

Project 1

Manipulating text, audio and image/video

Diogo Mendes^[88801], Lúcia Sousa^[93086], Rodrigo Martins^[93264]

Universidade de Aveiro

Abstract. This paper consists in a report of Project#1. Every exercise is briefly explained with the key points being explained. Each exercise also as a subsection with some results obtained.

Keywords: PSNR · BGR · Samples.

1 User guide

All source code can be found in the github: <https://github.com/rodrigo740/IC>

All programs explained below include the commands that must be ran to execute them individually. However, in the main directory the user can find two scripts. The first *compileAll* compiles all the programs (exercises 2 to 11) and the script *testAll* runs all of the previously compiled programs.

To compile some programs that use AudioFile it is necessary to add the AudioFile library (which is a git submodule), this can be achieved with the following commands while in the git main directory:

```
$ git submodule init
$ cd AudioFile/
$ git submodule update
```

2 Part B

2.1 Exercise 2

In exercise 2 it is requested that the students come up with the implementation of a program to copy a text file character by character. To compile this program it is necessary the following command:

```
$ g++ ex2.cpp -o ex2
```

After compiling, to run the program and copy the file *lusiadas.txt* character by character, to the file *lusiadasCocy.txt*, the command is as follows:

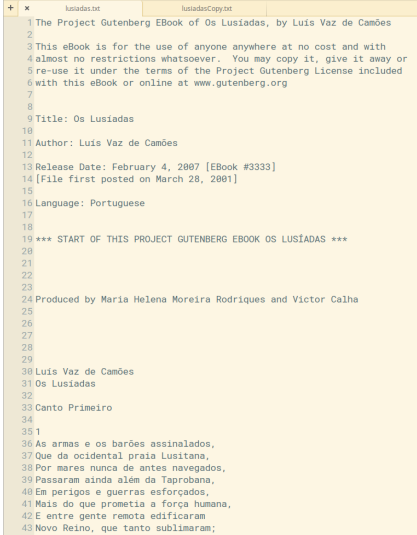
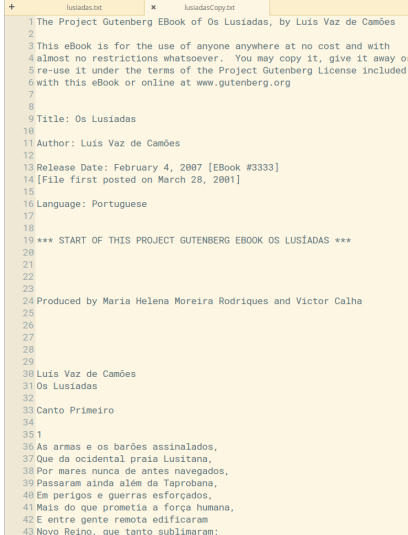
```
./ex2 <input_file> <output_file>
$ ./ex2 txts/lusiadas.txt txts/lusiadasCopy.txt
```

For this exercise, the input file name and the output file name are taken as arguments. Then, a vector of chars is created and the input file is read, character by character, putting every char into the vector. After reading the entire file, the vector is iterated and we pass these values to the output file.

Results

To measure the processing time, the program was executed ten times, the processing times were recorded and the average was calculated. In this case, the average processing time obtained was 21905 microseconds, approximately 0.02 seconds.

The result obtained after copying a text file represented in Fig1, is shown in Fig2.

	
: lusiadas.txt	: lusiadasCopy.txt

2.2 Exercise 3

In exercise 3 the goal is to create a program to copy an audio file in wav format, sample by sample. In this case, it was decided that the AudioFile library was going to be used. To compile the program the user must run the following command.

```
$ g++ ex3.cpp -o ex3
```

After compiling, to run the program and copy the audio file noise.wav sample by sample, to the audio file copyNoise.wav, the command is as follows:

```
./ex3 <input_audio> <output_audio>
$ ./ex3 audio/noise.wav audio/copyNoise.wav
```

For this exercise, the input audio file name and the output audio file name are received as arguments. Then, using the AudioFile library, it is possible to get the number of channels and samples per channel, with that and some cycles, each sample of each channel from the input file is copied to the output file.

