UNIVERSITY OF PADUA - ICT FOR INTERNET AND MULTIMEDIA
SOURCE CODING COURSE

Project 4: Image coding with a JPEG-like approach

Aurora Zabot, 1206742

A.Y. 2018/2019

1 ABSRACT

The main aim of this project is to develop a JPEG-like data compression procedure. This report would be divided in the following way:

- In chapter 2: I would provide a panoramic view on the JPEG procedure, focusing on the algorithm key points and on the weaknesses/strengths of the procedure
- In chapter 3: I investigate deeply the main passages of the procedure in order to grasp all the elements necessary to the development of the code
- In chapter 4: I explain briefly the code, paying particular attention on turning points of the algorithm and on the issues I've faced during the development
- In chapter 5: I expose the results of my computation
- In chapter 6: I just take stock of the work done so far

2 INTRODUCTION: JPEG

The lossy-JPEG (*Joint Photographic Expert Group*) is a lossy compression technique widely used in digital photography and in video editing standards. One of its main strength points is the high compression ratio, normally set around 60-75%, optimal to the web usage (that's the main reason of its popularity); moreover, its format can be correctly displayed almost on every device (whereas the lossless version is not so flexible). Further more, taking into account how the human eye perceives colors, this kind of algorithm works principally on the luminance component, compressing efficiently smoothed and realistic images. Finally, the degree of compression can be chosen by the user, that could decide to give priority to compression or to quality factors.
On the other hand, this kind of implementation is not suitable for sharp contrasts of adjacent pixels (that causes noticeable artifacts) or to multiple times edits (each re-compression step degrades the quality). Moreover, this format doesn't support the transparency.

2.1 Brief overview

In this section I would briefly explain the JPEG procedure, focusing on the key points of the algorithm that would be deeply investigate in the following sections. As it's specified in the assignment, firstly I have to make a transformation from the *RGB* to the YUV/YC_bC_r color space, in order to focus on the *luminance* and *chrominance* components; this step will be useful in the quantization part of the development, since the Y component (the luminance one) canalizes by its own nature most of the energy. Then, I have to partition the image I'm working on into sub-blocks of 8x8 pixels and applying *Transform Coding* techniques on each batch, in particular the *Discrete Cosine Transform (DCT)*, that would be finely explored in the 3 section. Moreover, the frequencies will be quantized by a *Uniform Scalar Quantizer*, having different quantization steps for each coefficient; as it was mention previously, implying a YUV/YC_bC_r color space is particularly useful in the sense that we can quantize higher frequencies (related to the chrominance components) with a lower accuracy, focusing instead on the lower ones. Taking into account the shape of the matrix, the algorithm requires to scan blocks in a "zigzag" way, starting from the second element of the upper left corner (for reasons that would be explained later on in 3) and involving *Run Length Coding*; in this way, we would obtain:

- Four bits representing the number of zeros (*runlength*)
- Four bits representing the number of bits necessary to represent the following coefficient (*category* or *size*)

Now, we need to apply *Entropy Coding Techniques*, in particular *Huffman Coding* on the eight bits obtained by the previous computation, i.e. *(runlength, size)*; we need also to append further bits to exploit sign and exact magnitude (*range*). The decoding procedure of course can be obtained doing the process explained above in reverse. The total process can be visualize through figure (2.1).

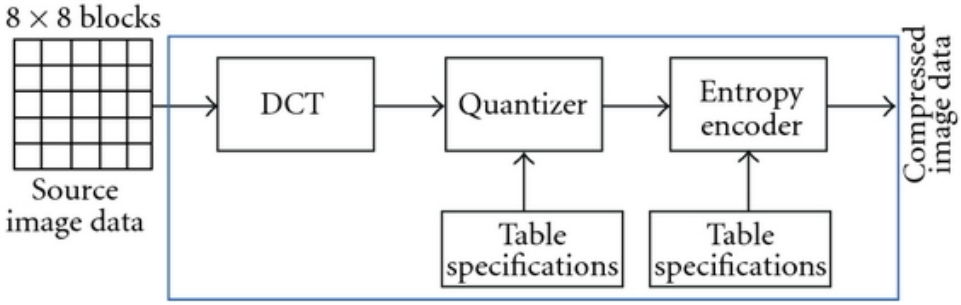


Figure 2.1: JPEG-baseline Encoder

3 THEORY

This section would provide a deep analysis of the main feature of the JPEG approach; in particular, I would investigate:

- Conversion from *RBG* to YUV color space
- Discrete Cosine Transform
- Quantization
- Coding procedure
- Decompression

3.1 Color space conversion

The first passage to take into account is the color space conversion from RGB to YUV/YC_bC_r. This color space is represented by three components: the luminance (Y) and the chrominances (C_b, C_r), split into blue and red components. Having a 8-bit representation, the conversion matrices can be expressed as follow:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.172 & -0.339 & 0.511 \\ 0.511 & -0.428 & 0.083 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.371 \\ 1 & -0.698 & -0.336 \\ 1 & 1.732 & 0 \end{bmatrix} \begin{bmatrix} Y \\ C_b - 128 \\ C_r - 128 \end{bmatrix} \tag{3.2}$$

This color space allows a additional compression without compromise too much the perceived quality of the image; this is due to the fact that the major portion of information is located just in one channel (the luminance one, associated to lower frequencies), and so we can downsample the chrominance components obtaining an efficient compression without losing too much information. In general, the ratios of downsampling in JPEG varying from 4:4:4 (no downsampling at all) to 4:2:2 (reduction only in the horizontal direction) and 4:2:0 (reduction on both directions). In my case, I would use a 4:4:4 ratio, without downsampling the two chromatic channels.
After this step, each channel needs to be partitioned in 8x8 sub-blocks (in the case of no downsampling). If the partition exceed the border of the image (i.e. the number of rows/columns is not a multiple of eight), the encoder pads the gaps with dummy data. The more used technique is to replicate the border pixels in order to avoid ringing artifacts, even thought that's not completely erase the effects in some cases. In my case, I've chosen just to add zeros.
The image also need to be "level shifted" by 2^{P-1}, i.e. subtracting 2^{P-1} from each pixel value, where P is the number of bits used to represent each pixel. Dealing with 8-bit images means that each entry falls into a range of $[0,255]$, and so there's the need to subtract the value 128 in order to obtain a range of $[-128,127]$ and centering in this way the levels.

3.2 DCT: Discrete Cosine Transform

Each 8x8 block would be converted in the frequency domain through a normalized two dimensional type *Discrete Cosine Transform*, a particular case of *Transform Coding*.
In general, the NxN cosine transform matrix **C** can be obtained as follow:

$$[C_{i,j}] = \begin{cases} \sqrt{(\frac{1}{N})} cos\frac{(2j+1)i\pi}{2N} & i = 0, j = 0, .., N-1 \\ \sqrt{(\frac{2}{N})} cos\frac{(2j+1)i\pi}{2N} & i = 1, ..N-1, j = 0, .., N-1 \end{cases} \tag{3.3}$$

The formula (3.3) can be rewritten in compact form, considering $N = 8$, in order to adapt the formula to my particular case:

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y} cos\frac{(2x+1)u\pi}{16} cos\frac{(2y+1)v\pi}{16} \tag{3.4}$$

Where: $\alpha(\cdot)$:= normalizing factor for orthonormality = $\begin{cases} \frac{1}{\sqrt{2}} & if\, u = 0 \\ 1 & elsewhere \end{cases}$

As it can be seen from figure (3.1), there's a huge change in the variation from the upper-left

corner to the bottom-right one. This “shape" of the coefficients would be crucial in the coding part. This point would be more clear after seeing the matrices computation in the chapter 5; those ones have the particular characteristic of concentrating the major portion of the signal in one corner, the upper-left one. The biggest component, in position $(0,0)$, is called *DC* component (constant) and it determines the hue of the entire block. The other elements of the matrix are called *AC* (or alternating) components. The quantization would take advantage of this shape, enhancing this effect while decreasing the overall coefficient sizes.

