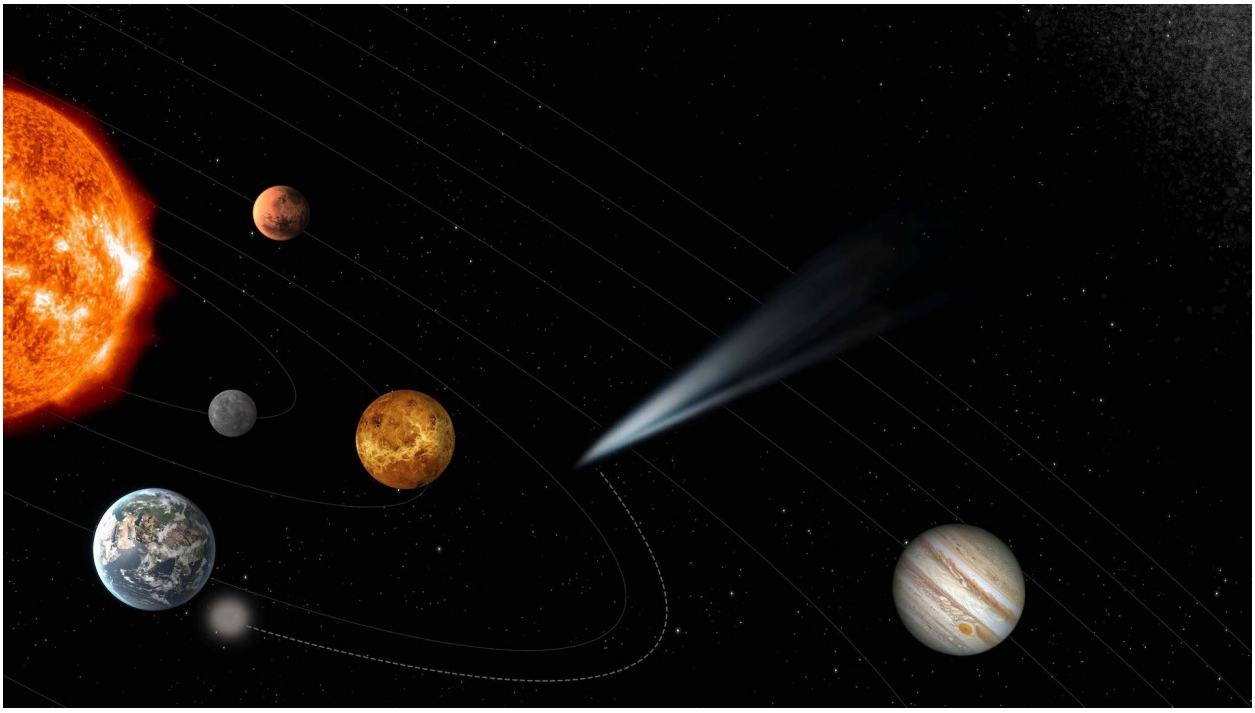

Compression of Space Magnetometer data using Differential coding

Name: Taran Attavar

CID: 01352694

Project code: 30

Supervisor Name: Christopher M. Carr

Assessor Name: Marina Galand

Word Count: 3246

Blackett Laboratory, Imperial College London, SW7 2BW, United Kingdom

Abstract

Acknowledgements

I would first like to thank my project partner, Parth Patel, for his many significant contributions to this project. He was responsible for most of the work on the Huffman and Golomb coding aspects of the report while I worked on the Differential coding part along with improving the graphs and making them look more readable. In addition to this, we had a good understanding of the work each of us had undertaken and this led to good progress on the project as a whole. We also assisted each other in our respective parts of the project if we felt confused or unsure along the way.

I would also like to thank our project supervisor, Christopher Carr, for his continued guidance on how to proceed with the project and also, providing us with the solar wind data on which to test our compression algorithms. He was always eager to arrange meetings with us to discuss our progress each week and point us in the right direction.

Table Of Contents

I Introduction

This project investigates existing compression algorithms and their compression ratios on space magnetometer data. The motivation for this project comes from the ESA Comet Interceptor (classified as an **F**-type mission) where the comet data would be measured within ~ 2 hours. From this, we find the need to compress the raw data for faster transmission. Our project is different from other fly-by missions in which the magnetometer data would evolve as a time-series. We would then have to consider different algorithms to compress the time-varying aspects of the data.

Similar efforts for compressing space magnetometer data have been undertaken, the most notable of which is the Bepicolombo mission. The compression algorithm used for that particular mission was a combination of Huffman, Differential and Golomb-Rice coding. That mission achieved a compression factor of ~ 0.6 but did so without a time-evolving data series. In this project, we want to see if we can achieve similar compression factors but for time-series magnetic field data.

II Background

The fundamental concept of this project is compression and the simple idea that compressed data (when done right!) is quicker to transmit. Before delving into the different compression techniques used in this project in detail, we first have to understand compression and its different types.

