UNIVERSITAT POLITÈCNICA DE CATALUNYA

Lossless data compression of radio astronomy data

by

David Pascual Arnáiz

in the

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

Departament de Física

Advisor: Enrique García-Berro Montilla

Co-advisor: Jordi Portell i Mora

October 2016

Abstract

The goal of this thesis is to develop a pre-processing and post-processing algorithm that prepares radio astronomy data to be compressed by the FAPEC data compressor. To achieve this, an adequate analysis is required on representative datasets obtained from the ALMA and SKA projects. The developed algorithm must be as simple as possible, since FAPEC is meant to be fast. The commitment between performance and speed will be key to achieve our objective. The implementation into FAPEC requires a deep knowledge of the compressor. The results from compression tests on both ALMA and SKA data with FAPEC are finally compared to other compressors, such as GZIP and BZIP2, to evaluate its performance.

Resumen

El objetivo de este proyecto es desarrollar un algoritmo de pre-procesado y post-procesado para datos de radio astronomía para ser comprimidos por el compresor de datos FAPEC. Para ello se deberán analizar en profundidad los conjuntos de datos obtenidos de los proyectos ALMA y SKA. El algoritmo desarrollado deberá ser simple, puesto que FAPEC es un compresor diseñado para ser rápido. Mantener el compromiso entre rendimiento y velocidad es un factor clave para cumplir nuestro objetivo. Deberemos conocer en profundidad el funcionamiento de FAPEC para poder realizar la implementación lo más eficiente posible. Para finalizar, compararemos los resultados obtenidos de la compresión de los conjuntos de datos de ALMA y SKA con los de otros compresores, como GZIP y BZIP2, para poder valorar el rendimiento de nuestra solución.

Acknowledgments

El final de una larga etapa de mi vida llega a su fin. Dejo atrás grandes recuerdos pero miro hacia adelante con mucha ilusión. Muchas horas de trabajo que por fin dan sus frutos. Muchas experiencias y vivencias que jamás olvidaré.

El camino ha sido largo y jamás lo anduve solo. En primer lugar quiero agradecer a mis tutores Enrique García-Berro y Jordi Portell, que confiaron en mi desde el primer día y me dieron esta gran oportunidad. Agradecer sobretodo su infinita paciencia y todos los conocimientos que han compartido conmigo que me han hecho crecer intelectualmente. No quisiera olvidarme de Marcial Clotet, el cual siempre aportó su punto de vista en las reuniones para ayudarme a llegar a donde he llegado.

No puedo olvidar también el apoyo de mi familia. Mi madre, que siempre que ha podido me ha quitado trabajo para que pudiera centrarme en estudiar. Mi hermana Marina, que siempre ha estado allí para echar una mano cuando más la necesitaba. Y mi padre, que me ha ayudado en aquellos momentos de incertidumbre transmitiéndome tranquilidad. A todos, os quiero.

Tampoco me olvido de mis grandes amigos Josep María Vilà y Xavi Linares, quienes me han servido de gran apoyo en esta etapa de mi vida. Ya fuera ayudándome a relajarme o aportando desde su experiencia. Sois muy grandes.

David

Contents

List of Figures

List of Tables

List of Equations

Acronyms

A

ALMA Atacama Large Millimeter Array

AVI Audio Video Interleaved

C

CRC Cyclic Redundancy Check

D

DS Double-Smoothed

E

ESO European Organization for Astronomical Research in the Southern Hemisphere

I

IBM International Business Machines

IEEE Institute of Electrical and Electronics Engineers

F

FAPEC Fully Adaptive Prediction Error Coder

J

JAO Joint Alma Observatory

JPEG Joint Photographic Experts Group

L

LC Large Coding

LE Low Entropy

M

MeerKAT Karoo Array Telescope

MP3 MPEG Layer-3

MPEG Motion Picture Experts Group

N

NaN Not a Number

NRAO National Radio Astronomy Observatory

NAOJ National Astronomical Observatory of Japan

P

PDF Probability Density Function

PEC Prediction Error Coder

PNG Portable Network Graphics

S

SKA Square Kilometer Array

V

VLC Variable Length Codes

Chapter 1

1 Introduction

1.1 Lossless compression and radio astronomy data

In the last few years, the constant development and evolution of technology has largely improved our day-by-day life and helped scientists to gain a better understanding of what the universe holds. Many data from space is continuously being collected, stored and processed. Discovering new stars, planets and galaxies and analyzing them requires a large amount of observing time and a state-of-the-art technology. In order to be able to manage all this information, it is highly recommendable to compress it so that it uses less storage and bandwidth.

There are many types of data compression systems. The strategy we choose to compress data depends on the information we want to process. The more we know about this information, the better the results we can achieve when compressing it by adapting ourselves to its exact needs. In addition, since human senses are imperfect, it is sometimes not necessary to recover all the information. This is what happens with images, audio and video, where suppressing parts of the information will probably not be noticed, while saving even more storage volume. This is what we call

lossy compression. On the other hand, lossless compression requires that the original data is perfectly reconstructed from the compressed data. Lossless compression is key in many cases. Suppressing information from the data collected by telescopes and spatial probes to save storage could make us miss relevant information. If we consider the precise information we can collect from hundreds of lightyears away, it is relevant to keep everything for a deeper analysis.

Deep analysis of information requires computational resources and time. Millions of terabytes generated every day, if compressed, will save vast amounts of storage, and in the end, money. Other restrictions might be applicable, since the information generated on board a probe around Jupiter will not have the same computational needs as the one generated on ground in the desert of Atacama. Any tiny extra improvement in compression ratio can result into many terabytes saved, but also the time needed to achieve that cannot be neglected. It is crucial to choose our algorithm properly according to the situation, looking for the appropriate balance between ratio and speed.

FIGURE. 1.1: The Very Large Array (VLA) in Socorro, New Mexico

There are several types of data generated, depending on which spectrum our instruments are working on and the type of sensor. Depending on the frequency, we can classify it into optical, radio, infrared, ultraviolet, x-ray and gamma. Every spectrum will need a different treatment of its data, but on this work we will focus on radio astronomy data. It is often generated using a floating-point format, thus generating huge data volumes which may not always contain relevant information. Many times we may just have random noise. Lossless compression of data that is mostly random results in low compression ratios. Finding patterns on the data to save storage is crucial for compression schemes to work properly, which is not an easy task.

Here we will focus mostly on radioastronomical data. Fig. 1.1 shows an example of a radiotelescope. In this work we have selected FAPEC as the data compression core to use. FAPEC is an algorithm based on compressing the sample prediction errors. The more similar (or better predictable) the samples are, the better compression ratio we will achieve. However, as stated, noisy data will significantly limit the achievable ratios, so we will need to analyze and understand the information to treat it adequately.

1.2 Objectives and Motivation

Since FAPEC performs best when prediction errors are as close to zero as possible, we must develop an algorithm able to pre-processes the radio astronomy data, bringing the values to compress as close to zero as possible. When this process is done, FAPEC will compress it giving a good compression ratio in a low time. This process must also be reversible, since any compression would be useless without its decompression, so the post-processing process will also need to be designed.

Besides ratios, we must also consider speed. FAPEC is designed to be very fast specially on compression. It is meant to compete with other compressors for space missions, so our pre-processing will have to be as simple as possible so it does not significantly delay the whole compression process.

In order to develop this algorithm, real data from astronomical facilities, such as ALMA and SKA will be used. This will be explained in detail in Chapter 2. These

samples will be closely examined and analyzed to be able to achieve the best compression ratio possible whilst keeping the algorithm simple and quick.

1.3 Structure and plan of this work

This report is organized as follows. First, in Chapter 2 the selected datasets to develop and test this algorithm are presented. We also discuss where such data comes from and how it was generated. In Chapter 3, the format of the data and the standards defined are introduced. Next, in chapter 4, two commercial compressors and FAPEC are presented. Chapter 5 presents the analysis of the data and the strategy to develop the algorithm. Chapter 6 describes the implementation, tests and results obtained with our developed algorithm. Finally, in Chapter 7 our findings are summarized, our conclusions presented and we elaborate on forthcoming work.

Chapter 2

2 Radio astronomy data

In this chapter, the origin of the datasets used in this project to develop an algorithm for compressing floating-point binary files is presented. The data used to test the developed algorithm is real data. This data has very different origins and nature since it comes from different telescopes. Therefore, it allows us to cover a broader scope of the kind of binary data that can be generated and, therefore, eventually get better ratios for any kind of binary files to be compressed.

2.1 ALMA

ALMA stands for Atacama Large Millimeter Array. It is an array of radio telescopes operated under the partnership of several organizations [1]. These are the NRAO in North America, ESO in Europe and the NAOJ in Asia. Together they form the JAO (Joint Alma Observatory) that manages the construction, commissioning and operation of the radiotelescope. Some parts of it are still under construction in the desert of Atacama (Chile). It is located at 5,000 meters of altitude and will start with 66 high-precision antennas working together exploring the millimeter and submillimeter wavelengths, which belong to the highest part of the radio spectrum. The spectrum analyzed may be extended in the future.

The power of ALMA will allow astronomers to probe stars, galaxies and even exo-planets. This new precise information will allow scientists an in-depth research of the universe in a way that was not possible before.

ALMA will use interferometry to combine the signals from two or more antennae. Combining those signals, images can be constructed. These images are equivalent to the ones that could be obtained with a giant telescope of 14,000 meters of diameter. The interferometry needs all the antennae and devices to work in perfect synchrony. The precision required is of one millionth of a millionth of a second. The antennae must also follow the same path in the sky with the precision of hundredths of a millimeter. The atmospheric conditions must be known on every moment to be able to quantify the attenuation and perturbation suffered by the signal. When all this process is achieved, the collected data is digitized and transmitted to a central building. A supercomputer combines and analyzes them with high precision. A diagram of all this process can be found in Fig. 2.1.

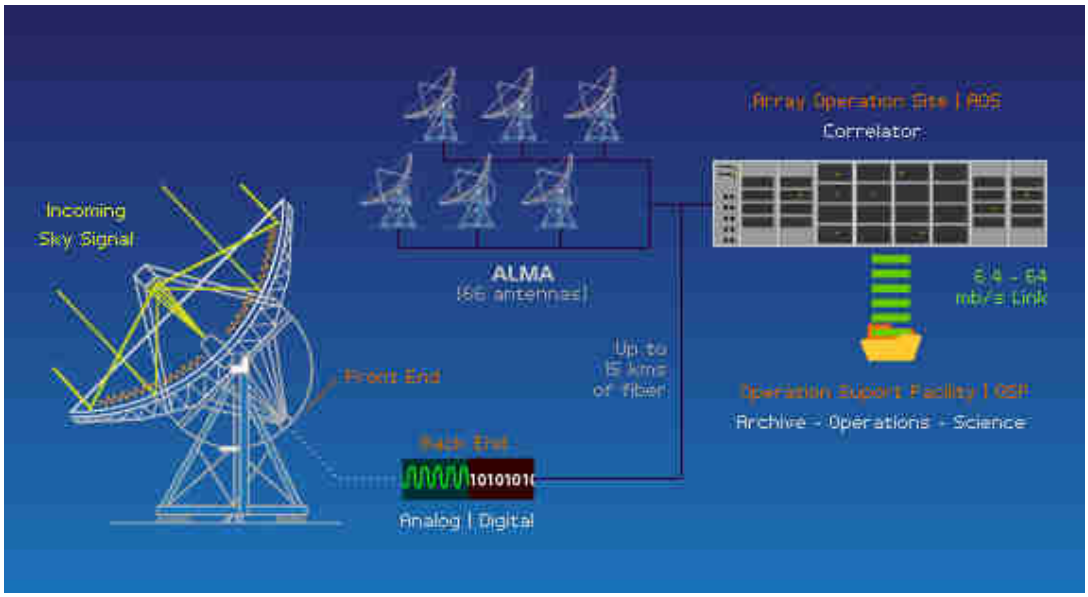

FIGURE 2.1: Diagram of the usage of the antennae of ALMA

To handle all the data generated by ALMA, powerful computing systems are needed. The first step of the processing is to convert the signal to a lower frequency, between 2 and 4 GHz. Then, the signal is digitized and sent to the main building. In the main building a correlator combines the signals and generates useful astronomical data for

its analysis. It multiplies the signals coming from different antennae and saves the data, forming a 2D image of the region of the sky observed.

The ALMA project is already generating single-precision data in little endian that is used in this project to test the algorithm developed. In Chapter 3 we explain what does this type of data imply and how it is structured.

2.2 SKA

SKA stands for Square Kilometer Array and it is an international collective work to build the largest radio telescope in the world [2]. When all the antennae will be built, it will have over a square kilometer of collecting area. This telescope will be operational on 2020 and it will allow astronomers to analyze millions of distant galaxies. This new and detailed information will help us on our understanding of our universe.

FIGURE. 2.2: MeerKAT antennae in South Africa

When considering the scale of this project, it is worth mentioning that thousands of dishes and up to a million antennae will be built. The image resolution of SKA will

be 50 times better than the one we can achieve with the Hubble Space Telescope. The SKA project will complement the range of other large telescopes in other spectrums that will be launched in the forthcoming decades. This will set a large amount of tools to explore the universe in a new way that has never been seen. This pharaonic project is supported by over 100 organizations in 20 different countries which are designing and developing the SKA project.

SKA will be located both in South Africa and Australia. The regions have been specifically chosen due to their atmospheric conditions and the minimal radio interferences. The core of the project will be built in the Karoo desert in South Africa and it will contain the high frequency antennae. The low frequency antennae will be located at the Australian Murchison Shire.

The data used to test our algorithm comes from the South African branch of the SKA project. Before the project is operational, several pathfinders have been designed and built. In our case, the MeerKAT telescope is already generating data. The MeerKAT uses seven dishes which have already produced the first images. An image of MeerKAT is shown in Fig. 2.2. More information about the project can be found in Ref. [3].

The SKA telescopes will generate the same amount of data in a day as the entire planet does in a year. It is expected to have more data flowing inside the network of SKA than in the entire internet in 2020. This means that we will need high-end supercomputers that shall be faster than any supercomputer ever built now. The calculations show that an exaflop-capable supercomputer will be required to handle all this data. For this goal, data centers are being constructed around the world. The main storage center will be located in Perth, Australia. The information will be split and handled in other centers located in North America, Asia and Europe. Those supercomputers will need to handle calculations of 200 teraflops and will require to store over 1.5 petabytes of information. It is estimated that an Exabyte of raw data will be generated every day.

Some companies such as IBM, Cisco or Intel have interest on providing their equipment as well as knowledge for this project. Being able to get this contract would

allow them to work with the researchers in the next four years to develop the designs of the SKA systems.