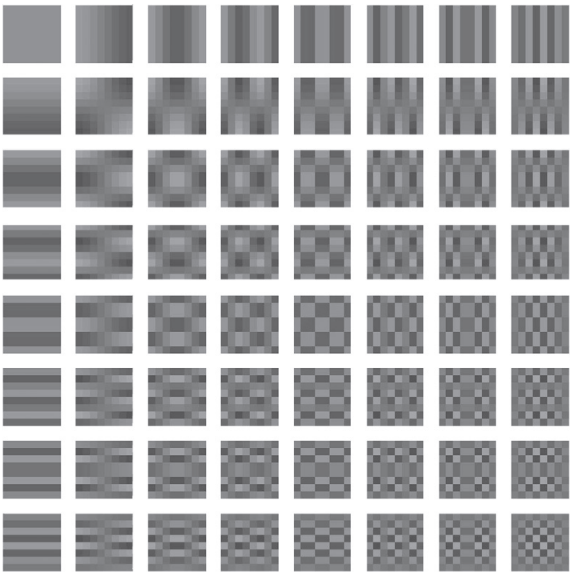


Figure 3.1: DCT basis

3.3 Scalar Quantization

Due to the fact that human eye perceives in a better way lower frequencies than the higher ones, as it was already underlined, the quantization can be adapted in order to get an higher reduction on the bottom-right portion of the matrices involved in the process. To do so, I would use a midtread quantizer. The quantizer step sizes are organized into some provided tables called *quantization tables* (see (3.6) respectively for chrominances and the luminance tables); then, the obtained value must be rounded as follow:

$$l_{i,j} = \left\lfloor \frac{\theta_{i,j}}{Q_{i,j}} + 0.5 \right\rfloor \tag{3.5}$$

Where $\theta_{i,j}$ represents the coefficient in the matrix in the position (i,j), $Q_{i,j}$ is the element in the same position in the *quantization tables* and finally $l_{i,j}$ is the quantized value. An alternative form consists just in round the $\frac{\theta_{i,j}}{Q_{i,j}}$ coefficient. From here, the reconstructed value is obtained by multiplying the quantized value to the correspective quantization element, i.e. $l_{i,j}Q_{i,j}$. The

quantization error can be computed as $\theta_{i,j} - l_{i,j}Q_{i,j}$.

$$\begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix} \quad \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 36 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \tag{3.6}$$

Note that, since we're using a midtread quantizers, we're also thresholding the matrices, since all the elements with the magnitude less than an half of the quantization step are setted to zero. From this, using a "zigzag" scan of the coefficients as the algorithm does in the coding step, we'll see that the probability of finding a long run of zeros increases step by step.
Finally, consider that choosing a more aggressive quantization matrix in this step would surely increase the compression, but it will also bring artifacts on the image. If this undesired effect is too heavy, the quality of the image would be highly compromised.

3.4 Coding procedure

The JPEG algorithm establishes a different coding procedure for the DC and the AC components.
Since the value of the DC component is related to the average value of the block, and given that each block would have almost the same mean, it makes sense not to encode the value itself, but the difference between neighboring labels. In order not to handle huge amount of values, each difference would be taken into a category, whose size would be a power of 2. Then, each category would be coded with an *Entropy Coding* technique, in our case *Huffman* (even thought it seems that *Arithmetic Coding* would improve the compression around a 5-7%). Since categories may contain more than one value, we need to add some bits to specify which element we're looking at.
The AC part is coded in a slightly different way. From the position (0,1), we start a "zigzag" scan with a *Run Length Coding* procedure that continue until a special value called *End Of Block*, EOB; from this point on, all the values would be zeros, so it's the last significant term in the zigzag scan. As we've already seen, we need four bits representing the *runlength* portion and another four to the *size* part; the portion (*runlength, size*) would be Huffman encoded as well. Also in this case we have to append some additional bits in order to identify the *sign* and the *magnitude*; those two are related to the belonging category as in the DC case.
Moreover, there are two additional special value to take into account: the above mentioned EOB (i.e. $(0,0)$) and the ZRL, that identify a sequence of more than 15 consecutive zeros (i.e. $(F,0)$).

Category Number	**Category**	**Category Size**	**Additional bits**
0	0	2^0	0
1	-1, 1	2^1	1
2	-3, -2, 2, 3	2^2	2
3	-7, .., -4, 4, .., 7	2^3	3
...	...	...	...
F	-32767, .., -16384, 16384, .., 32767	2^{15}	15

3.5 Decompression

This step fundamentally consists in reverse all the previous processes. So, the value obtained for each $l_{i,j}Q_{i,j}$ multiplication is used to construct the correspondent matrix. Then the DCT transformation mu be reversed in order to obtain the pixel coordinates:

$$f_{u,v} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7}\alpha(u)\alpha(v)F_{u,v}cos\frac{(2x+1)u\pi}{16}cos\frac{(2y+1)v\pi}{16} \tag{3.7}$$

Where: $\alpha(\cdot)$:= normalizing factor for orthonormality = $\begin{cases}\frac{1}{\sqrt{2}} & if\, u=0\\ 1 & elsewhere\end{cases}$

Inputs must be rounded as before and the remaining 128 values must be added.

4 MATLAB IMPLEMENTATION

Since I've implemented the procedure in two different ways, I'll explain each code separately (for the coding and decoding part), while the common part (the JPEG "structure") is exhibited first.

4.1 Common part

Function	**Description**
Main	In the main I just call both methods and compare them; I display the final result and the compression ratio
simpleJPEGcomp and JPEGcomp	**Input:** [image, compression factor]; **output:** [compressed image, compression ratio]; at the very beginning of the code I checked some parameters: the validity of the given inputs, the dimension of the input image (in particular, the total number of rows and columns must be a multiple of 8. If it's not the case, I zero-pad the image) and the number of channels of the image (since in the assignment was ask to develop a procedure for color images; in any case, the code can be easily adapt skipping the color conversion in case the size of the image has just two parameters). Then, I've splitted the three channels into the YCbCr components through the function *rgb2ycbcr*. After this, I've just focused on the 8x8 blocks using the combination *blockproc* and *block_struct*. I've applied the DCT using *dct2* on each block, then I've quantized them rounding the obtained DCT matrix divided for the quantization tables of chrominance (Cb, Cr) and luminance (Y). Finally, I've encoded the blocks. The process has been easily inverted computing the decoding procedure, multiplying each block for the aforementioned quantization tables and computing *idct2*. Eventually, I've reunited the three components, recomputed the color conversion through *ycbcr2rgb* and cut off the padded portion if the image was resized at the beginning

4.2 Huffman implementation

In this implementation I've just made a Huffman encoding procedure on each 8x8 block, without *runlength* coding and zigzag scan. The rest of the code just follow the description I've made above.

