

Project 2

Entropy coding of audio and images

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

Universidade de Aveiro
DETI - Departamento de Electrónica, Telecomunicações e Informática
Information and Coding

Abstract. This paper consists on a report of Project#2. Every exercise is briefly explained with the key points being explained. Each exercise also as a subsection with some results obtained.

Keywords: Golomb · Samples · Prediction.

1 User guide

All source code can be found in the github: <https://github.com/rodrigo740/IC-Project-2>

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, whereas 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 A

2.1 Exercise 1 - Class BitStream

For this exercise it was requested the implementation of a class BitStream to read/write bits from/to a file. This class is composed of a constructor that takes as parameters a string, which will correspond to the name of the file, and a character, 'r' or 'w', to decide which operation will be executed, read or write.

```
BitStream bs("example.txt", 'r');
```

Functions

The function **readbit** was implemented to read a single bit of a file.

```
int readbit(){
    int bit = 0;
    if(nbits == 8){
        buf=chars.at(pos);
        pos++;
        nbits = 0;
    }
    bit = (buf >> (7-nbits)) & 1;
    nbits++;
    return bit;
}
```

The function **readnbits** was implemented to read n bits of a file.

```
vector<int> readnbits(int n){
    vector<int> bit;
    for (int i=0; i < n; i++){
        if(nbits == 8){
            buf=chars.at(pos);
            pos++;
            nbits = 0;
        }
        bit.push_back((buf >> (7-nbits)) & 1);
        nbits++;
    }
    return bit;
}
```

The function **readstrings** was implemented to read a string of a file.

```
vector<int> readstrings(){
    vector<int> bit;
    string str;
    while(getline(ifs,str)){
        for(char x : str){
            while(nbits != 8){
                bit.push_back((x >> (7-nbits)) & 1);
                nbits++;
            }
            nbits=0;
        }
    }
    return bit;
}
```

The function **readFile** was implemented to read a file.

```
vector<int> readFile(){
    vector<int> bits;
    int b = readbit();
    bits.push_back(b);
    while (pos < chars.size()){
        b = readbit();
        bits.push_back(b);
    }
    return bits;
}
```

The function **writebit** was implemented to write a single bit in a file.

```
void writebit(int bit){
    buffer[nbits] = (bit << (7-nbits));
    nbits++;
    if (nbits == 8){
        char c = 0;
        for (int i = 0; i < 8; i++){
            c |= buffer[i];
        }
        ofs << c;
        nbits = 0;
    }
}
```

The function **writenbits** was implemented to write n bits in a file.

```
void writenbits(int n, int nbit){
    for (int i = 0; i < nbit; i++){
        writebit(n);
    }
}
```

The function **writestrings** was implemented to write a string in a file.

```
void writestrings(string s){
    for(int i=0; i<s.length() ;i++){
        writebit(s[i]-48);
    }
}
```

2.2 Exercise 2 - Test BitStream Class

In exercise 2 the goal is to create a program to test the class BitStream previously created and implemented. To compile the program the user must run the following command:

```
$ g++ ex2a.cpp -o ex2a
```

After compiling, to run the program the command is as follows:

```
$ ./ex2a
```

To test the class BitStream a simple program was implemented that calls the functions write one bit, write n bits, read one bit, read n bits, etc.

Results

Since the program tests several actions by analysing the output it is possible to conclude that the class BitStream is operational and working as intended.

2.3 Exercise 3 - Class Golomb

For this exercise it was requested the implementation of a class Golomb. This class is composed of a constructor that takes as parameter a integer, m .

```
Golomb g(4);
```

This class has an encoding function, so it is possible to encode a number, whether this number is positive or negative, which is achieved by folding the values. Positive values are folded into even values while negative values are folded into odd values.

The value to be encoded is divided by m and the quotient is encoded in unary code. If m is a power of 2, the remainder is converted to binary, otherwise the remainder is converted to truncated binary.

The Golomb class also has a decoding function, given the code of an encoded value, it returns the decoded value.

2.4 Exercise 4 - Test Golomb Class

In exercise 4 the goal is to create a program to test the class Golomb. To compile the program the user must run the following command:

```
$ g++ ex4a.cpp -o ex4a
```

After compiling, to run the program the command is as follows:

```
$ ./ex4a
```

The Golomb class was tested by varying the value of m between 2 and 1500, and for each value of m encoded values between -2000 and 2000.

