Advanced Master in Space Communications Systems

PROJECT 1 : DATA COMPRESSION APPLIED TO IMAGES

Maël Barthe & Arthur Louchart

Project 1 : Data compression applied to images
Supervised by M. Tarik Benaddi

Special thanks to : Lenna Sjööblom

In the early seventies, an unknown researcher at the University of Southern California working on compression technologies scanned in the image of Lenna Sjööblom centerfold from Playboy magazine (playmate of the month November 1972).

Since that time, images of the Playmate have been used as the industry standard for testing ways in which pictures can be manipulated and transmitted electronically.

Over the past 25 years, no image has been more important in the history of imaging and electronic communications.

Because of the ubiquity of her Playboy photo scan, she has been called the "first lady of the internet". The title was given to her by Jeff Seideman in a press release he issued announcing her appearance at the 50th annual IS&T Conference (Imaging Science & Technology).

This project is dedicated to Lenna.

Table of contents

INTRODUCTION **3**

1. WHAT IS DATA COMPRESSION? 4
   1.1. Definition *4*
   1.2. Back in the past : a brief history of data compression *4*
2. HOW DOES COMPRESSION DATA WORK? 6
   2.1. Example : Morse code *6*
   2.2. Shannon information theory and source coding *7*
   a. Entropy 7
   b. Average number of bits per symbols 8
   c. Shannon's bound 8
   2.3. Lossless data compression *9*
   2.4. Lossy data compression *9*
3. ALGORITHMS USED FOR DATA COMPRESSION 10
   3.1. Lossless data compression *10*
   a. Huffman code 10
   b. Arithmetic code 13
   c. LZ & variants 13
   3.2. Lossy data compression : JPEG *18*
4. TECHNICAL IMPLEMENTATION 21
   4.1. Using LabVIEW *21*
   4.2. LabVIEW Design for data compression *21*
   4.3. Lossless data compression : Huffman code *22*
   4.4. Lossy data compression : JPEG *23*
   4.5. Optimization algorithms for lossless compression *24*
   4.6. Optimization algorithms for lossy compression *24*
   a. Algorithm 1 : Fixed EOB 24
   b. Algorithm 1+ : Zigzag reading with defined EOB location for each block 25
   c. Algorithm 2 : RLE DCT block 25
   d. Algorithm 3 : Errors detection and correction capacity 25
   4.7. Tests campaign *26*
   a. Lossless data compression : Huffman 27
   b. Lossy data compression : JPEG like 27
   c. Lossy data compression : real JPEG with additional stuff 27
   d. Analyze of results, performances & complexity 27
   4.8. Going further *28*

CONCLUSION **29**

APPENDIXES I : MATLAB AND LABVIEW SOURCE CODES **30**

APPENDIX II : CAMPAIGN TEST LOSSLESS COMPRESSION **31**

APPENDIX III : CAMPAIGN TEST LOSSY COMPRESSION **32**

APPENDIX IV : CAMPAIGN TEST LOSSY COMPRESSION OPTIMIZED **33**

BIBLIOGRAPHY **34**

Introduction

When you look at computers and the internet, data compression is everywhere. The music we listen to, the pictures we see, the movies, all that is data, and all that needs to be compressed in some way or another so it "fits" into our computer memory and so it can be downloaded or sent to someone else in a reasonable amount of time.

Regarding to our advanced master in Space Communication Systems, we had for assignment to prepare a state of the art about data compression for fall December 2017.

The main purpose of this project is to evaluate the performance of data compression techniques on images. This project has been split into three parts. The first part is dedicated on the state of the art on data compression. The second part is to recreate to evaluate the performance and the complexity of the reconstruction by transmitting compressed images (report to come after January 2018). The third part is to evaluate the impact of using different kind of channels on the reconstructed image (report to come after January 2018).

1. What is data compression?

1.1. Definition

In signal processing, data compression, (A.K.A "source coding" or "bit-rate reduction") involves encoding information using fewer bits than the original representation. Compression data can be either lossless or lossy.

- Lossless compression reduces bits by identifying and eliminating statistical redundancy. **No information is lost in lossless compression**.
- Lossy compression reduces bits by **removing unnecessary or less important information**. **Some information is lost.**

The process of reducing the size of a data file is often referred to as data compression. In the context of data transmission, it is called source coding (encoding done at the source of the data before it is stored or transmitted) in opposition to channel coding.

Compression is useful because it reduces resources required to store and transmit data. Computational resources are consumed in the compression process and, usually, in the reversal of the process (decompression). Data compression is subject to a space–time complexity trade-off.

Data compression has been developed since a long period (XIXe century) before being so used so often nowadays.

1.2. Back in the past : a brief history of data compression

In 1838, Samuel Morse invented the telegraph [1]. It was a very simple way of communication consisting of an electric battery, two buttons, two bells and a very, very, very long wire between them. Transmitting information through this system was easy: you press the button, the bell rings on the other side of the wire, hundreds of miles away. The guy at the other side presses his button, and your bell rings.

Later in 1949, the Shannon Fado algorithm was introduced by Claude Shannon, and Robert Fano. It assigns codes to symbols in a given block of data based on the probability of the symbol occurring.

Three years later, in 1952, revolution came up with a student at M.I.T. His idea: is it possible to build the best code possible to write a given message in its shortest representation? His name was David A. Huffman, and the set of codes he described was named after him. Huffman proved mathematically that no code can be shorter than his in representing a message. Unfortunately for him, other brilliant minds found better methods of coding.

In 1977, Abraham Lempel and Jacob Ziv introduced the LZ77 algorithm, which was the first to use a dictionary to compress data. It used a dynamic dictionary called a sliding window. One year later in 1978, they published the LZ78 algorithm which includes a static dictionary. Two year later, in 1980,

Terry Welch made changes which led to the LZW algorithm. This one became the most popular for many general purpose compression systems and is used in modems for example.

Late in the 80's, digital images became so popular that standards for image compression started evolving. The TIFF file format was published in 1986 and is still used for high color-depth images.

In 1987, the low bit rate audio encoding was introduced. A german company Fraunhofer-Gesellshaft began researching high quality, low bit-rate audio coding. They aimed to compress CD quality song without affecting the sound (e.g. lossless data compression). The same year, CompuServe introduced the Graphics interchange Format (GIF). It LZW used data compression to provide an image format for their files. This replaced their RLE format, which was black and white only. GIF can also be used to display animation.

In 1988 : first digital video coding. The first truly practical digital video coding standard was introduced by the ITU. This formed the basis for all subsequent video coding standards.

In 1989, a revolution came in. Fraunhofer-Gesellshaft received a German patent for MP3. They now license the patent rights to the audio compression technology. Note that MP3 is lossy compression data. Meanwhile, the same year, another famous data compression algorithm born: Phil Katz created the ZIP file format, which supports lossless compression and permits a number of compression algorithms. The name ZIP means to move at fast speed.

1992 is the year of creation of another famous file format: JPEG by the Joint Photographic Experts Group. The degree of compression can be adjusted, allowing a tradeoff between storage size and image quality. The same year, GZIP file format was released and is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. DEFLATE was intended as a replacement for LZW and other patented algorithms.

In 1993, Eugene Roshal, a Russian software engineer released the RAR format.

In 1996, version 1.0 of the Portable Networks Graphics (PNG) raster graphics file format was authored by a group of computers graphics experts and enthusiasts. While GIF has a limit of 256 colors, PNG is a lossless image compression format.

2. How does compression data work?

Answering this question is quite complicated without preliminary requirements of Shannon information theory [2]. We suggest having a basic quick look on this topic before getting started with the hard stuff.

2.1. Example : Morse code

In order to be practical, we will present a simple example with the Morse code to understand the signal processing approach and the process for common data compression. We will see in details in the other what lies behind the data compression.

We have seen up head that the Morse code was the first code to compress data information. But Morse code as its limits. How to send a whole word? A full sentence ? A huge text with several pages ? The solution is quite easy : you code it in bell rings, some short, some long. Each letter was assigned a sequence of bell rings:

- one short ring and one long ring means "A",
- one long and three short means "B",
- And so on...[3]

Nice, but… how does this relates to data compression? That's simple: Morse realized that some letters occur more frequently in the English language, while others occur rarely. So to save the time of the telegraphers, he made the more frequent letters shorter! Letter "E", for example, the most common one, uses only one single ring (and a short one). Letters like "Q", "Y" and "Z" take four rings, three of them long.