4.2.1 Algorithm

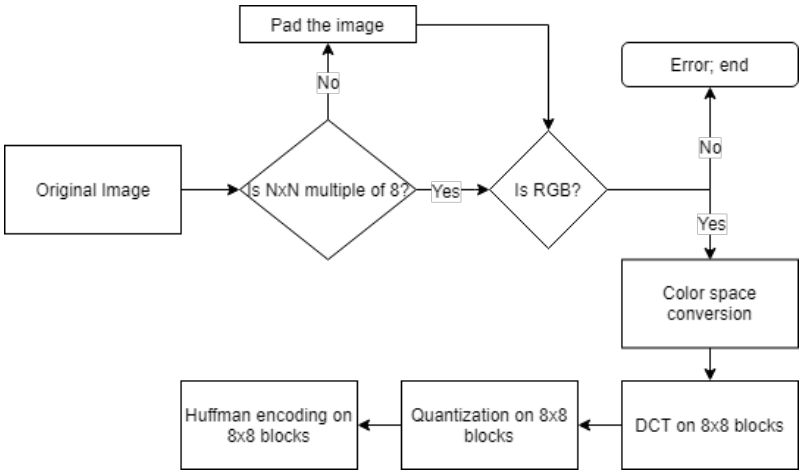


Figure 4.1: Coding Scheme

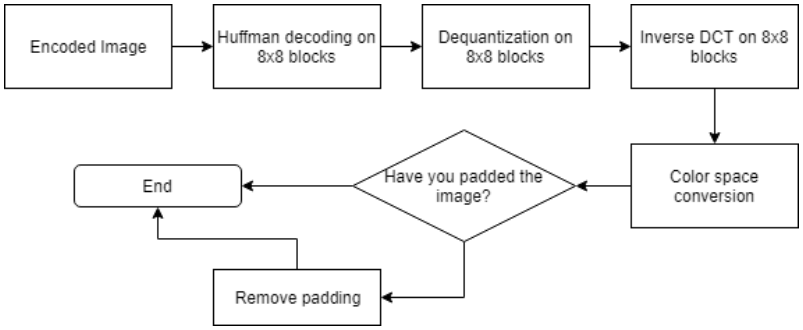


Figure 4.2: Decoding Scheme

4.2.2 Functions

Function	**Description**
simpleJPEGcomp	this is the main function for the computation of the JPEG procedure; here I call the various methods, following the algorithm 4.1 and 4.2
simpleHuffman	**Input:** [matrix block]; **output:** [code, dictionary, bitstream]; here I construct the dictionary for the encoding and decoding procedure, and then encode each 8x8 block. In order to obtain the symbol frequency to construct the dictionary, I've used the method *hist*, that returns the number associated of each bin given in input in a certain vector; in my specific case, I've obtained the symbols of the input matrix through *unique*, that cancels repetitions, and I've used it as *bin* in hist. After this, I've calculated the probability associated to each symbol as the common probability formula (p(a) = a / sum) and then I've used the *huffmandict* and *huffmanenco* methods offered by *Matlab*. The only issue I've found was when I've just one bin (normally a zero matrix); in this case I've just use a dummy element to differentiate the encoding process

simpleDecHuffman	**Input:** [encoded stream, dictionary]; **output:** [decoded matrix]; here I just decoded the matrix having the code and the associated dictionary. If the dummy matrix is received instead, I just give back the same input matrix without any probabilistic construction
compressionRatio	**Input:** [original ratio, compressed ratio]; **output:** [ratio]; this is just an auxiliary function to compute the final compression ratio

4.3 Runlength Huffman implementation

In this implementation, I've tried to follow the JPEG approach in the encoding and decoding part; I've just applied Huffman on *runlength* coding, after a zigzag procedure and add some additional bits in order to identify the *range*. In the decoding procedure, I've just inverted the process. I've used Huffman Standard Tables for JPEG in the coding part. The only thing that differs from the JPEG approach is that I didn't make any difference from AC and DC components.

4.3.1 Algorithm

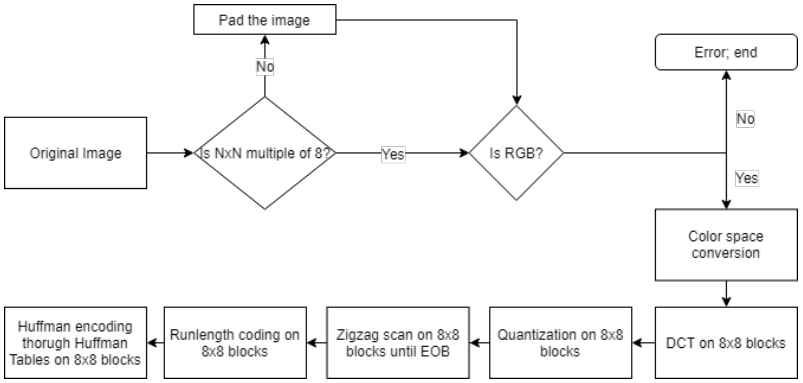


Figure 4.3: Coding Scheme

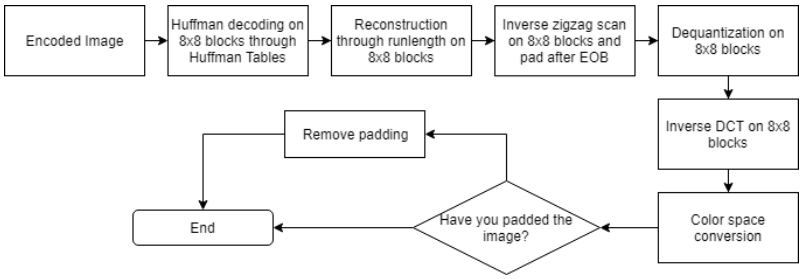


Figure 4.4: Decoding Scheme

4.3.2 Functions

Function	Description
JPEGcomp	as before, this is the main function that compute the procedure; the implementation follows the scheme proposed in 4.3 and 4.4
zigzag	**Input:** [matrix block]; **output:** [zigzag stream]; here I develop the zigzag scan procedure on the 8x8 blocks; in order to do so, I've considered what happened when the cursor is in the first/last row/column in a odd or even position, or inside the matrix in a odd/even position. On example, in the upper triangle of the matrix, if I'm in the first row (odd position) in a odd column position (so, in a even position considering the sum), I need to shift to the right of one step, and so on. I also found the EOB in order to code it with a key value (1010), cropping out all the remaining zero elements
encode	**Input:** [zigzag vector]; **output:** [number of zeros, category, range]; this part compute the *runlength* procedure and output the elements that would be necessary to the computation of the Huffman code on the 8x8 blocks; it has been developed as a *while cycle* in which we just skim the zigzag vector and categorize number of zeros in a run and the associated *range* and *category* for non-zero elements
findCategory	**Input:** [value of the matrix]; **output:** [category]; this is an auxiliary function needed to compute the *category* or *size* proportional to the value; I just created a matching table to do so, stopping at the 10-th *category*
huffman	**Input:** [number of zeros, category, range]; **output:** [code, bitstream]; in this part I've used the JPEG Huffman tables to encode the *runlength*; I also added the additional bits to find the *range* of our value; everything would be made on the 8x8 blocks
ihuffman	**Input:** [code]; **output:** [number of zeros, category, range]; here I just decode the encoded symbols on the 8x8 blocks, matching the provided Huffman code to the associated *runlength, category* and *range*
decode	**Input:** [number of zeros, category, range]; **output:** [decoded stream]; I call this function inside the **ihuffman** run: here, I just reunite the code (still in zigzag) using the information provided by the input values; to compute the dimension of the output vector, I've evaluated the dimension of the *range* vector, adding the number of consecutive zeros (so when the *runlength* vector was different from 0)
invzigzag	**Input:** [decoded zigzag vector]; **output:** [decoded inverse zigzag matrix]; here I compute the inverse zigzag scan; of course the procedure is slightly similar to the zigzag one. In this first part I pad the array with zeros (considering that in the previous part we've cut this portion with the EOB); having an 8x8 matrix, I just pad the vector with (8x8 - the current dimension of the reconstructed vector).
compressionRatio	**Input:** [original ratio, compressed ratio]; **output:** [ratio]; this is just an auxiliary function to compute the final compression ratio

4.4 Main issues

During the development of the code I've faced many problems, especially related to the management of the matrices (in particular when they were empty or with just one element) and with the scan procedures.

