Multimedia

Practical work no 1

Image Destructive Compression – *JPEG*

Report

Sancho Amaral Simões | 2019217590

Tiago Filipe Santa Ventura | 2019243695

Rui Bernardo Lopes Rodrigues | 2019217573

Coimbra, 25th March 2022

1. Index

2. Goal

The goal of the present report is to explain and summarize the work done in the practical project no 1 within the scope of the Multimedia subject of the Computer Engineering Degree at the University of Coimbra. The project main goals consist in the consolidation and application of basic concepts of *Multimedia*, such as information, compression, (particularly the destructive compression) initially acquired in theoretical/theoretical-practical classes *Multimedia* classes.

To put this knowledge into practice, the *Python* programming language and some of its available modules, such as *Numpy, Scipy* and *Astropy* were used in order to build a partial and modified version of the standard destructive *codec JPEG*. It is important to say that much of the concepts acquired in the subject of *Information Theory* (2nd year, 1st semester), such as statistical analysis and entropy coding, were very useful in order to better develop this practical work.

This report, in addition to briefly explaining the concepts related with the work in question, will make known the results obtained and interpretations formed regarding the resolution of the exercises in practical sheet 1, carried out over six weeks of work.

3. Introduction

3.1. Image compression

Image compression is the process of "encoding" an image file in such a way that it will consume less space than the original file.

The Image compression may be:

- **non-destructive**: the size of the image file is reduced without affecting or degrading its quality (i.e. *PNG*, *BMP*, *GIF*).
- **destructive**: the size of the image file is reduced by lowering its quality based on human perceptual knowledge (i.e. *JPEG*).

Image compression is typically performed via an image/data compression algorithm/*codec*. In this work, the analysis/experimentation process was only focused on the destructive image codec *JPEG*, whose steps will be briefly explained.

3.2. *JPEG*

JPEG – Joint Photographic Experts Group – is a standardized image format that allows the destructive compression of images. Its main motivations are the scientific facts concerning the perceptual characteristics of the human eye, particularly what it can usually best distinguish in images. For example, the human eye can better detect variations in green than in red or blue and is less capable of visualize abrupt changes in color that correspond to higher frequencies. All this *a priori knowledge* is the crucial key for the *JPEG* format to be effective as it is.

The *JPEG* compression stack is composed of the following operations:

- *RGB* to *YcbCr* color model conversion.

- Down sampling.

- Division in *8x8* blocks.

- *DCT* (*Discrete Cosine Transform*) applied on the blocks.

- Quantization.

- Compression
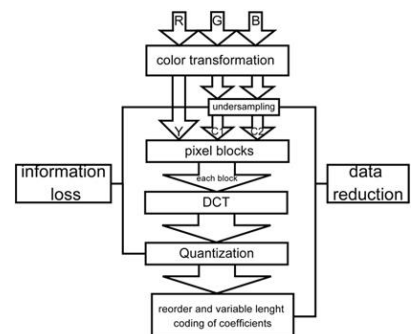
3.2.1. *RGB* to *YcbCr* color model conversion

A color space is a specific organization of colors, and has a correspondent color model that represents the mathematical abstract formula for how those colors can be represented (e.g. triples in *RGB*, quads in *CMYK*).

The interesting thing about this process is that you can convert one color model to another, which means you can change the mathematical representation of a color to totally different values.

R: 233   C: 0
G: 30   M : 0.877
B: 99   Y : 0.593
             K : 0.0471

In *JPEG,* the pixels in *RGB* are converted into pixels in *YcbCr*, a color model which concerns *Luminance* (Y), *Chroma Blue* (*Cb*) and *Chroma Red* (Cr). This process takes advantage of the fact that the human eye is proven to be more sensitive to luminance than chrominance. Hence, it is possible to perform considerable data suppressions in the *CbCr* channels without the human eye noticing it. Another fact that motivates the choice of the *YcbCr* color model as the one used in *JPEG*, and not *RGB*, is that the color information among its channels is not redundant, meaning that it differs greatly from channel to channel, specifically, the luminance component is separated from the chrominance one. On the other hand, in *RGB*, there is a lot of redundancy in its channels, since the luminance is visible in all of them. This will be demonstrated further ahead when separating an image *RGB* channels and viewing each one of them by applying a colormap to them.

3.2.2. Down sampling

As mentioned in the previous section, the channels *Cb* and *Cr* contain less relevant information. Therefore, it is possible to remove some of their data, by resizing them to about one half/quarter of their original size. This removal might be performed by simply truncating rows/columns or. More advanced ways that might give better results is the replacement of two rows/columns rows by their mean or by their interpolation of order n (linear, quadratic, cubic, …). This is the first step in *JPEG* compression stack that is destructive.
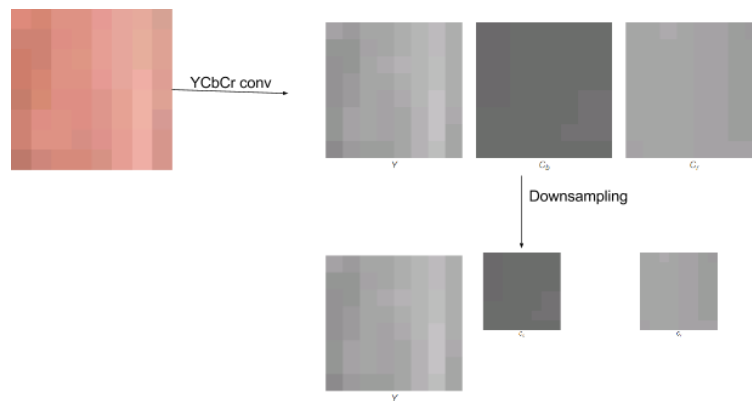
3.2.3. Division in 8x8 blocks

After down sampling the less relevant chrominance channels, the image is divided into blocks of 8x8 pixels. The number 8 may look like a magic number, but it has a very specific reason to be: in such small part of the image, the pixel values are less probable to vary considerably and so, the next steps of the *JPEG* pipeline, such as the *DCT*, will have better results.

Input image   **8x8 block**
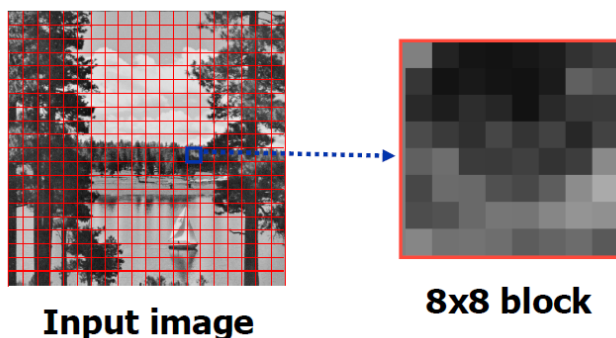
3.2.4. *DCT (Discrete Cosine Transform)*

The core of the *JPEG* starts here, with the application of the *DCT – Discrete Cosine Transform*. The key idea of the *DCT* is that it assumes that any numeric signal can be recreated using a combination of cosine functions. This transform can be applied to any discrete signal and implies the storage of the coefficients of each cosine function. But why using the *DCT* and not other discrete transform such as the *DFT – Discrete Fourier Transform*? The reason is, without going through further details, that the *DCT* shows better energy-compaction properties than the *DFT*[1].

A generic sampled signal

The modulus of its DFT

Its DCT
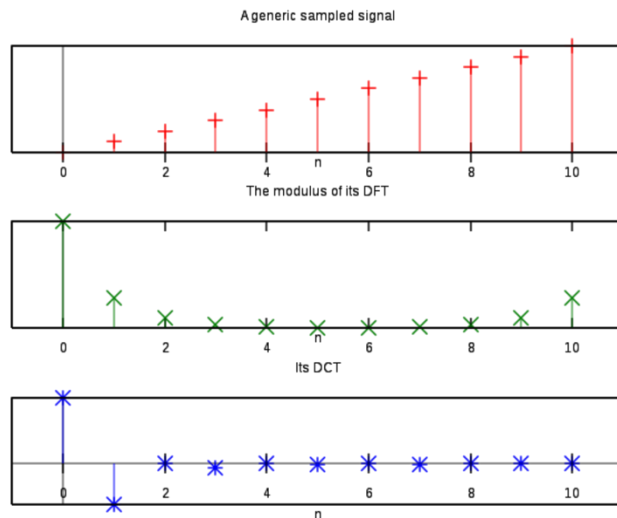
The main idea is that any 8x8 block can be represented as a sum of weighted cosine transforms at various frequencies.

But how should they be weighted together? The following formula must be applied to each block to accomplish this goal.

