# Audio and Video Coding
# Assignment 1

André Pinho
Email: andre.s.pinho@ua.pt
Nmec: 80313
DETI
UA

Diego Hernandez
Email: dc.hernandez@ua.pt
Nmec: 77013
DETI
UA

Margarida Silva
Email: margaridaocs@ua.pt
Nmec: 77752
DETI
UA

***Abstract*—Report of our implementation of the Assignment 1 of Audio and Video Coding.**

## I. How to compile

Some exercises require gnuplot to be installed. To install gnuplot on Ubuntu, run *sudo apt-get install gnuplot*. We are using CMake all exercises, so to compile an exercise:

1) Set the current dir to the exercise dir;
2) (Optional) Create and enter a *build/* folder;
3) Call *cmake* .

A *bin/* folder should appear on the current directory. The binaries reside inside it.

## II. Exercises

### A. Part I - Audio and Image/Video manipulation

*1) Exercise 1:* The implementation of this is exercise is carried out in the project wav. In this project it is defined a C++ class named Wav. This project does not take advantage of any available sound library, thus each type of processing and sound manipulation is done with the raw sound data. This class is in charge to store and process through different methods with different purposes the data of a Waveform Audio File Format (WAV) file.

First, to make use of this class and thus its methods, it is needed to instance an object of class Wav. To accomplish this, it is needed to pass an input argument containing the path of the WAV file of interest. As soon as this is done, the header and the data of a WAV file is interpreted and stored, processing such data to register essential information needed by the functioning of the class's methods.

In order to take context of the WAV data, it is crucial to first understand the given information disposed at the header file, and thus it is crucial to understand its format.

It is of valuable use, by the class methods, the following fields:

- **NumChannels**: The number of channels of a WAVE file;
- **BitsPerSample**: bits per sample;
- **Subchubk2Size**: The number of bytes in the data;
- **Data**: Sound data.

For the processing of the sound data, it was crucial to be careful with the default byte ordering assumed by WAV data files, which is little-endian. Files written using the big-endian byte ordering scheme have the identifier RIFX instead of Resource Interchange File Format (RIFF). During the development of the project, we only focused on RIFF, since it is the most popular.

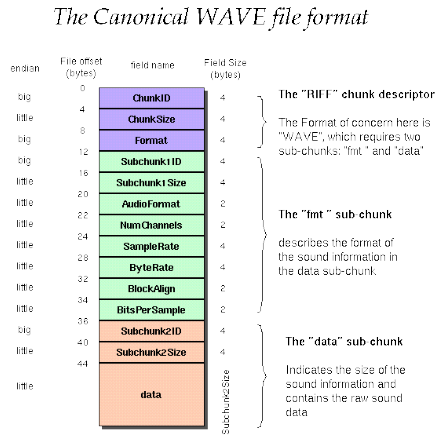

Fig. 1. WAV data format.

To gather such header data, of a WAV file, in the corresponding fields, a C++ structure was developed, named *wav_hdr*, to easily collect the information of interest. Once a WAV file is open to read the information is extracted and stored on the specified fields of the structure. The variable *wavHeader* contains all the obtained header information. Then, the program proceeds with the reading of the WAV file sound data and stored it in memory pointed out by the char pointer named *wavData*. Finally after this extraction, it is calculated crucial information such as the total number of samples and bytes per samples, which are used by some methods of the class.

To visualize the collected header information the programmer can make use of the Wav method *readHeader*.

The method responsible to copy a WAV file, sample by sample, is *cpBySample* function, whose input parameter is the output WAV file where the copy write must be done. First it is copied the integral part of the header data to the output file. Then, because it is requested, iteratively the program goes through each sample, taking in mind the length of the WAV file and the length of a sample in bytes. It is pointed out the memory location of the correspondent sample data at interest and it is written to the output WAV file. This method neither opens the output file neither closes it. These tasks must perform outside the mentioned method.