Results

After comparing the values to be encoded with the decoded values, it is possible to conclude that the Golomb class is functional, since the values obtained were the same.

3 Part B

3.1 Exercise 1

The first exercise of this part requested the development of a lossless predictive audio codec followed by Golomb encoding. For the prediction of the next sample one predictor was implemented. The predictor tries to, as the name itself states, predict the next value. In this case, it returns the average of the current sample and the previous sample.

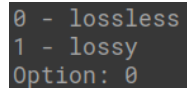
To compile this program it is necessary to run the following command:

```
$ g++ ex1b.cpp -o ex1b -lstdc++fs
```

After compiling, to run the program the command is as follows:

```
./ex1b <input_audio> <output_audio>
$ ./ex1b audio/sample01.wav audio/output_sample01.wav
```

Right after that, the user will be asked which option should be selected, if the user wants lossless coding or lossy coding. For this exercise, the user will have to choose the lossless option, so the user will have to click on the 0 key.



```
0 - lossless
1 - lossy
Option: 0
```

Fig. 1: Capture of the terminal with the option to select

The input file name and the output file name are taken as arguments. The first step is to calculate the optimal m for the Golomb code. The optimal m is calculated based on the following mathematical expressions:

$$\begin{aligned} \text{meanSamples} &= \frac{\text{sumSamples}}{\text{numSamples} * \text{numChannels}} \\ \alpha &= \frac{\text{meanSamples}}{\text{meanSamples} + 1.0} \\ m &= \lceil \frac{-1}{\log_2(\alpha)} \rceil \end{aligned}$$

The next step is to read each sample from the two channels and make the average of the two, predict the next sample and calculate the residual value. The residual value is calculated by subtracting from the value of the average of the current samples from the R and L channels the predicted value. The predicted value is predicted based on the average of the previous two samples. Finally the residual values are encoded using the class Golomb.

For the decoding process, a file is passed as an argument. This file contains the bits of the encoded audio. To read each bit of the encoded audio the class Bit-Stream. Code words are then decoded using the Golomb class. This code words are formed by a unary code part and a binary code part, each part represents the quotient and the remainder, respectively.

Results

The table below presents the ideal m value for each audio file.

Audio File	Ideal m
sample01	1900
sample02	2162
sample03	37
sample04	1085
sample05	70
sample06	98
sample07	7

After encoding the seven different audios and listening to the originals it is possible to compare them with the respective encoded audio. A slight difference is noticeable, despite the name of this method being lossless, losses still do occur. For example, in the audio file *sample01.wav* the sound has less volume compared to the original.

3.2 Exercise 2

In exercise 2 the goal is to implement a program that calculates the histograms of the residuals obtained after the prediction, as well as the entropy value. First of all, it is possible to choose the option lossless or lossy on the terminal. And for both the entropy and the histogram of either the residual values or the original values is calculated. The histograms are saved in text files with the following names: *histogram_ex1b_residual_lossless.txt*, *histogram_ex1b_residual_lossy.txt* and *histogram_ex1b_original.txt*.

To compile this program it is necessary to run the following command:

```
$ g++ ex1b.cpp -o ex1b -lstdc++fs
```

After compiling, to run the program and calculate the histograms of sample01.wav audio file, the command is as follows:

```
./ex1b <input_audio> <output_audio>
$ ./ex1b audio/sample01.wav audio/output_sample01.wav
```

In this exercise, the program starts by receiving the original audio file and the output audio file as arguments. The entropy for the original data is then calculated, afterwards the audio file will be encoded. The residual value will be calculated by calculating the difference between the sample value and the predicted value. Using these residual values, before being coded, the entropy and the histogram will be determined.

Results

The table below presents the entropy for residual values, for both lossy and lossless options, and original values for each audio file. In the lossy option, the residual quantization taken into account was 8.