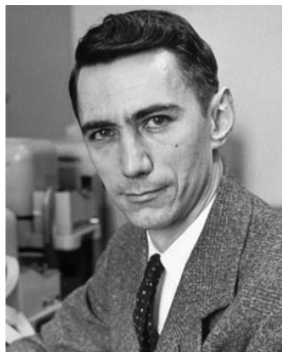
What does it mean? Let's say that we only have 3 letters, "E", "Q" and "Y". "E" occurs 80% of the time, while "Q" and "Y" occur 10% of the time. The telegrapher is a good one: he can "type" one ring every second (ok, he is a really bad one, but this is an example). And our message has 100 letters. We have two choices: A code in which each letter has the same size (which is 2 rings), and a code in which letter "E" takes 1 ring, letter "Q" takes 2 and letter "Y" takes 3 (similar to what the Morse code does). Then we have:

- 80 letters "E", 10 letters "Q" and 10 letters "Y", which means:
    - in the first code, 200 rings, in 200 seconds.
    - in the Morse like code, 80 rings plus 10 double rings (20 rings) plus 10 triple rings (30 rings), which amounts to 130 rings, in 130 seconds!

That's a 35% saving in time. So there's **REALLY** a compression. And a good one. And better yet: **No single letter was lost**. The problem is for this kind of code: "what if what I need to transmit is not written in English, but in French?". So in, this case, you have to study this language, figure out what are the good proportions among letters and build a code for it. But that's not sufficient to

Considering this example as a base for the coding, we are now going to have a look on the information theory we need in order to study data compression algorithms models.

2.2. Shannon information theory and source coding



The theoretical background of compression [3] is provided by information theory (which is closely related to algorithmic information theory) for lossless compression and rate–distortion theory for lossy compression. These areas of study were essentially forged by Claude Shannon, who published fundamental papers on the topic in the late 1940s and early 1950s.

The research works of Shannon have revolutionized the world of communications for the half of the 20th century.

Theses research work allowed the rise of Internet. Without all these research work, it would be impossible to go on holidays with your entire library in your e-reader and all the Games of Throne episodes inside your tablet !

Shannon's theory is dedicated to the analysis of all the technical performances of these coding techniques (e.g.: the number of bits required for coding) doing a random modeling for the message to code (composed of a stream of symbols).

a. <u>Entropy</u>

"So, how can I code a message with a minimum number of bits?" Shannon started with this sentence and introduces a fundamental mathematical object: the entropy [4]. The entropy has been invented by Ludwig Boltzmann in the theory of thermodynamics. His concept has been reused by Claude Shannon in order to explain (or at least: develop!) his theory of information. Entropy of a distribution of a source is defined by this formula:

$$H(X) = -\sum_{i=1}^{n} p(x_i)\,(\log_2 p(x_i))$$

This formula means that you do the sum (for all the V symbols possible) of the frequency of occurrence $\mathrm{p}(\mathrm{x_i})$ of the symbol, times the logarithm $\log_2 p(x_i)$ of this frequency of occurrence. Then, a minus sign is placed in front of the sum because, as it is known, a probability is always between 0 and 1. And as the logarithm is negative between the interval [0; 1], you have to add a minus sign in order to respect the fact that the entropy is always a positive value.

The aim of the entropy is to quantize the incertitude on the possible symbols sequence generated by the source V. It is possible to show that the entropy verifies the following formula:

$$0 \leq H(X) \leq \log_2 N$$

(Where N is the number of symbols.)

Let's take a simple example : let X be a random variable with the following distribution and codeword assignment :

$$\mathrm{Pr(X = 1)} = \frac{1}{2}, \mathrm{codeword\ C(1) = 0}$$

$$\Pr(X = 2) = \frac{1}{4}, \text{codeword } C(2) = 10$$

$$\Pr(X = 3) = \frac{1}{8}, \text{codeword } C(3) = 110$$

$$\Pr(X = 4) = \frac{1}{8}, \text{codeword } C(4) = 111$$

and we get the entropy $H(X)$ =1.75 bits. In our case, $H(X) \leq \ log_2(N = 4 \ symbols)$ is verified.

b. *Average number of bits per symbols*

In the following part, we will consider c_v, the associated code to a symbol v. We will note $L(c_v)$ the length (number of bits) of each code word c_v. For a uniform code, the length is constant, with $L(c_v) = \log_2(N)$. It is possible to compute the number of average bits L_v of a message using the empirical formula :

$$L_v = \sum_{v=0}^{N-1} p_v \, L(c_v)$$

This formula means that you do the sum (for all the possible symbols) of the frequency occurrence p_v of a symbol times the length $L(c_v)$ of the code word c_v. L_v is the average length over all code words.

c. *Shannon's bound*

Shannon showed that the entropy can be used to bound the average number of bits (average length L_v). Actually, Shannon showed that for all coding prefix, we have : $H_v = L_v$. It is a lower bound. It means that whatever the coding, it is impossible to do better than this bound.

This result is fundamental because it describes an unbreakable limit, whatever the technique used to code. The mathematical proof is too complicated to be described here in the document. However, it is also possible to bound L_v with an upper bound. It is defined by it exist a code without loss which can be written by the formula $H_v + 1$. At the end, it is assumed that:

$$H_v \leq \ L_v \ \leq \ H_v + 1$$

The Shannon's theory allows limiting the average length, which provides a powerful information on the performance of a coding technique ! Regarding this theoretical part, it is possible to understand what is the link between Shannon's information theory and lossless data compression.

2.3. Lossless data compression

Two simple sentences can summarize what is lossless data compression: "*Data compression is lossless when there is no loss of data from the original information. Indeed, the quantity of information is the same before and after lossless compression*" [3].

In information theory field, this requirement limits the compression efficiency. It is mandatory that the entropy be the same before and after compression. This requirement limits the encoding capacity. Actually, if the average codeword length L_v is below the source's entropy, the entropy of coded message will be reduced and we will loose data, which is not the aim of lossless data compression. Concept of lossless compression is to reorganize bits in order to achieve Shannon limit.

Many lossless data compression schemes exist, and we will describe some of the most popular algorithms in part 3 of the following report.

2.4. Lossy data compression

Lossy data compression [6] is the contrary of lossless data compression. In these schemes, some loss of information is acceptable. Actually, human perceptions are focused on some information. So dropping nonessential detail from the data source can save storage space. In the case of Lena for example, we are more interested in some shapes and curves of her body than the hat she's wearing.

It means that lossy data compression schemes are designed based on how people "sense" the data in question. For example, the human eye is more sensitive to little variations in luminance than it is to the variations in color (chrominance). JPEG image compression works in part by rounding off nonessential bits of information. There is a corresponding trade-off between preserving information and reducing size. A number of popular compression formats exploit these perceptual differences, including those used in music files, images, and video.

We will illustrate lossy data compression schemes with JPEG in the following part 3 in this report.

3. Algorithms used for data compression

For this part, we limit our research on gray level pictures, but we can easily extrapolate our results on RGB picture. We illustrate all compression schemes with the famous picture lena256, which size is 256 by 256 with 256 level of gray. So each pixel is coded on 8 bits.

3.1. Lossless data compression

a. Huffman code

Huffman code is a particular type of optimal prefix code that is used for lossless data compression. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a pixel in a picture). The algorithm creates his table from the estimated probability or frequency of occurrence (in our case, apparition frequency of a pixel) for each possible value of the source symbol.