2) *Exercise 2:* Using OpenCV, this program iterates over the source image and copies the value of the pixel to a Matrix that has the same configuration as the source one. Since the program can't assume the depth of the image, we use a template in which it can be specified the depth of the images, using the Vector datastructure. After copying it, the program also shows the image in a window.

3) *Exercise 3:* This program is able to either show different images or play different videos, making use of OpenCV. The program will expect image files if the user provides it with the flag "-i" and video if the flag is "-v", but only one can be present. Any number of files can be displayed and the image/video being shown can be changed by using the left and right arrow keys. The program terminates when the user presses "q" or "ESC".

4) *Exercise 4:* To process audio files for this exercise, the Sndfile library was used. The program receives one audio file as an argument and reads each sample, storing the contents in a vector, using the even indexes for the first channel and the odd indexes for the second channel. After reading all samples, the number of occurrences of a certain value is calculated for each channel individually; the mono version is calculated at the same time, by counting the occurrences of the average of both channels. Finally, to show the histogram, we made use of Gnuplot. This is accomplished by piping to Gnuplot through an external function developed by Martin Rünz (https://github.com/martinruenz/gnuplot-cpp). For each of the channels, the value and the corresponding frequency are sent to gnuplot and in the end the three histograms are finally drawn.

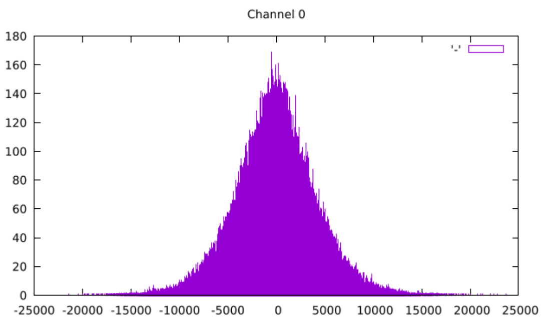

Fig. 2. Histogram of one channel of sample01.wav

These results seem reasonable, since a regular audio file should have a close to binomial distribution in terms of frequency and a sine wave will reach all frequencies uniformly.

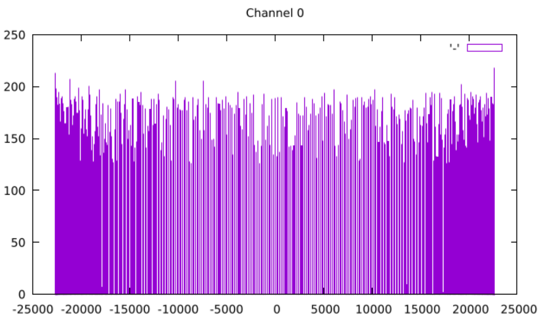

Fig. 3. Histogram of a 1kHz sine wave

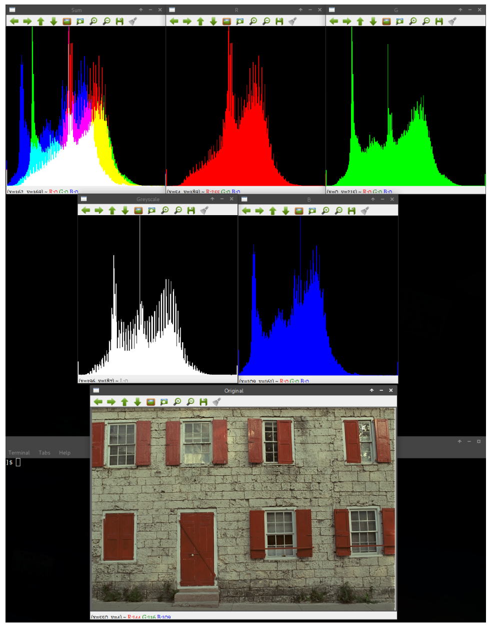

Fig. 4. Example histogram of an image.
Command: *../bin/hist ../../images/kodim01.png 400 400*

5) *Exercise 5:* Since OpenCV doesn't have a built-in way drawing histograms, we implemented one. In the *hist.cpp* resides the code that accomplishes this task. Its main function (*create_hist(Mat&,Mat&)*) receives the input matrix to be processed, and the output matrix where the histogram is supposed to be drawn. It takes to account the size of the output matrix and calculates the histogram accordingly to those. We used C++ templates to better generalise our code. The main program receives as arguments the image/video to process, and the dimensions of the window. When run, the program displays the image, the histogram of each channel, the grayscale histogram, and the RGB mix of the histograms (Fig. 4).