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y} \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right]$$

With this, it is possible to generate a new 8x8 matrix containing the weights of each cosine component. The following diagram summarizes this process.
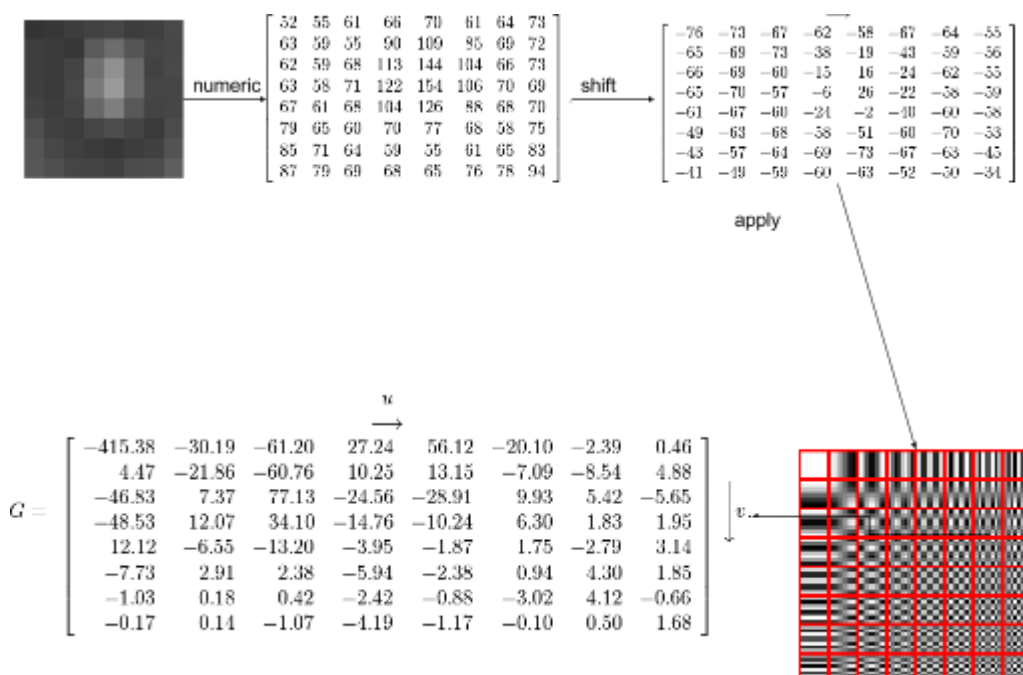
With a matrix of converted byte-aligned integer values into real numbers, the next important destructive step is the quantization one.

3.2.5. Quantization

To convert the weights-matrix back to values in the space of *[0,255]* a standard precalculated matrix of quantization factors like this one shall be used. It is noticeable that, as we go to its bottom right corner, the factors grow. This happens due to the appearance of the higher frequencies' coefficients in the same place as the quantization matrix. Since the human eye is less sensitive to those frequencies, they can be partially discarded without implying great perceptual changes in the decompressed image.

$$Q_l = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Now, both matrices Q and G are used to compute the quantized *DCT* quantized matrix, using the following formula:

$$B_{j,k} = \text{round}\left(\frac{G_{j,k}}{Q_{j,k}}\right) \text{ for } j = 0,1,2,\ldots,7; k = 0,1,2,\ldots,7$$

The result of the previous operation in the previous image is the following matrix:

$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

This process is applied to all channels: *Y, Cb* and *Cr*. Since the last two channels contain less relevant information, a different quantization matrix is applied to them, containing higher factors that will cause heavier data loss.

3.2.6. *DPCM* encoding

For this step it is important to have in mind the two types of coefficients present in each block after the *DCT*/quantization is applied:

- *DC coefficient* ([0, 0]) – the one in the top left corner. It is proportional to the mean of all the block values.

- *AC coefficients* – the remaining coefficients.

For the *DC coefficients* of all blocks, the *DPCM* method (stands for Differential Pulse-Code Modulation), also known as *sub filtering and* delta encoding, is applied, in order to decorrelate the values and, hence, enhance the compression potential for the next step: entropic coding.

3.2.7. Non-destructive compression

The *DCT*, quantization and *DPCM* encoding are all steps that augment the statistical redundancy of the image in overall. In other words, its entropy is reduced, causing it to be more compressible. This redundancy is local, and it is translated for example, by long sequences of zeros and values close to zero. This is an optimal or close-optimal situation for an algorithm such as *RLE – Run-Length-Encoding* – to be applied. Then, entropic coding algorithms, such as the *Huffman Encoding* and *Arithmetic Encoding* might be executed. It is important to point out that this procedure, that causes no data loss, is achieved by running through all the cells in each block in a *zig-zag* way.

4. Procedure

The main goal of this practical work is to know how *JPEG* compression stack is performed in order to destructively compress images, achieving high compression rates. To accomplish this, a program was developed in *Python* that partially recreates the algorithm present in the *JPEG* format. This task was executed through six weeks of group-work, investigation, analysis and experimentation. The following timeline was followed in order to accomplish the mentioned goal:

Week 1

- Creation of a *Git* repository in order to enhance the workflow – *https://github.com/smartlord7/Multimedia-TP1.*

- *S*uperficial analysis of the *JPEG* format to better understand the concepts applied in this project.

- Perceptual analysis of the dataset.

- Implementation of a function that applies *JPEG* compression to the dataset using a premade *Python* library.

- Definition of the *JPEG codec* wrapper functions *encoder* and *decoder*.

- Implementation of functions that allow the image visualization with the *RGB* color model.

- Implementation of a function that converts an image from the *RGB* color model to the *YcbCr* color model and *vice-versa.*

- Implementation of a function that pads an image with sides that are multiple of a certain number of pixels and the inverse one.

Week 2

- Implementation of a function that applies down sampling/up sampling to an image.

- Implementation of a function that applies the *DCT* (applied in all the image in question).

Week 3

- Implementation of a first version of the blockwise *DCT*, using loops.

- Research on how to split a matrix in submatrices and then join them back into the original matrix.

- Implementation of the final optimized version of the blockwise *DCT* function.

- Benchmarking of the implemented blockwise *DCT* functions.

Week 4

- Implementation of a function that applies quantization to each *DCT* block.

- Implementation of a function that *DPCM* encodes the *DC coefficients*.

Week 5

- Implementation of functions that calculate the distortion metrics of the resulting decompressed image – *MSE, RMSE*, *SNR*, *PSNR*.

- *WIP* – Code refactor in order to improve its modularity and organization.

- *WIP* – Code documentation and formatting.

Week 6

- Implementation of a mechanism that allows the automatic generation of results (under the form of plots and log files), based on the given compression parameters.

- Integration of compression time benchmarking.

- Code refactor in order to improve its modularity and organization.

- Code documentation and formatting.

- Final adjustments and tuning.

It is relevant to say that the redaction of this report was continuous and carried out along with the development of the code developed in the scope of this project.

5. Discussion and results

5e.1. Dataset analysis

In order to test the developed *JPEG* partial *codec*, a set of *.bmp* images was provided, being them *barn_mountains.bmp*, *logo.bmp* and *peppers.bmp*
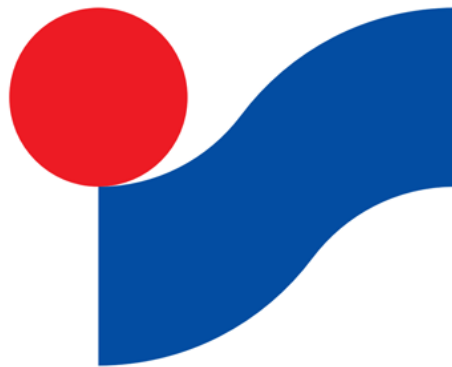
Even though the choice of the dataset may look like random, the group came to the conclusion that it has a very specific reason to be: it contains three images that allows the analysis of JPEG performance in most scenarios. There are apparently three natural and photorealistic images: *barn_mountains.bmp* and *peppers.bmp*. The first one comprises many abrupt changes in colour, e. g. in the mountains/house sections. The second one seems to be much softer in terms of colour transitions. The *logo.bmp* image is clearly an image made using some kind of image editing/graphical tool, being artificial. It is solely composed of polygons, being their boundaries very well defined and the colour changes in their limits very abrupt.

After this analysis made in the beginning of this project and having in mind the core functioning of *JPEG* and some information theory concepts, such as entropy and histogram, the