The figure 1 below shows us the histogram of gray level in the picture **lena256** :

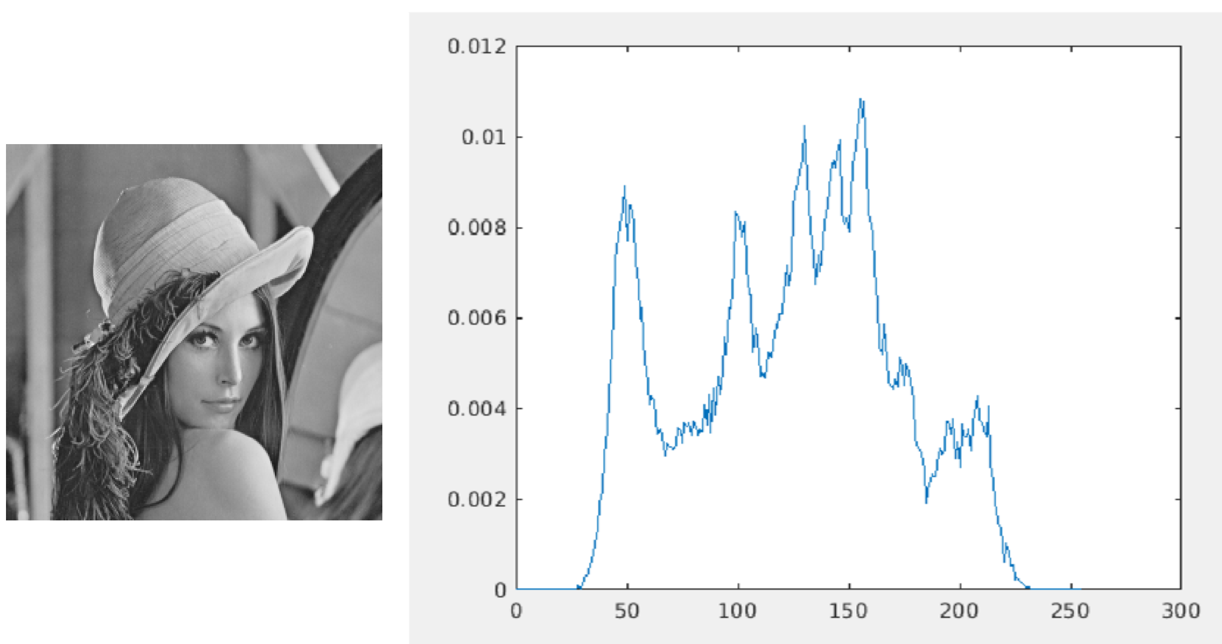


Figure 1 : *Histogram of a 256-gray level picture: lena256*

The aim of Huffman algorithm is to attribute smaller prefix code for level of gray which has higher apparition frequency. If we zoom in, pixels with the value '155' have the largest frequency of apparition, so they will have the smallest code which has a length of 6 bits. In comparison the pixel with the value '29' appears only one time in the picture, so it will be coded with the largest code of 16-bit length. An extract of the Huffman dictionary can be seen below in figure 2

38	39	40	41	42	43	44	45
[1 0 0 1 1 1 1 1 0]	[0 1 1 0 0 0 1 0 0]	[1 1 1 0 1 0 0 1]	[1 0 1 0 1 0 1 1]	[0 1 0 0 0 1 0 1]	[0 0 0 0 0 0 1 1]	[1 1 0 1 1 0 1]	[1 0 0 1 1 0 1]

Figure 2 : *Each column indicates the gray level and the corresponding prefix code*

To construct the Huffman table, we use a tree. First we put the symbol and their relative probability. Then we connect together the smallest symbols and reiterate this operation. We obtain the following tree.

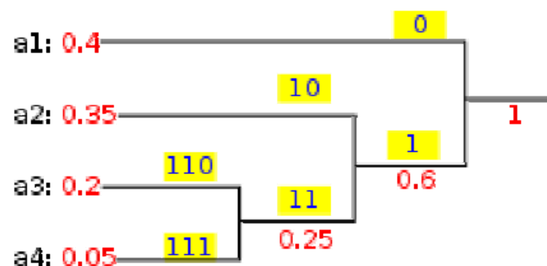


Figure 3 : *ak is the symbol with its probability in red*

Then we construct the prefix code by attributing '0' for upper branch and '1' for lower branch. In this example, the symbol a1 will be represented by '0', a2 by '10', a3 by '110' and a4 by '111'. When the table is constructed, we can calculate the average codeword L_v length in binary element per symbol, where Ω is the Huffman code set, $\mathrm{L}(\mathrm{c_v})$ the length of the code and $\mathrm{p_v}$ its probability.

$$L_v = \sum_{x \in \Omega} \mathrm{p_v}\,\mathrm{L}(\mathrm{c_v})$$

The L value indicates the number of binary element per symbol. In our case, we want that this value is less than 8 binary element per symbol (original value of the sequence) and is as close as possible the entropy of the source. The efficiency of a code is computed by the following expression.

$$\epsilon = \frac{H(X)}{L}$$

Encoding process is easy: read the source sequence and convert each symbol with the corresponding code. The encoding sequence length in binary element will be directly $\mathrm{L} * \mathrm{numberOfSymbol}$ in our case because we work with observed symbol frequency.

For the decoding, you must have the corresponding table (the dictionary) or the statistical distribution of symbol. Then the algorithm constructs the Huffman tree and the sequence is placed in input of this tree.

Results of Huffman coding :

Sequence length before coding	256 * 256 * 8 = 524288 binary element
Source entropy H(X)	7,4223 b.e. / symbol
Average length of a codeword L	7,4458 b.e. / symbol
Sequence length after coding	487966 binary element
Data compression ratio	6,9 %
Number of different pixel	0 pixel

From now on, it is possible to state on Huffman coding.

- Firstly : the encoder need to read twice the picture : one time to establish the dictionary, a second time to encode input sequence.
- Secondly : the decoder must know the statistical distribution of symbol which will be transmitted, or the Huffman tree. So the compression gain is lost with the obligation to send the tree. An adaptive version of Huffman coding exists : the encoder and the decoder begin with the same tree, and at each new symbol coded and decoded the tree is updated on each side.
- Finally, Huffman code is close to the limit of Shannon, but we can use a better code to be closer : the arithmetic coding.

Another way to obtain a better compression rate is to take into account that each pixel is correlated with its neighbors. So it is possible to code only the difference between two pixels. This method change the distribution of the symbol, as it is possible to see below on figure 4 :

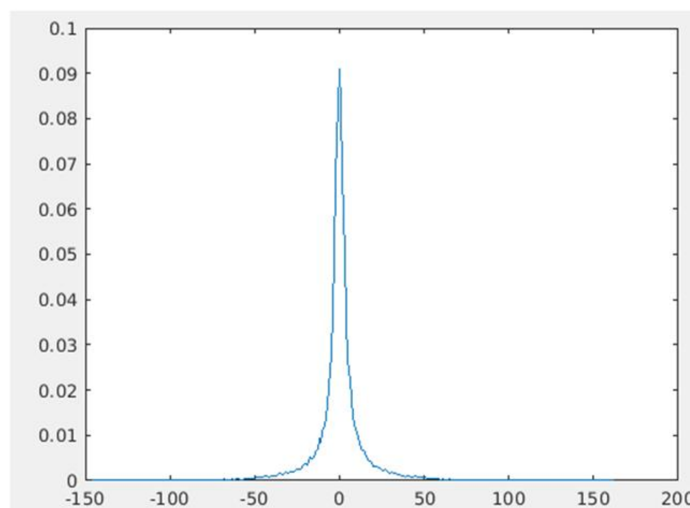


Figure 4 : *Frequency (in Y-axis) corresponding to each symbol (X-axis)*

So the Huffman coding efficiency will increase. Results are reported in the following table :

Sequence length before coding	256 * 256 * 8 = 524288 binary element
Source entropy H(X)	5,3676 b.e. / symbol
Average length of a codeword L	5,4025 b.e. / symbol
Sequence length after coding	354064 binary element
Data compression ratio	33 %
Number of different pixel	0 pixel

Exploitation of some properties of the source can increase the compression efficiency. The Huffman coding efficiency has reached **33%** (remind : the efficiency was equal to 7% without correlation consideration).

b. Arithmetic code

Arithmetic coding [7] differs from Huffman coding in that : rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, an arbitrary-precision fraction q where 0.0 ≤ q < 1.0. It represents the current information as a range, defined by two numbers.