6) *Exercise 6:* Quantization is representing the sampled values of the amplitude by a finite set of levels, which means converting a continuous-amplitude sample into a discrete-

time signal. This lossy compression techniques are usually irreversible, degrading what it is introduced to it.

There are two types of uniform quantization. They are Midrise type and Midtread type.

The Midrise type is so called because the origin lies in the middle of a raising part of the stair-case like graph. The quantization levels in this type are even in number.

The Midtread type is so called because the origin lies in the middle of a tread of the stair-case like graph. The quantization levels in this type are odd in number.

Both the Midrise and Midtread type of uniform quantizers are symmetric about the origin.

For this exercise it has been implemented both of the mentioned quantization methods.

For the Midrise quantization, we extend all intervals with the same size, and thus the reconstruction of the sample values are also spaced. This extension consists in truncating the least significant bits to zero, leading to an appropriate mapping of the values to the correspondent output values.

For the Midtread quantization, we also extend all intervals with the same size, and consequently the reconstruction of the sample values are also spaced in multiple of uniform intervals.However, the approach for this methodology is different, since the value of the sample will modify to the most approximate possible mapped output value resulted by the quantization.

At the wav project in the Wav class there are methods responsible for the appliance of the mentioned quantization methods to the Wav file sound data.

The *midriseQuant* is a function that modifies the values of each sample of the WAV file according to the methodology mentioned for the Midrise uniform quantization.

As input arguments there is: the number of least significant bits to switch to zero; a char pointer to the allocated memory in order to store the resulting quantized samples; and the total length in bytes of the mentioned allocated memory.

In this function we calculate the number of bytes to directly truncate to zero, and the bit mask to apply at the last byte of the initial sample who suffers from bit switching, resulted by the wanted quantization.

Then iteratively we go through each sample of the WAV file initially specificated at the instantiation of the Wav class object, truncate to zero the bytes which bits are supposed to fully be switched to zero, and apply the bit mask to the last least significant byte which some of their bits suffers from the quantization bit truncation. This method considers the default little-endian byte ordering scheme of RIFF.

The *midtreadQuant* function modifies the values of each sample of the WAV file according to the methodology mentioned for the Midtread uniform Quantization. Input arguments are the same as the midriseQuant function, having the same purpose.

At this function, according to the number of least significate bits to truncate to zero, it is calculated the resulting spacing interval of the output sample value. It is also determined the bitmask to apply to the sample for the quantization operation.

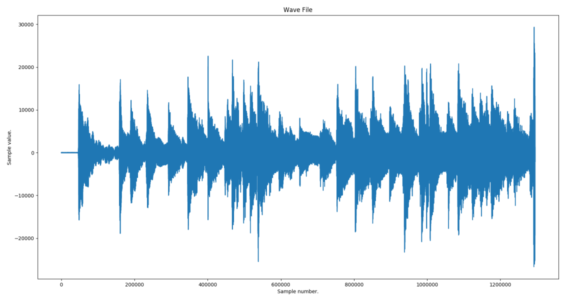

Fig. 5. Graph of samples' values of WAV file sample02.wav

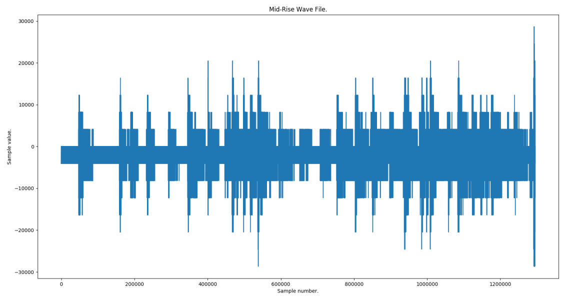

Fig. 6. Graph of samples' values resulted from the Midrise quantization of WAV file sample02.wav with truncation of the 12 first least significant bits

Iteratively the function goes through each sample, sums half of the mentioned interval and apply the bitmask, in order to follow the Midtread operation methodology. All sample's bytes are written sequentially to the allocated memory in a little-endian approach.