group was able to establish some predictions about the effectiveness of *JPEG* in each image. Having in account the previous perceptual analysis made about the dataset and the use *JPEG* does with that knowledge, a plausible prediction for the order of the quality of the resulting decompression images would be: *logo.bmp < barn_mountains.bmp < peppers.bmp*. This prediction is purely qualitative. However, a statistical analysis of each image can be performed, using concepts the group learned in the 1st semester of the previous year, in the scope of the *Information Theory* subject. This statistical analysis led to a prediction for the order of the compression ratio of each image in the dataset.
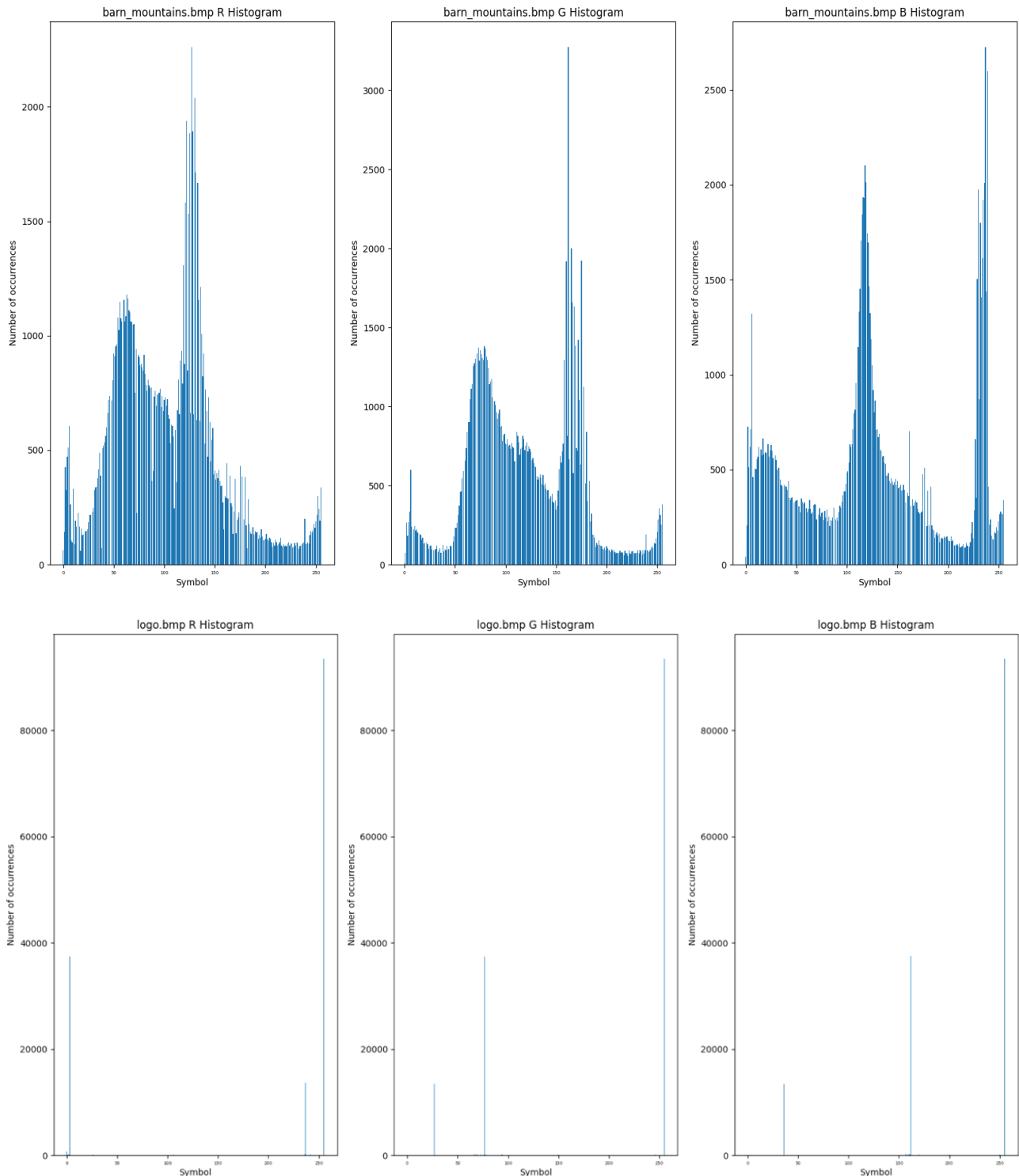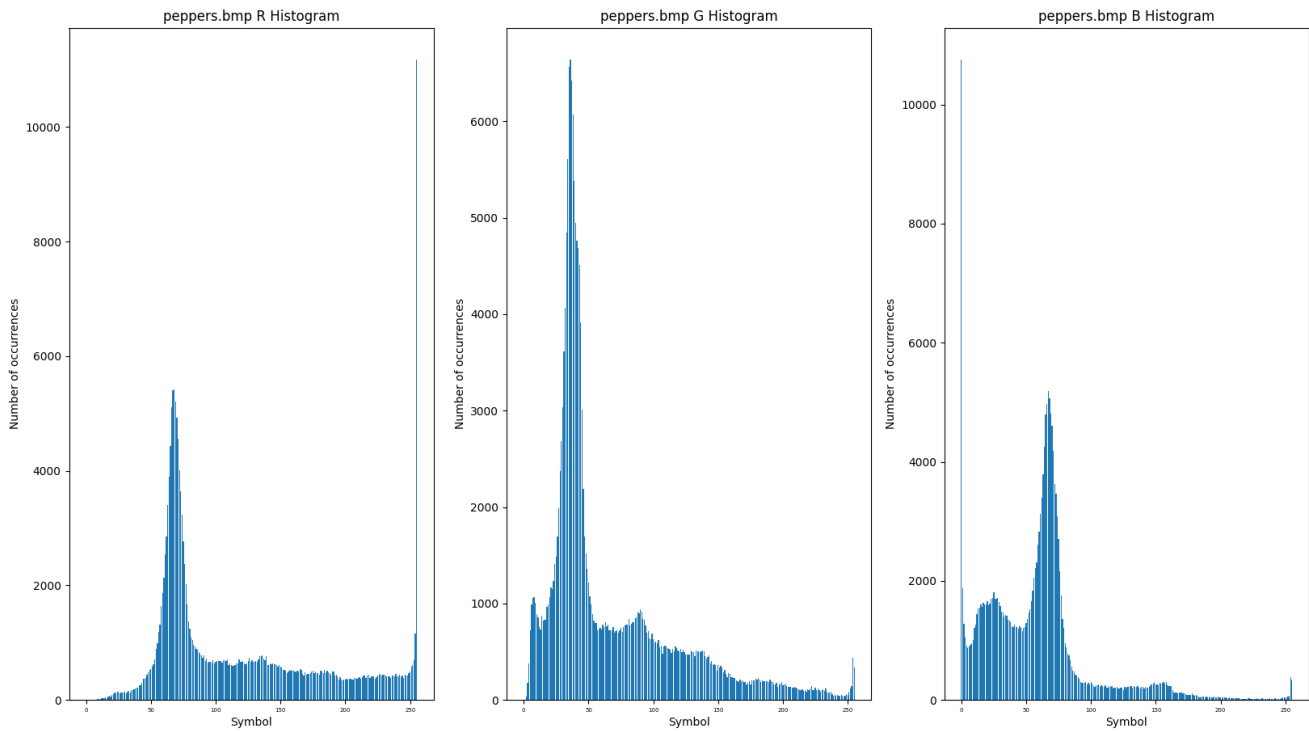
Image \ Channel Entropy (bits)	R	G	B
barn_mountains.bmp	7.33	7.24	7.40
logo.bmp	1.45	1.50	1.48
peppers.bmp	7.19	7.02	6.79

The image *logo.bmp* is clearly the one with most statistical redundancy, followed by *peppers.bmp* and *barn_mountains.bmp*. These results perfectly overlap with the analysis we perform by just looking at each one of the images.

By analyzing the histogram plots and the entropy table for each *RGB* channel of each image, it is possible to say that the prediction for the order of the compression ratio of each image in the dataset would be *barn_mountains.bmp < peppers.bmp < logo.bmp*.

A relevant observation to note is the similarity between the entropies and histograms of each channel of each image. This leads us to point 3.2.1, which says that the RGB color model has high redundancy, since the luminance component is present in all channels.

5.2. Image *JPEG* compression using an existing software

To have a notion of how the developed *JPEG* partial *codec* would behave for the dataset, the group found it important to execute a *JPEG* conversion on it by using existing software, such as an image editor or a library. We choose not to use any image editor and instead used the *Python* library *Pill*. The tables below have all the compression ratios of the different images with the qualities *low* (quality factor of 25), *medium* (quality factor of 50) and *high* (quality factor of 75).

5.2.1. *barn_mountains.bmp (356456B)*

Quality \ Stats	Size (bytes)	Compression ratio
Low	11976	96,640% (30:1)
Medium	18666	94,763% (19:1)
High	28477	92,011% (13:1)

barn_mountains25.jpg   *barn_mountains50.jpg*   *barn_mountains75.jpg*
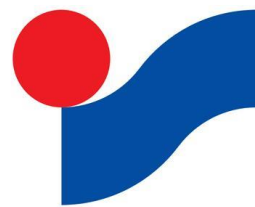
5.2.2. *logo.bmp* (421556B)

Quality \ Stats	Size (bytes)	Compression ratio
Low	5188	98,769% (81:1)
Medium	6328	98,499% (67:1)
High	7838	98,141% (54:1)
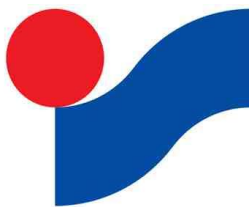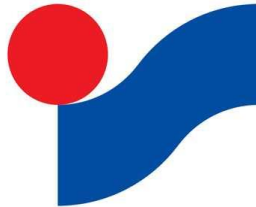
logo25.jpg   *logo50.jgp*   *logo75.jpg*

5.2.3. *Peppers.bmp* (589880B):

Quality \ Stats	Size (bytes)	Compression ratio
Low	10553	98,211% (56:1)
Medium	15671	97,343% (38:1)
High	23509	96,015% (25:1)

peppers25.jpg   *peppers50.jgp*   *peppers75.jpg*

High quality

- *barn_mountains.jpg* – Almost unnoticeable **blockiness** and **contouring** in the mountains-sky transition.

- *logo.bmp* – Some **graining** and **contouring** along the polygon's boundaries.

- *peppers.bmp* – Practically no differences are observable.

Medium quality

- *barn_mountains.jpg* – Intensified **graining** and **contouring** in the mountains-sky transition. Some **blocks** are perceptible in the sky region.

- *logo.bmp* – **Graining** and **contouring** intensify along the polygon's boundaries. Some **phantom frequencies** appear in those regions

- *peppers.bmp* – Some **graining** and **contouring** appearing mostly in the peppers' boundaries with the purple background.

Low quality

- *barn_mountains.jpg* – Even more intensified **graining** and **contouring** in the mountains-sky transition. **Blocks** and **bands** are perceptible in the sky region. **Blocks** are abundant in the mountains region.

- *logo.bmp*   Heavy **phantom frequencies** appearance in the polygons boundaries. The image in general gets very distorted.

- *peppers.bmp* – **Blockiness** is considerably visible over all the image. Since this image is rich in colors and soft color variations, the phenomenon **posterizing** is also observable in regions such as the purple background.

The results obtained in terms of compression ratio and quality are consensual with predictions done in 4.1.