Arithmetic coding must know the probability of each symbol. To show the encoding process, we will code the sentence “ACFD”, with the table of probability below:

A	B	C	D	E	F
0,4	**0,2**	**0,15**	**0,15**	**0,05**	**0,05**

First, the algorithm decompose a range between 0 and 1 according to the probability (first line in the figure 5 below) :

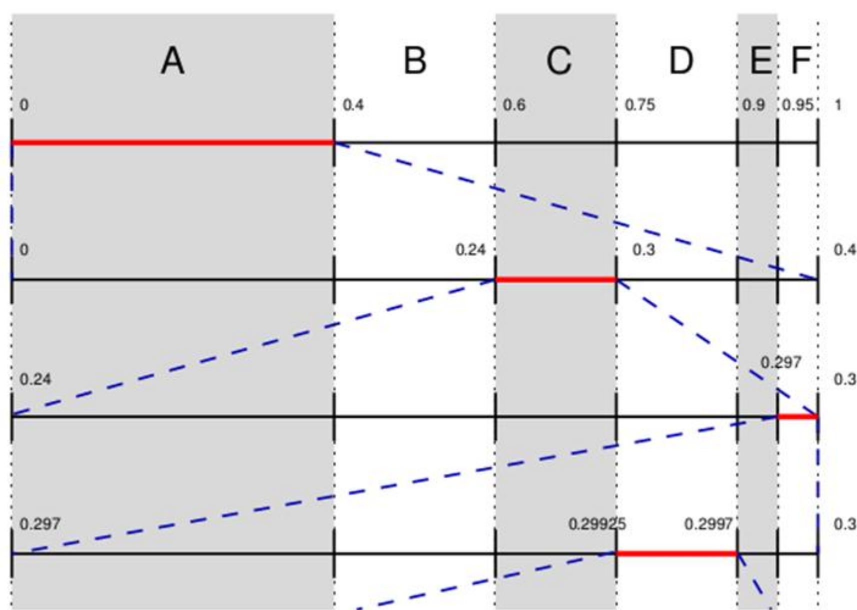


***Figure 5** : decompose of a range between 0 and 1 according to the probability*

Then we choose the range corresponding to the symbol to code. This new range is also subdivided according to the probability of each symbol. The process continues until the final symbol, which gives the final range: [0.29925; 0.2997]. **Any number into this range is the compressed version of the sentence “ACFD”**. This number is coded in bit to obtain the encoded sequence.

As Huffman coding, it exists an adaptive arithmetic coding in order to suppress the first pass. However using floating number can introduce error and difficulties in IT & Electronic fields.

c. LZ & variants

Lempel-Ziv data compression (abbreviate LZ) is theoretically **dictionary coder**. All dictionary data compression are lossless, they do not introduce loose of information. It exists a lot of compression algorithms based on LZ : LZ77, LZ78, LZW … But all of them used the same idea. This type of data

compression is widely used in the IT field, for example GIF and ZIP format implement a variation of LZ.

This category of data compression will create the dictionary and the encoded output sequence at the same time. But the most important feature is that the dictionary is not transmitted to the receiver. Indeed decoder will reconstruct the same dictionary during decoding process.

We will present the LZW algorithm in detail because it is used in GIF format. The coding process is explained below.

First LZW algorithm begins with a known dictionary which contains all symbols. In this example there are two symbols : 0 and 1

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Table *Output*
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1

We test the first symbol, which is inevitably present in the dictionary

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Present 0
Table *Output*
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1

So we increase the window. Now the sentence 00 is not in the dictionary.

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
New 0 0
Table *Output*
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1

The algorithm add this new sentence into the dictionary

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Add 0 0
Table *Output*
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1 0
0

Then it will remove the last symbol of the window and code it. Here it will code the symbol 0 by the code 0

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Output 0 0
Table *Output*
0 1 2 3 4 5 6 7 8 9 A B C D E *0*
0 1 0
0

Then it increases the start index of the window by the previous number of coded symbol (here: 1 symbol). We repeat previous step : the sentence 0 is present into dictionary

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Present 0
Table *Output*
0 1 2 3 4 5 6 7 8 9 A B C D E *0*
0 1 0
0

Algorithm increases the window and test if sentence 00 is present into dictionary

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Present 0 0
Table Output
0 1 2 3 4 5 6 7 8 9 A B C D E 0
0 1 0
0

Now the sentence 00 is present, so it will increase the window. But now the sentence 001 is not into dictionary

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
New 0 0 1
Table Output
0 1 2 3 4 5 6 7 8 9 A B C D E 0
0 1 0
0

So it add this new sentence in dictionary.

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Add 0 0 1
Table Output
0 1 2 3 4 5 6 7 8 9 A B C D E 0
0 1 0 0
0 0
1

Then it remove the last symbol of the window and code it. So the algorithms code 00 by the code 2.

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Output 0 0 X
Table Output
0 1 2 3 4 5 6 7 8 9 A B C D E 0 2
0 1 0 0
0 0
1

Next the window start index increases by 2 (because we have coded 2 symbol previously), and we restart the process.

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Present 1
Table Output
0 1 2 3 4 5 6 7 8 9 A B C D E 0 2
0 1 0 0
0 0
1

The sentence 10 is not in dictionary

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
New 1 0
Table Output
0 1 2 3 4 5 6 7 8 9 A B C D E 0 2
0 1 0 0
0 0
1

So the algorithm add it in.

Input
0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1
Add 1 0
Table Output
0 1 2 3 4 5 6 7 8 9 A B C D E 0 2
0 1 0 0 1
0 0 0
1

Then it codes the sentence 1 because we remove to the window the last symbol.

Input

0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1

Output 1 0

Table															Output
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	0 2 1
0	1	0	0	1											
		0	0	0											
			1												

Moreover, dictionary algorithm change the distribution of each symbol in a sequence. So after a dictionary compression, there is often a entropy compression (Huffman, arithmetic, …) on the output sequence.

For decompression, only encoded sequence is needed. The decoder will reconstruct step by step the dictionary and the original sequence.

First the decoder begins with the known dictionary: all possible symbols are in the dictionary. Here there are 0 and 1

Table: 0 1 2 3 4 5 6 7 8 9 A B C D E
0 1

Input: 0 2 1 2 2 0

New

Output

Now decoder starts to read the first code which is 0 here. So the decoder knows that the code 0 correspond to the symbol 0, but because of the remove of the last symbol in the sliding window during encoding process, he cannot determine next symbol yet. This symbol is written by a variable x.

Table: 0 1 2 3 4 5 6 7 8 9 A B C D E
0 1

Input: **0** 2 1 2 2 0

New: 2 0x

Output: 0x

Thanks to the encoding process, the decoder also knows that 0x is included in the dictionary. So it adds 0x into.

Table: 0 1 2 3 4 5 6 7 8 9 A B C D E
0 1 0
x

Input: **0** 2 1 2 2 0

New: 2 0x

Output: 0x

Then decoder reads next code, which is 2. It looks to its dictionary and finds that the code 2 correspond to the symbol 0x. As the previous step, because of encoding process, the output is 0xy. But now, we can determine that x = 0. So we update x with is real value.

Table: 0 1 2 3 4 5 6 7 8 9 A B C D E
0 1 0
x

Input: 0 **2** 1 2 2 0

New: 3 0xy

Output: 0x
00xy

The decoder adds the sentence 00y into the dictionary. Variable y will be determined at the next step

Table
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1 0 0
0 0
y

Input
0 2 1 2 2 0

New
3 00y

Output
000y

Then decoder reads next code, which is 2. It find the corresponding symbol 1. It represents the unknown next symbol by the variable z. Now it is possible to determine the value of y which is 1.

Table
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1 0 0
0 0
y

Input
0 2 1 2 2 0

New
4 1z

Output
000y
0001z

The sentence 1z is added into the dictionary

Table
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1 0 0 1
0 0 z
1

Input
0 2 1 2 2 0

New
4 1z

Output
0001z

The decoder takes next code which is 2. This code corresponds to the sentence 00. The unknown symbol is represented by variable w. Now it is possible to determine the value of z.