Additionally it has been implemented the function *encMidtreadUniQuant* and *encMidtreadUniQuant*, which by specifying the number of bits to switch to zero and the output WAV file, it applies the Midrise Quantizer or Midtread Quantizer methods, respectively, to the samples of the WAV file. It writes them out, in a little-endian approach according to RIFF, to the output file. This lets the programmer experience the results of such modification of the sound data, and allows to compare the different outputted audio files with different sample values spacing, by hearing the output file or by displaying the values of each samples of the WAV file through graphs as shown in the figures 5, 6 and 7.

*7) Exercise 7:* In this exercise we also implemented the Mid-Rise and Mid-Tread quantizing algorithms. For the Mid-Rise algorithm, the N least significant amount of bits of each channel, are simply truncated, leaving only the most significant ones. For the Mid-Rise, before truncating the data, half the amount of the interval is added. This way, the values are not floored, but rounded to the nearest neighbour. To complement these two algorithms we also adventured into a more complex one: Linde-Buzo-Gray. In this quantization algorithm, the image is simplified to a defined amount of (N) vectors. The

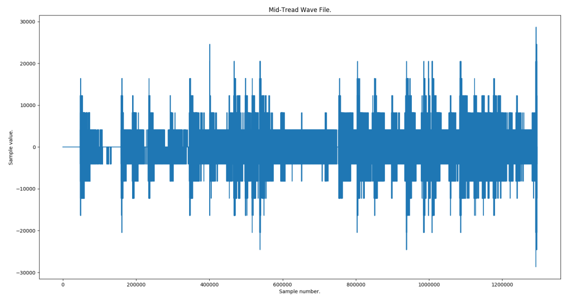

Fig. 7. Graph of samples' values resulted from the Midtread quantization of WAV file sample02.wav with truncation of the 12 first least significant bits

algorithm's job is to find the N most representative colors of that image, so that it can be represented with those. This algorithm resembles a Unsupervised Learning Clustering Machine Learning algorithm, and uses a calculation of the image's distortion to know when to stop converging.

Our approach to this algorithm was to implement a simplified version, basing it on our knowledge of clustering algorithms. Instead of a calculation of distortion, we assumed that by 20 iterations of the algorithm, the image was in it's final form. The initial values of the vectors resembles a black to white gradient of N steps (function *lbg_init*). We also used templates to allow parametrization on compile-time.

In each iteration, the algorithm calculates the distance from each pixel to each representative vector, finds the closest one, and clusters every pixel to each nearest vector. After that, it calculates the new representative vector, as the centroid of the pixels. This way, the vectors will (hopefuly) converge into the most important ones, improving the outcome.

The output of the main algorithm (function *lbg_calculate*) is a matrix (with the same dimentions of the input image), where each element (pixel) indexes the list of vector. After that, we use another function to map the vectors to the image, alowing us to see the result (function *lbg_apply*).

As expected, the Linde-Buzo-Gray (inspired) algorithm is the one that yields the best results. And can even come incredibly close to the original in complex images with only a handful of colors (Fig. 8).

It also behaves particularly well when only working with a very limited amount of colors (Fig. 9).