In the first implementation the main problem was related to empty matrices, because when you have just one element, the function *hist*, that I've used, fails. In order to manage this situation, I've just consider that if the *symbol* array had a size equal to [1, 1], then I just report the same matrix in output, since the symbol probability is 100%.
The second implementation was, of course, hardly to develop. The first problem that I've faced was related to the zigzag implementation scheme, but the only way to solve it was to consider all the possible scenarios during the run of the code. The second problem concerned the categorization of the elements for the *runlength*, i.e. [R S NZ] referring to the code. As the zigzag, the only way to compute this part was to try many matrices and look what happened on each computation. The last problem was related to the management of Huffman tables, in particular on the categorization of the *range* element.
In any case, the main issues that I've found during the development were linked to the fact that I'm not so confident using *Matlab*.

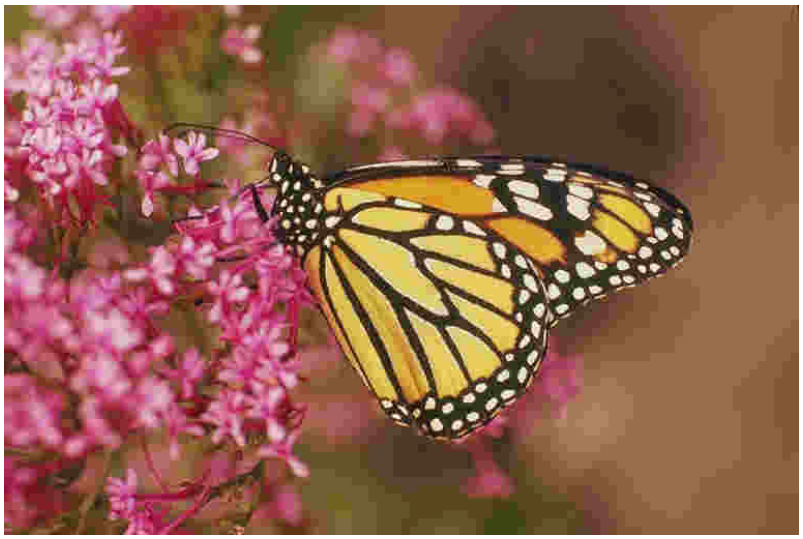
5 Results

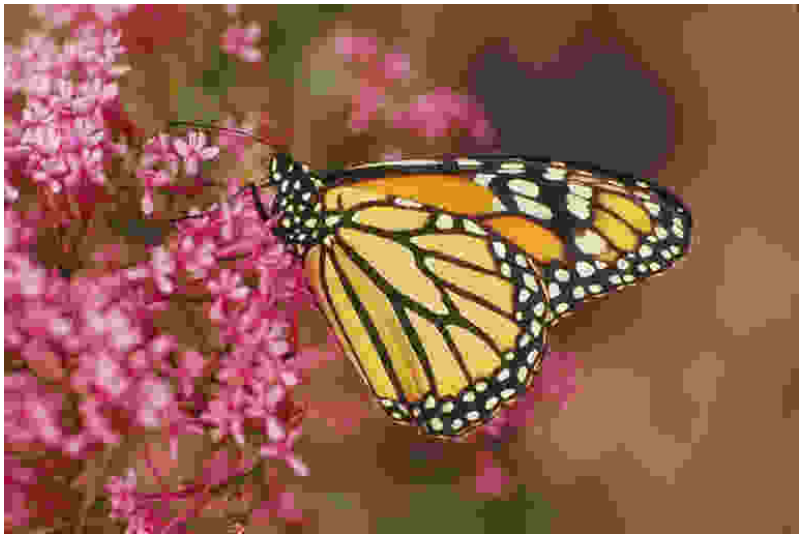
5.1 Data, results and parameters

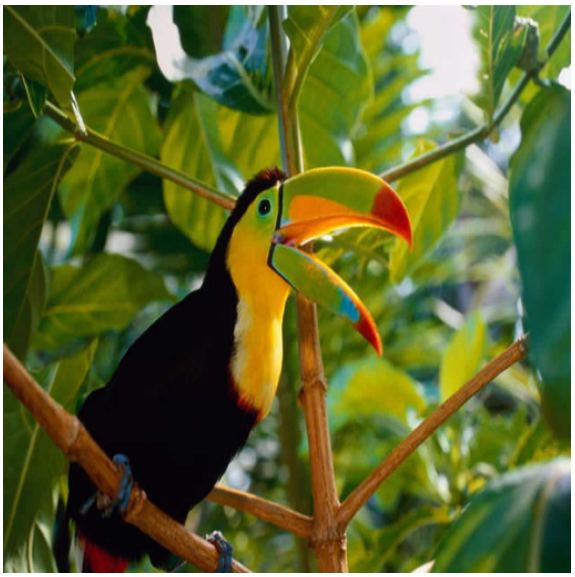
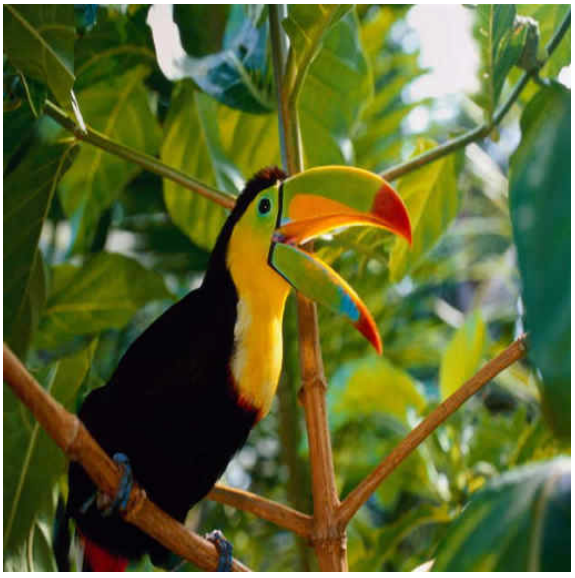
Here I just put the results of my computation; as I did before, I differentiate from the two different implementations