5.3. Visualization of each image with the *RGB* color model

To visualize the images using the *RGB* color model, the image should be separated into three channels *R*, *G*, and *B* with the *separate_channels()* function. Then, by using the *show_images()* function, each channel would be plotted with the appropriate given colormap.
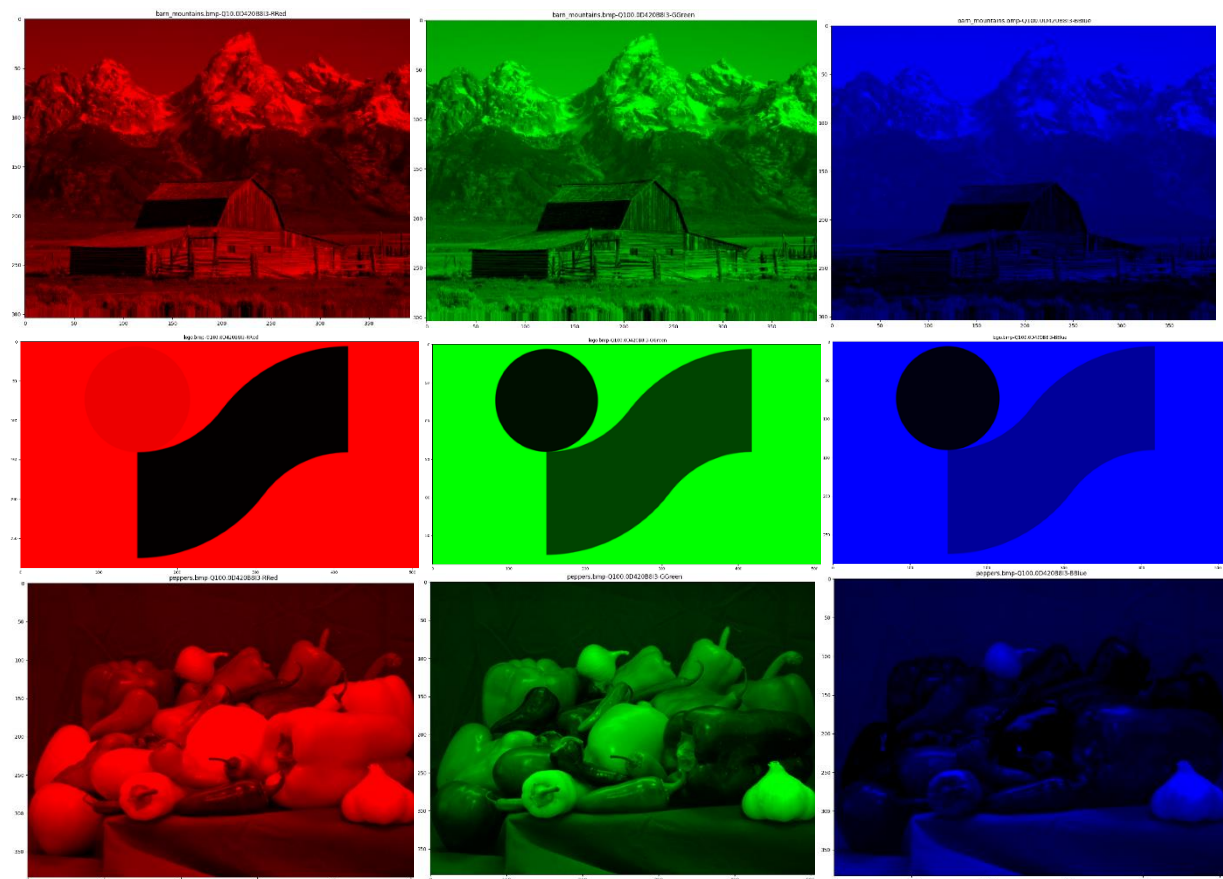
This time, the redundancy of the *RGB* channels is visually proven in the sense that the luminance component is present in each one of them, a point referred to in 3.2.1 and 4.1. It is also noticeable the difficulty of the human eye on distinguishing the red/blue variations compared to the green ones.

5.4. Padding

In *JPEG*, it is standard for the *DCT* to be executed on blocks 8x8. For this to happen, the image width and height must be multiple of at least 16. This is because of the down sampling applied after the conversion to the *YcbCr* color model, which typically reducesthe image size to a half/quarter. Therefore, when an image's dimension is not multiple of 16x16, it is required the application of the padding, which consists in adding lines or columns to the image border till it reaches our intended dimension. To add theses rows/columns the *numpy* functions *np.repeat()* and *np.vstack()/ np.tile()* and *np.hstack()* are used to replicate the last rows/columns, in the function *apply_padding()*.
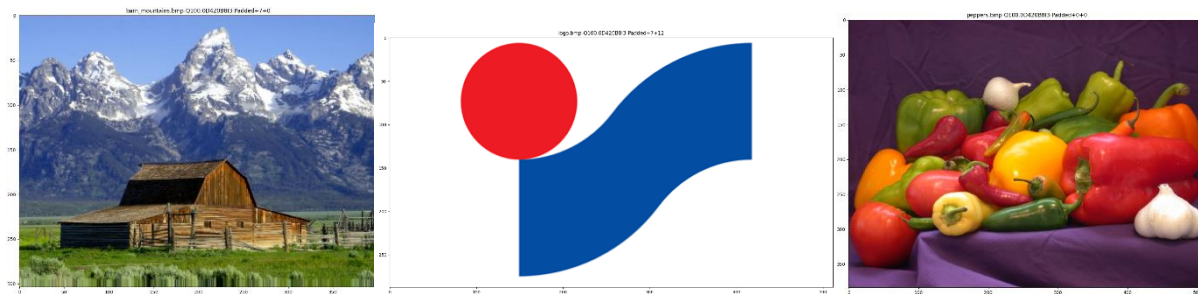

As we can see from the images above, the added last lines and columns are all the same due to the padding process.

5.5. Conversion to the *YcbCr* color model

To convert the image to the *YcbCr* color model, the function *rgb_to_y_cb_cr()* was created. In this function, the following operation is implemented:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$
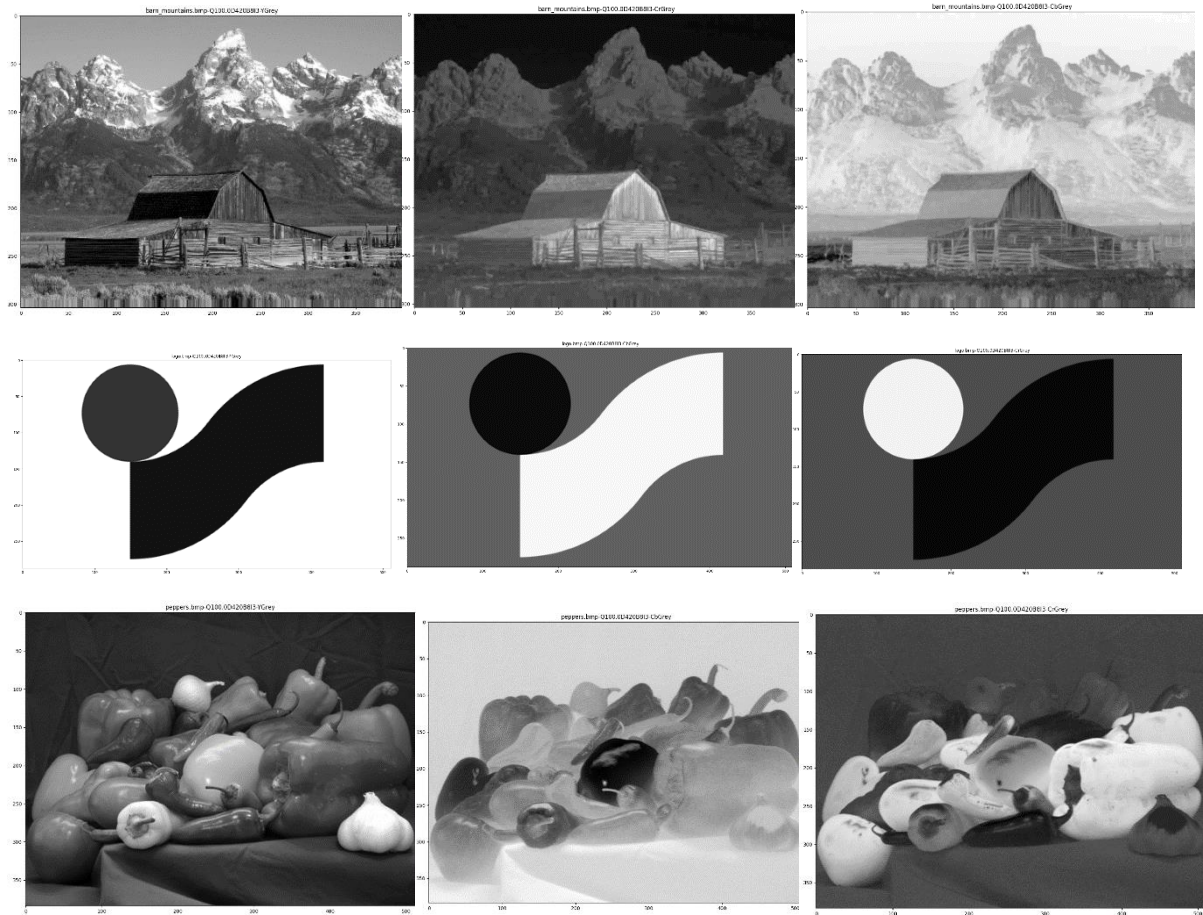
In the *RGB* images, the luminance is equally perceptible in each of the 3 channels, contrary to the *YcbCr* channels where the *Y* channel is the one who has the most perceptible luminance. On the other hand, the chrominance is most perceptible in the *Cb* and *Cr* channels compared to the others (*Y* and *RGB*).