*8) Exercise 8:* Signal to Noise Ratio is a measure of signal quality, is related to signal energy and noise energy, is represented in decibel (dB) and is given by the following formula:

$$SNR = 10 * \log_{10}(\frac{Es}{Er}) \tag{1}$$

The implementation to this problem is at the function *getSNR* from the Wav class of the project wav. The input arguments is the type of quantization (in order to this function to operate, it must be passed one of the following defined values:

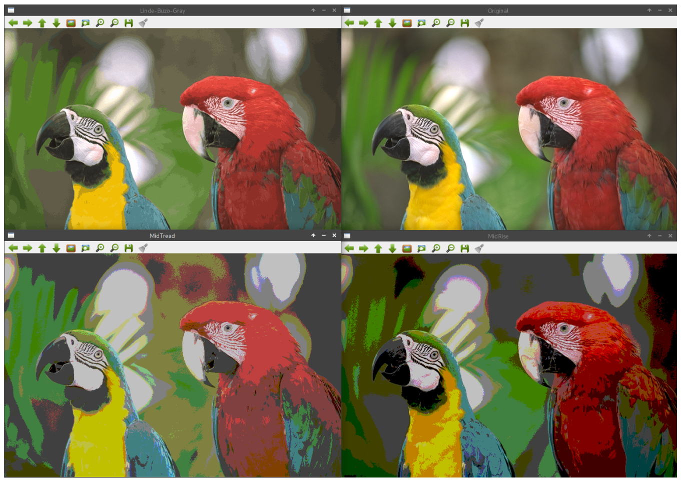

Fig. 8. 64 Colors (2bit per channel) representation of kodim23.png
Clockwise starting on top left: LBG, Original, MidRise, MidTread
Command: *../bin/quant -i ../../images/kodim23.png midtread—midrise—lsb 2 out.png*

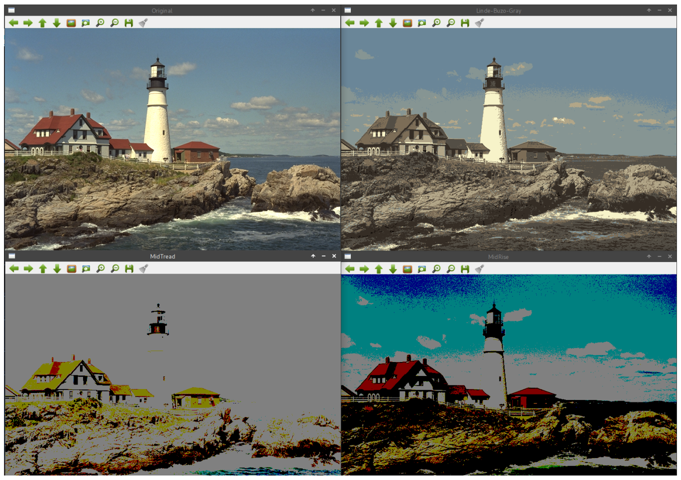

Fig. 9. 8 Colors (1bit per channel) representation of kodim21.png
Clockwise starting on top left: Original, LBG, MidRise, MidTread
Command: *../bin/quant -i ../../images/kodim21.png midtread—midrise—lsb 1 out.png*

*MIDRISE_QUANT* or *MIDTREAD_QUANT*); the number of bits to truncate to zero by the selected quantization approach; and the channel number to do such an operation.

Once the selected quantization method is applied to the data sound, it is collected the value samples, both quantized and non quantized, of the desired channel of the WAV file.

Once these values are collected, the function iterates in each one of the values and calculates the Energy of signal (*Es*) and the Energy of noise (*Er*), until it reaches the values of the last samples, given by the following formulas:

$$Es = \frac{1}{N} * \sum_{i=1}^{N} |x_i|^2 \tag{2}$$

$$Er = \frac{1}{N} * \sum_{i=1}^{N} |x_i - \overline{x_i}|^2 \quad (3)$$

Having calculated both energy of signal and noise the function proceeds to calculate the SNR and return it at the end of the function.

The same function also calculates and returns the maximum per sample absolute error as well.

*9) Exercise 9:* This exercise is similar to the previous one but applied to image. In this case, the PSNR (Peak Signal-to-Noise Ratio) was used instead, which means the Energy of Signal used in the formulas presented in the previous exercise is the maximum energy value. In the case of image, this value is 255 (8 bits at 1).