Results

To measure the processing time, the program was executed ten times, the processing times were recorded and the average is calculated. In this case, the average processing time was 210733 microseconds, approximately 0.21 seconds.

2.3 Exercise 4

Exercise 4 requests the implementation of a program to copy an image, pixel by pixel and a video, frame by frame. Both file names should be passed as command line arguments to the program. To compile this program the user must run the following command:

```
$ g++ ex4.cpp -o ex4 `pkg-config --cflags --libs opencv`
```

After compiling, to run the program and copy the image lena.ppm to lenaCopy.ppm and video teste.avi to testeCopy.avi, the command is as follows:

```
./ex4 <original_img> <copied_img> <original_vid> <copied_vid>
$ ./ex4 images/lena.ppm images/lenaCopy.ppm video/teste.avi video/testeCopy.avi
```

For this exercise, we start by accepting the input image and video names as well as their copies names as arguments. To copy the image, the program will create a empty Mat object with the same size as the original image and will populate it with the same pixel values as the original image. This is done by reading each pixel and copying their value to their respective position in the empty Mat.

For the video, the program creates a video encoded in AVI format with the same characteristics (fps count, frame width and height) as the original video then while the original video has frames it reads them and copies them to the output video.



: lena.ppm



: lenaCopy.ppm

Results

To measure the processing time, the program was executed ten times, the processing times were recorded and the average was calculated. In this case, the average processing time was 321176 microseconds, approximately 0.32 seconds.

The result obtained after copying an image file represented in lena.ppm, is shown in lenaCopy.ppm.

3 Part C

3.1 Exercise 5

In exercise 5 it is asked for a program that calculates the histogram of the letters that exist in a text file and the corresponding entropy. To compile this program it is necessary to run the following command:

```
$ g++ ex5.cpp -o ex5 -lstdc++fs
```

After compiling, to run the program and calculate the histogram to histogram.txt of the text file lusiadas.txt, the command is as follows:

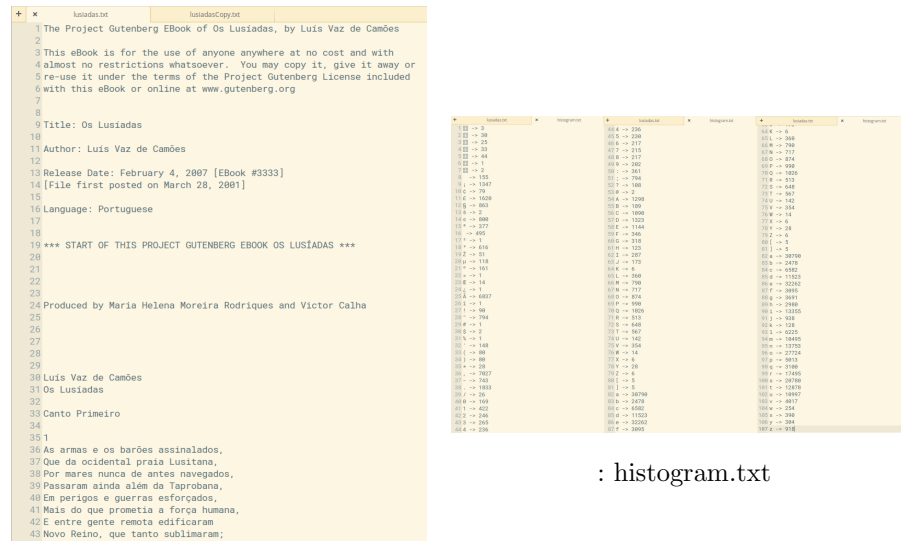
```
./ex5 <input_file> <output_file>
$ ./ex5 txts/lusiadas.txt txts/histogram.txt
```

For this exercise, the input file name and the output file name are received as input as arguments. Afterwards, using a map, it is possible to store each character and the respective number of times it appeared in the text file.