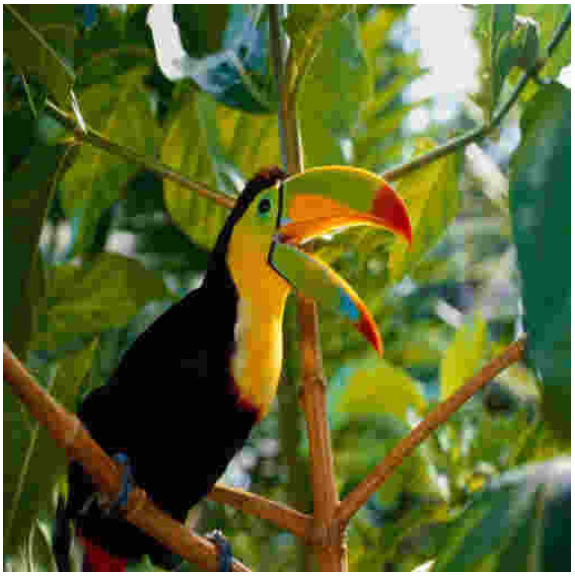
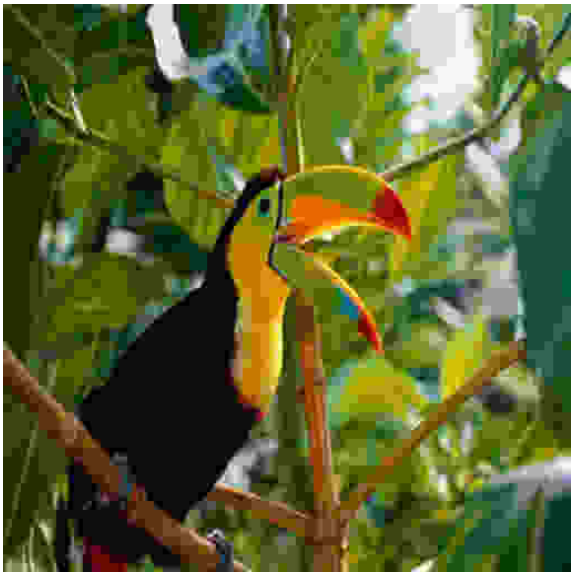
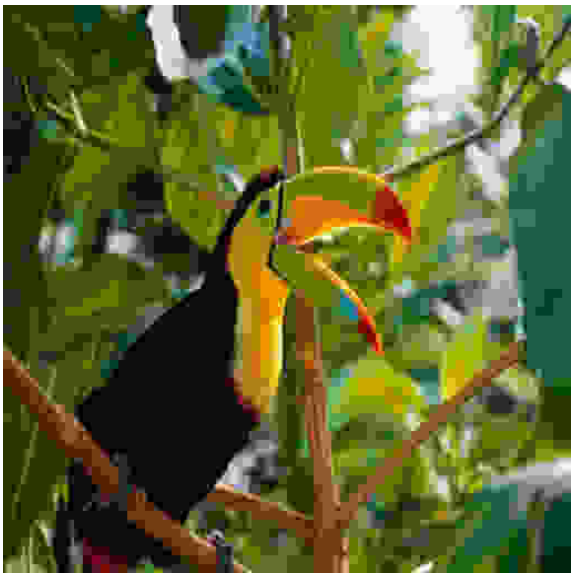
5.2 Huffman implementation

monarch image (png), 512x768x3	**Input**	**Ratio**
	-	-

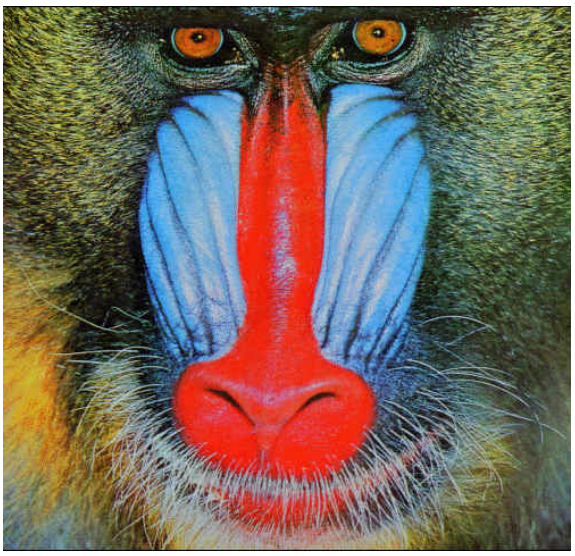
	90	7.2971
	60	7.544
	20	8.3032

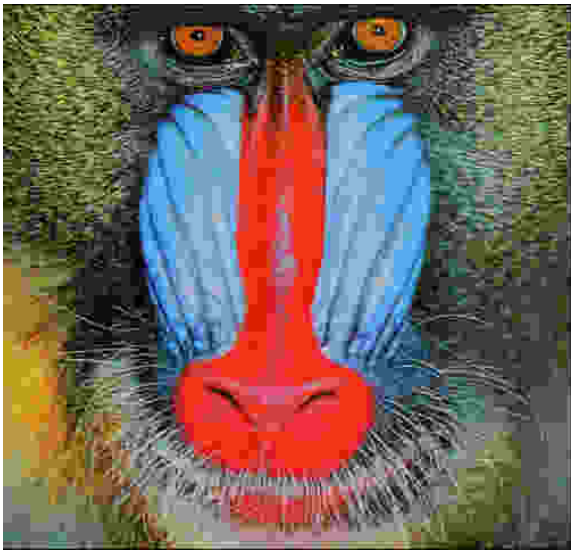
	10	9.6127
	8	10.5240
	5	15.4757

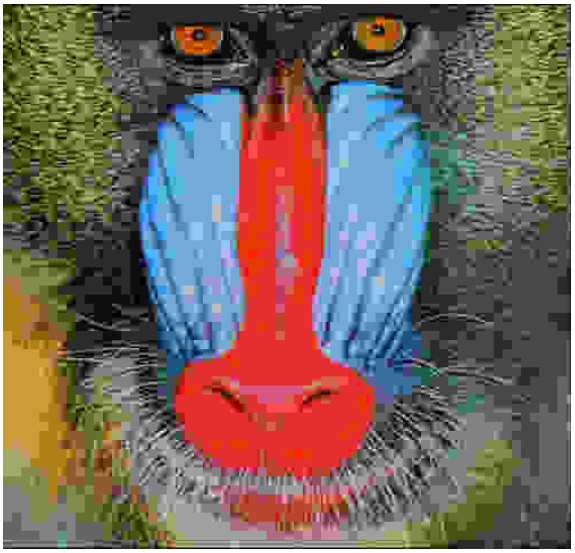
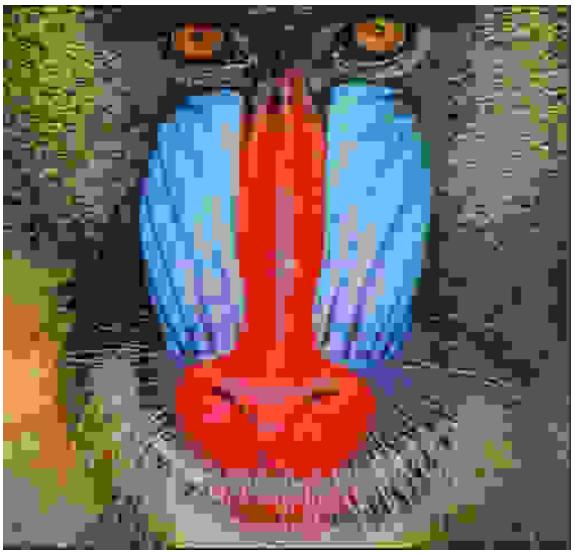
tucan image (bmp), 512x512x3	Input	Ratio
	-	-
	90	7.5268
	60	7.9413