This raw data needs to be compressed as much as possible since any small gain will save a large storage amount. The data used in this project for testing our algorithm comes from the MeerKAT and it is coded as single-precision floats (32 bits) in little endian. However, this data will also be converted to double precision binaries (64 bits) for testing the double-precision algorithm since we do not own 64 bits data of any other project.

Chapter 3

3 Floating-point data types

In this chapter we present the different types of floating-point data standards that are used for coding binary files. This is very important to understand how we developed our algorithm. The most used standards are explained here. There are manufacturers who use other definitions, and other standards are also being developed, but those are beyond the scope of this project.

3.1 Floating Point Data Types

Floating-point numbers are used for variables with a wide range of values, so that the value is represented by its sign, its mantissa, and its exponent. All of these components have assigned a fixed number of digits.

In the 1970's, William Kahan of the University of California at Berkeley defined a floating-point arithmetic standard. He joined a group of scientists whose purpose was to produce the best possible definition of floating point arithmetic. Nowadays, all manufacturers follow the representation of IEEE 754 [4] they designed.

The IEEE 754 contains single, extended single, double and extended double precision. It also explains how to manage the different mathematical operations (add,

subtract, multiply, divide, remainder and square root) as well as rounded format conversion (round down, round up, round toward zero and round to the nearest). It also considers five exception types (invalid operation, division by zero, overflow, underflow and inexact), which get signaled by a status flag.

3.1.1 IEEE single precision

Single precision is based on 32-bit words. It uses 1 bit for the sign (s), 8 bits for the biased exponent (e) and the remaining 23 bits form the mantissa (m).

Since we have 8 bits for the exponent, values from 0 to 255 can be coded. The range that is defined for the exponent is [-126, 127]. This means that coding 0 in this standard makes e = -127, and coding 255 makes e = 127. Both for coding and decoding this bias must be considered: adding and subtracting 127 will be needed to get our initial exponent back.

$$1.2345 = \underbrace{12345}_{\text{mantissa}} \times \underbrace{10^{\overbrace{-4}^{\text{exponent}}}}_{\text{base}}$$

EQUATION 3.1: Floating Point Example

The following cases can be found:

1. e = 255 and m ≠ 0 gives an x which is not a number (NaN).
2. e = 255 and m = 0 gives infinity with its sign.
3. e = 0 and m ≠ 0, gradual underflow, sub-normal numbers.
4. e = 0 and m = 0, zero with its sign.
5. 1 ≤ e ≤ 254 is the normal case.

The largest number to be represented is $(2 - 2^{-23}) \times 2^{127} \approx 3{,}4028 \times 10^{38}$ and the smallest positive number is $1 \times 2^{-126} \approx 1{,}1755 \times 10^{-38}$. The NaN can be used to represent zero divided by zero, infinity minus infinity and other quantities with an unknown value.

Extended precision is also available to allow evaluating subexpressions to full single precision. The details depend on the implementation, but the minimum numbers for the mantissa have to be at least 31 and 10 for the exponent.

3.1.2 IEEE double precision

Double precision is based on 64-bit words. It uses 1 bit for the sign (s), 11 bits for the biased exponent (e) and the remaining 52 bits form the mantissa (m). The following cases can be found:

1. $e = 2047$ and $m \neq 0$ gives an x which is not a number (NaN).
2. $e = 2047$ and $m = 0$ gives infinity with its sign.
3. $e = 0$ and $m \neq 0$, gradual underflow, sub-normal numbers.
4. $e = 0$ and $m = 0$, zero with its sign.
5. $1 \leq e \leq 2046$ is the normal case.

The largest number to be represented is $(2 - 2^{-52}) \times 2^{1023} \approx 1{,}7977 \times 10^{308}$ and the smallest positive number is $1 \times 2^{-1022} \approx 2{,}2251 \times 10^{-308}$.

Extended precision is also available to allow evaluating subexpressions to full single precision. The details depend on the implementation, but the minimum numbers for the mantissa have to be at least 63 and 15 for the exponent.

3.1.3 Future standards

Double precision is not always sufficient, so quadruple precision is available from many manufacturers. The official standard has not been defined yet, but a generalization of the IEEE 754 is used by most manufacturers. On the other hand, some use another convention named SGI quadruple precision, which is very different from the usual quadruple precision and has a very large range.

3.2 Endianness

The standard described above does not define how the bytes are read and written. This is an important issue and it is called endianness. The endianness defines that bytes can be read and written from left to the right or from right to left. It is worth emphasizing that it is done byte by byte instead of bit by bit.

The two ways of reading and writing the bytes are called big endian and little endian. Big endian indicates that the bytes are read from left to the right. This is the natural, intuitive order for human beings and was mostly used by Motorola. Contrary to what would seem logical, the most used one is that called little endian. Little endian is when the bytes are read from right to left. This system is the most common one and, amongst other manufacturers, used by Intel. Endianness will therefore be very important to correctly read and write the floating-point data.

Chapter 4

4 Data compression Algorithms

4.1 Introduction

Radio Astronomy has evolved a lot during recent years. Technological improvements in both precision and instrumentation has allowed us to generate a large volume of data at very high rates. The handling and storage of this information is costly. Reducing the amount of information to be handled is crucial and will be the focus of this project. In theory, removing the redundant information makes it possible to compress information without any loss. This is called lossless compression. Losing some information that may not be perceived by human senses is also a possibility. This is used mostly for images (JPEG, PNG), audio (MP3) or video (MPEG, AVI) and it is named lossy compression. In this work we will only work with lossless

compression since we want to keep all of our radio astronomical data with its original precision.

4.2 GZIP

GZIP [5] is a compressor based on the DEFLATE [6] algorithm, which is a combination of the LZ77 [7] and Huffman coding [8]. DEFLATE uses dynamic Huffman encoding, which creates an optimized Huffman tree for each block of data. The compression is achieved by the matching and replacement of duplicate strings with pointers and the replacement of symbols with new, weighted symbols based on their frequency of use. It is also able to detect repeated strings and create a back-reference with the location of the duplicated string.

The format of GZIP uses a 10-byte header which contains a magic number (which is a constant numerical used to identify the file format), a version number and a timestamp. There can be optional extra headers which may contain information such as the original file name. After the headers comes the body, which contains a DEFLATE-compressed payload. In the end, it has an 8-byte footer which contains a CRC-32 checksum and the length of the original uncompressed data. GZIP allows multiple files to be concatenated but is normally used to compress single files. The files to compress are normally assembled into collections of files creating a single tar archive and then compressing that with GZIP.

4.3 BZIP2

BZIP2 [9] is a free and open-source file compression program that uses the Burrows-Wheeler algorithm. This algorithm rearranges a character string into runs of similar characters. This leads into a good compression, since repeated characters are easy to compress by techniques such as move-to-front transform and run-length encoding. This transformation is reversible without needing any storage of additional data. It is a method of improving the efficiency of text compression algorithms at the cost of some extra computation. Thus, BZIP2 compresses files more effectively than GZIP in general, but it is considerably slower. It compresses data in blocks of size between 100 and 900 KB using the Burrows-Wheeler transform, then it applies move-to-front

transform and finally Huffman coding. It used to use arithmetic coding instead of Huffman, but it was changed due to a software patent restriction.

Its performance is asymmetric, as decompression is relatively fast. A multi-thread version was created on 2003, named PBZIP2 but has not been incorporated into the main project yet. BZIP2 only compresses single files opposite to GZIP, which could assemble several files and compress them. The tar archive tool can also be used to combine and compress several files.

4.4 FAPEC and PEC

FAPEC is the algorithm selected for our compression. We will build pre-processing and post-processing stages around it, so it is important to understand how it works to fully use its coding capability. FAPEC [10,11] stands for Fully Adaptive Prediction Error Coder and it is an upgraded (adaptive) version of PEC. PEC is an entropy coding algorithm, based on compressing the prediction error of samples.

4.4.1 PEC

Some solutions used for compressing lossless data come from the Golomb Coding [12], which is a method designed by Solomon W. Golomb in the 1960s. Variants from this method, like Rice coding [13], provide impressive results but only if the data that will be compressed follows a geometrical distribution. This may not always be the case, so the idea behind PEC has a wider scope.

PEC is focused on the compression of prediction errors. It has a pre-processing stage based on a data predictor plus a differentiator. The strategy behind its development was to build a very fast and robust compression algorithm. It is a partially adaptive entropy coder based on a segmentation strategy.

PEC is composed of three coding options: Low Entropy (LE), Double-Smoothed (DS) and Large Coding (LC) which are variable-length codes (VLC). The three coding options share the same principles: the range of the data to be coded is split into four small segments. Each segment is determined by its corresponding coding

parameter, which indicates the number of bits required to code the values of that segment. For each value coded, the adequate segment and number of bits is chosen. PEC follows the assumption that most values to be coded are close to zero and when this happens the coding parameters must be chosen in a way that the first segments are significantly smaller than the original symbol size, while the last segments are slightly larger. This leads to a compressed output and the ratio will be determined by the probability density function (PDF) of the data combined with the selected coding table.

PEC is flexible enough to adapt to data distributions with probability peaks far from zero, which is great for dealing with outliers. For PEC to work as optimally as possible it requires a pre-processing stage that brings the values to code close to zero, so the compression ratio increases.

Since the final user does not need to know how PEC works, a calibrator has been developed, which determines the best configuration for each case. This calibrator tests each of the possible PEC configurations based on the histogram of the values and selects the one that offers the highest compression ratio. This process is done in less than 1 second, which is affordable in most cases but not in space missions, where the calibrator must be run on-ground with simulated data before launch.

PEC can be considered as a partially adaptive algorithm since the code size is selected for each one of the values. This is an advantage over the Rice coder, which uses a fixed parameter for all the values. It also overcomes Rice coder by limiting the maximum code length to twice the symbol size in the worst cases. On the other side, PEC must be trained for each case in order to get the best compression ratios and to fix this issue, its fully adaptive version, FAPEC, was developed.

4.4.2 FAPEC

FAPEC is the Fully Adaptive Prediction Error Coder. It adds an adaptive layer to PEC in order to configure its coding table and coding option according to the statistics of each data block. This will give a nearly-optimal compression ratio without any preliminary configuration of PEC and without any information on the statistics of the data to be compressed.

The block length is configurable and not restricted to a power of two, with a typical value of 128 samples. FAPEC trades a slight decrease in the optimal PEC configuration to be as fast as possible and thus a slight decrease in the compression ratio. FAPEC accumulates the values to be coded and calculates the histogram on-the-fly. The histogram will look logarithmic so higher sample values are grouped and mapped to fewer bins, and values close to zero are mapped to independent bins. This reduces the memory and time required to analyze the histogram. Off this histogram, the best coding option (LE, DS or LC) is selected. A default threshold configuration has been fixed in the algorithm, which gives excellent ratios for almost any dataset with a decreasing trend in its PDF such as geometric, bigamma or Gaussian. This threshold could be modified if necessary to adapt to other statistical distributions if it was really needed. This feature is not available in other compressing algorithms. When the coding option and parameters have been determined, a small header is added to each block so it can be decompressed afterwards.

4.4.3 Pre-processing options available

As stated previously, PEC and hence FAPEC yield the best results when values are as close to zero as possible. Thus, a pre-processing stage is required to help with this. Some pre-processing options are already available for FAPEC:

Interleaving, a frequent option used in digital communication and storage systems to improve the performance of forward error correcting codes. It shuffles the source symbols across several code words providing a more uniform distribution of errors. This technique though degrades compression ratios if it is not determined properly, since it will randomize the samples. This pre-processing option can undo the interleaving for treating the file up to 32775 samples. This interleaving will be re-done in a post-processing stage equivalent to this one.

Filters can also be used. This stage does a simple differential and averaging process to the samples straight as they come, so PEC compresses a prediction error of what the next sample might be, depending on the order of the filter we choose. Finally, lossy compression can be used as well. This pre-processing removes a number of least significant bits that we choose. This is useful for images and audio.

The data to be compressed used on this work are floats, which makes none of the options available viable. The lossy option is completely discarded to treat the data, since our compression must be lossless. Interleaving is an option which would help in case our data was coded in that way, but there would not be a further treatment to bring values close to zero. Filters could be very useful, but it is not possible to use it in combination with another filter for now. These filters are an interesting option, so a similar idea could be used for the floating-point data. Thus, a new pre-processing option will be developed. This pre-processing stage will be focused on converting the float samples into values close to zero, which can be decompressed on a post-processing stage. This will optimize the FAPEC entropy coding.

Chapter 5

5 Strategy and algorithm

In this chapter, we describe the techniques used to analyze the datasets that support the development of our compression algorithm. It is important to emphasize that the objective of the pre-compression is to bring values as close to zero as possible so FAPEC can get the best possible ratios. It also requires that the algorithm is simple, since FAPEC was created as a quick algorithm so it can be run in environments with low computational power. The last requirement is, as stated, lossless compression and decompression.

In the analysis of both SKA and ALMA data, we applied the same techniques in order to find out their structure. However, each case has its particularities, so we analyze them in detail hereafter, first we study the case of ALMA, and then the data from SKA.

5.1 ALMA analysis

The first step needed to start the analysis consists on reading and plotting the ALMA data to visualize the signal with the aim of identifying patterns, peculiarities and periodicities of the samples by means of simple visual inspection.

The input file is a binary file following the standard IEEE 754 single precision in little endian. Therefore, we have to read the signal in groups of 32 bits and separate the sign, exponent and mantissa. The interpretation of this information allows us to plot the time sequence. Since no commercial program designed to interpret the binary file exists, the development of our own code was required. This program, named *read_binary.c*, can read binary files from both single and double precision and for both little and big endian. It separates the sign, exponent and mantissa, and writes them in decimal format. Once the data is separated, the time sequence of ALMA can be plotted (Fig. 5.1). However, it is impossible to visualize the entire file. Therefore, a sampling of 1,000 items was performed to be able to plot the signal and visualize its general appearance. This procedure is the only mean available to get an idea of how the data looks like. Consequently, further data plots will imply some sub-sampling too. Nevertheless, it must be taken into account that this sampling is not representative of the whole data file. Regarding the plotting, it has been done in logarithmic scale for a better understanding of the signal.