Results

To measure the processing time, the program was executed ten times, the processing times were recorded and the average was calculated. In this case, the average processing time was 69072 microseconds, approximately 0.07 seconds.

The entropy value calculated was 4.01765.



: histogram.txt

: lusiadas.txt

3.2 Exercise 6

In exercise 6 it is requested the implementation of a program that calculates the histogram of an audio sample and corresponding entropy. The user should visualize the histogram of the left and right channels, as well as, the histogram of the average of the channels (the mono version). To compile this program it is necessary to run the following command:

```
$ g++ ex6.cpp -o ex6
```

After compiling, to run the program and calculate the histogram to histogram_audio.txt of example.wav audio file, the command is as follows:

```
./ex6 <input_audio> <output_file>
$ ./ex6 audio/example.wav txts/histogram_audio.txt
```

For this exercise, the input file name and the output file name are taken as arguments. Then, using the AudioFile library, it is possible to get the number of channels and samples per channel, and using two maps, one for the right

channel, and other for the left channel, counting the times each sample appears in that channel and store that result. A third map is used, to store the samples from both channels and the number of times they appear.

Results

To measure the processing time, the program was executed ten times, the processing times were recorded and the average was calculated. In this case, the average processing time was 760349 microseconds, approximately 0.76 seconds.

The entropy for the left channel calculated was 14.0795, for the right channel was 14.1038 and for the mono version was 26.2886.