To demonstrate the reduction in redundancy after converting the image to the *YcbCr* color model, the work group decided to perform a statistical analysis of each channel, for each image.
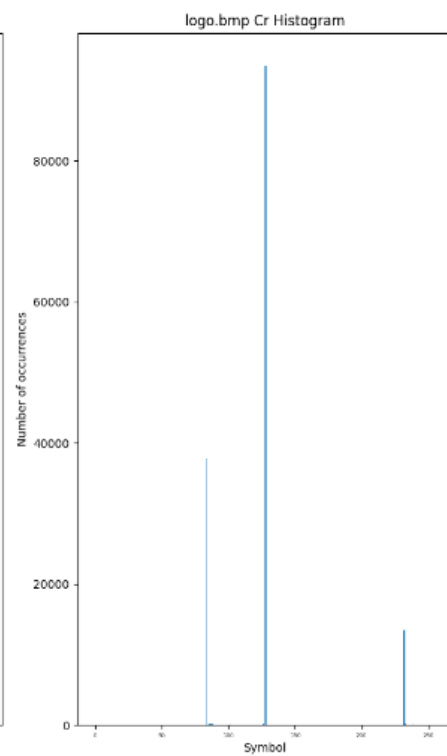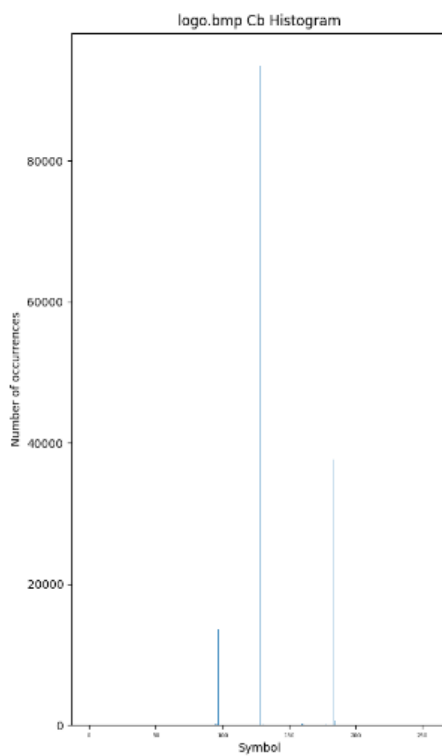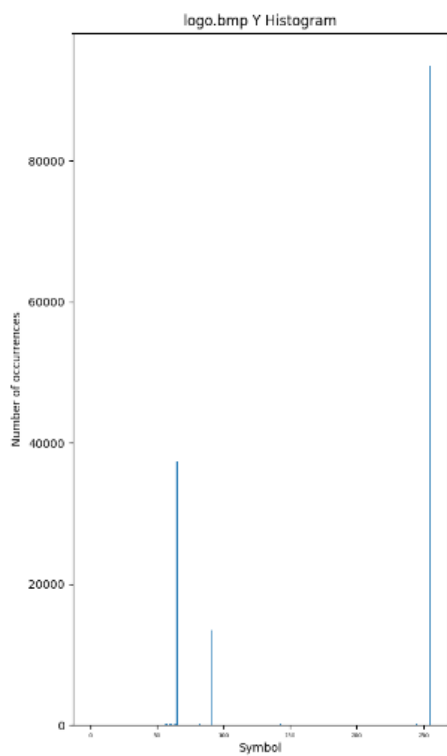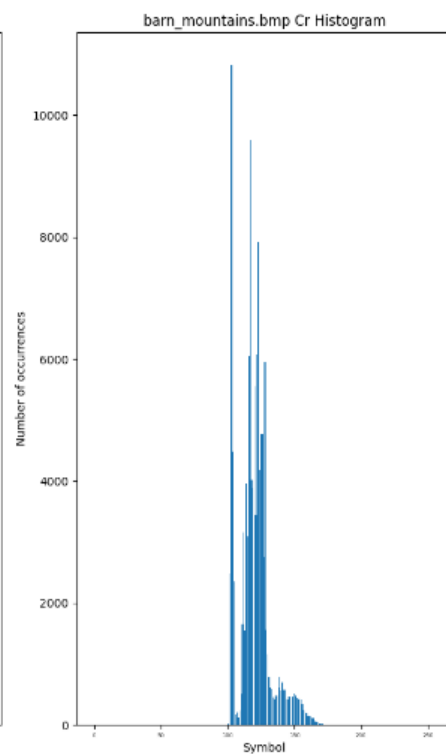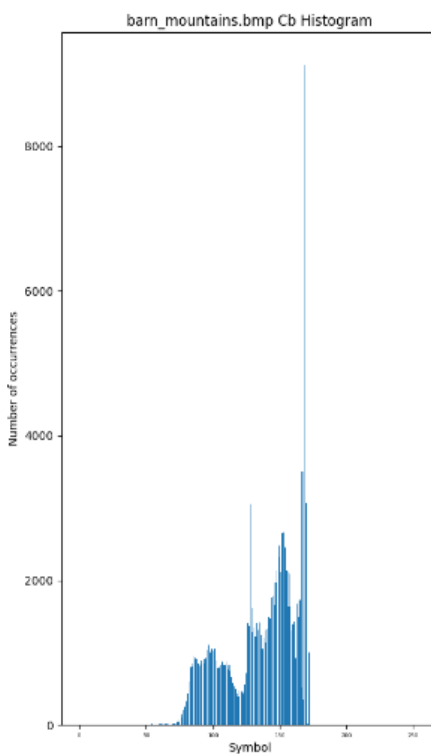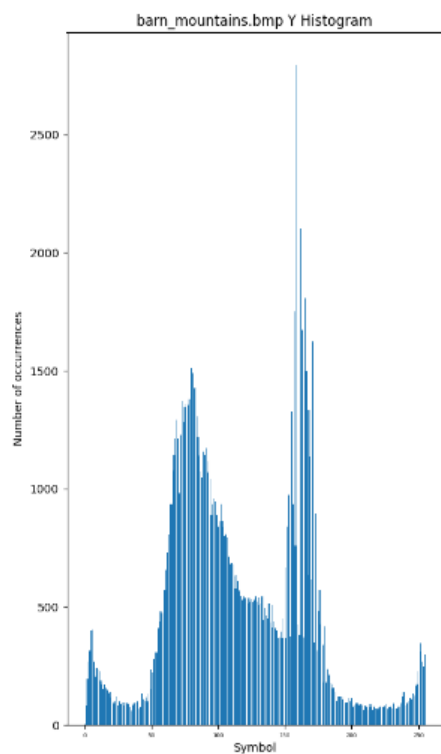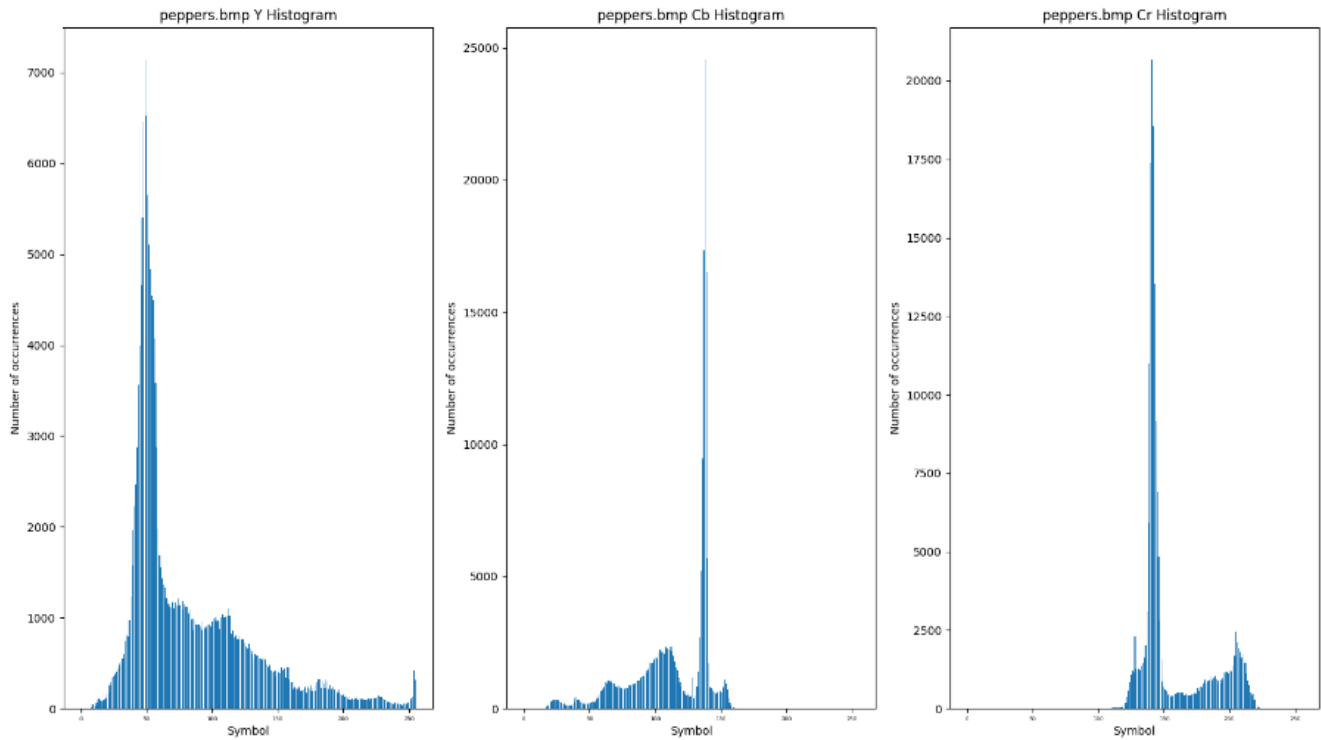
5.6. Down-sampling

To down sample the *Cb* and *Cr* channels, the *down_sample()* function was created. In this function, the channels *Cb* and *Cr* are down-sampled, having their number of columns and/or columns and lines reduced (or not), depending on the given variant (*4:4:4, 4:2:2, 4:2:0*, …). The group implemented this function in a way that the programmer can pick the manner the image is down-sampled: if its rows/columns are simply truncated or if the image is resized to the desired size, using the interpolation (*LINEAR, CUBIC, LINEAR*) primitives provided by the module *cv2*.To revert that process we resort to the function *up_sample()*. Again, two ways of doing this are provided: the image may be resized using simply *np.repeat()* or it can be interpolated using the cv2 module function *cv2.resize*.