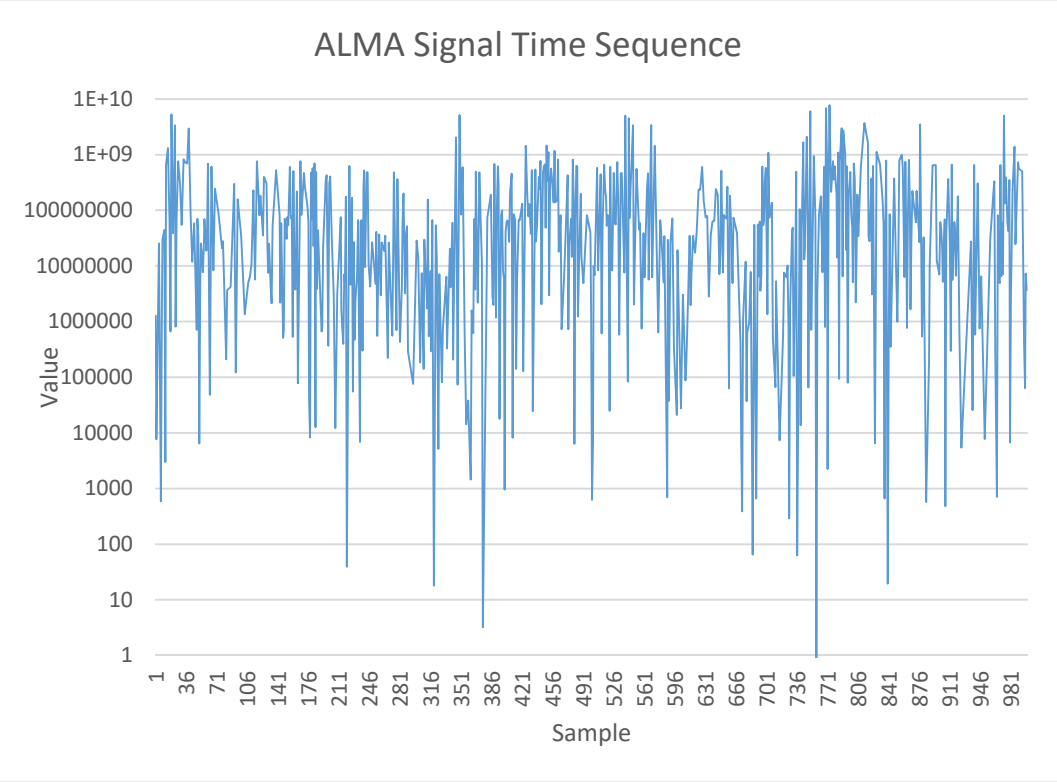

FIGURE 5.1: ALMA signal time sequence (logarithmic scale)

In Fig. 5.1 we can see that the values change suddenly from order of 10^5 to 10^9. The environment looks thus very noisy, and it is hard to appreciate any specific pattern. The joint analysis of the graph and the signal itself states that the exponent of the signal is quite stable. It does change a lot if we look at the values themselves, but the exponent does not have such big swings, in terms of numbers. The first conclusion is that we might be able to get a good ratio with the compression of the exponent by means of a simple differential, since this exponent does not vary excessively for consecutive samples. By applying the simple differential, many values should shrink to zero or to values close to zero. A deeper analysis is required to obtain clearer conclusions.

5.1.1 Sign

Focusing on the sign, it is important to state that on a logarithmic representation negative values cannot be plotted. As far as the signal is concerned, it can be seen that it is strongly random. The noise goes from positive to negative values. Performing a deeper analysis, by taking 100,000 samples it can be seen that 58,516 are zeroes and 41,484 are ones. Although there are significantly more zeroes than ones it will not be easy to gain much compression from the sign since it is just 1 bit. It can be combined with other bits to be able to gain some extra compression, but the sign does not seem to follow any pattern that allows us to gain a lot of compression.

5.1.2 Exponent

To gain a deeper insight into the distribution of exponents has been done (Fig. 5.2).

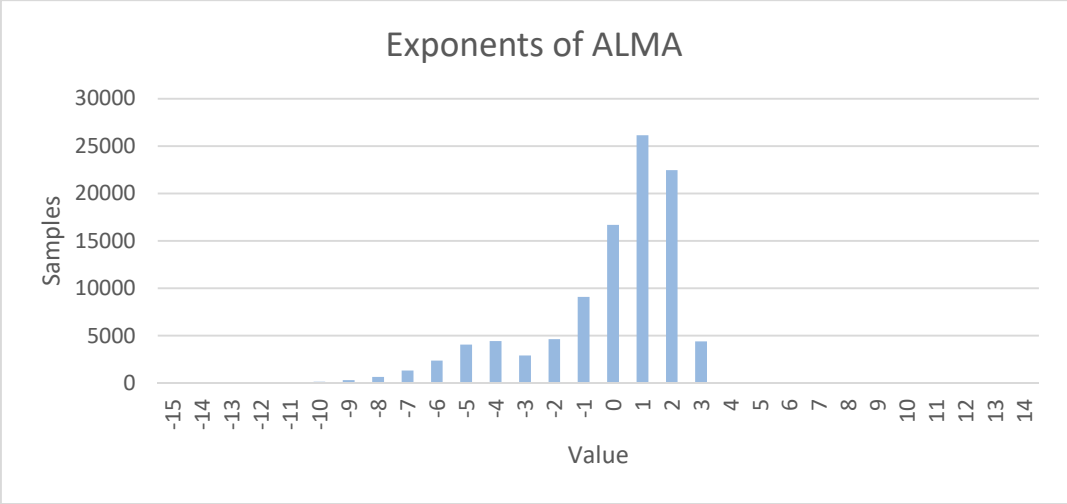

FIGURE 5.2: Histogram of the exponent values of ALMA

Fig. 5.2 displays a histogram of the exponent values of ALMA data. Since it is single precision, coded values go from -127 to 128, although those 2 values on both extremes would be special cases. In the plot we can see that 1 is the most repeated value with 26,158 occurrences, and 65% of the values are either 0, 1 or 2. If we add the -1 and -2 values it goes up to 79%. As it could be expected, the exponent has a lot of potential to be compressed. Since we want as many zeroes as possible or values as close to zero as possible, we will try the simple differential which should work very well in this case.

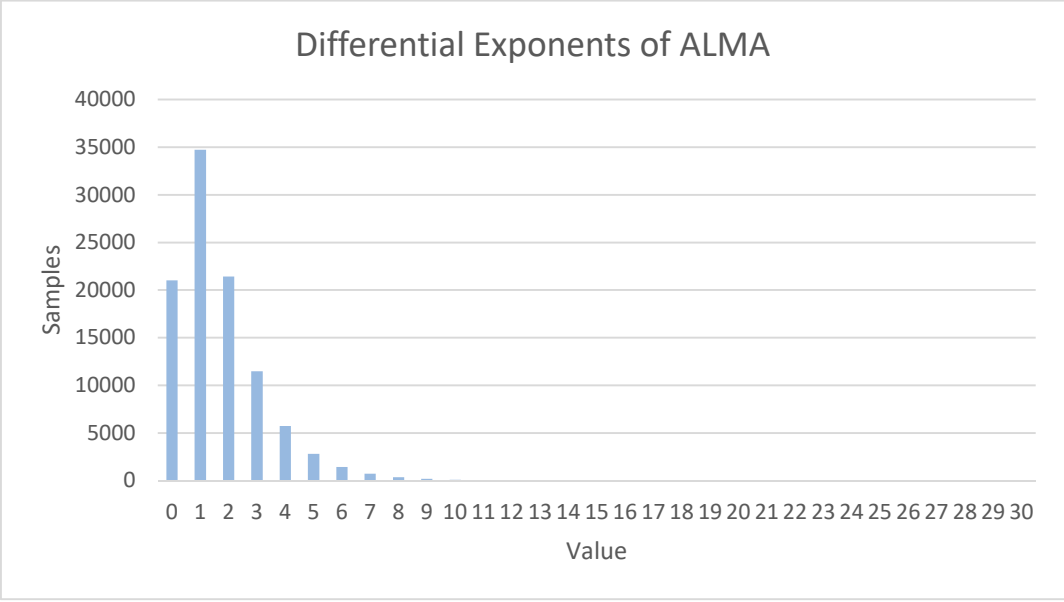

FIGURE 5.3: Histogram of the differential exponents of ALMA

Fig. 5.3 depicts the absolute value of the difference between each exponent and its predecessor, being therefore all values positive. This differential approach increases the number of zeroes we had by an extra 21%, although the values from 0 to 2, which include the negatives of those cases, represent now a 77%.

The 8 bits used originally for compressing the exponent will be highly reduced for most of the cases by using this technique. We should not forget that by coding the difference of any value we have to add an extra bit, which will keep the information of the sign of the difference, so compressing the difference will cost us 1 extra bit.

5.1.3 Mantissa

To analyze the mantissa of ALMA we will first plot a histogram of its values. Since the values have a wider range than in the case of the exponent, we need to group bins on the histogram. This means that every bar of the histogram contains 32,768 different values, so in the end we are looking at the 8 most significant bits of the mantissa and ignoring the other 15 (we expect them to be more random). Fig. 5.4 shows this histogram, again, considering 100,000 samples of the file.

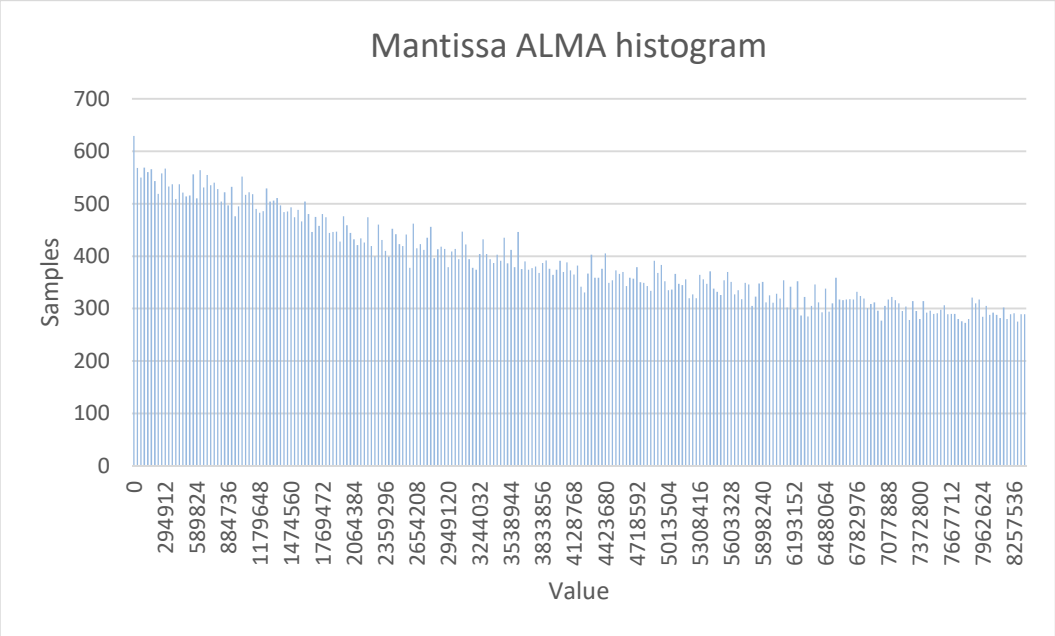

FIGURE 5.4: Histogram of the mantissa of ALMA data

The first conclusion is that the mantissa is not completely random. Low values are more frequent than high ones, but the difference between the first and the last bin is not as large as we would like it to be. We do see some potential of compression, but it is not as large as for the exponent. On top of that, considering that mantissa uses 23 bits while the exponent uses 8, the weight of mantissa is almost three times larger than that of the exponent. This implies that its compression will also have a weight three times larger with respect to the compression of the exponent.

Aiming at improving the compression of the mantissa, different filters have been applied following the same process as before. A simple differential was tested on the samples assuming that the changes from consecutive samples are not steep. Fig. 5.1 shows that the samples do not look completely random. They seem to have some trend. Taking advantage of this trend, we will try to gain compression.

Fig. 5.5 shows the histogram of simple differences of the mantissa using the same bins as in Fig. 5.4. As can be seen, the amount of higher values has largely decreased, and the amount of lower values increased. As we expected, the signal is not completely random so there is a chance of getting some compression.

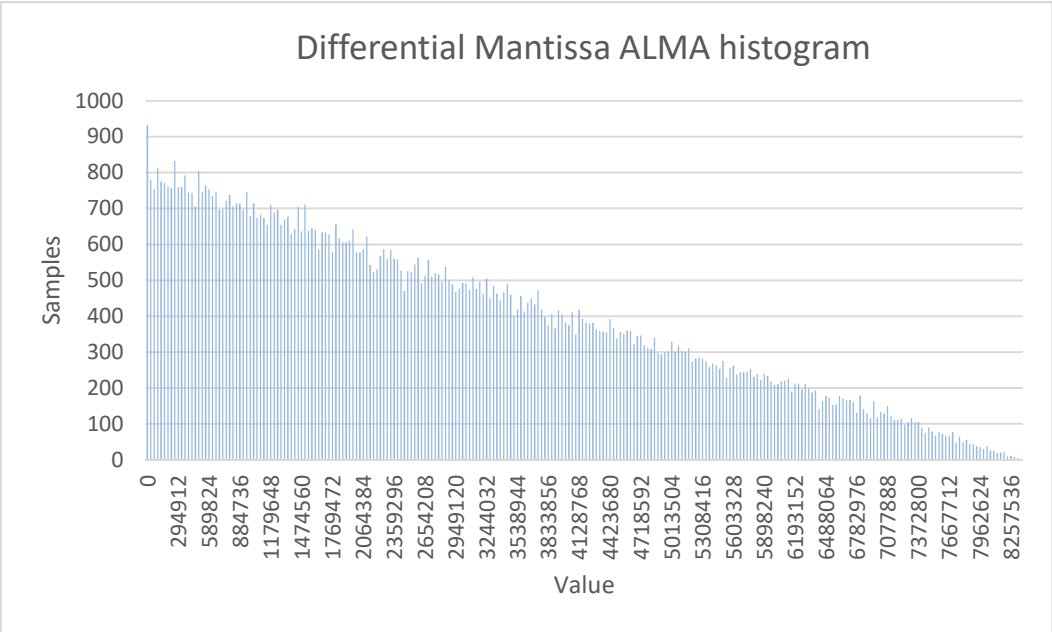

FIGURE 5.5: Histogram of the differences of the mantissa of ALMA

Although the approach of the simple differential seems to be working well, more complex options will be explored. A more accurate prediction will be made by using 2, 3 and 4 of the previous samples. We will take the average of these samples and compress the difference between the final sample and the prediction. The basic expression we used so far for the simple differential is:

$$X(i) = S(i) - P(i) \tag{5.1}$$

where *S(i)* is the signal, *P(i)* is the prediction of the signal and *X(i)* is the difference between them. In a first approach, the value of the following sample was set to be equal to the value of the previous sample and then the difference was obtained. In the new approach *P(i)* will be the average of the previous samples. Thus, for the second-order differential filter the prediction is

$$P(i) = \frac{S(i-1) + S(i-2)}{2} \tag{5.2}$$

The result of this second-order differential filter can be seen in Fig 5.6.