#	x	histogram_audio.txt	#	x	histogram_audio.txt	#	x	histogram_audio.txt
1501	R	-0.448002 -> 1	46652	L	0.8776002 -> 23	85168	M	0.289062 -> 7
1501	R	-0.439941 -> 2	46652	L	0.8776367 -> 23	85168	M	0.289062 -> 7
1506	R	-0.439941 -> 2	46653	L	0.8776672 -> 24	85170	M	0.289093 -> 2
1506	R	-0.43988 -> 3	46654	L	0.8776978 -> 16	85171	M	0.289124 -> 5
1507	R	-0.439789 -> 1	46655	L	0.8777283 -> 23	85172	M	0.289154 -> 5
1508	R	-0.439758 -> 1	46656	L	0.8777588 -> 14	85173	M	0.289185 -> 4
1509	R	-0.439728 -> 2	46657	L	0.8777893 -> 19	85174	M	0.289215 -> 8
1510	R	-0.439667 -> 1	46658	L	0.8778198 -> 29	85175	M	0.289246 -> 5
1511	R	-0.439545 -> 1	46659	L	0.8778503 -> 15	85176	M	0.289276 -> 4
1512	R	-0.439484 -> 1	46660	L	0.8778808 -> 22	85177	M	0.289307 -> 2
1513	R	-0.439423 -> 2	46661	L	0.8779114 -> 29	85178	M	0.289337 -> 3
1514	R	-0.439362 -> 1	46662	L	0.8779419 -> 21	85179	M	0.289368 -> 9
1515	R	-0.43927 -> 1	46663	L	0.8779724 -> 23	85180	M	0.289398 -> 4
1516	R	-0.43924 -> 1	46664	L	0.8780029 -> 26	85181	M	0.289429 -> 5
1517	R	-0.439209 -> 1	46665	L	0.8780334 -> 17	85182	M	0.289459 -> 4
1518	R	-0.439177 -> 1	46666	L	0.878064 -> 11	85183	M	0.28949 -> 4
1519	R	-0.439087 -> 1	46667	L	0.8780945 -> 22	85184	M	0.28952 -> 4
1520	R	-0.439026 -> 1	46668	L	0.878125 -> 20	85185	M	0.289551 -> 9
1521	R	-0.438995 -> 1	46669	L	0.8781555 -> 25	85186	M	0.289581 -> 7
1522	R	-0.438965 -> 2	46670	L	0.878186 -> 23	85187	M	0.289612 -> 4
1523	R	-0.438843 -> 1	46671	L	0.8782166 -> 22	85188	M	0.289642 -> 7
1524	R	-0.438782 -> 1	46672	L	0.8782471 -> 18	85189	M	0.289673 -> 6
1525	R	-0.438751 -> 1	46673	L	0.8782776 -> 23	85190	M	0.289703 -> 8
1526	R	-0.438721 -> 1	46674	L	0.8783081 -> 23	85191	M	0.289734 -> 18
1527	R	-0.43869 -> 1	46675	L	0.8783386 -> 20	85192	M	0.289764 -> 7
1528	R	-0.438629 -> 2	46676	L	0.8783691 -> 23	85193	M	0.289795 -> 6
1529	R	-0.438538 -> 2	46677	L	0.8783997 -> 26	85194	M	0.289825 -> 4
1530	R	-0.438507 -> 1	46678	L	0.8784302 -> 29	85195	M	0.289856 -> 8
1531	R	-0.438477 -> 1	46679	L	0.8784607 -> 13	85196	M	0.289886 -> 5
1532	R	-0.438446 -> 1	46680	L	0.8784912 -> 19	85197	M	0.289917 -> 4
1533	R	-0.438393 -> 1	46681	L	0.8785217 -> 13	85198	M	0.289948 -> 4
1534	R	-0.438363 -> 1	46682	L	0.8785522 -> 18	85199	M	0.289978 -> 6
1535	R	-0.438171 -> 1	46683	L	0.8785828 -> 17	85200	M	0.290009 -> 3
1536	R	-0.438141 -> 2	46684	L	0.8786133 -> 35	85201	M	0.290039 -> 8
1537	R	-0.43811 -> 2	46685	L	0.8786438 -> 18	85202	M	0.29007 -> 7
1538	R	-0.438049 -> 2	46686	L	0.8786743 -> 12	85203	M	0.2901 -> 8
1539	R	-0.438019 -> 1	46687	L	0.8787048 -> 12	85204	M	0.290131 -> 8
1540	R	-0.437864 -> 2	46688	L	0.8787354 -> 28	85205	M	0.290161 -> 6
1541	R	-0.437805 -> 1	46689	L	0.8787659 -> 20	85206	M	0.290192 -> 7
1542	R	-0.437775 -> 3	46690	L	0.8787964 -> 18	85207	M	0.290222 -> 5
1543	R	-0.437683 -> 2	46691	L	0.8788269 -> 14	85208	M	0.290253 -> 8
1544	R	-0.437622 -> 1	46692	L	0.8788574 -> 17	85209	M	0.290283 -> 6
1545	R	-0.437592 -> 1	46693	L	0.8788879 -> 28	85210	M	0.290314 -> 5
1546	R	-0.437561 -> 1	46694	L	0.8789185 -> 18	85211	M	0.290344 -> 4
1547	R	-0.4375 -> 1	46695	L	0.878949 -> 17	85212	M	0.290375 -> 8

: histogram_audio.txt

3.3 Exercise 7

In exercise 7 it is required that the students create a program that calculates the histogram of an image file and the corresponding entropy. Considering the histogram of each channel in color images, as well as the histogram of the corresponding grayscale version. To compile this program it is necessary to run the following command:

```
$ g++ ex7.cpp -o ex7 `pkg-config --cflags --libs opencv`
```

After compiling, to run the program and calculate the histograms of lena.ppm image file, the command is as follows:

```
./ex7 <input_img>
$ ./ex7 images/lena.ppm
```

In this exercise, the program starts by receiving the original image file as an argument, then creating a clone of that file, but in grayscale using the `cvtColor` (input, output, code) method of the openCV library.

```
cvtColor(image, gray, COLOR_BGR2GRAY);
```