	20	9.9061
	10	11.9850
	8	13.1295

	5	16.1141

baboon image (bmp), 480x500x3	**Input**	**Ratio**
	-	-
	90	6.5782

	60	7.2036
	20	9.4680
	10	11.8915

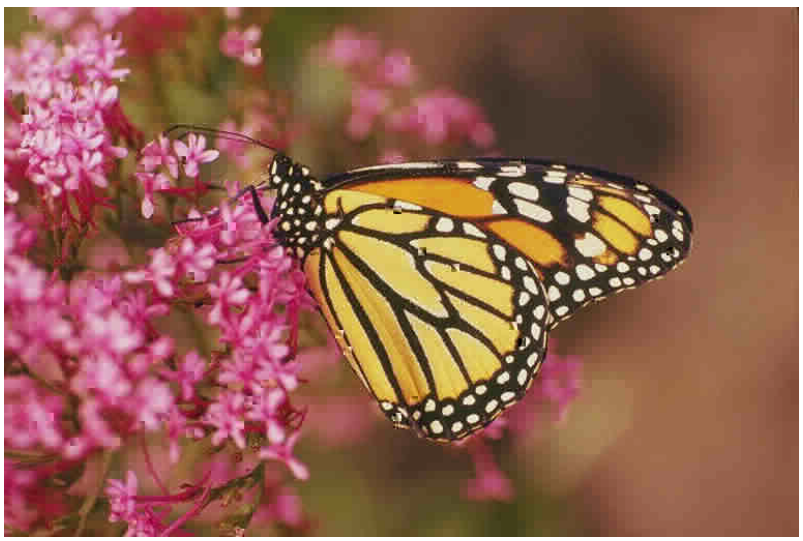
	8	12.9468
	5	17.0108

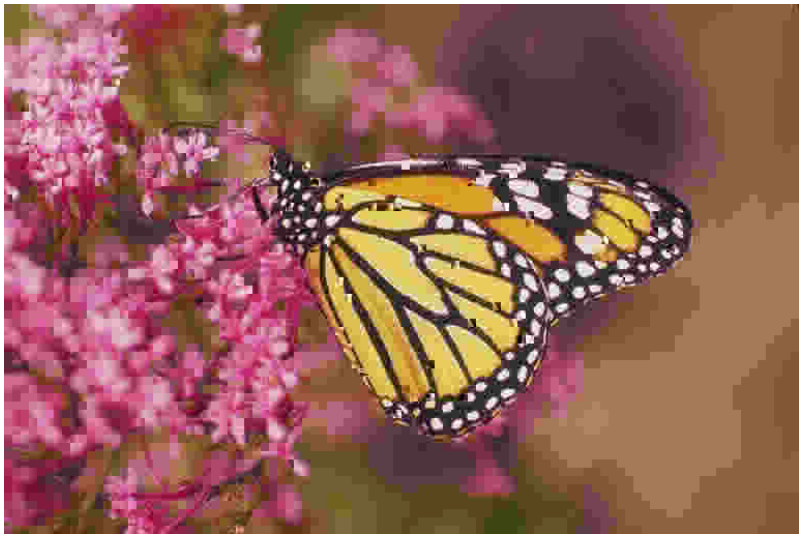
5.2.1 Comments

As we can clearly see, the images show more and more artifacts decreasing the input; at this is associated, as it is expected, an higher compression rate. In this implementation, the compression rate is not so high also at low inputs (i.e. 10, 8 or 5) and the quality of image start being really corrupted just around an input parameter of 20. For sake of clarity, I'll putting some results in a review table below, making also a comparison with the same quality compression given in input to a JPEG compression algorithm.

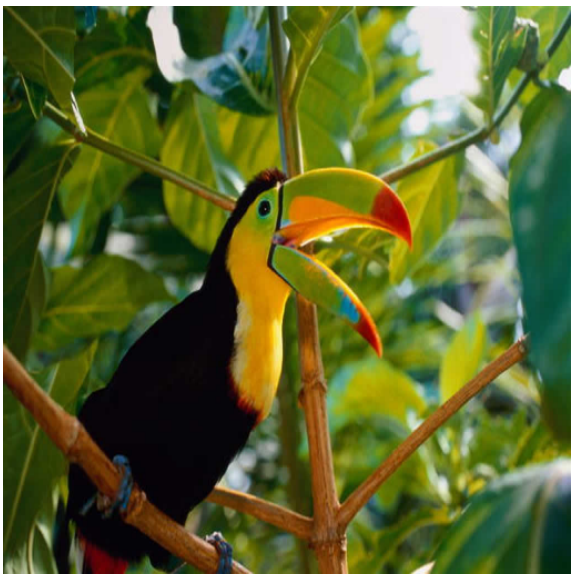
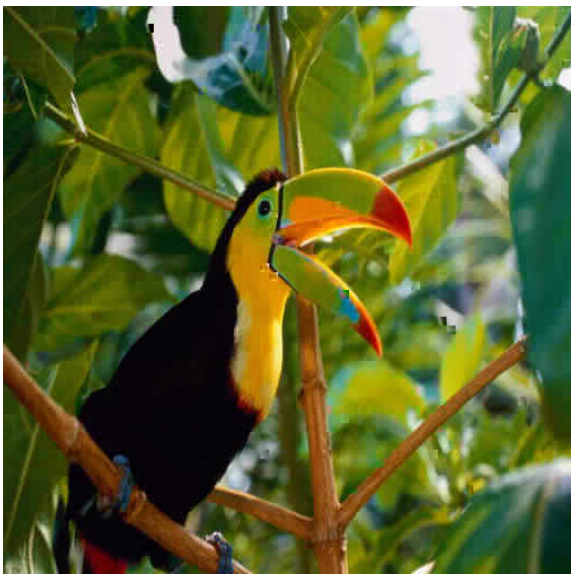
Image	**Input: 20**	**Input: 10**	**Input: 5**	**JPEG: 20**	**JPEG: 10**	**JPEG: 5**
monarch	8.3032	9.6127	15.4757	26.9377	38.9189	53.9985
tucan	9.9061	11.9850	16.1141	51.8107	76.0404	107.2091
baboon	9.4680	11.8915	17.9198	51.8107	76.0404	107.2091