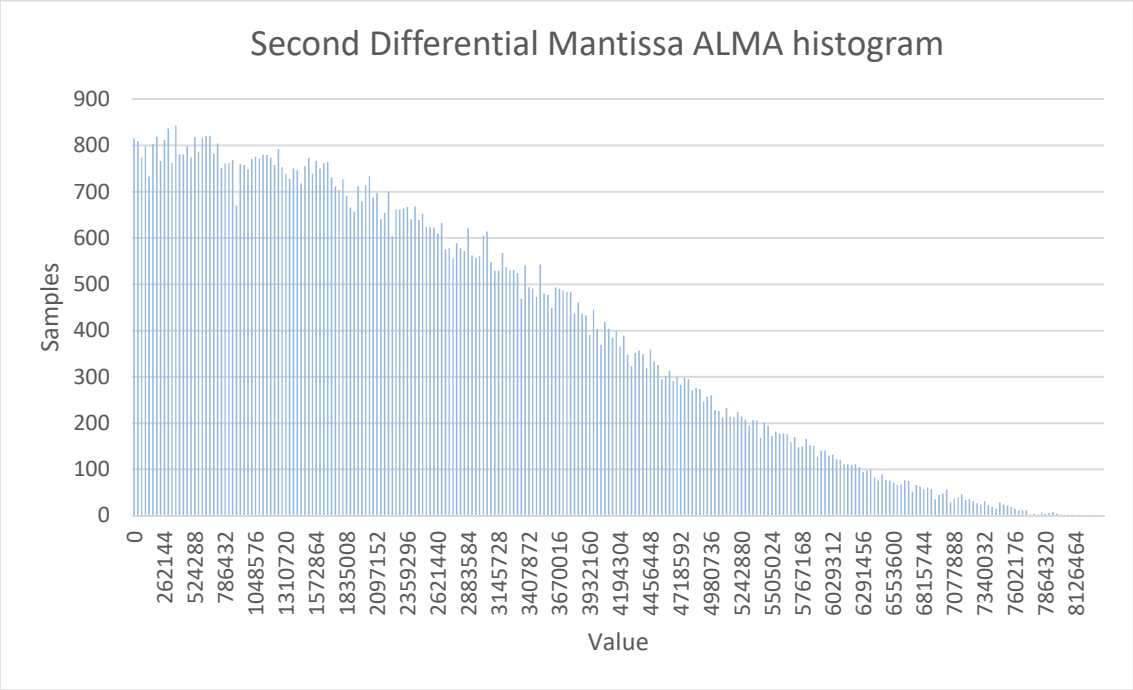

FIGURE 5.6: Histogram of the second-order differential filter of the mantissa of ALMA

Comparing Figs. 5.6 and 5.5 it can be seen that the number of higher values has further decreased and the number of lower values have increased. At first glance, it could imply that the second differential filter works better than the simple differential. However, if on closer inspection, it can be seen that, although there seems to be a gain, less values on the first bin have been obtained with respect to the simple differential (a loss close to 20%). This could mean that we are losing zeroes and could result in the simple differential being better than the second-order one. A further tool is needed to decide which one provides us with better results. Before using that tool, we will take the third and fourth order differential filters, using the 3 and 4 previous samples respectively.

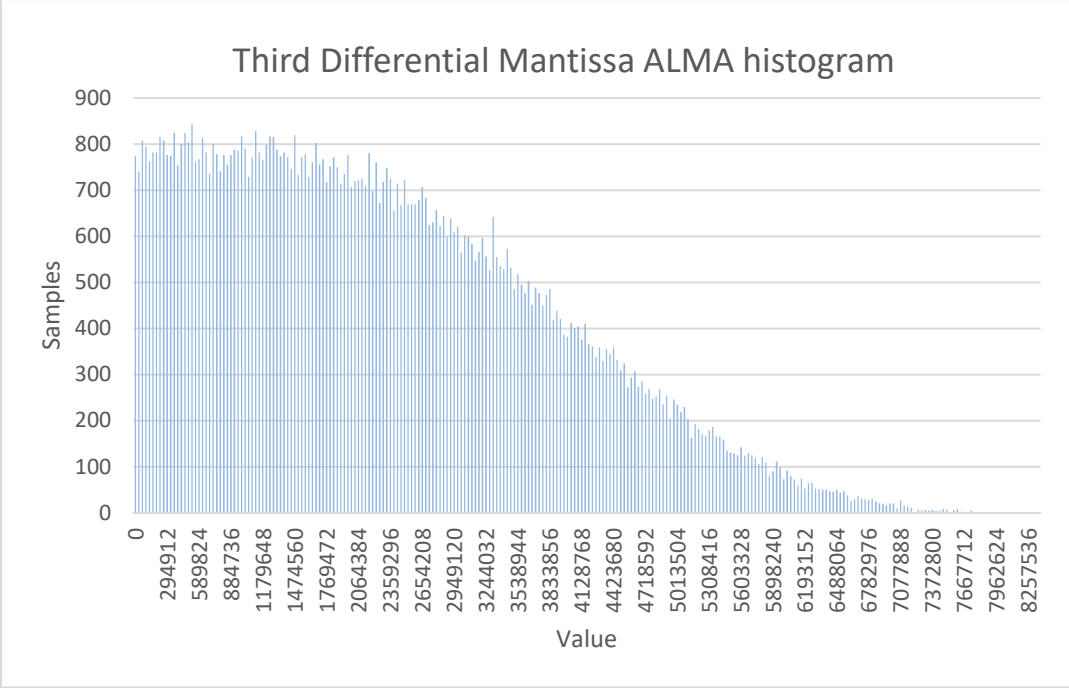

FIGURE 5.7: Histogram of the third order differential filter of the mantissa of ALMA

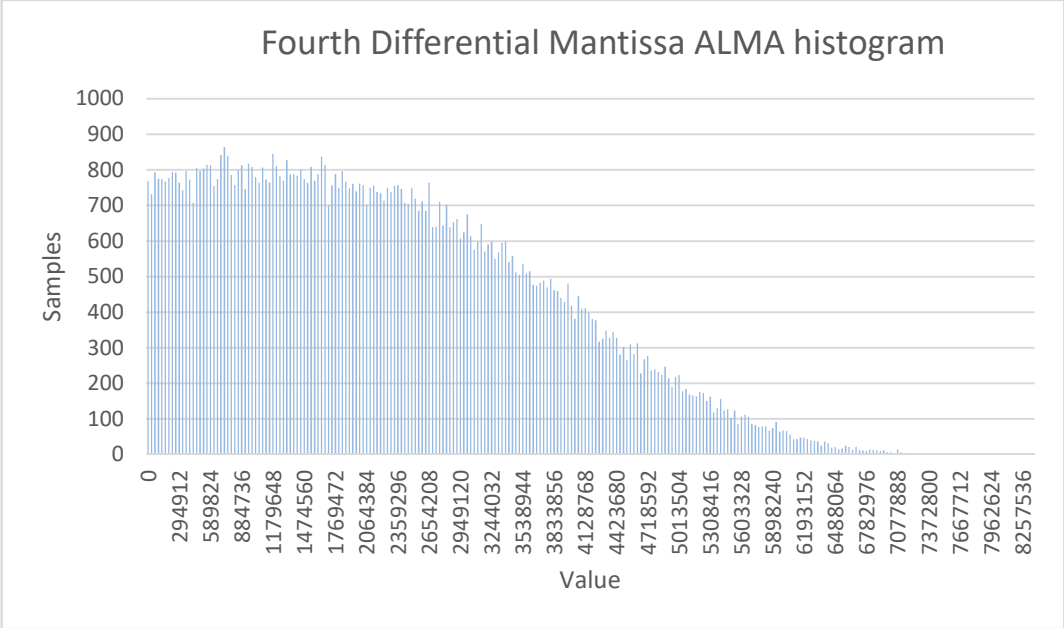

FIGURE 5.8: Histogram of the fourth order differential filter of the mantissa of ALMA

Figs. 5.7 and 5.8 appear quite similar. It seems that Fig. 5.8 gives slightly better results than Fig. 5.7, and that Fig. 5.7 looks slightly better than Fig. 5.6 too. Analyzing the entropy of the signal will allow us to determine which case is better.

5.1.4 Entropy and Shannon limit of the mantissa of ALMA

Entropy is the amount of information that a signal contains. It can be defined as:

$$H(x) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{5.3}$$

The entropy of each differential case will give us the Shannon Limit, which is the maximum compression ratio we can achieve with optimal coding for each case. The formula for its calculation is:

$$Shannon\ limit = \frac{Input\ (bits)}{Entropy\ (bits)} \tag{5.4}$$

A short code was developed in order to calculate the entropy of each signal so we can compare which one has the highest Shannon limit, and therefore which one will give us the best compression ratio. It is worth mentioning that all these entropy calculations are made on the sampled signals according to the algorithm described before. This means a completely different algorithm could lower the entropy and therefore increase the Shannon limit, which would lead to a better compression ratio.

	Entropy (bits)	**Shannon Limit**
Mantissa differential N=1	20.37	1.13
Mantissa differential N=2	20.30	1.13
Mantissa differential N=3	20.27	1.13
Mantissa differential N=4	20.25	1.13

TABLE 5.1: Entropy and Shannon limit of the mantissa of ALMA

Our program has some limitations in the calculations of the mantissa. These limitations are due to the size of the samples. Considering the hardware we are using, it is not feasible to process a histogram of 2^{23} items. Therefore, to represent those values, it was necessary to make bins which allowed us to represent 2^{20} different values, ignoring only the 3 least significant bits of the mantissa. This means that the final Shannon limit might still be slightly lower than the one calculated here. We estimated how big this impact could be by analyzing those 3 bits separately, and the result was that it could lower the entropy by 0.05, which means that the Shannon limit we are showing here would not change significantly. Therefore, we considered this estimate to be enough for our purposes. As can be seen in Table 5.1, the fourth differential is the one that gives the best results. Nonetheless, the differences between all four cases are too subtle and when calculating the Shannon limit only the third decimal would be modified. This means that the extra complexity of the higher differential preprocessing algorithms does not justify their implementation, since the changes in the final results would not be noticeable in the final ratio. The obtained result would almost be the same regardless of the degree of the differentials. At this point, it is important to make clear that the entropy has one extra bit, since this bit is needed to indicate whether the value must be added or subtracted to the previous one.

5.1.5 Overall entropy and Shannon limit of ALMA

The mantissa results were not very promising since a compression ratio of 1.13 is not very high. In addition, it is important to clarify that this ratio would be obtained in optimal conditions, which is not our case. Note that this is not the overall ratio we will have, due to the fact that the exponent has a higher potential of compression. Thus, it is necessary to calculate the whole entropy of the ALMA data.

	Entropy (bits)	**Shannon Limit**
Sign	0.98	1.02
Exponent	3.46	2.31
Mantissa	20.37	1.13
Exponent + sign	4.30	2.09
Sign + exponent	4.61	1.95
Mantissa + sign	19.94	1.20
Sign + mantissa	19.94	1.20

TABLE 5.2: Entropy and Shannon Limit of the parts of ALMA

Table 5.2 shows the entropy of the three parts of the signal: sign, exponent and mantissa as well as the value obtained after combining the sign with the other three parts. Exponent and mantissa have already added that extra bit for sending the difference of the prediction errors. Adding the sign to the other parts of the signal is needed, since we do not want to transmit the sign by itself. It is notable that sending the sign with the mantissa has less entropy than sending the mantissa itself, so it appears that the sign is allowing more compression when combined with the mantissa. As can be easily seen, the mantissa combined with the sign has lower entropy than the mantissa itself. This is not real, since our limit is 2^{20}, we are ignoring the last 4 bits instead of 3, but it is still a good approximation. Finally, it is worth combining the sign with another part of the signal since otherwise we would need to send it by itself without any compression. The final Shannon limit was obtained by combining all the previous parts of the signal and is shown in Table 5.3.

	Shannon Limit
Total Sign + Exponent + Mantissa	1.42
Total Exponent and Sign + Mantissa	1.40
Total Exponent + Mantissa and Sign	1.48

TABLE 5.3: Entropy and Shannon Limit of ALMA

Table 5.3 demonstrates that the best possibility is to leave the exponent by itself and combine the mantissa with the sign as a least significant bit. This means that the Sign is more random than the most significant bit of the mantissa. There could be better solutions by means of splitting the mantissa and placing the sign in the middle but it is not worth trying its theoretical analysis since the limitations of the implementation might force us to use other alternatives.

5.2 SKA analysis

In this section, the same analysis for SKA is performed considering its particularities. It is worth mentioning that the SKA dataset is a single precision binary file with little endian, therefore this is the one that will be analyzed. We will look for the compression ratio for this original file, but since we were unable to get a double precision binary file of real data, we will also convert this data into a double precision dataset and compress it as well to test our double precision algorithm. This means that the final ratio achieved will be higher because the conversion will add redundancy to the information, but in the end both the comparison with other algorithms and the analysis for the compression algorithm has the same value.

A particularity of the SKA dataset is that its coding has interleaving of 168 samples. In the case of the histograms it is irrelevant whether the signal is interleaved, since we are simply counting the amount of values that appear in the signal but we do not take into account the order. However, for time sequences it is different.

The analysis is done using the same procedure as before. We do this for single precision, since converting the data to double precision does not change the range of the values or the values themselves. The analysis follows the same steps as for

ALMA, plotting 1,000 samples of the temporal sequence – see Fig. 5.9. Note that the plot is in a logarithmic scale.