Audio File	Residual Lossless Entropy	Residual Lossy Entropy	Original Entropy	Original Size (bits)	Lossless Compressed Size	Lossy Compressed Size	Processing Time (s)
sample01	3.74618	2.74726	13.9086	5176796	1780505	1780514	12
sample02	3.58882	2.60135	13.035	2589596	971082	971082	6
sample03	0.874605	0.685209	13.2678	3530396	661941	661941	5
sample04	3.24602	2.26254	14.1619	2354396	809308	809308	5
sample05	0.928163	0.709486	12.4516	3647996	797989	797989	7
sample06	0.799144	0.645844	11.8349	4235996	926614	926614	8
sample07	0.872937	0.685098	15.1849	3765596	369730	361389	4

Avg. Original Size (bits)	Avg. Lossless Compressed Size (bits)	Avg. Lossy Compressed Size (bits)	Avg. Size Difference Orig. Lossless (%)	Avg. Size Difference Orig. Lossy (%)	Avg. Processing Time (s)
3614396	902452,71	901262,43	24,97	24,94	6,7

For all audio files, the entropy results for the original data were much larger than the entropy results for lossless and lossy encoding. This indicates that there is considerably more information in the original audio file than in the same file after encoding. Now comparing the entropy results for lossless encoding with the entropy results for lossy encoding, it can be observed that there is a slight difference between both. The entropy for lossless encoding is slightly larger, indicating that it contains more information.

As for the size of the file, original and after compression, a big difference is noticeable. The compressed file size is much smaller. Regarding lossless and lossy compression, not many differences were noticed. The only files *sample01.wav* and *sample07.wav* were the only which difference was noticeable, in this case the lossy compressed file has a smaller size than the lossless compressed one.

In addition, there is a drop of approximately 25% in the size of the original file for the compressed file.

3.3 Exercise 3

Exercise 3 requested the inclusion of an option for lossy coding in the developed codec, based on residual quantization.

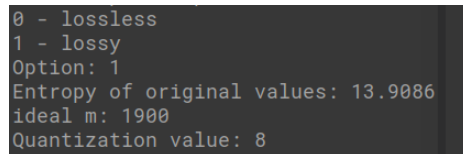
The command to compile is the same as shown before and that is shown below.

```
$ g++ ex1b.cpp -o ex1b -lstdc++fs
```

After compiling, to run the program the command is as follows:

```
./ex1b <input_audio> <output_audio>
$ ./ex1b audio/sample01.wav audio/output_sample01.wav
```

Now, to choose the lossy option, press 1 on the terminal. And then, write the quantization value.



```
0 - lossless
1 - lossy
Option: 1
Entropy of original values: 13.9086
ideal m: 1900
Quantization value: 8
```

Fig. 2: Capture of the terminal with the option and quantization value

The quantization value is the number of levels. The number of divisions is calculated using the equation:

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

Where div is the number of divisions and niveis the number of levels given by the user. It is calculated by starting with 32, which 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.

Afterwards, *div* (the variable mentioned above) number of shifts right and *textitdiv* number of shifts left are made to the residual values. Then, the manipulated residual values are encoded. Because of the shifts it is possible to be sure that the *div* bits further right are zero. The program ends after the audio file with the manipulated residual values decoded is saved.

Results

After encoding the seven different audios, listening to the originals and comparing them with the respective encoded audio, a greater difference is noticeable than the differences that were observed when it was lossless. For the same example, in the audio file *sample01.wav* the sound that used to have less volume, now also has more noise.

4 Part C

4.1 Exercise 1

In exercise 1 the students were requested to implement an image codec using the Golomb coding algorithm. This codec should consider a pre-processing stage in order to transform the image into the YUV 4:2:0 format before the encoding process. The codec is divided in two stages, the first one using lossless encoding and the second one using lossy encoding.

The lossless encoder complies to the following requirements:

- The frames should be encoded using spatial predictive coding based on the non-linear predictor of JPEG-LS or the 7 JPEG linear predictors;
- Entropy coding should be performed using Golomb codes;
- All the information required by the decoder should be included in the bit-stream (video format, frame size, encoder parameters, etc.).