Table
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1 0 0 1
0 0 z
1

Input
0 2 1 2 2 0

New
5 00w

Output
0001z
000100w

Then the sentence 00w is added into dictionary

Table
0 1 2 3 4 5 6 7 8 9 A B C D E
0 1 0 0 1 0
0 0 0 0
1 w

Input
0 2 1 2 2 0

New
5 00w

Output
000100w

Results of LZW coding :

Sequence length before coding	256 * 256 * 8 = 524288 binary element
Sequence length after coding	520800 binary element
Data compression ratio	0,66 %
Number of different pixel	0 pixel

From now on, it is possible to assess about LZ coding :

- Firstly the implementation must be adapted, according to which type of data you encode. For first tests and results we have taken a basic implementation of LZW, that's why the data compression ratio is not so good.
- Secondly there is no loose of gain in the data compression during the transmission, because the dictionary is not sent, but the encoder and decoder must have the same implementation of the LZ algorithm.
- Finally, the LZ coding reaches asymptotically entropy limi**t. So: the longer the input sequence is, the more efficient is the coding code**.

3.2. Lossy data compression : JPEG

We have seen previously that lossy data compression can be achieved using specific techniques which are based on human senses. In this part, we will introduce a basic concept: image lossy data compression. Let's start with one of the famous: the JPEG (Joint Photographic Experts Group) format.

JPEG is a commonly used method of lossy compression for digital images, particularly for those images produced by digital photography. The degree of compression can be adjusted, allowing a selectable trade-off between storage size and image quality. JPEG typically achieves 10:1 compression with little perceptible loss in image quality.

We focus our research on the compression way of the standard JPEG. Indeed, it exists a lot of standard about JPEG. The figure 6 below represents the encoding and decoding process used for JPEG.

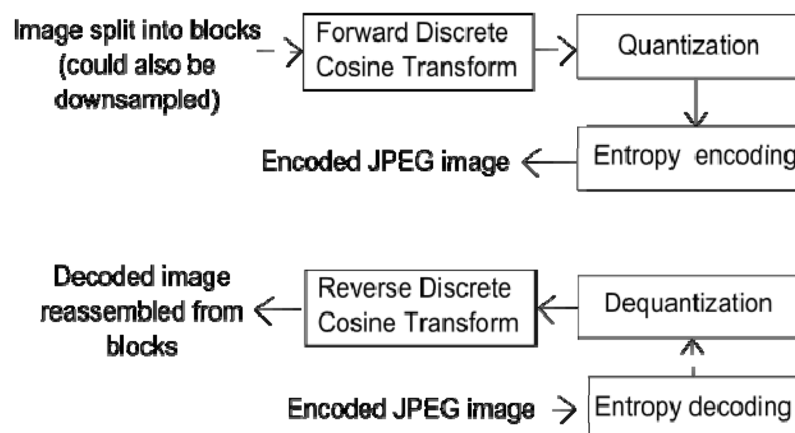


Figure 6 *: JPEG compression and decompression process*

First, the image should be converted from RGB into a different color space called Y'CbCr : the Y' component represents the brightness of a pixel, and the Cb and Cr components represent the chrominance.

Then it is possible to downsample (e.g : losing information). Due to the densities of color- and brightness-sensitive receptors in the human eye, humans are more sensitive to the brightness of an

image (the Y' component) than to the chrominance (the Cb and Cr components). Please not that we are not concerned by this step because we use gray-scale pictures.

Next step is to split the picture into small matrix of 8 by 8 and compute a discrete cosine transform (DCT) on each block in order to go in the frequency domain (see figure 7).

Figure 7 : *DCT on Lena*

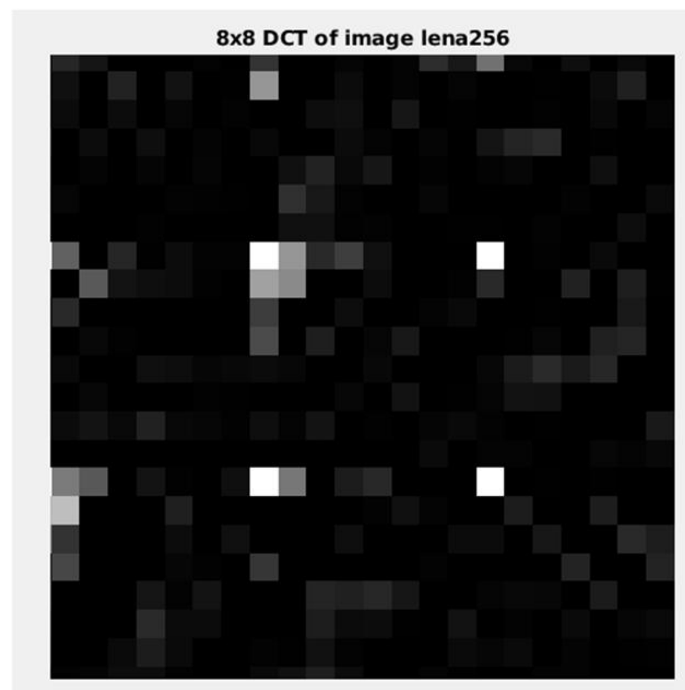


Figure 8 : *Zoom on 8-by-8 matrix. The most relevant information is at the top-left of each matrix*

The main process in JPEG compression is the quantization. From this step we are losing some information. This block rounds the frequency matrix, in order to remove high frequency information. So the 8-by-8 matrix will contain information only in its upper-left part : many zero elements are

introduced in the bottom-right. It is possible to control this round by the quantization matrix. This parameter will define final quality of the picture.

And last but not least, Huffman coding exploits this high number of zero to drastically reduce the final size.

To decompress the picture, the decoder has to know which quantization matrix have been used. It rebuilds the DCT matrix, which is different compare to the original. Then it computes the inverse DCT to construct the picture.

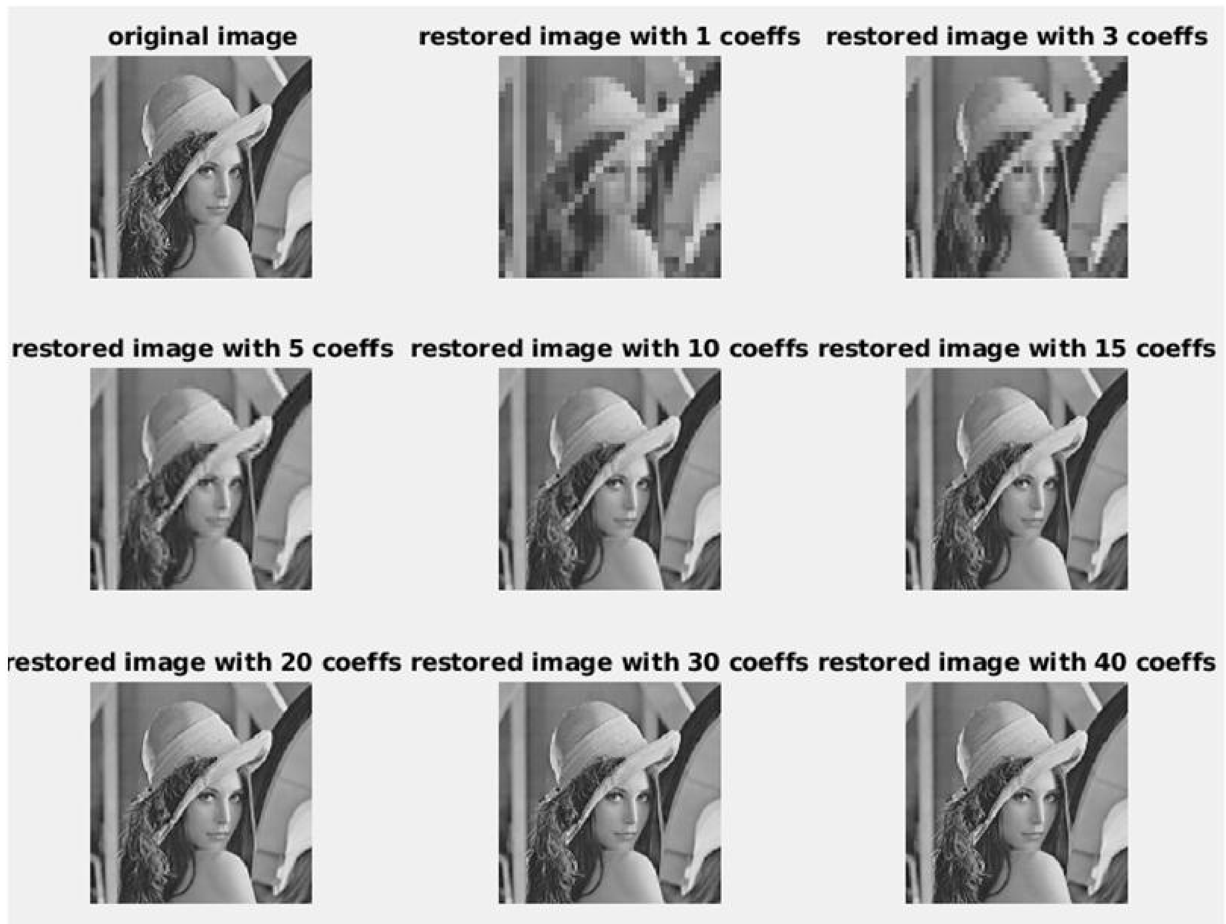


Figure 9 *: JPEG compression with variation of number of coefficient in the DCT matrix*