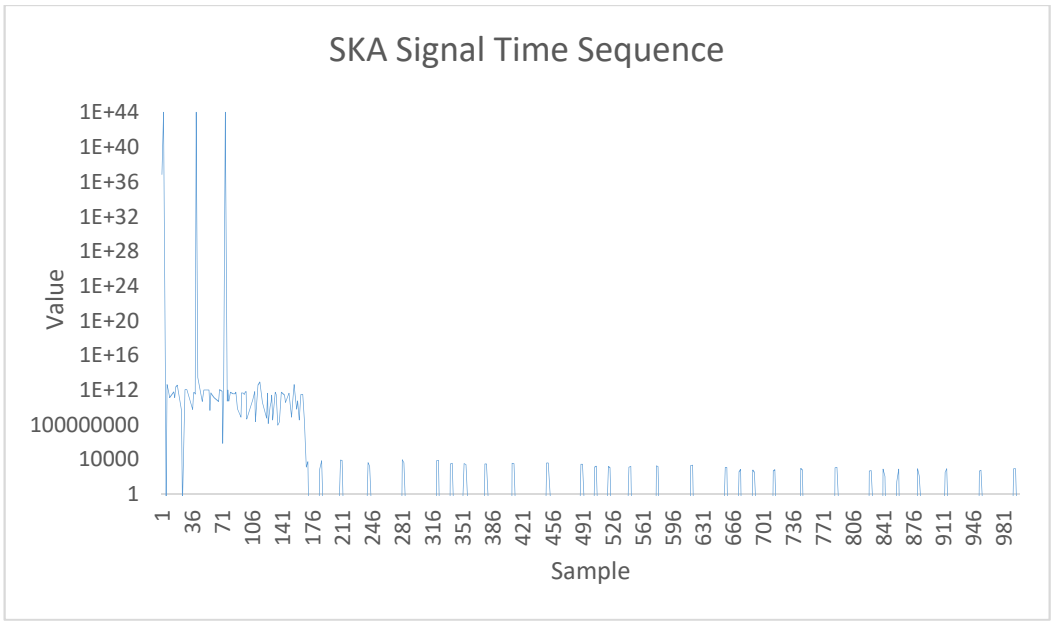

FIGURE 5.9: SKA signal time sequence (logarithmic scale)

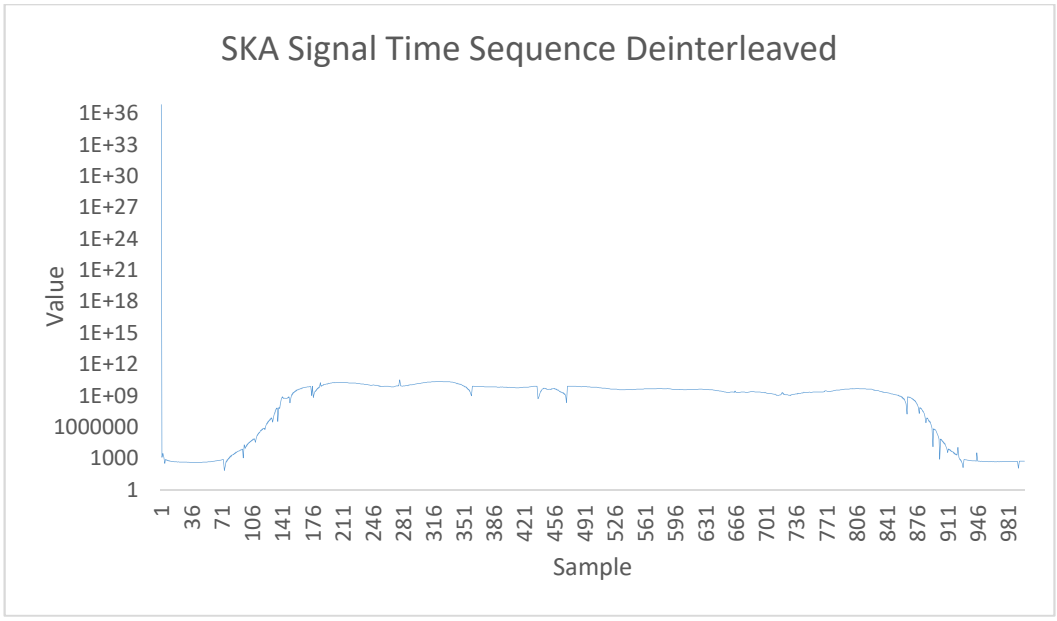

FIGURE 5.10: SKA signal time sequence de-interleaved (logarithmic scale)

Fig. 5.9 shows the signal without the interleaving, that is, the signal stemming directly from the dataset. We can see three very high peaks on the order of 10^{44}. We can also see some values around 10^{12} and some other values around 10^{3}. The rest of the values not shown are negative. The interleaved signal was also analyzed, which required of programming a code to undo the interleaving.

Fig. 5.10 shows only a peak of 7.3 x 10^{36} and the rest of the values show some continuity, which goes from 10^{3} to 10^{12} and then again to 10^{3}. The continuity is always positive in this case, since it shows a pattern that might turn into compression ratio.

Compressing the signal without interleaving will result in a much lower ratio since it randomizes a lot the received data and it makes us lose the continuity of the signal. We need a deeper analysis to find out the compression ratios we can obtain.

5.2.1 Sign

The analysis on the sign uses the same procedure as for ALMA. Taking 100,000 samples gives us a total of 55,981 zeroes and 44,019 ones. These results are similar to the results obtained for ALMA. They are close to 50%, which in theory would not represent much gain. When combined with the mantissa the compression ratio could be slightly increased.

5.2.2 Exponent

We now take a look at the histogram of the exponents of SKA the same way we did for ALMA. Fig. 5.11 shows that the values are centered around -5 and -6, with a peak on the 0. This is more irregular than the ALMA case but the difference should still help us gaining compression since the extremes are almost empty. 72% of the values are from -9 to 1, so there is a lot to gain on the exponent of SKA in terms of compression. There are some samples from out of the range represented on the graph, but it is shortened to make it easier to view.

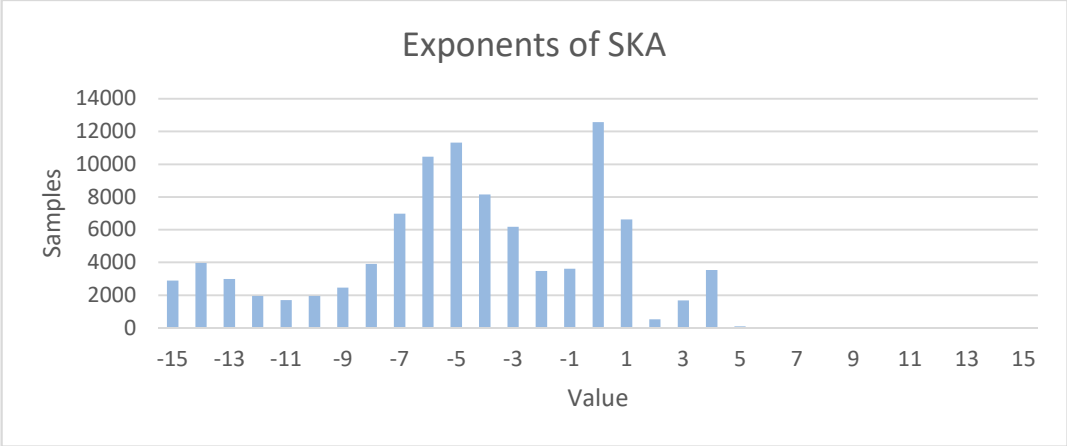

FIGURE 5.11: Exponents of SKA

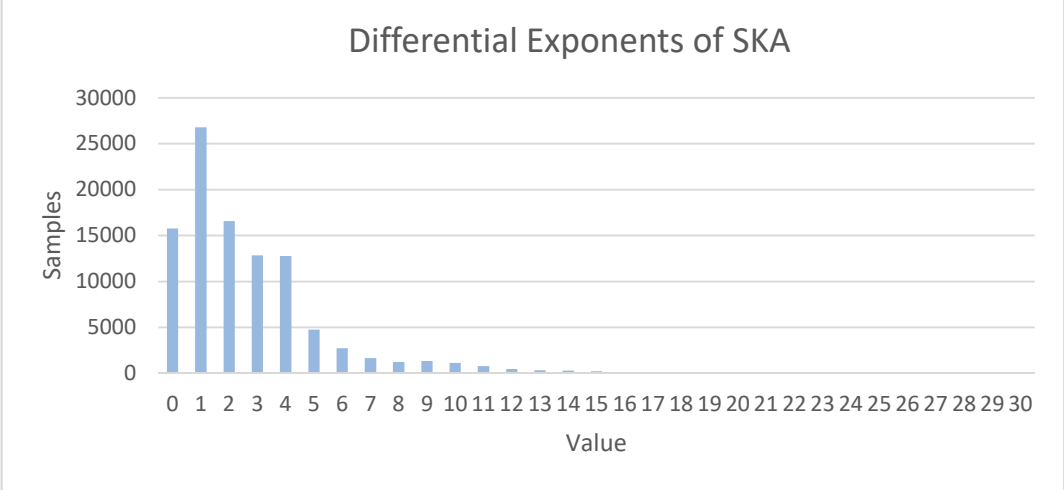

FIGURE 5.12: Differential exponents of SKA

Fig. 5.12 depicts a similar phenomenon to the one observed in ALMA. 84.5% of the values range from 0 to 5 and the zeroes have been increased by a 24%. These gains will be noticed when compressing, but we have to take into account that the exponent is only a fourth of the whole SKA signal and that for compressing differences we have to add an extra bit. Again, we need to look at the mantissa, which is almost three fourths of the signal.

5.2.3 Mantissa

The analysis of the mantissa of SKA was performed using a similar procedure to the one used for ALMA. First, a histogram of its values is plotted considering that every sample contains a range of 32,768 values. The first 100,000 samples of the file are plotted in Fig. 5.13.

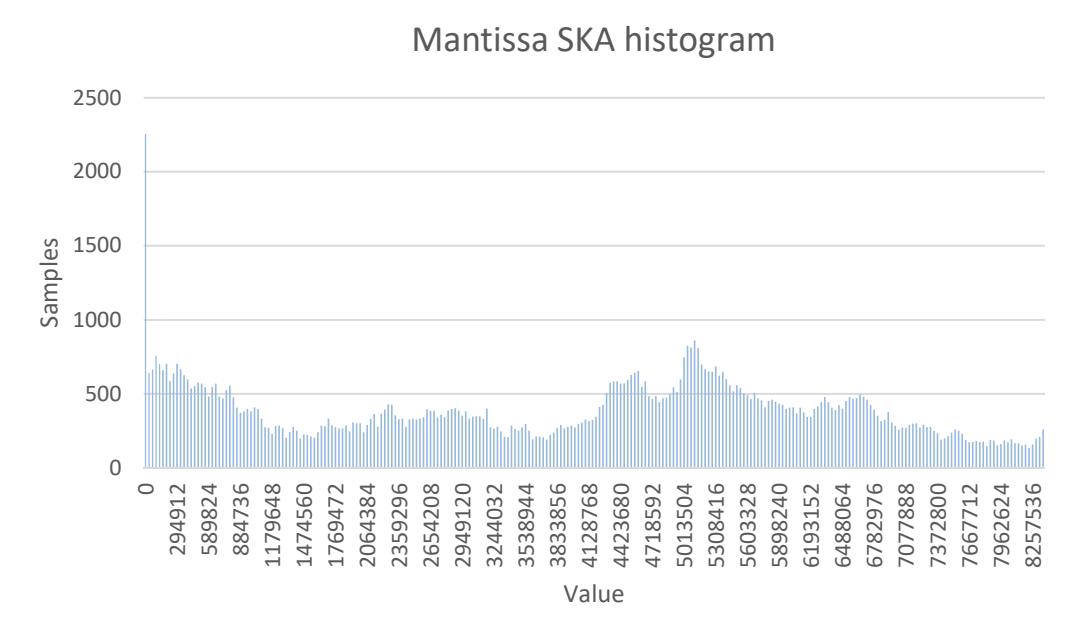

FIGURE 5.13: Histogram of the mantissa of SKA

The first feature that can be observed in the plot is the high number of samples that can be found in the first bin. There are around 4 times more samples in the first bin than in the others. This could mean that there is a relatively high amount of zeroes, but we can only state that there are 4 times more values from 0 to 32767 than in any other bin. Other than that, the rest of the values seem quite randomized. There are some bins with double the samples than others, but they are all spread without a recognizable pattern.

If we take a deeper insight, the first half of the plot has a decreasing number of samples for most cases. That behavior can be seen once again after the second peak, so the differential should provide us with better results than the raw plot. Just as

before, different filters are tested to improve compression and to discover which one has a better performance for the mantissa.

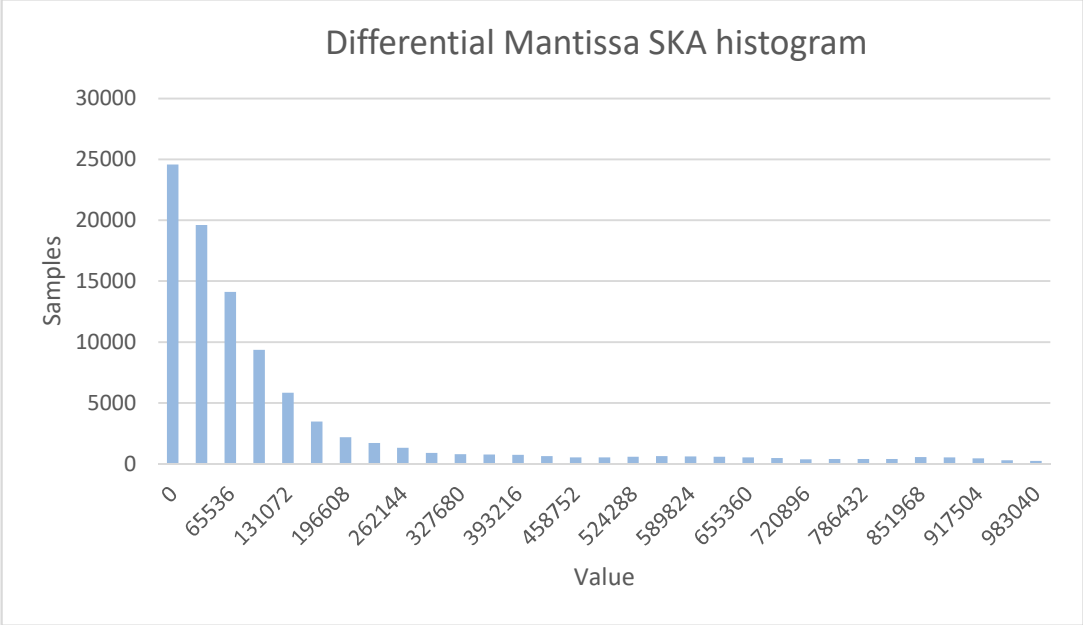

FIGURE 5.14: Histogram of the first differential of the mantissa of SKA

Fig. 5.14 shows a large improvement. After the 37th bin, values are barely visible on this scale, thus meaning very few occurrences. The graph shows a quite high potential of compression. However, it is needless to say that since the data is grouped in bins, we are only checking 3 out of 23 bits, so the compression seems good for these ones, but we cannot tell about the other 20 bits.