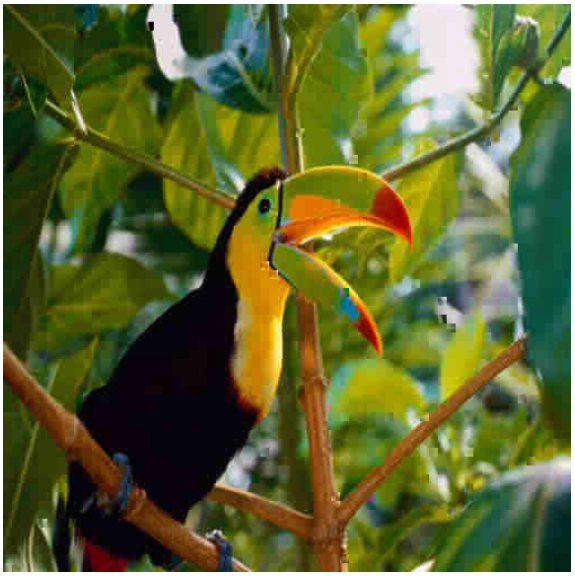
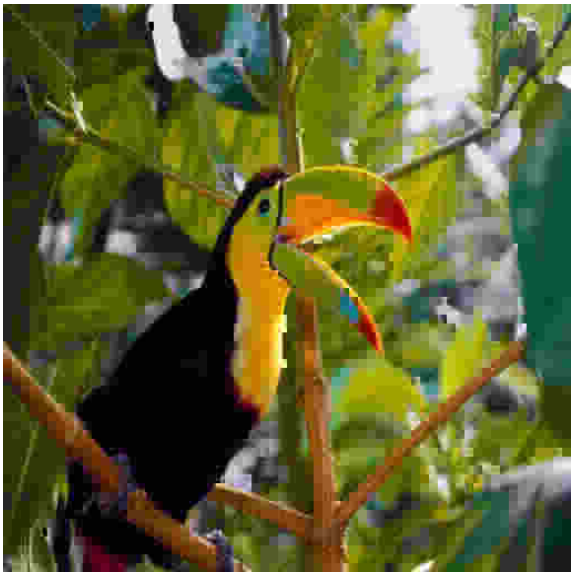
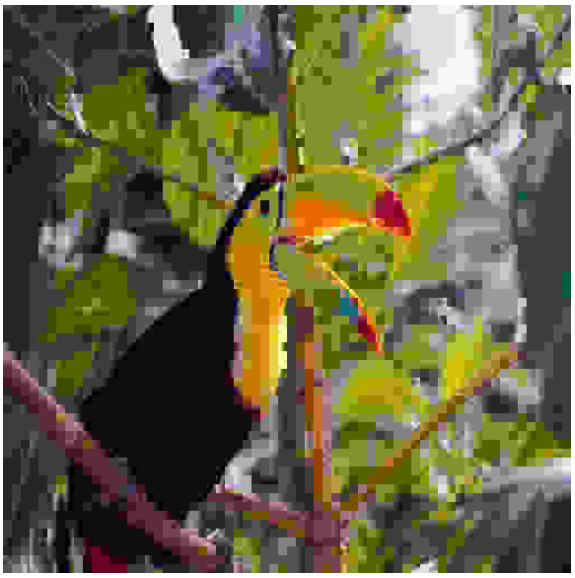
5.3 Runlength Huffman implementation

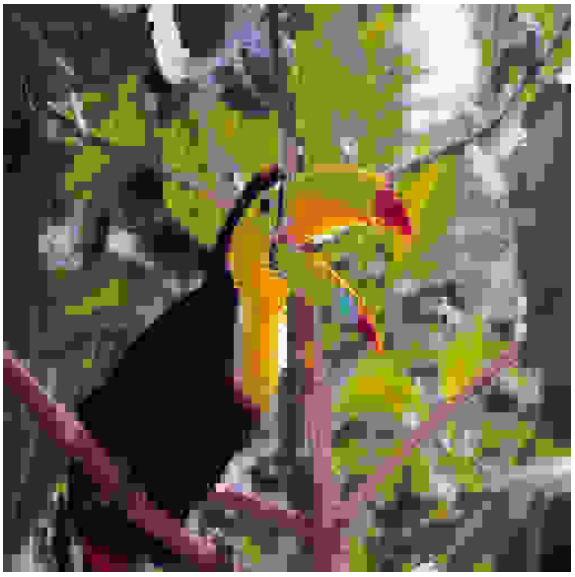
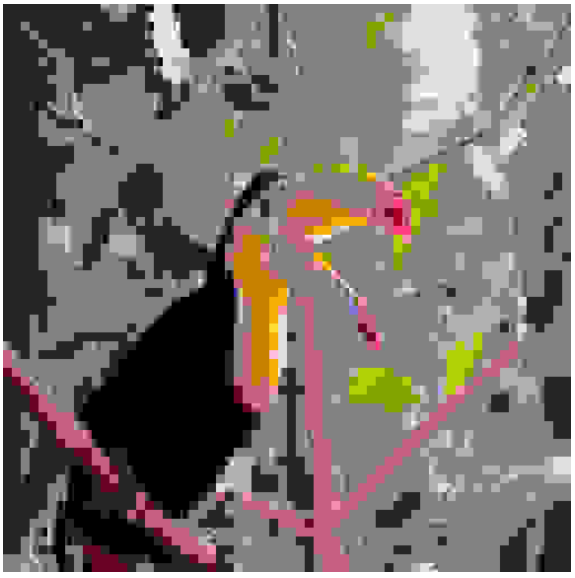
monarch image (png), 512x768x3	Input	Ratio
	-	-
	90	22.2845
	60	26.7286

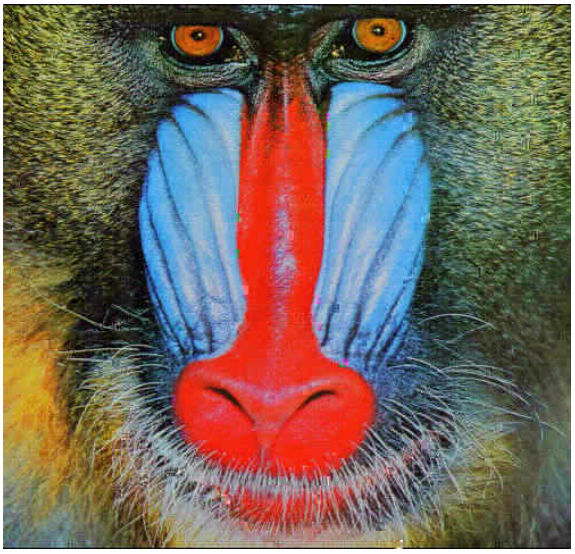
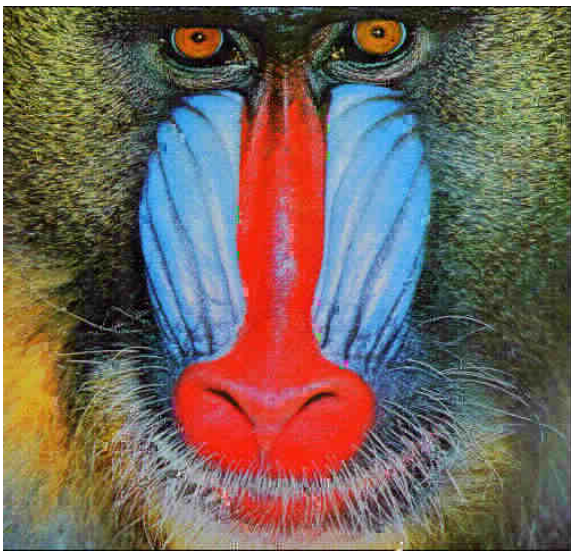
	20	41.4484
	10	54.4734
	8	59.9182

