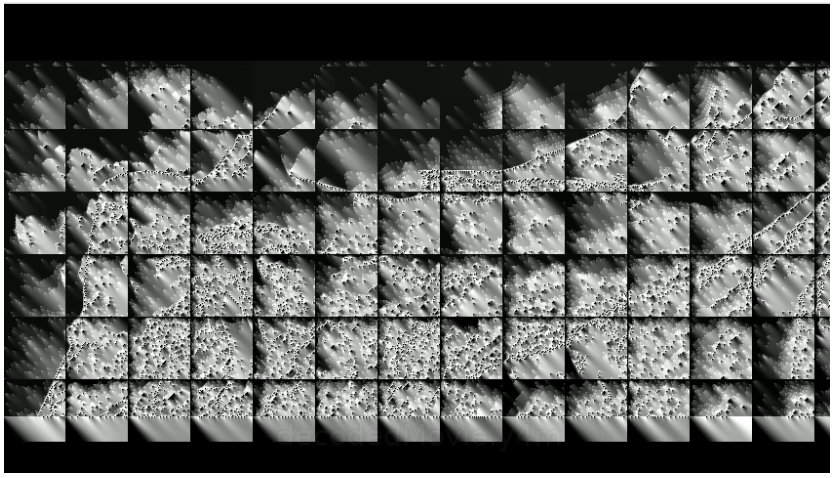

Figure 3.1: Lossy frame from movie

Contributions

The work carried out in this is distributed as follows:

1. Gonçalo Silva - 48%
2. Tiago Alves - 28%
3. Samuel Teixeira - 24%