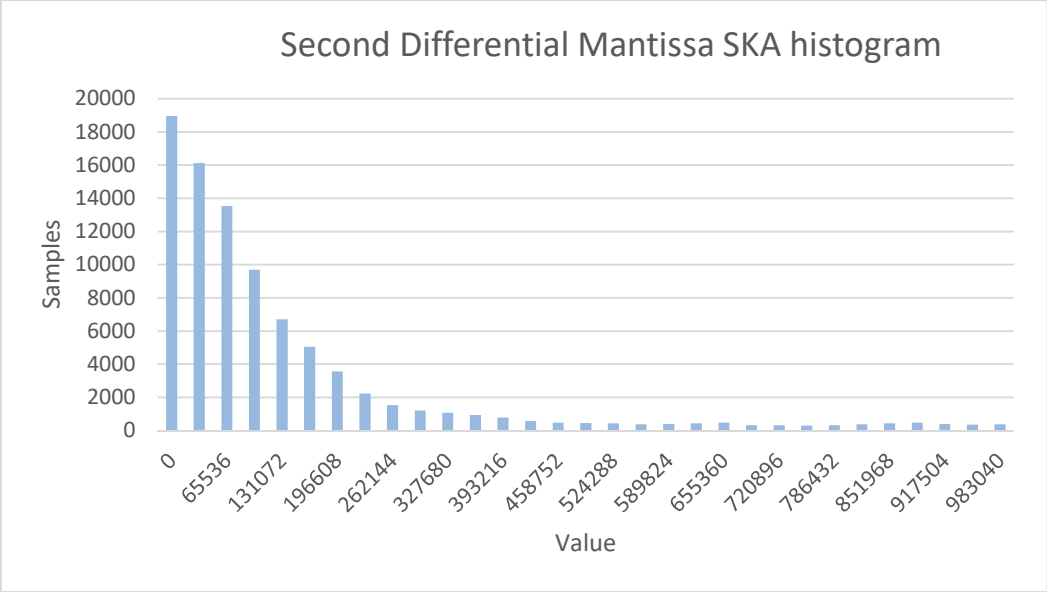

FIGURE 5.15: Histogram of the second differential of the mantissa of SKA

Fig. 5.15 shows the second differential of the mantissa of SKA. In this case, the shape of the graph is quite similar to figure 5.14 but the values are slightly more spread in this one. In Fig. 5.14 there were more values in the first 3 bins while in Fig. 5.15 from the fourth to the end the number of values is slightly higher. Therefore, a better compression is expected by using the first differential rather than the second one, since it is desirable to have as many values as closest to zero as possible.

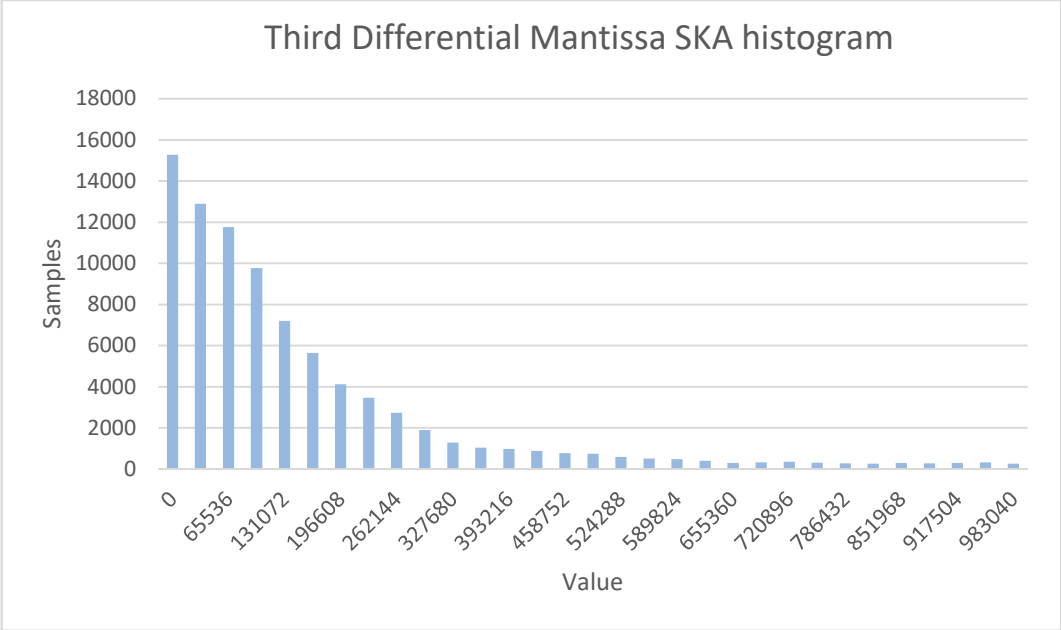

FIGURE 5.16: Histogram of the third differential of the mantissa of SKA

Fig. 5.16 shows the third differential of the mantissa of SKA. The same progression that went from the first to the second differential is seen here if we compare the second with the third. Once again, the values are getting worse in terms of potential of compression. So far, the first differential seems to be the best one at first sight, but a deeper analysis is needed to determine which one is actually better. The fourth differential is also expected to be slightly worse.

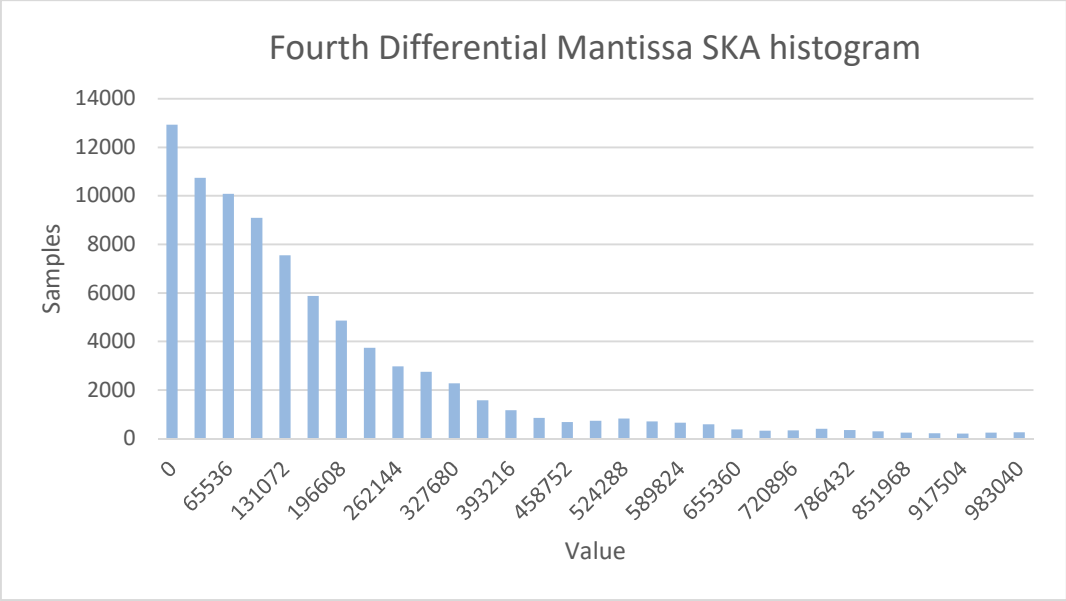

FIGURE 5.17: Histogram of the fourth differential of the mantissa of SKA

Taking a look at Fig. 5.17 it can be confirmed that the fourth differential is worse than the third. This means that the correlation is weaker when increasing the distance with the predicted sample. Simply taking the last value and using a simple differential provides us with quite a good result to predict the next sample, therefore, some compression is gained.

5.2.4 Entropy and Shannon limit of the mantissa of SKA

	Entropy (bits)	**Shannon Limit**
Mantissa differential N=1	15.33	1.50
Mantissa differential N=2	16.03	1.43
Mantissa differential N=3	16.49	1.39
Mantissa differential N=4	16.69	1.38

TABLE 5.4: Entropy and Shannon limit of the mantissa of SKA

Several conclusions were drawn from the graphs of the SKA values for the mantissa. However, as we did with ALMA, it is necessary to calculate the entropy and Shannon limit to conclude which scenario will provide us with a better compression. Table 5.4 shows that the first differential is the one offering the best results. On ALMA the

results were very close to each other, so any of the differentials could have been used. In general, there could be cases where other filters got a better compression ratio. However, we are looking for a quick pre-processing and analyzing on the fly which option is better would take too much time for only a potential small improvement. Consequently, the first differential will be used in our implementation for all cases.

5.2.5 Overall entropy and Shannon Limit of SKA

The mantissa results for SKA were much more interesting than the ones for ALMA. The entropy and Shannon limit of the sign and the exponent will now be added to calculate the expected ratio in optimal conditions.

	Entropy (bits)	**Shannon Limit**
Sign	0.99	1.01
Exponent	3.99	2.01
Mantissa	15.33	1.50
Exponent + sign	4.86	1.85
Sign + exponent	4.30	2.09
Mantissa + sign	16.25	1.48
Sign + mantissa	16.88	1.42

TABLE 5.5: Entropy and Shannon limit of the parts of SKA

Table 5.5 shows the entropy of the three parts of the signal: sign, exponent and mantissa as well as the combination of the sign with the other three parts, in the same way that was done for ALMA. Exponent and mantissa have already added that extra bit for sending the differential of the signal. Adding the sign to the other parts of the signal is needed since we do not want to transmit the sign by itself. In ALMA we could see that, when we wanted to transmit the exponent and the sign together, the best option was to transmit the difference of exponent with sign at the bottom with its previous values. But for SKA this changes, and it is better to put the sign at the beginning. This shows that the sign is more stable than the exponent, and in the end it is better to send the sign with the exponent as it can be seen in table 5.6.

	Shannon Limit
Total Sign + Exponent + Mantissa	1.61
Total Sign and Exponent + Mantissa	1.67
Total Exponent + Mantissa and Sign	1.61

TABLE 5.6: Entropy and Shannon limit of SKA

In this case, the best option would be the second one. The problem is that we cannot invest time in these calculations every time a file is going to be compressed. Since for ALMA putting the sign in front of the exponent would degrade a lot the compression ratio, but for SKA putting the sign at the end of the mantissa decreases the ratio less than a tenth, the third option will be implemented.

This result for SKA was surprising, since our observations showed that the exponent was highly compressive, and mixing it with the sign was expected to make the final compression worse. However, the results were better. Having to choose one of them forced us to look for the best option in general instead of being able to choose it for every case.

5.2.6 Entropy and Shannon limit of SKA on 64 bits and interleaved

All the analysis shown so far was done on the original data. However, since the aim was to make the algorithm work also for 64 bits signals and since it was not possible to get real data in this format, the SKA 32 bits data was converted into 64 bits data. This was done byte by byte converting the format type of each one and writing a new file. In this process it was also important to undo the interleaving, since that would make our algorithm achieve higher compression ratios. This interleaving process will not be automatic in the implementation of the algorithm, but it is something to consider in future releases.

We were unable to recalculate the entropy of the mantissa since we could only take the first 20 bits out of 52, which are not real information as we are adding 29 bits on the mantissa just for the conversion to 64 bits. This means that the calculation would not give any relevant result, but we can assume that the entropy was the same since the information was not changed.

Since the information that is being sent is the same, but using extra bits, the entropy, as stated, will remain the same, but the Shannon limit will change, as it can be seen in table 5.7. Nevertheless, joining the several parts of the data it does change. Putting the sign in front of a larger type of data will degrade the compression since the difference is being compressed, so those cases will not be considered on the table, as they are much worse.

	Entropy (bits)	**Shannon Limit**
Sign	0.99	1.01
Exponent	3.99	2.76
Mantissa	15.33	3.39
Exponent + sign	4.86	2.47
Mantissa + sign	16.25	3.26

TABLE 5.7: Entropy and Shannon limit of the parts of SKA for 64 bits

As one can see, the Shannon limit has increased significantly for the mantissa and slightly for the exponent. This is because we are using now 52 bits on the mantissa instead of the former 23, which means that the information remains the same, but those extra empty bits make it much more compressible. In the case of the exponent, we are switching from 8 to 11, which is a smaller increase. The final Shannon limit is listed in table 5.8.

	Shannon Limit
Total Sign + Exponent + Mantissa	3.24
Total Exponent and Sign + Mantissa	3.22
Total Exponent + Mantissa and Sign	3.17

TABLE 5.8: Entropy and Shannon limit of SKA for 64 bits

The first option gives the best ratio in this case, very close to the second one. We have to take into account that this is converted information. This means that on real 64 bits data there will not be such redundancy on the mantissa and the ratios would never be this good. The implementation might give much worse results, since several limitations that will keep us away from the ideal will be faced.

Chapter 6

6 Implementation

This chapter is devoted to show the process that turns the analysis from chapter 5 into the implementation of the real software program and the adaptations of the algorithm. The theoretical results and the reality often differ due to certain limitations and problems faced. On the implementation, there is a need to write the code for the pre-processor and post-processor for both floats and doubles. Those stages are right after the input of the data and before the compressing and decompressing stages respectively. This can be seen in Figs. 6.1 and 6.2.

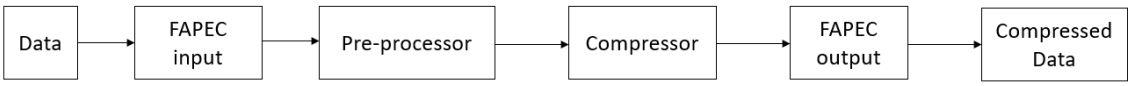

FIGURE 6.1: Model of the FAPEC compressor

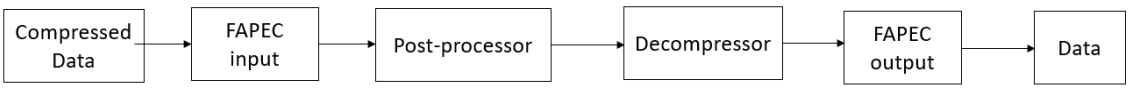

FIGURE 6.2: Model of the FAPEC decompressor

We start with the implementation of the pre-compressor for floats. We then adapt the code for the doubles and will write both decompressors.

6.1 Pre-processing of floats

To pre-process the floats the algorithm will have two stages. The first stage will divide the input data into arrays of information to be compressed. The second stage will send the data to the compressor.

6.1.1 Assembling for compression

First of all the data from the input file needs to be read. The bytes will be read in groups of 4 to generate the full float. This can be done in two ways depending on whether the input data is big or little endian. This is a known information for the user and he will need to select which is the case so that the algorithm gives the best results. When the float is successfully read, we can start assembling it.