After that the image is split by its 3 channels to 3 different Mats (blue, green and red) and 3 empty Mats with 256 rows (because there are 256 different values for each pixel) and 1 column are created. These empty Mats are associated with one of the image channels. All of the empty Mats and the grayscale Mat of the image are converted to CV 32F, this step is done to facilitate the calculations needed for the histograms. Afterwards, the program will read each pixel and increment the value stored by each Mat, then 4 more Mats will be created to visualize the information gathered, but before the program starts to manipulate the data from values in a Mat to actual viewable data it is necessary to normalize the values acquired. Such process is done by using the `normalize` function in each Mat. To insert the data in the Mats the program uses the `rectangle` method (output, input, point1, point2, color, thickness):

```
rectangle(hist_img_blue,
Point(2 * i, hist_img_blue.rows - normalized_hist_blue.at<float>(i)),
Point(2 * (i + 1), hist_img_blue.rows), Scalar(255,0,0), -1);
```

Finally, the histograms are ready to be shown, but before the program does that it calculates the entropy of each channel and the grayscale version, when this calculations end the program stores each histogram in the histograms folder.

Results To measure the processing time, the program was executed ten times, the processing times were recorded and the average was calculated. In this case, the average processing time was 42588 microseconds, approximately 0.42 seconds.

The entropy values are:

- Entropy Blue: 6.96843
- Entropy Red: 5.02747
- Entropy Green: 7.59404
- Entropy Grayscale: 7.44504

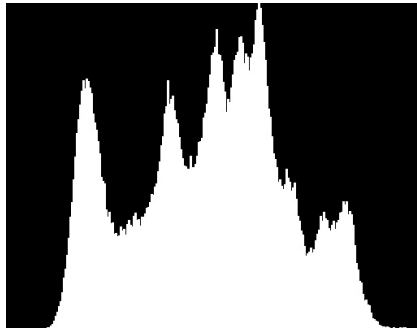
The histograms of `lena.ppm` can be seen in the following images.



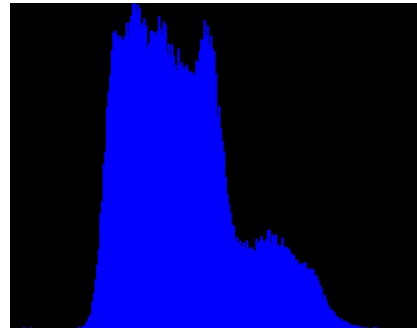
(a) Original image



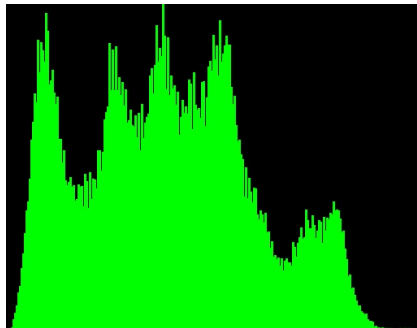
(b) Grayscale image



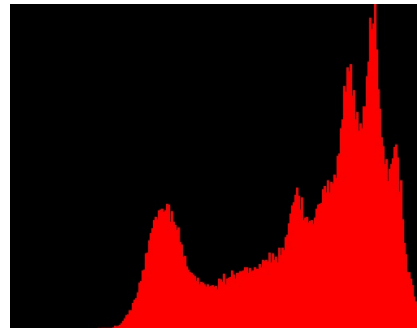
(c) Grayscale histogram



(d) Blue histogram



(a) Green histogram



(b) Red histogram

4 Part D

4.1 Exercise 8

The goal of exercise 8 is to reduce the number of bits used to represent each audio sample, a less complex uniform scalar quantization. To compile this program it is necessary to run the following command:


```
$ g++ ex8.cpp -o ex8
```

After compiling, to run the program and calculate the histogram to histogram_audio.txt of example.wav audio file, the command is as follows:

```
./ex8 <input_audio> <number of levels>
$ ./ex8 audio/noise.wav 8
```

For this exercise, the program starts with receiving the original audio file and the number of levels that the user wishes as arguments. Afterwards, using the AudioFile library a copy of the input audio file is created. The number of divisions is calculated using the equation:

```
int div = (32/niveis)/2;
```

Where *div* is the number of divisions and *niveis* is the number of levels given by the user. The logic behind this is that 32 is the number of bits of an audio sample, divided by the number of levels wished for, and divided again by 2. The division by 2 is necessary because removing one bit divided the size of the sample in half, removing two bits divides the size of the sample in four, etc. Therefore, the equation above was reached.