To obtain the energy of noise, the difference between the matrix of the original image and the modified image is calculated, all elements are then individually squared and finally summed. Using the SNR formula as previously stated, the value is calculated. This is done for each of the three channels

The maximum per pixel error reuses the matrices that contain the difference between both images for each channel. For each pixel, the distance to the original image is calculated as follows:

$$d = \sqrt{R^2 + G^2 + B^2} \quad (4)$$

In which R, G, B are the modulus of the difference in the corresponding channel. In the end, the highest of this values is determined, as that is the Maximum Per Pixel Error.

The program can either receive and image or a video, according to the flag (-i or -v respectively). In the case of video, the PSNR and the MPPE will be printed for each frame. Using the sample kodim01.png, these values were tested using a quantized image and one image in which one specific pixel had been manually switched to another color.

TABLE I
OBTAINED PSNR AND MPPE RESULTS

| | **Quantized image** | **One pixel difference** |
|---|---|---|
| **PSNR** | 16.9841dB | 56.8733dB |
| **MPPE** | 109 | 397 |

We can observe that the quantized image held a much lower signal than the image that only had one pixel changed, which is expected since the former was modified overall. In the case of maximum per pixel error, it was also expected that the image with one pixel changed would have a higher value, since the chosen color was completely different from the pixels surrounding it, while in the quantized image that effect is in some way smoothed out.

## *B. Part II - Entropy*

*1) Exercise 10:* The average information per symbol is known as *entropy*, and can be calculated as in the following formula:

$$H(X) = -\sum_{i=1}^{N} P(x_i) \log_2 P(x_i) \quad (5)$$

Where P(x) is the probability of that symbol occurring.

In the context of audio, the entropy can be translated as the number of bits that are actually required to represent all the range of frequencies in the file.

Our program receives an audio file an reads the samples using the Sndfile library. For each sample, the average of channels is obtained and the probability of that value occurring after the one that was previously registered is increased. At the same time, the frequency of the pair [previous, current] is incremented in a separate data structure. Once all samples have been processed, it's time to iterate through each value and partially apply the formula to each of the possible next symbols to gradually accumulate the final entropy of each symbol. Knowing how many times each symbol occurred, the weighted average of the entropy of all symbols is calculated.

TABLE II
OBTAINED ENTROPY RESULTS

| | **sample07.wav** | **440Hz Sine wave** | **noise** |
|---|---|---|---|
| **Entropy** | 4.74981 | 0.571115 | 5.43869 |

A regular WAV file still required some bits to be represented, but could still be compressed to save some space, as some symbols can be determined by the previous one. Noise, although random, can eventually form some sort of pattern and that allows for an entropy so close to a regular sound file. As expected, a clear sine wave held very low entropy, since it is extremely predictable.

*2) Exercise 11:* This exercise has been divided into two parts. The entropy calculation of an image and the entropy calculation of a video.

Entropy is a measure of image information content, which is interpreted as the average uncertainty of the source of information. In the figure, the entropy states is defined as the level of intensity of individual pixels that can be corresponding adapted. It is used in image analysis and evaluation of quantitative data.

The image entropy calculation is based in the use of the formula presented in the previous exercise, where *n* is the number of gray levels and *P(x)* is the probability of a pixel having gray level *x*.

We have implemented two types of entropy:

An entropy with no direct dependency of selected pixel neighbours intensities, but rather taking into account, at the calculation of the image entropy, the probability of existence of an specific intensity in the image, based on the historic