Firstly, we need to choose how we send the information for compression, and we will found limitations in the ways to do this. The first limitation we had to face was that FAPEC could not handle symbols larger than 24 bits due to its internal design. This was not a severe limitation for floats, but it was so for doubles. In this case, we expected to divide the float into 2 parts, one with the 8 bits of the exponent and the other one with the 23 bits of the mantissa plus the sign, so a total of 24 bits. This structure is shown in figure 6.3. Those two parts would be compressed individually to achieve the best result possible.

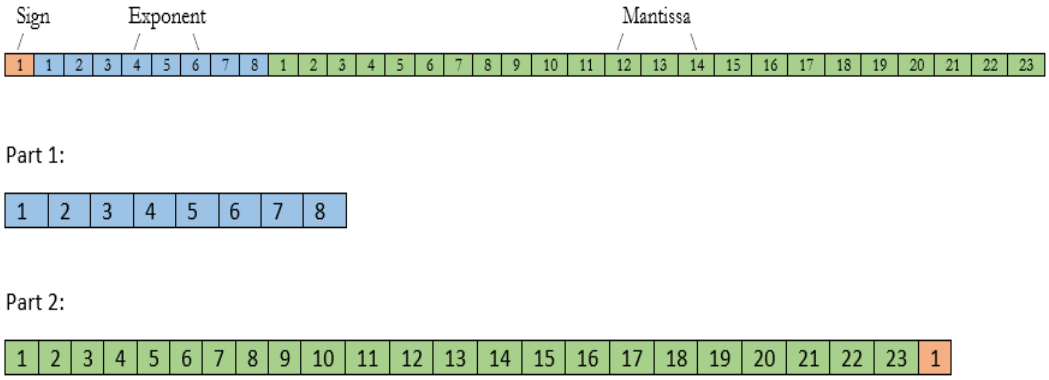

FIGURE 6.3: Theoretical strategy to compress floats

6.1.2 Compressing

This second stage will mainly pick the data from the first stage and use the FAPEC algorithm to compress consecutive samples. Since we had two divided sources of pre-processed data, we would do the largest compression on the fly and store into an array all the exponents to compress them later.

For this process, we need to store 50% of the data. Considering the processing is divided into chunks, which by default are 4096 KB, we needed 2048 KB of storage, which is low enough to avoid impacting at all the overall performance on the hardware. Headers will be added on every chunk so that we can decompress it later.

6.2 Tests and results on floats

In our first approach, we decided to implement the best result we had achieved with the theoretical analysis and check how far the results were from the optimal approximations we had. All the tests of the different options for the implementation will be done in the second stage of the pre-processing. This implementation is the one of figure 6.3.

The implementation of the first version yielded very poor results, too far from those expected theoretically. We used the 32 bit version for both ALMA and SKA, for both the original data samples. We compressed separately the 8 bits of the exponent and the 23 on the mantissa with the sign added at the end. The compression ratio for ALMA was 1.04 and for SKA 1.06. Based on the theoretical calculations, the best results we could achieve were 1.48 for ALMA and 1.61 for SKA in this situation, so we were very far from that.

One of the reasons the results were being so bad was that in too many cases, compressing the second array would result in the use of 25 or 26 bits for it, when we originally had 24. Not compressing the mantissa was actually better in general, increasing the compression ratio by 8%. Consequently, this needed to be fixed. The second problem we found was that it was much better to compress blocks of the same size than to have blocks with very different sizes. In this case, compressing the mantissa with the sign was 24 bits and the exponent 8 bits. With one block being three times the other block makes it very inefficient when compressing the exponent.

We decided to test different options. One would split the signal in four parts, which would mean to keep the exponent on the first part, the first 8 bits of the mantissa on the second part, the next 8 bits of the mantissa in the third part and finally the remaining 7 bits of the mantissa plus the sign in the last part. . This structure can be seen in figure 6.4.

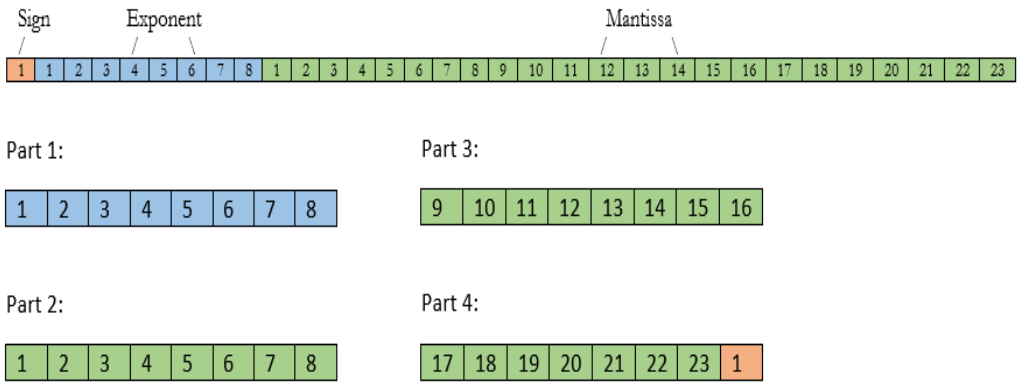

FIGURE 6.4: Second strategy to compress floats

This would keep constant blocks of 8 bits and would be efficient, but the more blocks involved, the more compression we lost due to having to use a bit on each block for sending the difference of each sample with its predecessor.

A second option was developed. This option would split the signal into two parts. The first part would contain 8 bits of the exponent plus the first 8 bits of the mantissa, which are the ones that have more potential of compression. The second part would contain the remaining 15 bits of the mantissa plus the sign. This can be seen in figure 6.5. With this implementation the ratios significantly increased to up to a 40% for ALMA. However they would stay equal for SKA.

The problem with SKA was using the data straight as we had it. It was known that SKA had been interleaved by 168 samples, which made it worthless to compress straight and it was better to use other filters than the one that we were developing. This happened due to the fact that our algorithm is based in using the continuity of the signal to compress. Without undoing the interleaving all our efforts would be useless. The interleaving would be undone for the doubles, since we decided to convert SKA to 64 bits data to test that algorithm. However, to develop the algorithm for floats we tested it as well to see how it would work knowing the ratio would still be bad.

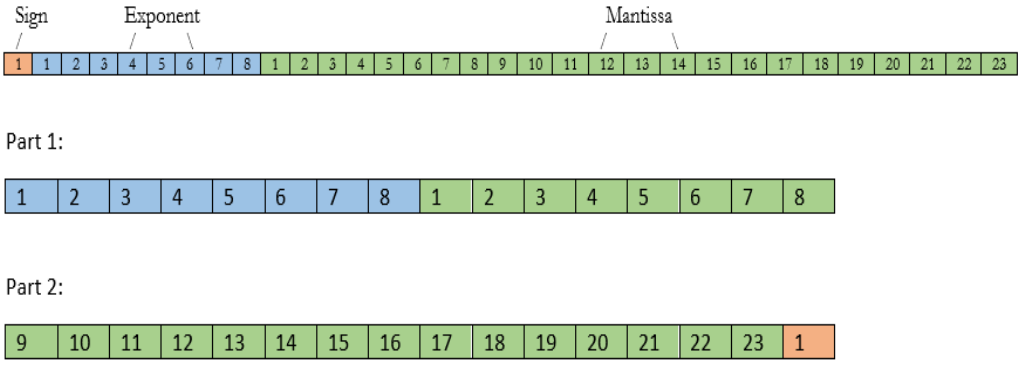

FIGURE 6.5: Third strategy to compress floats

We found a third option, which gave better results. It consisted in compressing the sign with the first block. The first block would contain the exponent, the 7 first bits of mantissa plus the sign and the second block would contain the remaining 16 bits

of the mantissa. This can be seen in figure 6.6. This raised ALMA's ratio to 1.15 but kept the previous ratio for SKA.

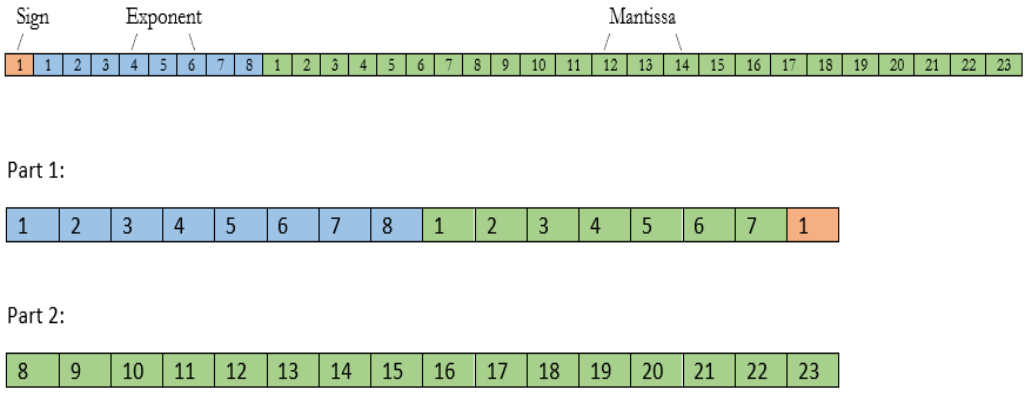

FIGURE 6.6: Final strategy to compress floats

This would be the final implementation used for doubles, since the optimal implementation for ALMA was 1.48. Getting a 1.15 was very far from our theoretical calculations. The reasons for this could be our sample not being very representative of the full signal, or the limitations we were facing dealing in a degradation for our compression ratio. The algorithm was simple, quick and its results, as it will be seen later, were solid, so we simply needed to be able to decompress this information.

6.3 Post-processing of floats

Regardless of how good a compression algorithm is, it is completely worthless if the original data cannot be recovered. The decompression has to be able to get all the data back without losing a single bit and reassemble it so that the final file is exactly the same as the original one. Decompressing in theory is very simple. We will have to reproduce the compressing process but doing all the steps in an inverted order. If not all the steps can be done backwards, the decompression will fail. We will take a look at the 2 stages for the decompression and we will invert them.

6.3.1 Decompressing

To undo the compression we need to know how it was done. We know that the data was split into two parts. The first part would contain the first 8 bits of the exponent, the first 7 of the mantissa and the sign, while the second part would contain the remaining 16 bits of the mantissa. The first part was compressed on the fly, right after reading and splitting, but the second part was compressed after the first part was done. This means that our compressed file contains, on the first half those first 16 bits of the whole signal, and on the second part, the other 16. Therefore, we will need to read the first 16 bits, store them and wait for the other 16 bits to reassemble every float of the data. We will decompress the first half of the signal and store it. Then we will decompress the second half while doing the second stage on the fly.

6.3.2 Reassembling the bytes

This stage is done while we are decompressing the second half of the signal. We need to take the 32 bits which were split when compressing and rebuild the original float by using hexadecimal masks. This sounds very simple in theory but it is hard to assemble accurately each half of the float with the other half.

While we are reassembling all the floats, to make this process as efficient as possible, we write the output file. Once again, we need to know the endianness we are using to write the file the same way as the original one. When this process is done, we compare the output file with the original data file bit by bit. If this process is successful, our compression worked.

After several adjustments on this decompression algorithm, we were able to recover both ALMA and SKA and the recovered files would be exactly the same as the original ones bit by bit.

6.4 Pre-processing of doubles

The structure of the program for pre-processing the doubles is the same as it was for the floats. Since we are working with 64 bits words instead of 32, we will see some differences which will be explained in the following sections.

6.4.1 Assembling for compression

In order to read the input data this time we simply need to pick the groups of 4 bytes until 64 instead of 32 and consider the endianness. When the process is done, we are ready to start with the assembling.

As explained for floats, this stage is the most important one and has some relevant differences. We know our limitation to handle strings larger than 24 bits so we are not allowed to divide the signal in 2 parts of 32 bits. We considered two different options. The first one would be the symmetrical one, which would split the signal into four groups of 16 bits with the considerations from the theoretical analysis to give the best result possible. This option is shown on figure 6.7. However, this option was overcome by the asymmetrical option. This second option would split the signal into a group of 22 bits and two more groups of 21 bits. Splitting the signal into 3 instead of 4 would be better since we are using less headers and one less extra bit for sending the difference of the samples. This structure can be seen in figure 6.8. The first 22 bits would contain the 11 bits of the exponent, followed by the 10 first bits of the mantissa and the sign. The other two groups of 21 bits would contain the rest of the mantissa.

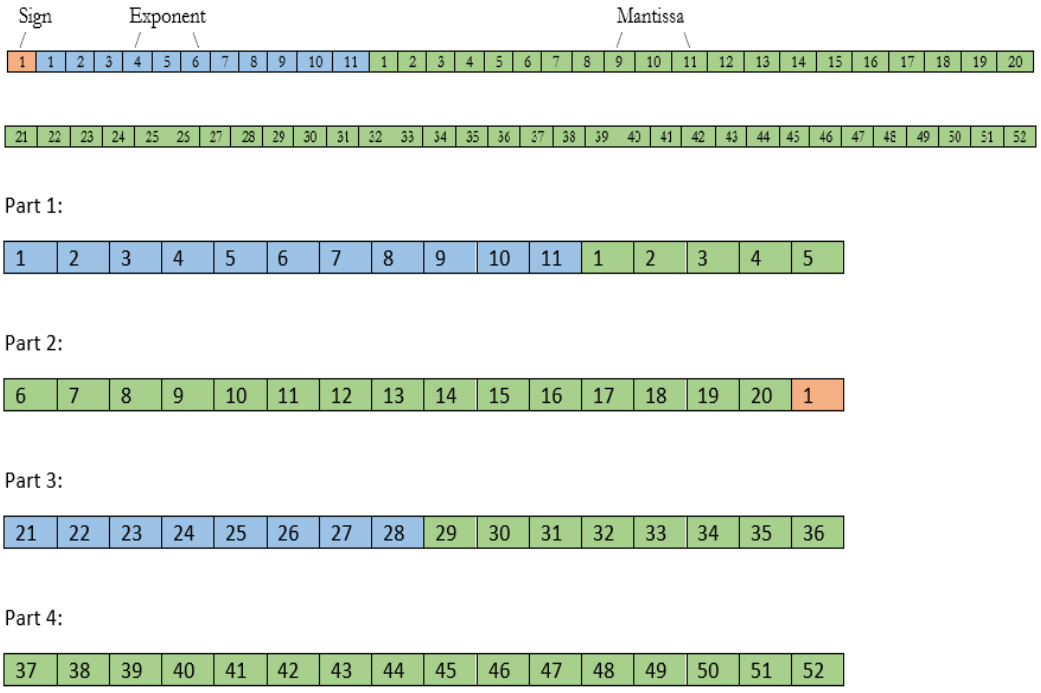

FIGURE 6.7: Symmetrical strategy for doubles

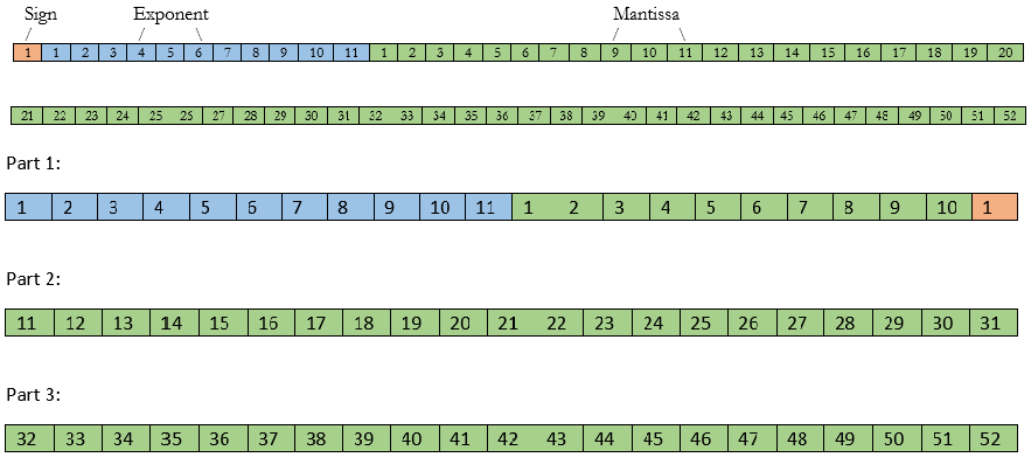

FIGURE 6.8: Asymmetrical strategy for doubles

6.4.2 Compressing

The final stage will work similarly to the one that treated floats. In this case, we have to process three different groups of bits and since we can only do one of them on the fly, the other two groups will need to be stored in arrays and compress them later. On the code for the floats we needed to store 50% of the signal but this time, we need a 66%, which with the default chunks would need 2703 KB for this process. It is still a low number but in case we wanted to use a larger chunk, we would need to consider this limitation. When the process is finished, we need to work on the decompression to be able to recover our information.