It is expected that the *4:2:0* variant will have a higher compression ratio than the *4:2:2* because, in the first variant, the image has half of its lines and columns removed while the other keeps its lines and removes half the columns. Therefore, the *4:2:2* variant will output approximately 50% compression ratio and the *4:2:0* approximately 75% (Note: with the padding the compression ratio may be slightly smaller).
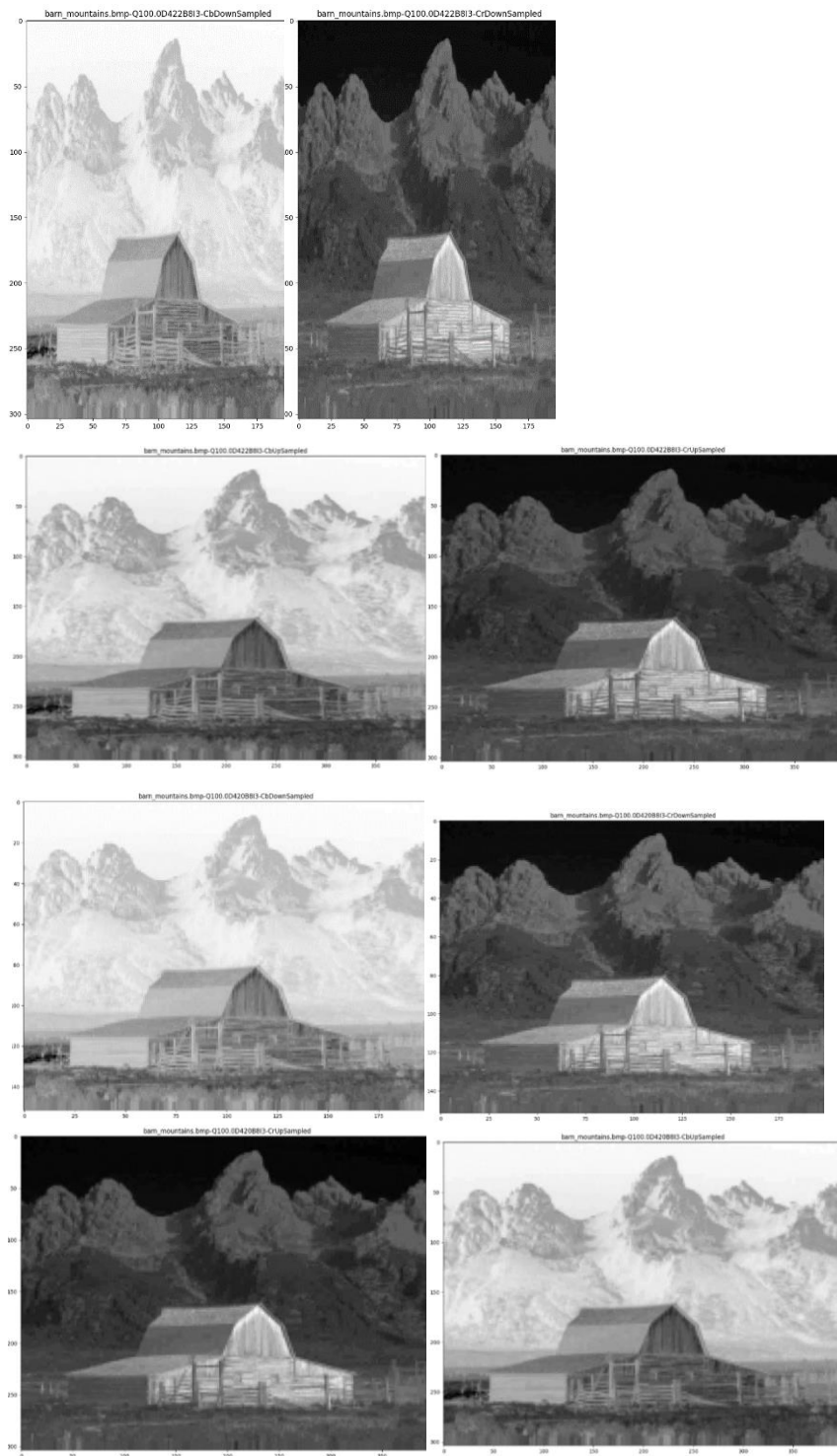
5.7. *Discrete Cosine Transform (DCT)*

The usage of the *DCT* starts the core of the *JPEG* compression stack.

Note: For simplicity purposes, only the image *barn_mountains.bmp* is taken into account in the images of this section.

5.7.1 Total *DCT/IDCT*

In order to calculate this transform and the correspondent inverse of a complete channel, the functions *dct()* and *idct()* imported from the *scipy.fftpac*k library were used. Since both functions are one-dimensional and each channel is a two-dimensional array, they were applied on top of each other in order to capture both dimensions.
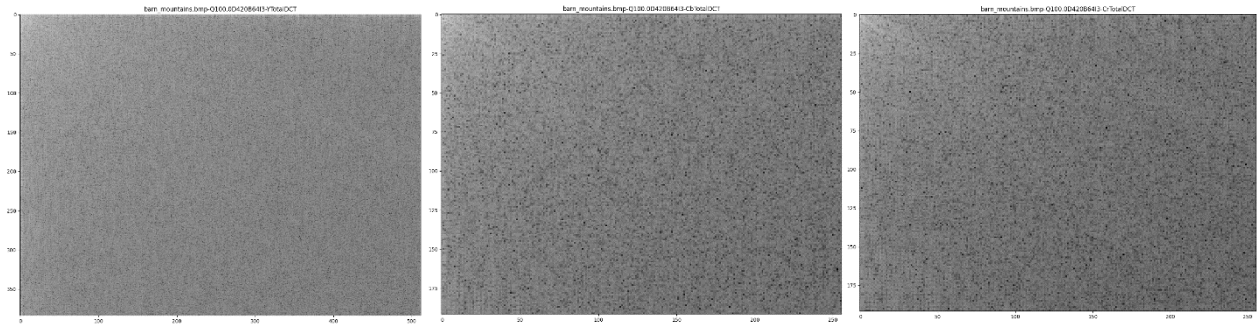
5.7.2 *DCT/IDCT* in NxN blocks

To calculate the *DCT/IDCT* in NxN blocks, the group implemented three possible solutions and benchmarked them. These implementations and their temporal efficiency study is shown below.

These three approaches were implemented using the following mechanisms:

- *Python* range loop

- *Numpy r_*

- *Numpy/Astropy*

Python *r*ange Loop

Using the range loop way (the intermediate effective one based in the above graphics), a for loop was used to go through the columns and another to loop trough the rows of the image. Within those two loops, the *DCT* is applied in a block of the specified size. The same goes for the *IDCT.*

Numpy r_ loop

Using the *numpy r_* iterable, a for loop is also used to go through the columns and another one to loop trough the rows. The r_ that is used to concatenate any number of array slices along

the axis and within that scope we apply the *DCT* with the selected block size. The process for the *IDCT* is similar.

Numpy/Astropy

Using the a*stropy* function *reshape_as_blocks*(), a mix of *Python* structures such as *tuples* and *numpy* functions like *reshape()* and *transpose()* are used in order to divide the image into NxN blocks. The *DCT* is then applied in dimensions 2 and 3. The process of the *IDCT* very different from before: the IDCT is first applied in dimensions 2 and 3 and then, the *numpy* function *concatenate()* is called twice, inside each other, in order to reach both dimensions and join all blocks.

IDCT in blocks 8x8: range loop vs r_ loop vs numpy/astropy
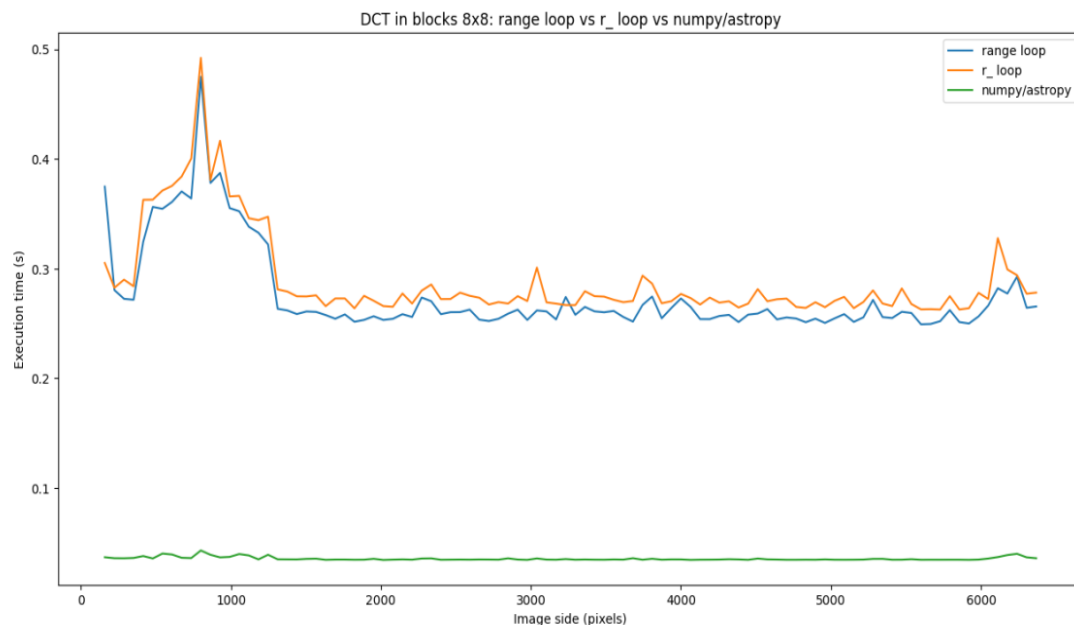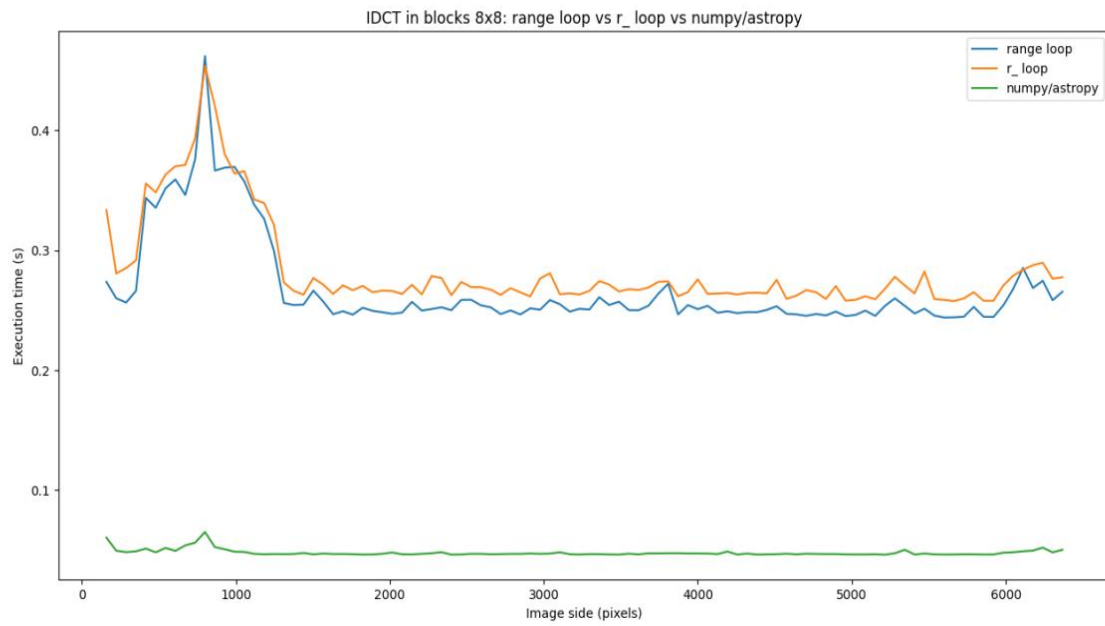
The above graphics represent the execution time (s) per image side length for each one of the approaches and for the *DCT* and *IDCT*. By observing them, it is possible to conclude that the *astropy/numpy* approach is the most efficient one and the one that uses *numpy r_* is the less efficient one. This was the most difficult step in *JPEG* pipeline the group developed in order to avoid the usage of *Python* loops, since they are very inefficient compared to the ones in other programming languages, such as *C, C++, Java*, etc.