To compile this program it is necessary to run the following command:

```
$ g++ ex1c.cpp -o ex1c -lstdc++fs `pkg-config --cflags --libs opencv`
```

After compiling, to run the program, the command is as follows:

```
./ex1c <input_img> <output_img>
$ ./ex1c images/lena.ppm images/output.ppm
```

The program starts by receiving 2 input arguments, the first one being the path to the input image and the second one the path to save the decompressed image, if the image width or height aren't a multiple of 2, the program will resize the image to fit that requirement (increase by 1 the width, height or both depending on the case), this operation is done both on the lossless and the lossy version of the codec.

After this the program will split the input image by its three color components (RGB) and perform the necessary calculations to obtain the YUV values.

The YUV values are obtained with the following equations:

- $Y = 0.299 * r + 0.587 * g + 0.114 * b + 16$
- $U = -0.147 * r - 0.289 * g + 0.436 * b + 128$
- $V = 0.615 * r - 0.515 * g - 0.100 * b + 128$

r, g, b are the values of the 3 color components (0-255)

Y will be a value between 16 and 235

U and V will be values between 16 and 240

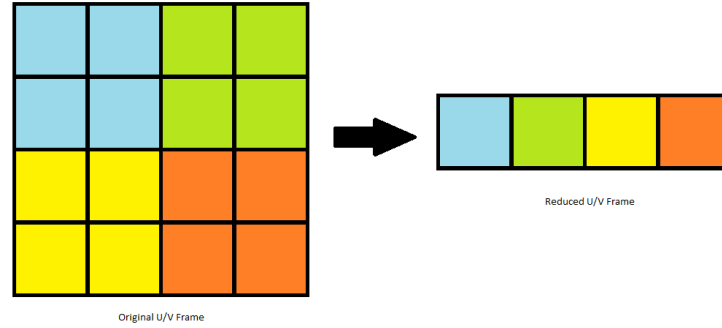


Fig. 3: U/V reduce operation

After the YUV values are obtained the program will reduce the size of the U and V frames to comply to the YUV 4:2:0, to achieve this the program will calculate the mean value for each 4 pixels and assign that value to a new U and V frame, this will result in 2 frames each with half of the width and height of the original image and a quarter of the amount of pixels.

Following this operation the program will now form the final YUV frame. This frame consists in grouping the Y, U and V values together to form a cohesive frame that is more compressible. In this frame all the Y values come first, followed by all the U values, followed finally by all the V values. This will result in a frame with as many Y values as there are pixels in the original image. Where X equals the height multiplied by the width, the first X indices in the array are Y values that correspond to each individual pixel. However, there are only one fourth as many U and V values. The U and V values correspond to each 2 by 2 block of the image, meaning each U and V entry applies to four pixels. After the Y values, the next $X/4$ indices are the U values for each 2 by 2 block, and the next $X/4$ indices after that are the V values that also apply to each 2 by 2 block. Its important to notice that this frame will always have the U and V values all present in one row, this means that the amount of U and V values will be the same as the amount of columns present in the frame.

Single Frame YUV420:



Fig. 4: YUV420 Frame

Position in byte stream:

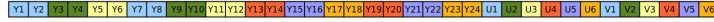


Fig. 5: YUV420 Frame in byte stream format

As shown in the above image, the Y, U and V components in YUV420 are encoded separately in sequential blocks. A Y value is stored for every pixel, followed by a U value for each 2×2 square block of pixels, and finally a V value for each 2×2 block. Corresponding Y, U and V values are shown using the same color in the diagram above. Read line-by-line as a byte stream from a device, the Y block would be found at position 0, the U block at position $x \times y$ ($6 \times 4 = 24$ in this example) and the V block at position $x \times y + (x \times y)/4$ (here, $64 + (64)/4 = 30$).

With the YUV420 frame done the program will start the encoding process. The first step in this process is to calculate the optimal M for the frame and for that the following equations are used:

$$\begin{aligned}
 mean &= \frac{sumValues}{originalImage.height * originalImage.width} \\
 \alpha &= \frac{mean}{mean + 1.0} \\
 m &= \lceil \frac{-1}{\log_2(\alpha)} \rceil
 \end{aligned}$$

The table below presents the ideal m value for each image file.

Audio File	Ideal m
airplane	180
anemone	138
arial	135
baboon	145
bike3	120
boat	142
girl	196
house	153
lena	149
monarch	132

With the optimal M calculated the encoding process begins following the first linear predictor (Predictor = a) mode of the lossless mode of JPEG. The results of the encoding process are written to a file named `encoded.bit` using the Bit-Stream Class. When the compressed file is completed the program shows to the user a prompt with the size of the original image and the compressed one. This concludes the compressing process of the codec.