6.5 Tests and results on doubles

With all the experience acquired with the testing of the floats, we were already working with the best solution for the doubles. We did tests dividing the signal in 4 parts but it was quickly noticed that making it into 3 was better. In this case, we only had the SKA code and as it was mentioned earlier, we needed to convert this signal from floats to doubles. We would also undo the interleaving since our code relies strongly on the continuity of the signal and without doing this process, the pre-processing would show poor results.

We did not have the information on how the interleaving had been done, but after several tests we discovered it was done with 168 samples. Considering all this our compression limit for this data was 3.17. On our first approach, we got a compression ratio of 1.5, which was very far from the limit but already promising. After some analysis we found out that the ratio was lowered due to the data being unaligned. This happens when the header is not multiple of the type of data and this causes the assembling stage to fail since we are not dividing the data correctly.

When adjusting the header properly, our compression ratio went up to 2.15. This result may seem solid but not that good, considering the ideal was set on 3.17. However, the ideal result was too optimistic since we had done a conversion which was not being treated as well as it could be. The data we will compress will never come from a 32 bits one. In this case, it was definitely better to develop another algorithm which would in the end turn the signal into a 32 bits one and have a huge

compression, since the extra information added is irrelevant. But treating it as a normal signal, assuming it is all relevant information will make us stay far from the ideal but with solid results when compared with other compressors.

6.6 Post-processing of doubles

Our algorithm was providing solid results on compression, but as explained earlier, if the decompression is not possible and the recovered file is not exact as the original one all this work will be useless. We will reproduce systematically the compression algorithm backwards just as we did for the floats, but in this case, for our doubles.

6.6.1 Decompressing

We will need to read two thirds of the signal and store it, and then decompress on the fly the last one. Since the third group of the data was the one written on the fly, this will be the one we will read first. This group contains the bottom of the mantissa, the most randomized data. The second group that was written was the one containing the exponent and the sign with the first bits of the mantissa. As alredy said, this will be stored as well. The last group is the rest of the mantissa and will be decompressed while the reassembling is done.

6.6.2 Reassembling the bytes

We are now decompressing on the fly the last group of the compressed signal, which contains 21 bits of the mantissa. When decompressing them we need to look for the rest of the bits of this mantissa and rewrite it. We will need to prepare this restored information to be written, but due to internal building of FAPEC, we will need to prepare two arrays of 32 bits instead of one of 64. When these two arrays are prepared with the information in order and aligned, we will proceed to write the output file.

Writing the file for doubles is very similar to our previous case. We need to know which endianness we are working with. When we have this information, all we need to do is take the two arrays of 32 and write them dividing those 32 into strings of 4

bits. After the writing is done, we just need to compare our output file with the original one bit by bit. The process was successful and both files were exactly the same.

6.7 Comparing FAPEC with GZIP and BZIP2

In this section we compare the results of FAPEC with other popular commercial compressors such as GZIP and BZIP2. Both algorithms are very powerful. This will show us the quality of the performance of FAPEC.

It is worth mentioning that GZIP has different options for the compression, so one can choose from fastest compression to best compression. For this comparison, we chose to show the fastest, the best and one in the middle. The time of compression and decompression will be shown on a relative scale. The reason behind this is that in every computer in which the algorithms are run the time might differ considerably, but the ratio between them is the same.

File: alma.raw (268,435,456 bytes)				
Compressor	Compression Ratio	Size (bytes)	Compression Time (t/tmin)	Decompression Time (t/tmin)
FAPEC	**1.15**	**232,576,980**	**1**	1.89
GZIP1	1.07	250,579,112	2.53	**1**
GZIP5	1.07	250,579,112	2.53	1
GZIP9	1.07	250,579,112	2.53	1
BZIP2	1.04	257,321,315	8.81	8.12

TABLE 6.1: Comparison of ALMA

We start discussing the ALMA file. As it can be seen on table 6.1, FAPEC gives the best compression ratio compared to BZIP2 and GZIP. GZIP gives exactly the same ratio on all his options which is not very usual and shows that the algorithm is not able to improve its initial results.

When comparing the processing times, FAPEC is clearly the fastest compressor. All the GZIPs takes over double the time than FAPEC. It is not usual that the settings of GZIP do not take more time when increasing the value. On the other hand, the fastest decompressor is GZIP, which shows that the decompression does not have significant differences despite the level of compression. For FAPEC, the decompression time is quite close to that of GZIP. BZIP2 is the slowest on both compression and decompression, giving a very poor compression ratio.

File: ska32.bin (66,060,288 bytes)				
Compressor	Compression Ratio	Size (bytes)	Compression Time (t/tmin)	Decompression Time (t/tmin)
FAPEC	1.08	60,991,565	**1**	1.90
GZIP1	1.43	46,053,375	2.63	**1**
GZIP5	1.42	46,302,356	3.33	1.01
GZIP9	1.44	45,911,619	7.96	**1**
BZIP2	**1.75**	**37,685,603**	7.40	5.75

TABLE 6.2: Comparison of SKA (float)

In Table 6.2, we show the comparison for SKA. It is worth mentioning here that this file had been interleaved with 168 samples so its continuity was lost and we already knew FAPEC was going to have a poor performance for this file. Having stated this, BZIP2 has the best compression ratio overall. It clearly beats GZIP and of course FAPEC, which has a very low ratio. BZIP2 even beats our optimal predictions on the theoretical analysis. This happens due to the fact that our theoretical analysis is based on the algorithm we are implementing, so that limit only works for our algorithm. BZIP2 uses a completely different algorithm which seems to perform much better than the ones from GZIP and FAPEC. Strangely, GZIP5 gives worse results than GZIP1.

Looking at the CPU time they take to compress, FAPEC wins here, followed by GZIP1 and GZIP5. Surprisingly BZIP2 turns to be faster than GZIP9. However, the win of FAPEC here is quite irrelevant, considering its low ratio. However, although the algorithm is not working properly due to the interleaving of the dataset, it is still fast.

On the decompression, once again GZIP performs better than any of the other options. As mentioned before, the decompression algorithm works very similarly for all of the levels of GZIP.

In this case there is not a clear winner. If we want a high compression ratio, BZIP2 is the best option. If we want a decent compression ratio and a quick method, GZIP1 will provide us with the best results.

File: ska64deinterleaved.bin (132,120,592 bytes)				
Compressor	Compression Ratio	Size (bytes)	Compression Time (t/tmin)	Decompression Time (t/tmin)
FAPEC	2.15	61,391,074	**1**	1.92
GZIP1	2.14	61,657,073	1.89	1.24
GZIP5	2.19	60,390,124	3.51	**1**
GZIP9	2.59	51,051,083	97.22	1.16
BZIP2	**3.69**	**35,806,755**	7.98	5.92

TABLE 6.3: Comparison of SKA (double interleaved)

In Table 6.3 we can see the comparison of the SKA file converted into double and interleaved. Looking at the compression ratio we now have a clear winner. BZIP2 clearly gives the best compression ratio, beating both FAPEC and GZIPs. Once again, BZIP2 gives a better ratio than the optimal calculated for the FAPEC algorithm. We have to consider that the data is the same as on the previous case, but with a lot of redundancy and interleaved. This was done, as explained earlier, to test the FAPEC algorithm for doubles and at the same time check the results without the interleaving interfering with the algorithm of FAPEC, so it makes sense that BZIP2 is again beating both algorithms.

If we look at the compression time, FAPEC is once again the fastest one. GZIP takes at least double the time to give the same compression ratio. BZIP has a much better ratio but takes almost 8 executions of FAPEC to reach it. It is mentionable that GZIP9 takes 97 FAPEC executions to show all the power of its algorithm. While this gives solid results, it cannot compete with BZIP2.

Looking at the decompression time, GZIP5 is the fastest, with a similar execution time with all the GZIPs. FAPEC takes almost double the time to decompress and BZIP2 close to 6 times longer.

We do not have a clear winner here either. If a high compression ratio is needed, BZIP2 is the best option. If we are looking for a solid ratio with a quick algorithm, FAPEC is the best one.

Chapter 7

7 Conclusions

7.1 Conclusions

In the present work, a pre-compression algorithm and subsequently a post-compression one have been developed for FAPEC. The aim was to create a simple and quick algorithm intended to prepare floats and doubles so that in a further processing step they can be compressed by FAPEC, using its strengths in order to obtain the best results possible. A deep analysis of real data from different projects was necessary to identify the best and simplest way to process that data.

Understanding how the core of FAPEC works has been a real challenge. It was developed and programmed by many different people, which made it a complex and optimized algorithm with a fast performance. A proper understanding of it has been key to develop and integrate the pre and post-processing algorithms.

The ability to split and convert the original data at a bit level using simple operations has been critical for its analysis. Disarraying the input data and being able to

rearrange it afterwards, finding every pair of information at a bit level, has been problematic and challenging.

Another clear conclusion has been that there is a big gap between the theoretical analysis and the real implementation. When dealing with data, one can encounter many unexpected limitations which in order to be obviated require rethinking the strategy. An algorithm can always be improved and therefore it is important to figure out how to implement those improvements.

The final implementation showed a good performance for ALMA, beating all the pre-existing commercial algorithms, which has been a big success. The differences with SKA strive on the fact that one of its conditions has not been observed on our particular implementation. Being able to find a solution to the problem given the present framework without that constraint required a lot of extra work to bypass that problem. It was a challenging task which was eventually successfully achieved. The final result has been solid compared to GZIP but hard beaten by BZIP2. It definitely showed that the input data types were very different, which means our tests were covering a big sample of the floats and doubles to be compressed. In the end, all of this means that the algorithm is solid and complete.

Any small gain in compression is crucial for the systems of ALMA and SKA, which work with huge amounts of data and need a lot of storage for them. But the compression time is also very important to be able to process all incoming data, and this is what makes FAPEC better than the other algorithms. Being able to do it in at least half of the time with the same compression ratio is a huge achievement.

Finally, FAPEC has been designed to work with low computation load so it can be put into space missions. Our algorithm keeps FAPEC fast while providing more features so it can cover a broader scope of data types.

7.2 Future lines of research

During the execution of this work, multiple interesting research lines were uncovered. However, these are out of the scope of the current work but are presented here as they might be part of future essays.

The algorithm has shown to be weak when the continuity of the signal is lost due to processes like interleaving. An integration into FAPEC to deal with this issue may be needed so it can fully work with any type of data.

The final results and performance are solid but the manual parameters to be configured are complicated to be used at a user level. A system which sets automatically the best settings may be developed in the future to make it both easier to utilize and much faster to configure.

It would also be interesting to develop other pre-processing filters. In particular, a study using similar techniques as the ones used by BZIP2 but in a simpler way so the algorithm can be kept as fast as possible but the compression ratio can be better. This would allow the system to choose between performance and results in a similar way as GZIP does it but with two completely different algorithms.

8 Bibliography

[1] Hales, Antonio. "Atacama Large Millimeter/submillimeter Array" almaobservatory.org. EPO team, 2016.

[2] "SKA Project" skatelescope.org. SKA organization, 2016.

[3] Dodson, R., Vinsen, K., Wu, C., Popping, A., Meyer, M., Wicenec, A., Quinn, P., van Gorkom, J., Momjian, E. 2016, "Imaging SKA-scale data in three different computing environments", Astronomy and Computing, 14, pp. 8-22.

[4] "IEEE Standard for Floating-Point Arithmetic", IEEE Computer Society, 29 Aug. 2008.

[5] "GNU Gzip: Advanced usage", 28 Nov. 2012.

[6] Feldspar, Antaeus. "An Explanation of the Deflate Algorithm", 23 Aug. 1997.

[7] "The LZ77 algorithm", Data Compression Reference Center: RASIP working group. Faculty of Electrical Engineering and Computing, University of Zagreb. 1997.

[8] Huffman, D. "A Method for the Construction of Minimum-Redundancy Codes". Proc. IRE 40, 1098-1101. 1952.

[9] Seward, Julian. "Bzip2 and libbzip2". 27 Sep. 2008.

[10] Portell, J., Villafranca, A. G., García-Berro, E., "Quick outlier-resilent entropy coder for space missions". 2010.

[11] Portell, J. "Fapec Core 2015.1 User Manual". DAPCOM Data Services S.L. 9 Nov. 2015.

[12] Golomb, S. W. "Run-length encodings", IEEE Trans. Info. Theory 12, 399-401. 1966.

[13] Rice, R. F. "Some practical universal noiseless coding techniques", JPL Tech. Rep., 79-22, Jet Propulsion Laboratory. 1979.