In order to conclude on the JPEG format, the main idea of the JPEG is to perform in the frequency domain and remove the high frequencies. Using this deception, the distribution of each symbol is changed. Indeed, Huffman coding is more efficient with a non-homogeneous distribution of symbols.

4. Technical implementation

4.1. Using LabVIEW

For the second part of our project, we will focus on the technical implementation for the data compression. According to our tutor Tarik, the main goal is to create a digital transmission/reception communication chain in order to test the data compression algorithms (both lossless and lossy) and analyze the performances and the impact of the channel.

In order to design the communication chain, we will use the G language (LabVIEW). Dedicated to instrumentation at the very beginning, LabVIEW has evolved to a new level: the LabVIEW Communication Design Suite, dedicated to radio communications and signal processing.

From a personal point of view, working with LabVIEW is a … nightmare[1]. This software has been designed for SDR (Software Defined Radio). However, using LabVIEW instead of powerful tool like MATLAB is like heresy. During this second part of our project, we have faced a huge number a problems starting with the complex ability of LabVIEW to transmit 65536 values with a resolution of 8 bits (which corresponds to an image of 256 times 256 for each pixel having a resolution on 8 bits). Working with these values conducted to a crash of our platform with a loss of our project.

So, we changed our computer and used the National Instrument Personal Computer of the SATCOM lab regarding the recommendations of our tutor Tarik. However, even if equipped with 8 Go of RAM and Gen 5 CPU 2.8 GHz (and Win 7 as operating system), some little changes occurred allowing to test and obtain results.

Using entirely MATLAB would probably have conducted us to be more efficient. Using the communication chain from the project 3 (DCOM) and adding the process for data compression (lossless and lossy), should probably permits to go further in this project. We strongly recommend for the next year project to have LabVIEW courses before starting the Project 1.

4.2. LabVIEW Design for data compression

Implement the data compression on LabVIEW is a hard challenge. At the very first beginning, our idea was to use LabVIEW with MATLAB just for reading ".m" files, meaning: using LabVIEW for the communication chain (modulation and emitting with NI USRP and reverse process for the communication chain). However, using MATLAB code in LabVIEW is pure rip-off. Indeed, using LabVIEW block containing MATLAB code is limited to basic functions of MATLAB and not the ones you can create under MATLAB. Nice tried.

That's why we jumped to another solution. We split the job to do into parts:

- First: implement a full communication chain under LabVIEW,

[1] LabVIEW requires an eagle's eye view!

- Second: transform the result of data compressed file under MATLAB into a binary file (readable by LabVIEW) and then inject it in the input signal to transmit (without noise in the channel) using compression lossless,
- Third: same as part 2 but with an AWGN channel simulated by Labview,
- Fourth: same as part 3 but with lossy compression (instead of lossless),
- Fifth: same as part 4 but with the use of two USRP (for a more realistic channel).

4.3. Lossless data compression : Huffman code

The complicated part (creating Huffman tree in LabVIEW) has been reduced dramatically due to the use of a binary file provided to LabVIEW. Following this idea, it is not mandatory to provide to LabVIEW something related to the compression part. The compression part is done by MATLAB and the transmission is done by LabVIEW. The decompression is also done by MATLAB. The results are observed on both sides: LabVIEW for the BER for example, and MALTAB for the figures.

The MATLAB code is provided in appendix meanwhile the LabVIEW code is provided separately on the upload platform SourceForge.

So, we obtain as results this image, with E_b/N_0 = 7 dB in BPSK:

Figure 10 *: Huffman compression with Lena 256*

Regarding to the results above, we can directly conclude that the value of E_b/N_0 has a powerful impact on the quality. Indeed Huffman compression scheme creates a code word which corresponds to a pixel. If a code word bit is altered, this code word will become two new code words. So one original pixel generates two pixels and the image is shifted. Moreover if the first (or final) bit of a code word is altered, this code word will be merged with is neighbor, so the two originals pixels generate only one pixel.

4.4. Lossy data compression : JPEG

For the lossy compression, we will implement the JPEG encoding. Using the previous part of this report, we implemented an algorithm with normalization, the DCT, the quantization. Normally, a Huffman coding or an RLE are added in the JPEG. For our project, we decided at the very first step to not implement the Huffman coding after the zigzag process in the JPEG lossy compression.

Still using MATLAB and LabVIEW softwares, we implemented compression/decompression process under MATLAB and transmission process under LabVIEW such as lossless part. The MATLAB codes are provided in appendix. Please note that the quantized matrix is not defined randomly. It is also possible to define your own quantized matrix. However, for all the following tests, we considered the quantized matrix defined for the JPEG format:

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

JPEG can tune the quality of the picture with a quality factor q_JPEG. This factor plays an important role in the compression rate. The quality factor generates a coefficient α with is multiplied by the quantization matrix.

$$\alpha = \begin{cases} \dfrac{50}{q_JPEG} & 1 \leq q_JPEG \leq 50 \\ 2 - \dfrac{2q_JPEG}{100} & 51 \leq q_JPEG \leq 99 \end{cases}$$

As you can see below on figure 11, for E_b/N_0 = 7 dB and a BPSK modulation, we have:

Figure 11 *: JPEG compression with Lena 256*

We can see impairments in the image, due to altered bits. However the image is not shifted because we don't introduce RLE compression yet. So it is possible to know exactly the beginning (start) and the end of each 8-by-8 block. However, the image is not compressed, that's why we add the RLE scheme after the quantization process. By introducing RLE (so in practice the End Of Block (EOB) symbol) the end of each block is not fixed; and if the EOB is altered, we will obtain a shift like in lossless coding scheme.

4.5. Optimization algorithms for lossless compression

Shifted images are not acceptable. So, we decided to introduce some processing tricks to retrieve the original image.

First thing we can do is to introduce a new symbol: EOC (End Of Codeword). This symbol is introduced after each pixel and takes the value 256. This, results in increasing the size of the image twice. However Huffman coding scheme will attribute a very small codeword in terms of length. By performing the new image with Huffman scheme, we can retrieve the end of each codeword and avoid the shift in the picture.

Finally, we can generalize this algorithm by adding pilot pixels in the image. For example we can introduce a symbol each 8 pixel block, where it is possible to tune the length of the block, to find a trade-off between increasing the size of the image without shifting the image.

4.6. Optimization algorithms for lossy compression

During the tests on data compression and data decompression, we had several discussions in order to reduce dramatically the errors occurring during the transmission process through the noisy channel.

Multiple solutions can be founded (we were even talking about magic!) to decrease the numbers of errors and increasing the compression factor. The idea here for the lossy compression is to examine carefully on which part can be optimized for the coding and the decoding. Regarding the schematics on page 18 (figure 6), it is possible to play on: the rate compression factor, the size of transmitted information, the quantized matrix, the EOB location for example.