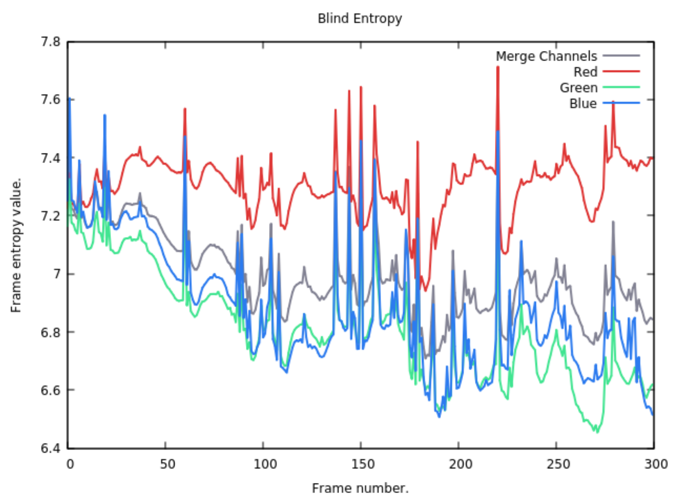

Fig. 10. Graph containing the calculated Blind Entropy values of each frame of the video file crew.y4m. It includes plots that represent each entropy frame values for: the average on channel's entropy (gray plot); the Red channel entropy (red plot); the Blue Channel entropy (blue plot); the Green channel entropy (green plot).

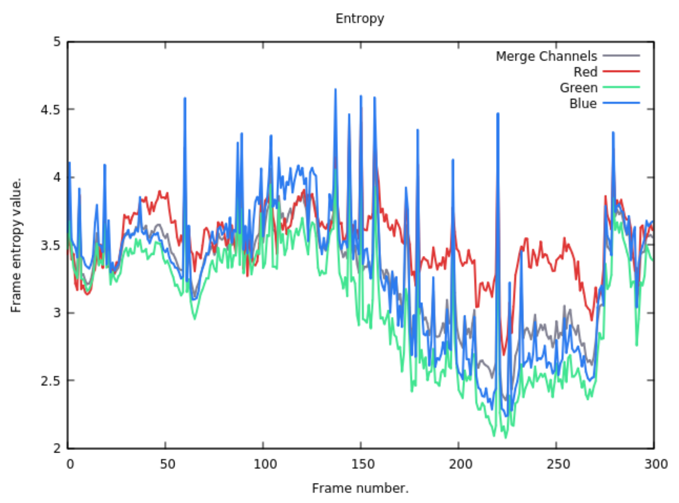

Fig. 11. Graph containing the calculated Entropy values of each frame of the video file crew.y4m. It includes plots that represent each entropy frame values for: the average on channel's entropy (gray plot); the Red channel entropy (red plot); the Blue Channel entropy (blue plot); the Green channel entropy (green plot).

of all the intensities that appears on an image channel, and their quantity (in number of pixels). We named this entropy as Blind Entropy, once no direct relation with neighbour pixels has been established for the entropy calculation. In the ex11 project such entropy calculation is done at the functions: *getChannelBlindEntropy* and *getBlindEntropy*

The other entropy targets the difference between neighbour regions. The region may be described as a pixel or groups of pixels. For this exercise, it has been decided to consider the previous pixel intensity. For the calculation of the image entropy, based on the collected history of the image pixel intensities and their neighbour's ones, and thus their relations, it is taking into account, given the pixel intensity, the probability of the next pixel (from the same row) to have a specific intensity. At the end, having all entropy values, given a previous pixel intensity, we calculate the average value of the entropy of a given image channel. In this exercise we named this entropy Entropy. In the ex11 project such entropy calculation is done at the functions: *getChannelEntropy* and *getEntropy*

For this exercise we calculate the image entropy for each channel, although the programmer can also calculate the average between the entropy's values of the image's channels. These are calculated at functions: *getBlindEntropy* and *getEntropy*.

For the entropy calculation of the videos, this exercise's program is able to get each frame of the video and calculates each channel entropy of the frames, based on a selected entropy calculation method, such as the ones implemented in the functions *getBlindEntropy* and *getEntropy* from the ex11 project. Once the entropy of each frame is collected, is is calculated the average value of entropy for the total entropy (involving the average entropy of each frame channel), the entropy values from the Red channel, the Green channel and finally the Blue channel.