Afterwards, the usage of two *for* cycles allows going through all the audio samples. Then, *div* (the variable talked above) shifts right and *textitdiv* shifts left are made to the samples. Because of the shifts the *div* bits further right are zero. The program ends after the audio file with the manipulated samples is saved in the folder audio with the name *ex8.wav*.

Results

To measure the processing time, the program was run ten times, each time recording the processing times and then the average was calculated. In this case, the average processing time was 342024 microseconds, approximately 0,34 seconds.

4.2 Exercise 9

In exercise 9 it is requested the implementation of a program to reduce the number of bits used to represent each pixel of an image. To compile this program it is necessary to run the following command:

```
$ g++ ex9.cpp -o ex9 `pkg-config --cflags --libs opencv`
```

After compiling, to run the program and calculate the histograms of lena.ppm image file, the command is as follows:

```
./ex9 <input_img> <number_of_levels>
$ ./ex9 images/lena.ppm 32
```

For this exercise, the program starts by receiving the image file name and the number of levels of quantization creating a clone of that file but in grayscale using the `cvtColor` (input, output, code) method of the `openCV` library.

```
cvtColor(image, gray, COLOR_BGR2GRAY);
```

After that, the image is split by its 3 channels to 3 different Mats (blue, green and red) all of these empty Mats and the grayscale Mat of the image are converted to CV 32F, this step is done to facilitate the calculations. Next the program will prepare an array with the values per level that the pixels will assume. It is importante to notice that to minimize the error the value of the pixels will be changed to the middle value of the level.

```
const int d = 256/stoi(argv[2]);

double valueByLevel[stoi(argv[2])];

for (int i = 0; i < stoi(argv[2]); i++)
{
    valueByLevel[i] = (i+1)*d/2 + (i*d)/2;
}
```

After this it is just a matter of reading each value of the pixels, find which level it belongs to and change its value.

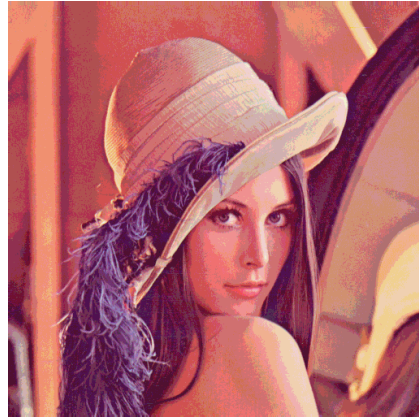
```
for (int i = 0; i < gray.rows; i++)
{
    for (int j = 0; j < gray.cols; j++)
    {
        // Grayscale
        value = gray.at<float>(i,j);
        temp = trunc(value/d);
        gray.at<float>(i,j) = valueByLevel[temp];
    }
}
```

Finally, the program will merge the 3 channels back into 1 Mat and convert the colored image and the grayscale version to *CV 8UC3*, this needs to be done or the images won't be able to be displayed nor saved. The colored version is saved in *color.ppm* and the grayscale one is saved in *grayscale.pm* both in the images folder. The program will also calculate the PSNR and the maximum per pixel absolute error using the same method as described further below in exercise 11.

Results



(a) Original Image



(b) Colored with 8 level



(c) Grayscale with 8 levels

To measure the processing time, the program was executed ten times, the processing times were recorded and the average was calculated. In this case, the average processing time was 100875 microseconds, approximately 0.10 seconds.