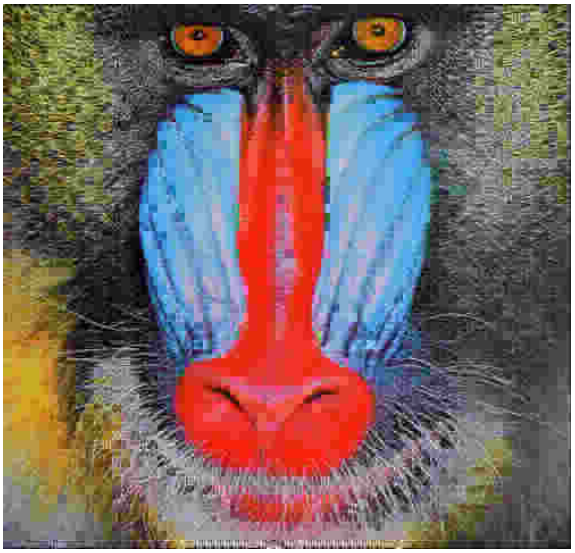
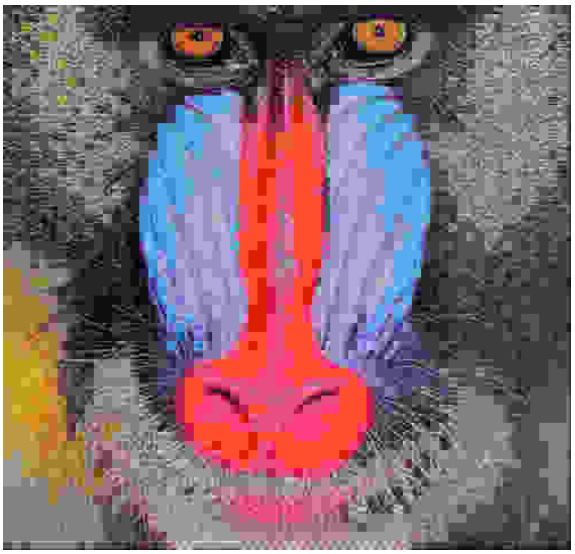
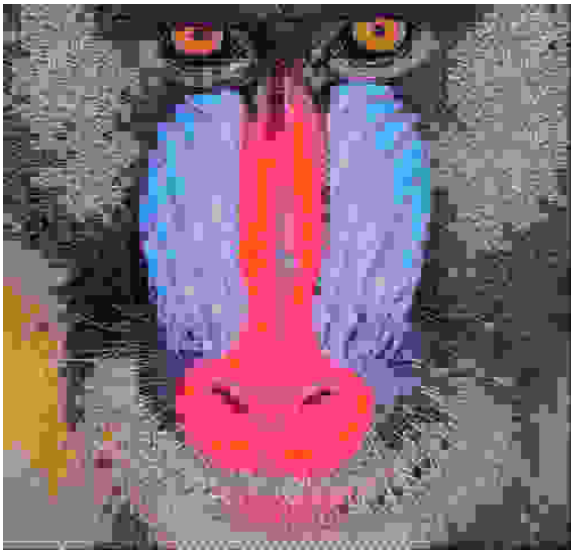
	5	79.4797

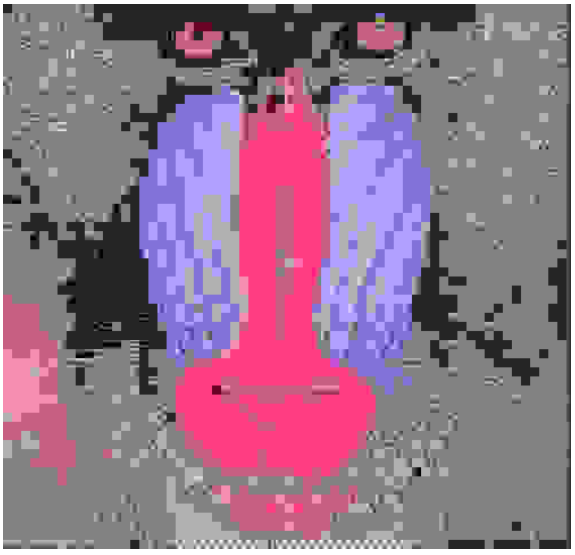
tucan image (bmp), 512x512x3	**Input**	**Ratio**
	-	-
	90	22.5186

	60	27.4657
	20	46.1552
	10	62.1833

	8	67.9863
	5	82.1575

baboon image (bmp), 480x500x3	Input	Ratio
	-	-
	90	13.0441
	60	17.0767

	20	35.0403
	10	55.1447
	8	63.7891

	5	83.7009

5.3.1 Comments

In this case, the compression rate is really high already at the beginning; the image start showing really annoying artifacts at an input parameter equal to 25/20, but the degradation of the image is really high from here on: with a compression factor around 60, images are hardly recognized. I'll make a reviewing table as I've done before, in order to make a comparison with this implementation and JPEG.

Image	**Input: 20**	**Input: 10**	**Input: 5**	**JPEG: 20**	**JPEG: 10**	**JPEG: 5**
monarch	41.4484	54.4734	79.4797	26.9377	38.9189	53.9985
tucan	46.1552	62.1833	82.1575	51.8107	76.0404	107.2091
baboon	35.0403	55.1447	83.7009	51.8107	76.0404	107.2091

5.4 Final observations on results

Since it is impossible that my computations (in particular I'm referring to the second one) has an higher compression rate to the JPEG one, I've supposed two reasons to motivate this fact: the first is that I don't manage in the same way as JPEG do the conversion of the input to a compression/quality factor (and that's could be the main reason) or, probably, I don't grasp correctly the compression factor of the JPEG computation (this is suggested by the fact that two images have the same results). In both cases, the main result is that our second implementation of course has a better compression than the first one, but obviously couldn't have better results than JPEG.

5.5 Optimization

Considering what I've read on books and online, I've thought that there could be some improvement that could be done on the code in order to obtain better performances; I'm going to list them in the following:

- Books suggest to use Arithmetic coding instead of Huffman, in order to improve the performances around 5-7%

- It is also recommend to downsample the two chrominance components; we can choose on three types of compression ratio, as 4:4:4 (no subsampling), 4:2:2 (reduction of a factor

2 on the horizontal direction) or 4:2:0 (reduction of a factor 2 on both horizontal and vertical directions). The better choice may could be to let the user choose which one he/she want to use

- May would give better result using directly Huffman and not the Huffman tables, that of course are convenient for their simplicity but not applicable in the particular case

- Even though is not strictly required in this case, a different implementation for RGB and B&W images should be proposed, in order to manage both of them in the computation; it is just needed to skip, in the second case, the color conversion and work just on one channel

6 Conclusions

In this work I've developed two kind of JPEG implementations: in the first, I've used just a Huffman encoding procedure, in the second one I've used Huffman on runlength coding. In both cases, I've obtained good results in terms of compression, in particular in the second one, even though this is related to an higher and faster degradation of the image. I've tried to make a comparison with the JPEG effective approach, but for reasons probably related to conversions or erroneous grasped information, I wasn't capable to make this point accurately.

References

[1] Khalid Sayood, *Introduction to data compression*, Elsevier, 4th edition, 2012

[2] Giancarlo Calvagno, *Lossy Coding 6 - Source Coding*, University of Padua, 2019

[3] Wikipedia, https://en.wikipedia.org/wiki/JPEG

[4] Wikipedia, https://en.wikipedia.org/wiki/YUV

[5] Wikipedia, https://en.wikipedia.org/wiki/YCbCr

[6] Vocal, https://www.vocal.com/video/rgb-and-yuv-color-space-conversion/

[7] Wikibooks, https://en.wikibooks.org/wiki/JPEG_-_Idea_and_Practice/The_Huffman_coding

[8] Cristi Cuturicu, 1999, https://www.opennet.ru/docs/formats/jpeg.txt

[9] Andrew B. Lewis, University of Cambridge, https://www.cl.cam.ac.uk/teaching/1011/R08/jpeg/acs10-jpeg.pdf