DCT 8x8
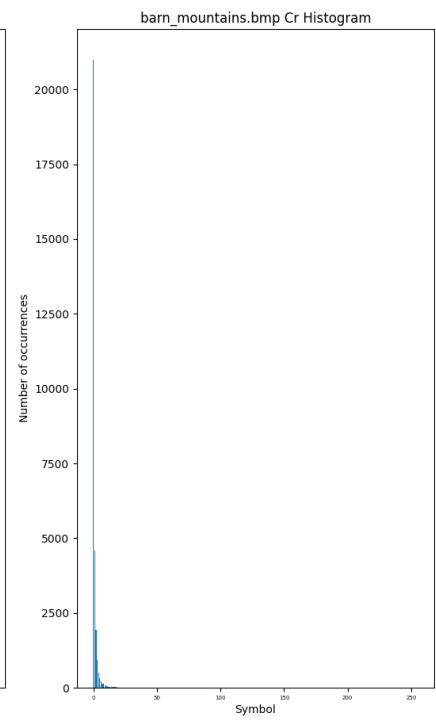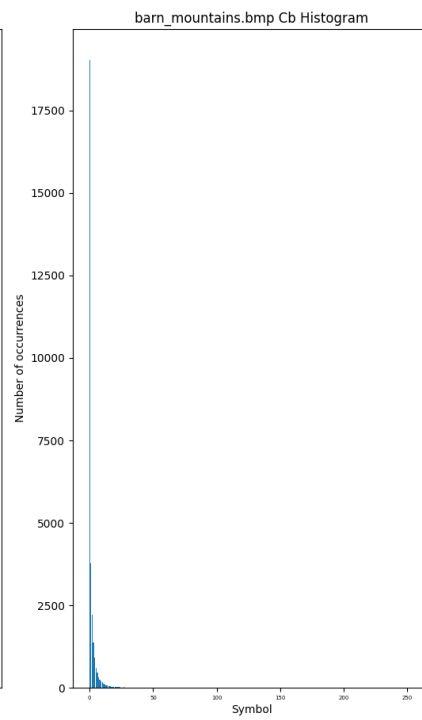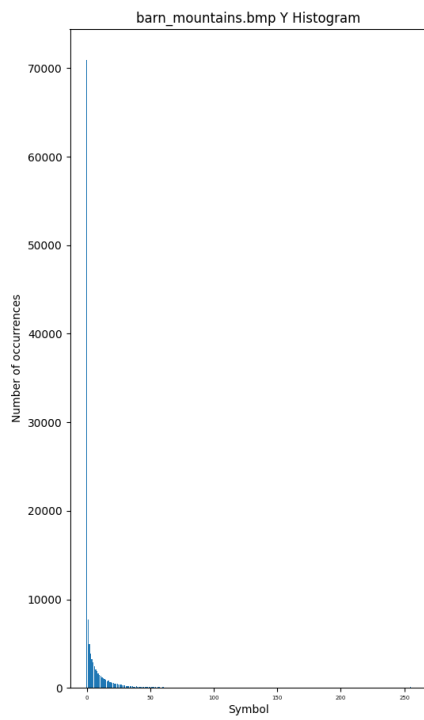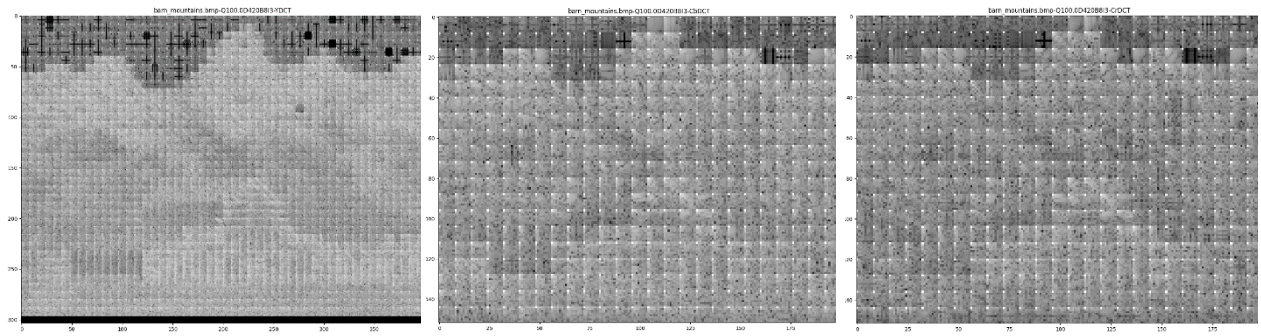
DCT 64x64
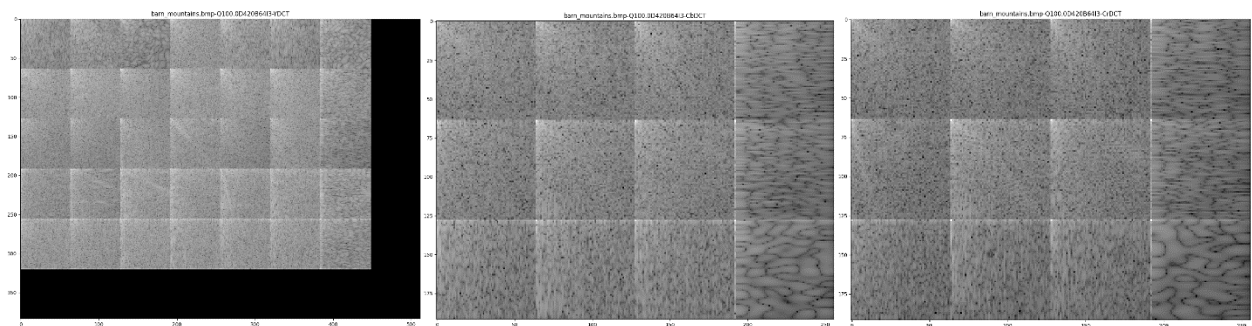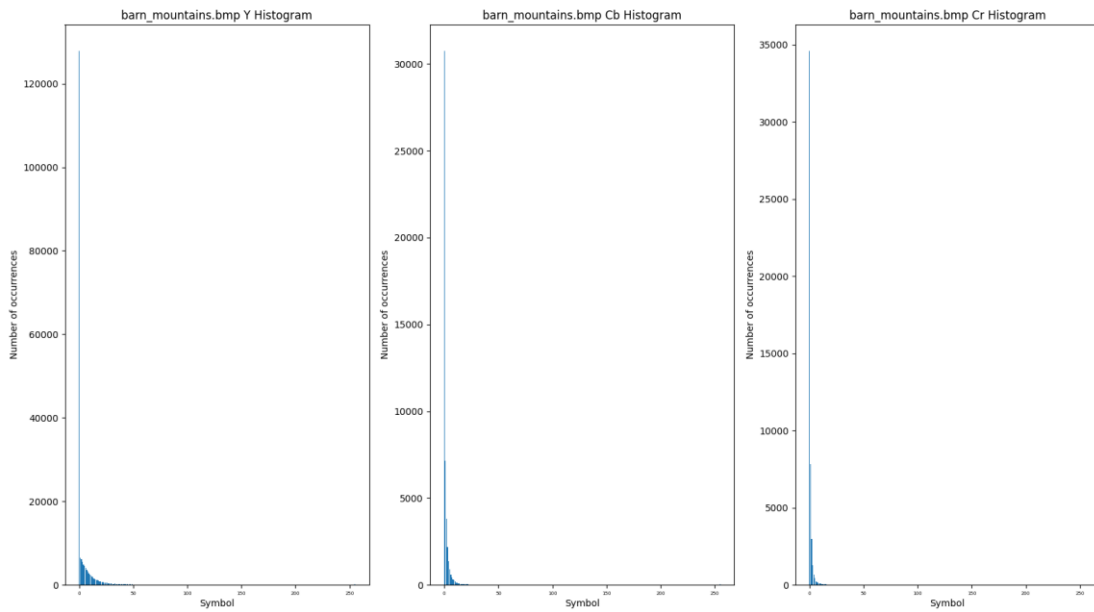
Image \ Channel Entropy (bits)	Y (8x8)	Y (64x64)	Cb (8x8)	Cb (64x64)	Cr (8x8)	Cr (64x64)
barn_mountains.bmp	3.11	2.70	2.25	2.07	1.73	1.58
logo.bmp	0.50	1.27	0.66	1.38	0.65	1.58
peppers.bmp	2.03	2.23	1.86	2.19	1.83	2.14