The SNR value calculated was 21.0962 dB for the colored version and 22.2611 dB for the Grayscale version, the maximum per pixel absolute error for the colored version was 88.8061 and for the grayscale version 22.2611.

The result of quantizing the lena.ppm image by 8 levels can be seen in the following images.

4.3 Exercise 10

In exercise 10 it is requested to implement a program that prints the signal-to-noise ratio (SNR) of a certain audio file in relation to the original file, as well as the maximum per sample absolute error. To compile this program it is necessary the following command:

```
$ g++ ex10.cpp -o ex10
```

After compiling, to run the program and print the SNR and the maximum per sample absolute error of the original file noise.wav and the output file of Exercise 8 ex8.wav, the command is as follows:

```
./ex10 <audio_file_original> <audio_file_exercise8>
$ ./ex10 audio/noise.wav audio/ex8.wav
```

For this exercise, the audio file names of the original audio and the audio file output of Exercise 8 are taken as arguments. Then, using the AudioFile library, it is possible to get the number of channels and samples per channel.

To calculate the SNR we calculate the sum of the absolute value of each sample from each channel squared, for the original audio and the quantized audio. Then, we use the formula:

$$SNR = 10 * \log \frac{\sum |(original_samples[i])|^2|}{\sum |(quantized_samples[i])|^2|}$$

To calculate the maximum per sample absolute error:

$$AbsoluteError = \sum |(original_samples[i])^2 - (quantized_samples[i])^2| * \frac{1}{N_c * N_s}$$

N_c - Number of channels; N_s - Number of samples.

Results

To measure the processing time, the program is executed ten times, the processing times are recorded and the average is calculated. In this case, the average processing time was 170051 microseconds, approximately 0.17 seconds.

The SNR value calculated was -38.9298 and the maximum per sample absolute error was 0.349607.

4.4 Exercise 11

The goal of exercise 11 is to implement a program that prints the signal-to-noise ratio (SNR) of a certain image file in relation to the original file, as well as the maximum absolute error per pixel. To compile this program it is necessary the following command:

```
$ g++ ex11.cpp -o ex11 `pkg-config --cflags --libs opencv`
```

After compiling, to run the program and print the SNR and the maximum per pixel absolute error of the original image lena.ppm and the output image of Exercise 9 color.ppm, the command is as follows:

```
./ex11 <original_image> <quantized_image>
$ ./ex11 images/lena.ppm images/color.ppm
```

For this exercise, the program starts with receiving the name of the 2 images to be compared as input, then separates each image by each color channel and reads each pixel value, following the formula:

$$e^2 = \frac{1}{NrNc} \sum_{i=1}^{Nr} \sum_{j=1}^{Nc} [f(i, j) - f'(i, j)]^2$$

Nr - Number of rows; Nc - Number of columns; f - Value of original image; f' - Value of reconstructed image.

The PSNR is calculated using the following formula:

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

A - Maximum value of 1 pixel(255) e - Mean squared error

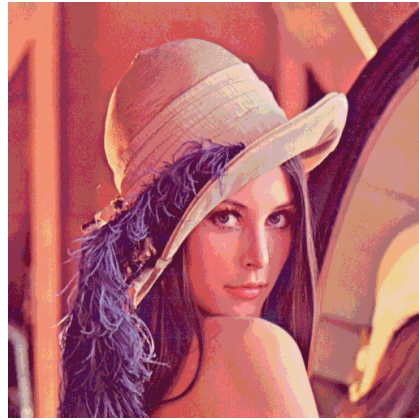
Results

To measure the processing time, the program was executed ten times, the processing times were recorded and the average was calculated. In this case, the average processing time was 42324 microseconds, approximately 0.04 seconds.

The PSNR value between the lena.ppm (original image) and color.ppm (reconstructed image) is 65.9607 dB and the maximum per pixel absolute error was 88.8061.



(a) Original Image



(b) Reconstructed Image