With the compression done the program will start the decompression and image reconstruction phase. This phase consists in 3 parts. First the decoder will read the compressed image and recreate the YUV frame with the residual values. Secondly this YUV frame will be used to recreate the original frame accordingly with the predictor used. Finally with the original frame recreated the image will be reconstructed, shown to the user and saved to the output file specified in the beginning, with the values present in the frame. The RGB values of the image are obtained with the following equations:

- $yval = yval - 16$
- $uval = uval - 128$
- $vval = vval - 128$
- $R = 1.164 * yval + 1.596 * vval$
- $G = yval - 0.813 * uval - 0.391 * vval$
- $B = 1.164 * yval + 2.018 * uval$

Results

To test the lossless image codec a bash script was created. The purpose of this script is to test the image codec with all images from the image folder.

To use this script it is necessary to run the following command:

```
$ ./testAll_ex1c.sh
```

The results of this test run can be seen in the table below:

Image File	Original Entropy	Residual Entropy	Original Size(bits)	Compress size(bits)	Size Difference(bits)	Process. Time(s)
airplane	6.54614	4.22516	786447	394140	392307	2
anemone	7.20738	5.01427	1020201	512102	508099	2
anemone	7.26806	5.04085	1087080	546158	540922	3
baboon	7.2898	5.99569	786447	394339	392108	2
bike3	7.23702	4.48716	2153232	986812	1166420	5
boat	7.41037	5.23296	786447	393630	392817	2
girl	5.67639	3.79425	786447	394012	392435	2
house	6.56985	4.16797	196623	98394	98229	0
lena	7.32958	4.72938	786447	393494	392953	2
monarch	7.12174	4.13265	1179663	590401	589262	3

Avg. Original Size(bits)	Avg. Compressed size(bits)	Avg. Size Difference(%)	Avg. Processing Time(s)
956903,4	470348,2	49,15	2,3

After analyzing the results it is possible to conclude that the compression aspect of the image codec works as intended, averaging a 49% size drop when comparing the compressed image to the original image. For all image files, the entropy results for the original data were much larger than the entropy results for lossless encoding. This indicates that there is considerably more information in the original image file than in the compressed file after encoding. To take conclusions about the fidelity of the output image we need to compare the original image to the output image, both images can be seen in the figures below:



(a) Original Image



(b) Decompressed image

After analyzing both images we can conclude that the fidelity of the decompressed image is very high. Both images aren't 100% equal because when the YUV values are calculated an integer arithmetic is used, this means that any decimal value will be discarded (example: if $y = 4.3$ the `c++` will always truncate y , making it equal to 4 instead of 4.3), this means that some error is introduced in the pre processing stage of the codec. This value discrepancy will affect the YUV to RGB conversion making the output image have a slight difference in the image hue when compared to the original image. This hue difference can better be seen in the face of the person in the image, the face on the original picture is slightly more darker than the face on the decompressed image.

4.2 Exercise 2

In exercise 2 it is requested the implementation the second stage of the image codec, this is the lossy version of the codec. The lossy encoder receives 3 levels of quantization, one for each color component(YUV). Each level can be any positive number, but the results for any number greater than 8 will be the same as if the level chosen was 8, this results will be better explained further in the report.

To compile this program it is necessary to run the following command:

```
$ g++ ex2c.cpp -o ex2c -lstdc++fs `pkg-config --cflags --libs opencv`
```

After compiling, to run the program, the command is as follows:

```
./ex2c <input_img> <output_img> <nlevels_Y> <nlevels_U> <nlevels_V>
$ ./ex2c images/lena.ppm images/output.ppm 8 8 8
```

The program starts by receiving 5 input arguments, the first one being the input image, the second one the output image, the third one being the number of levels of the Y color component, the fourth one the number of levels of the U color component and the final one the number of levels of the V color component. With the number of levels for each component defined the program will calculate how many bits the residual values need to be shifted. This result is obtained following the equation:

```
const int d_y = (8/nlvl_y)/2;
const int d_u = (8/nlvl_u)/2;
const int d_v = (8/nlvl_v)/2;
```

Where d_y, d_u, d_v is the number of divisions of the corresponding color component and nlvl_y, nlvl_u, nlvl_v the number of levels given by the user. The logic behind this is that 8 is the number of bits of a pixel value, 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 pixel value in half, removing two bits divides the size of the pixel value in four, etc. For numbers greater or equal than 8 the division number will remain the same (divisions = 0) therefore producing the same results.