Comparing the images and entropy table related with the *DCT* in complete channels, 8x8 and 64x64 blocks, and using some *Multimedia a priori* knowledge, the group concluded that the best in general, in terms of compression potential, is 8x8, in almost every case, because of how small the difference between pixels. In other words, it means that, in such small area, the probability of occurring an abrupt change of color intensity is smaller than in a larger area, such as a 64x64 block or even the entire image. However, regarding the *barn_mountains.bmp* image it can be deducted that the DCT by 64x64 blocks is more proficient due to the lower entropy and less disperse histogram presented above.

5.8. Quantization

To quantize the *DCT* coefficients for each block the function *apply_quantization()* was created. It by calculating the scale factor (s_f), using the given quality factor (q_f) and the expressions below. Next, the scale factor is utilized to retrieve the scaled quantized matrix that will then divide the matrix correspondent to the 8x8 block in question. This operation is performed in the three channels.

For the inverse process (dequantization) the quantized block (G) is simply multiplied by the scaled quantization matrix (Q_s).

$$s_f = \begin{cases} \dfrac{100 - q_f}{50} & , q_f \geq 50 \\[2em] \dfrac{50}{q_f} & , q_f < 50 \end{cases} \qquad Q_s = \begin{cases} \text{round}(Q * s_f) & , s_f \neq 0 \\[1.5em] Q_1 & , s_f = 0 \end{cases}$$
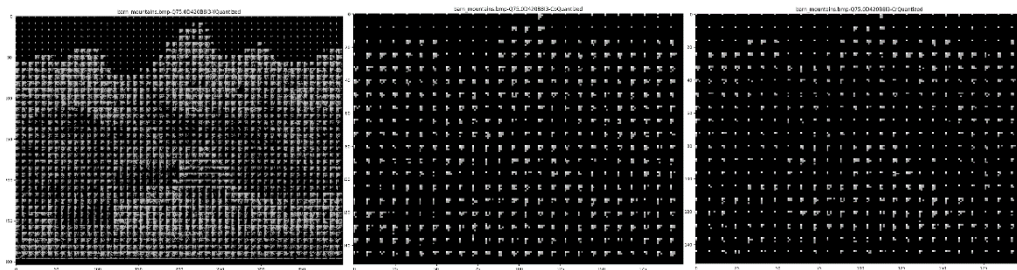
Image \ Channel Entropy (bits)	Y (8x8)	Cb (8x8)	Cr (8x8)
barn_mountains.bmp	3.11	2.25	1.73
logo.bmp	0.50	0.66	0.65
peppers.bmp	2.03	1.86	1.83

As concludable by the analysis of the previous table, after applying quantization, the entropy of all images lowered, meaning that each one got more compressible.

5.9. *DPCM* encoding of the *DC* coefficients

To *DPCM* encode the *DC* coefficients of every block it is necessary to replace the *DC* coefficient by the difference between it and the previous one, starting on the second block. For this purpose, the function *apply_dpcm_encoding()* was implemented. It takes an array of blocks and retrieves the first element of each block (*DC* coefficient = B(0, 0)). Then, using the *numpy* function *np,diff()*, the *delta encoding* is propagated to each block. The inverse function simply uses the *numpy* function *np.cumsum()*, which calculates the cumulative sum of the residuals, in order to reconstruct the original *DC* coefficients.
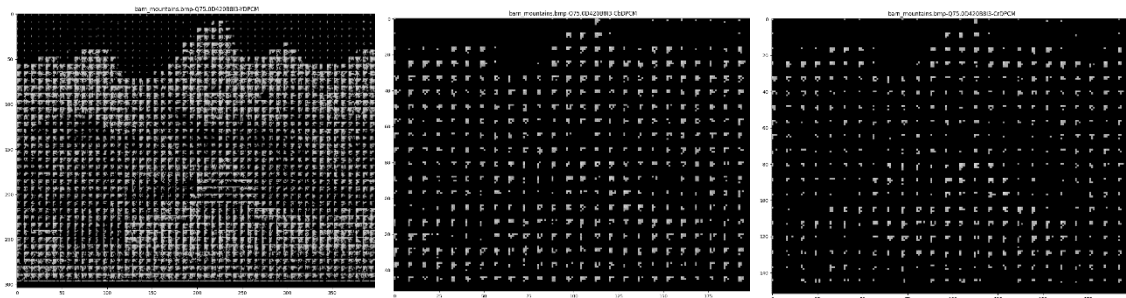
Image \ Channel Entropy (bits)	Y (8x8)	Cb (8x8)	Cr (8x8)
barn_mountains.bmp	3.08	2.20	1.69
logo.bmp	0.40	0.57	0.56
peppers.bmp	2.00	1.82	1.79

As concludable by the analysis of the previous table, after applying the *DPCM*, the entropy of all images lowered, meaning that each one got more compressible.

5.10. Encoding and end-to-end decoding

In order to quantitively evaluate the quality of the decompressed images, some functions that calculate distortion metrics such as the *MSE*, *RMSE*, *SNR* and *PSNR* were implemented. The following tables will demonstrate the difference between the distortion metrics values with the various quality factors and down-sampling variants.

Barn_mountains.bmp									
	4:2:0					**4:2:2**			
QF	10	25	50	75	100	10	25	50	75	100
MSE	743.96	426.20	286.85	177.25	28.97	710.31	401.49	264.13	155.23	12.92
RMSE	27.28	20.64	16.94	13.31	5.38	26.65	20.04	16.25	12.46	3.60
SNR	18.47	20.89	22.61	24.70	32.57	18.67	21.15	22.97	25.28	36.07
PSNR	19.42	21.83	23.55	25.64	33.51	19.62	22.09	23.91	26.22	37.02

logo.bmp									
	4:2:0				4:2:2				
QF	10	25	50	75	100	10	25	50	75	100
MSE	191.55	91.57	63.72	43.05	18.88	164.18	68.94	47.44	28.01	8.55
RMSE	13.84	9.57	7.98	6.56	4.34	12.81	8.30	6.89	5.29	2.92
SNR	28.51	31.71	33.29	34.99	38.57	29.18	32.95	34.57	36.86	42.01
PSNR	25.30	28.51	30.01	31.79	35.37	25.98	29.75	31.37	33.66	38.81

peppers.bmp									
	4:2:0					4:2:2				
QF	10	25	50	75	100	10	25	50	75	100
MSE	354.49	174.94	119.65	85.91	30.32	293.41	135.57	84.15	57.62	13.60
RMSE	18.83	13.23	10.94	9.27	5.51	17.13	11.64	9.17	7.59	3.69
SNR	19.43	22.50	24.15	25.58	30.11	20.25	23.60	25.68	27.32	33.59
PSNR	22.63	25.70	27.35	28.79	33.31	23.46	26.81	28.88	30.53	36.80

By analyzing the above tables, it is affirmable that the results are the ones expected: with the increase of the quality factor, *SNR* and *PSNR* are larger and *MSE* and RMSE are smaller. Taking into account the down-sampling, since the *4:2:0* variant causes more data loss than the *4:2:2* (because it removes half of the image rows/columns), it is expectable the increase of both *MSE* and *RMSE* and decrease of *SNR* and *PSNR* from one variant to another.

The uniform noise in the first and third pictures above (error between the original and reconstructed *Y* channel) is due to the fact both *barn_mountains.bmp* and *peppers.bmp* have color variations in all of their extension, while the image *logo.bmp* only presents variation at the border of the contained polygons. The referred uniform noise is a direct consequence of the fact that no quantization is applied (quality factor of 100) and the only sources of data destructiveness are the down-sampling and round operation before the application of the *IDCT*.

6. Conclusion

Upon finishing this practical work, the following familiarities were acquired:

- The scenarios in which the *JPEG* performs well or not.

- The artifacts that *JPEG* may cause when applied to an image.

- The perceptual knowledge and analysis behind the *JPEG codec*.

- The reason why the *YCbCr* colour model is used in *JPEG* instead of the *RGB*.

- The motivation to prefer the *DCT* instead of other discrete transforms such as the *DFT*.

- Trying various parameters in some of this modified *JPEG* steps, such as the down sample variant, interpolation type, block size can give us different compression rates and different times of compression and we can make use of that to reach the final state we want.

7. Bibliography

-https://stackoverflow.com/questions/30597869/what-does-np-r-do-numpy, acedido em 25/03/2022.

-https://en.wikipedia.org/wiki/Compression_artifact, acedido em 24/03/2022.

-https://www.dsprelated.com/showthread/comp.dsp/18074-1.php, acedido em 23/03/2022.

-https://www.intel.com/content/www/us/en/develop/documentation/ipp-dev-reference/top/volume-2-image-processing/image-color-conversion/image-downsampling.html, acedido em 23/03/2022.