So, we created different algorithms in order to optimize the lossy compression.

a. Algorithm 1 : Fixed EOB

This one is dedicated to define an EOB position (End Of Block). The EOB is a used to inform that all the remaining values are "0" in the 8*8 matrix. Here, we decided to keep only the 36 first values of the matrix. This corresponds to the upper-left triangle of the 8-by-8 matrix. By doing this we remove coefficients which are not so relevant, and increase the compression rate. We obtain a rate

compression of 43.75% (meaning that an image with size = 64 bits before compression will size 36 bits after compression).

b. Algorithm 1+ : Zigzag reading with defined EOB location for each block

This algorithm can be presented as a "generalization of the algorithm 1". In this optimized algorithm, the idea is to define a specific location of the EOB symbol which will be the same for all the 8-by-8 block. The position of EOB is known from both sides (compress and decompress). This powerful idea permits to avoid a shift of blocks because there is no EOB symbol transmitted.

Moreover, the position of the EOB tunes the quality of the picture, and we decide to bound the coefficient at 16, otherwise the quality will be clearly degraded. So the compression factor oscillates between 16, the lowest quality, and 64, the best quality (and no compression). The compression factor is independent of the quality factor with this algorithm, so setting the quality factor to the highest value is the best choice.

Caution! If this number is too small, we will lose information and degrade the image at the receiving!

c. Algorithm 2 : RLE DCT block

This one is focusing also on the EOB. Contrary to algorithm 1, the position of the EOB will change for each block. A zigzag read is performed on the 8-by-8 matrix (following the rules of zigzag seen in part 3 of this report), and the EOB symbol replaces all the zero final values.

As we have seen in previous parts, values presents in the 8-by-8 matrix are bounded between -128 and 127 (so 8-bit coded). It is not necessary to code values with 9 bits in order to be able to introduce a new symbol to represent EOB. We decided to code the EOB symbol with the greatest values (all of the 9 bits with the value '1').

For the decompression process, we may be confident with the EOB. Indeed if we failed to detect EOB, we will obtain a shift in the picture. So we need to consider a threshold to detect the EOB. The threshold is computed in order to have the less probability of alteration. This threshold has to be enough high if one of pixel is altered, because it can become an EOB (due to the introduction of noise).

We can also introduce some processing tricks: check if the EOB is followed by high values, or introduce one '0' before the EOB.

If the noise is too high, it won't be possible to recover all EOB symbol, and some pixels will become EOB symbol. So the usage of algo2 is restricted to a relative high E_b/N_0.

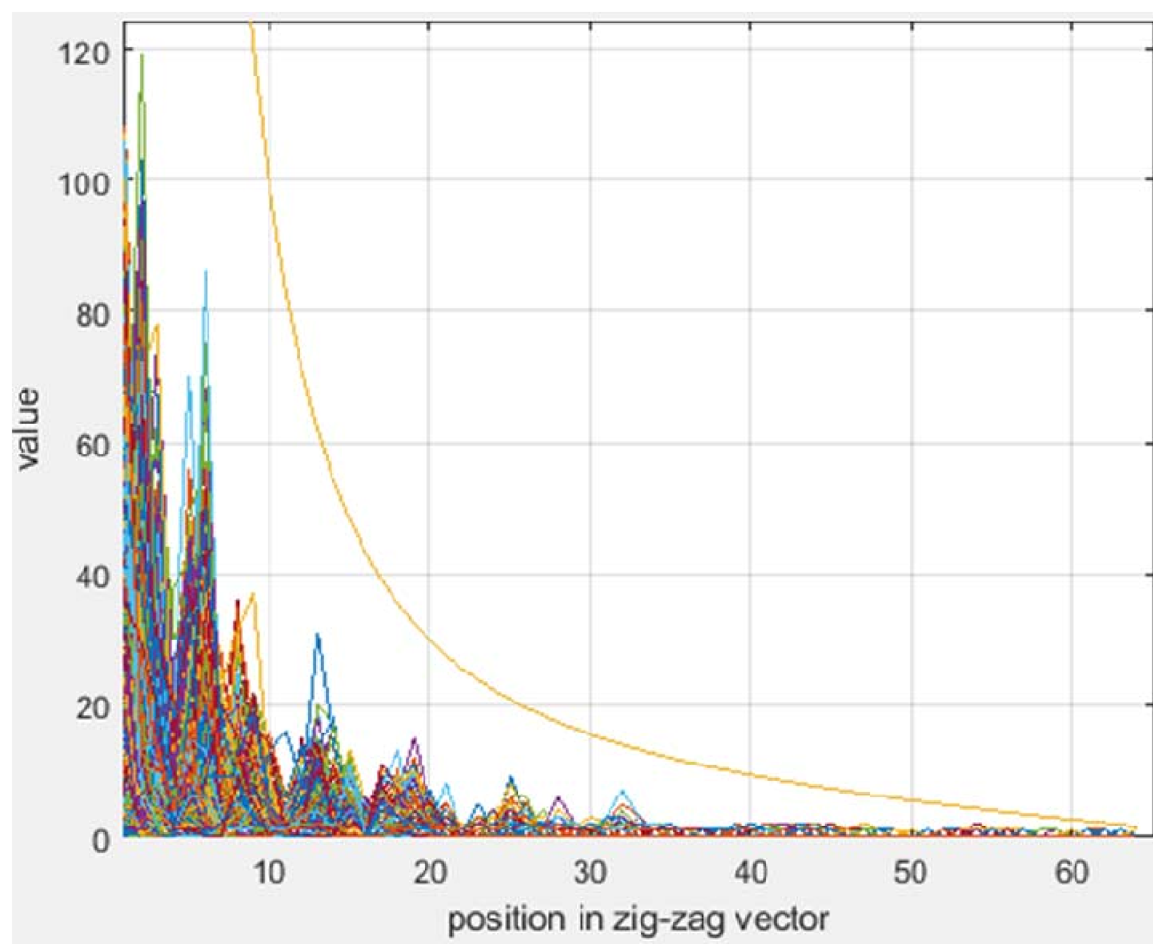
d. Algorithm 3 : Errors detection and correction capacity

This algorithm is dedicated to the detection and correction capacity. Here, the idea is to characterize as a decreasing mathematical function the error detection ability and adjust levels of threshold in

order to correct values that are supposed wrong. For the algorithm 3, the zigzag algorithm is required to read the values in the matrix quantized.

Please note that algorithm 3 alone is not useful for compression. It must be added to algorithm 1 or 2.

In order to explain this algorithm we need to represent the decreasing characteristic of values created by DCT. The graph below shows us the spatial dispersion of the coefficient.



So on the decompression side, we can detect if some values are above the orange curve. In this case, it means that the most significant bit (MSB) has been altered (in practice we are not sure but there is a high probability that it is true). So we can correct this value by transform the first '1' value into a '0'. By doing this we reduce the apparition of error block and their impact.

4.7. Tests campaign

Please note that this campaign test took a lot of time to be performed on the computer we used. Moreover, the reader may notice that we have a high number of errors introduced with the AGWN that can be clearly seen on the picture. Don't forget this is also due to the small size of the picture taken.

a. ***Lossless data compression : Huffman***

Using our best photographed model for the data compression (Lena), we proceeded to a step by step campaign for the lossless compression with AWGN:

- Lossless compression coding and decoding,
- Size of Lena : 256 * 256 (= 65536 matrix values) with 8 bits of resolution for each values,
- Modulation : BPSK,
- E_b/N_0 = 7 db and 10 dB.

The results are provided in appendix II.

b. ***Lossy data compression : JPEG like***

Again, we proceeded as the same method in order to make comparison between modulations and coding using our best model Lena:

- Lossy compression coding and decoding,
- Size of Lena : 256 * 256 (= 65536 matrix values) with 8 bits of resolution for each values,
- Modulation : BPSK,
- E_b/N_0 = 7 db and 10 dB
- Quality factor = 80

The results are provided in appendix III.

c. ***Lossy data compression : real JPEG with additional stuff***

Here, the idea is to test our algorithms dedicated to optimize the coding and decoding process. As usual, we proceeded as the same method in order to make comparison between modulations and coding using our best model Lena:

- Lossy compression coding and decoding,
- Size of Lena : 256 * 256 (= 65536 matrix values) with 8 bits of resolution for each values,
- Modulation : BPSK,
- E_b/N_0 = 7 db,
- Quality factor = 80,
- Algorithm 1, 2, 3.