From this point onwards the program follows the same steps as the lossless version, calculating the YUV values, reducing the U and V frames and creating the YUV Frame will remain equal to before. The differences comes when the residual values are calculated, in the lossy version the residual values for each component are calculated independently in order to apply the correct quantization to each component. This quantization is applied in the form of a right shift in the encoding process and in the form of a left shift in the decoding process. When the decoder is recreating the YUV Frame with the residual values it will also recreate each component individually. After this the program will perform the same steps as the lossless version ending when the image is recreated, shown to the user and saved in the file specified in the beginning.

Results

To test the lossy image codec a bash script was created. The purpose of this script is to test the image codec with all images from the image folder.

To use this script it is necessary to run the following command:

```
$ ./testAll_ex2c.sh
```

The results of this test run can be seen on the table below:

Image File	Original Entropy	Residual Entropy	Original Size(bits)	Compressed size(bits)	Size Difference(bits)	Process Time(s)
airplane	6.54614	4.0197	786447	394079	392368	4
anemone	7.20738	4.749	1020201	512092	508109	5
anemone	7.26806	4.85891	1087080	546158	540922	5
baboon	7.2898	5.81821	786447	394310	392137	4
bike3	7.23702	4.21103	2153232	980294	1172938	11
boat	7.41037	5.02604	786447	393609	392838	4
girl	5.67639	3.48295	786447	393892	392555	4
house	6.56985	3.94931	196623	98393	98230	1
lena	7.32958	4.5408	786447	393495	392952	4
monarch	7.12174	3.92303	1179663	590401	589262	6

Avg. Original Size(bits)	Avg. Compressed size(bits)	Avg. Size Difference(%)	Avg. Processing Time(s)
956903,4	469672,3	49,08	4,8

After analyzing the results it is possible to conclude that the compression aspect of the image codec works as intended, averaging a 49,08% size difference when comparing the compressed image to the original image. This value is lower than the value of the lossless codec, which is as expected, because in this codec some bits are saved when the quantization happens. The average processing time is higher than the lossless version, which is also to be expected because in this codec more operations are done to the values of the YUV frame which increases the processing time.

For all image files, the entropy results for the original data were much larger than the entropy results for lossy encoding. This indicates that there is considerably more information in the original image file than in the compressed file after encoding.

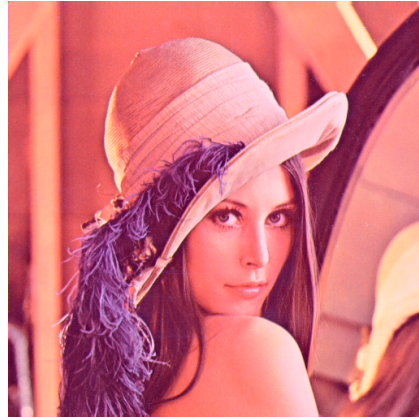
Now comparing the entropy results for lossless encoding with the entropy results for lossy encoding, it can be observed that there is a slight difference between both. The entropy for lossless encoding is slightly larger, indicating that it contains more information. This result is expected because in the lossy

encoding exists quantization and therefore some information is lost making the entropy get lower.

To take conclusions about the fidelity of the output image we need to compare the original image to the output image, both images can be seen in the figures below:



(a) Original Image



(b) Output with 8 8 8 levels



(c) Output with 7 4 4 levels

After analyzing all of the images we can conclude that the fidelity of the decompressed image is very high when the 8 levels per pixel are selected. Minor differences are seen in the hue of images (a) and (b) which is explained in the lossless coded. With more quantization introduced (image (c)), the distortion of the image will increase, this is as expected because if more bits are being lost in the quantization, the noise will increase and therefore the distortion will increase too.