The results are provided in appendix IV.

d. ***Analyze of results, performances & complexity***

We have seen so far that it is possible to compress data, in a lossless way or lossy way. We have seen also that without a transmission channel (supposed perfect), we can decompress files (pictures in our

case) and obtain zero errors whatever the E_b/N_0 is. The first tests (which are not included in this report) were dedicated to test our technical implementation of coding and decoding (both compression) using MATLAB and LabVIEW.

The real challenge was in the addition of the AWGN channel in the transmission process. For the lossless, we use the “normal” design for Huffman code for lossless compression, and normal JPEG for the lossy compression.

Regarding the complexity of the system, the lossy implementation combined with algo1 is the simplest to implement (implementation and number of operations). This combination provides a good compression rate and a good quality, even if the BER of the channel is high. The received picture is not shifted and it is possible to perform algo3 is order to remove extravagant values and improve global quality.

Note: in order to analyze the performances, we computed the rate compression, which is a good indicator of the compression regarding the quality factor. Reminder: in order to compute the rate compression, we use the formula T = (1 – (Final volume/Initial volume)) *100. For the campaign test realized, we obtained these values:

Huffman	6.93%
Algo1 (with compression factor of 36)	43.75%
Algo2 (with quality factor of 80)	54.35%
Algo1+ (with compression factor of 20)	68.75%
Algo2 (with quality factor of 50)	71%

4.8. Going further

Due to the time we had for our project, we didn’t manage to proceed with part 5, meaning the use of two USRP for a more realistic transmission channel. However, this part could be implemented by others students next year using our project as a basis of work.

Moreover, it would have been also possible to design a specific quantized matrix. Some researchers have explained how to define the optimal quantization matrix from psycho visual threshold [4]. We strongly recommend our successors in this project to review and analyze this document. Indeed the quantization matrix can be optimized. Of course, this quantization matrix won’t the one for the JPEG compression but something different. This may be an opportunity

Another point to develop in order to go further is the idea to reuse the job done by others students group this year in order to add more tests and opportunities to develop specific chain of transmission such as integrate an LDPC, use of optical transmission instead of radio frequency links (in case of the BER of optical transmission is equal to 0).

CONCLUSION

It exist multiple data compression algorithms, each one having his particularities and especially a type of target in each case. All of the data cannot be compressed using the same way (e.g. same algorithm). A compression data algorithm of text will be based on a recurrence of the number of characters or pieces of sentences. Another algorithm used for data image compression will be based on the difference between two pixels co-located.

If we would discuss and make a choice between all of data compression algorithm and the best rate for compression, this would be useless. Actually, there is no best compression algorithm used to provide the best compression rate. It all depends on the source you have in input of your compression block. It all depends on the structure of your data you want to compress.

However, all of these algorithms have a common ground between them: their objective is to recover the initial data (partially or integrally).

During the second part of our project, we implemented under LabVIEW and MATLAB a full communication process using compression algorithms. We saw that the use of AWGN channel introduces a lot of errors in your original image. To correct these errors, multiple solutions can be implemented using several algorithms.

Thanks to our tests campaign, we have concluded that it is possible to use both (lossless and lossy) compression when you want to transmit something. However, the use of lossless or lossy is conditioned most of the time by your E_b/N_0:

- small value of Eb/N0 : : it is recommended to use lossy compression with fix EOB position.
- high Eb/N0: it is recommended to use lossless data compression.

Moreover, the use of a robust modulation (such as BPSK) is mandatory if you want to limit the errors added by the channel.

This project was really exciting from a technical and human point of view. We had to face multiple difficulties but it was worth challenging. We hope that our project will be useful for others students and further applications. Our job is free of use, the only requirement we are asking is to be added in bibliography of future projects.

Appendixes I : MATLAB and LabVIEW source codes

MATCLAB and LabVIEW codes are available on SourceForge.

Appendix II : campaign test lossless compression

Parameters :
BPSK // E_b/N_0 = 7 dB
Lena = 256 * 256
# of errors : 459
BER = 9 *10^{-4}

Parameters :
BPSK // E_b/N_0 = 10 dB
Lena = 256 * 256
# of errors : 3
BER = 6 * 10^{-6}

Appendix III : campaign test lossy compression

Parameters :
BPSK // E_b/N_0 = 7 dB
Lena = 256 * 256
Quality factor =80
# of errors : 7
BER =9 *10^{-4}

Parameters :
BPSK // E_b/N_0 = 10 dB
Lena = 256 * 256
Quality factor =80
# of errors : 1
BER =6 * 10^{-6}

Appendix IV : campaign test lossy compression optimized

Parameters :
Algo1
BPSK//E_b/N_0=7 dB
Lena = 256 * 256
Quality factor =80
# of errors : 265
BER =8 * 10^{-4}

Parameters :
Algo2
BPSK//E_b/N_0=7 dB
Lena = 256 * 256
Quality factor =80
# of errors : 222
BER =9* 10^{-4}

Parameters :
Algo3
BPSK//E_b/N_0=7 dB
Lena = 256 * 256
Quality factor =80
# of errors : 466 !
BER =8 * 10^{-4}

(If you can find the 466 errors, you're definitely a boss !)

Bibliography

[1] Girino, «Data Compression: A little introduction for beginners,» [En ligne]. Available: http://blog.girino.org/tutoriais/data-compression-a-little-introduction-for-beginers/.

[2] M. Cagnazzo, «Principes du codage sans perte, Codage d'Huffman, Lempel-Ziv, arithmétique,» 2013.

[3] WebService'Est, «Mieux comprendre les deux principales familles de compression,» 25 February 2015. [En ligne]. Available: http://www.webservice-est.fr/partie-2-mieux-comprendre-les-deux-principales-familles-de-compression/.

[4] Wikipedia, «Huffman coding,» 6 11 2017. [En ligne]. Available: https://en.wikipedia.org/wiki/Huffman_coding.

[5] S. H. N. Ferda Ernawan, «The Optimal Quantization Matrices for JPEG Image Compression From Psychovisual Threshold,» 2014.

[6] FAQ, «Comp.compression Frequently Asked Questions (part 1/3),» 05 09 1999. [En ligne]. Available: http://www.faqs.org/faqs/compression-faq/part1/index.html.

[7] G. Peyré, «Claude Shannon et la compression des données,» 2016.

[8] P.-H. Wang, «Data Compression,» 2012.

[9] J. Fors, «Information Theory Lecture 3: Data Compression,» Linköping University, 2013.

[10] Wikipedia, «Entropy (information theory),» 19 11 2017. [En ligne]. Available: https://en.wikipedia.org/wiki/Entropy_(information_theory).

[11] D. Delaunay, «Compression et entropie,» [En ligne]. Available: http://mp.cpgedupuydelome.fr/document.php?doc=Compression%20et%20entropie.txt.

[12] Y. Ollivier, «La théorie de l'information : l'origine de l'entropie,» [En ligne]. Available: http://www.yann-ollivier.org/entropie/entropie1.

[13] D. M. N. Joseph, «Principes généraux de codage entropique d'une source».

[14] ENS, «Compression de données (compléments)».

[15] P. Pansu, «Entropie,» 2012.

[16] M. G. Urban, «Voyager Image Data Compression and Block Encoding,» California Institute of Technology, 1987.

[17] Wikipedia, «Codage arithmétique,» 27 09 2017. [En ligne]. Available: https://fr.wikipedia.org/wiki/Codage_arithm%C3%A9tique.

[18] Wikipedia, «Codage entropique,» 21 05 2017. [En ligne]. Available: https://fr.wikipedia.org/wiki/Codage_entropique.

[19] Wikipedia, «Data compression,» 08 03 2018. [En ligne]. Available: https://en.wikipedia.org/wiki/Data_compression.

[20] Shaaban, «Data Compression Basics,» 2000.

[21] G. Dallas, «Data Compression: What it is and how it works,» 04 08 2013. [En ligne]. Available: https://georgemdallas.wordpress.com/2013/08/14/data-compression-what-it-is-and-how-it-works/.

[22] L. Wehenkell, «Introduction to information theory and coding».

[23] Divers, «Introduction à la théorie de l'information,» [En ligne]. Available: http://www.bibmath.net/crypto/index.php?action=affiche&quoi=complements/entropie.