There are 2 types of compression: lossy and lossless compression. Lossy compression techniques ignore less useful information or repetitive data to compress the entire data set. These techniques are likely to offer higher compression factors but at the cost of raw data. In this project, we are looking to ensure that raw data can still be retrieved after decoding and so instead, we are only considering lossless techniques. Lossless compression, on the other hand, use statistical redundancy to store data in fewer bits. There are many techniques including Huffman coding, Lempel-Ziv (LZ), arithmetic coding, etc. Most compression software used by consumers are often a combination of these algorithms, e.g. `.zip` files use a combination of Huffman and LZ77 coding.

As mentioned above, this project only considers lossless techniques and we will only be looking at individual algorithms and how they perform on their own due to the time constraint. However, it is mentioned in Section V how this project can be extended by considering combinations of the techniques discussed later.

In this project and by convention, the level of compression is measured using a compression factor which is given by the following expression:

$$\kappa = \frac{d_\mathrm{c}}{d_\mathrm{u}} \tag{1}$$

where $\kappa \to$ compression factor, $d_\mathrm{c} \to$ compressed file size and d_u uncompressed file size. From this formula, it is implied that lower the compression factor, higher the compression, i.e. better!

Before moving on to the different techniques investigated, let us consider the data used for this project. Since, we do not have any actual data, solar wind data has been used and it has been taken into account that the actual raw data would be in the form of 14-bit signed integers. This information is key because most techniques exploit this which means that compression is data-dependent and so, this leads to another extension to obtain simulated data under actual mission conditions again discussed in Section V.

For this project, we assume $d_\mathrm{u} = 14 \times 3 \times N$. The value 14 comes from the fact that the data is 14-bit and the 3 comes from the 3 magnetic field axes since this is the total uncompressed file size. N is the number of samples chosen from the data set and we have choice in what this value can take. Ideally, we want it such that it yields optimal compression factors but to find the optimum N is beyond the scope of this project for some of the techniques used.

Lastly, we find that the magnetic field evolves with time since the satellite orbiting the comet

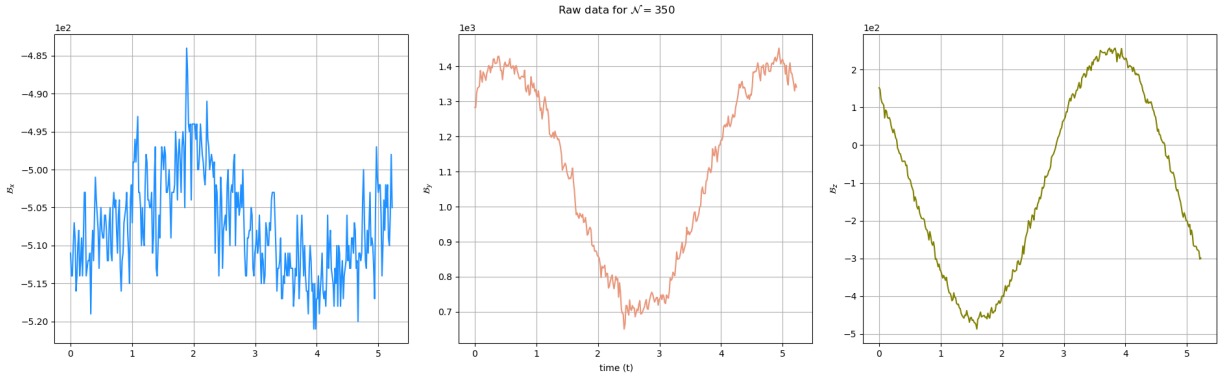

Figure 1: Raw solar wind data with sample size $N = 350$

would be rotating and so, this makes our project unique since we would be looking at compression on time-series data which adds another layer of complexity to the code and almost certainly impacts the final compression factors acheived. A small sample from the data set has been shown in the figure below (Fig 1) to illustrate how the data evolves with time.

III Compression Algorithms

In this project, we will look at 3 compression techniques in great detail: Huffman coding, Differential coding and Golomb encoding. We will look at how each technique works and how they differ from each other by investigating their pros and cons and in the following section, discuss the compression factors achieved. Rather than include the code in this report, I have created schematics of how each technique works which is far easier to comprehend.

Before discussing the techniques investigated in the report, it should be noted here that other algorithms were also considered and initial tests were conducted using existing Python libraries. The tests conducted on these libraries gave us an idea of the compression times of different libraries on the solar wind data. These libraries include Lempel-Ziv, i.e. LZ77/LZ78 on `lzma`, `zlib` and `snappy`.

3.1 Huffman Coding

Huffman coding uses the frequency of the characters in a string to assign a Huffman code, i.e. in binary representation, to each character. The length of the Huffman code depends on the number of characters in the string and since, we are looking at numbers, we will have 10 characters to encode, i.e. 0-9 and a -ve sign. The following schematic shows how Huffman coding on a simple string.

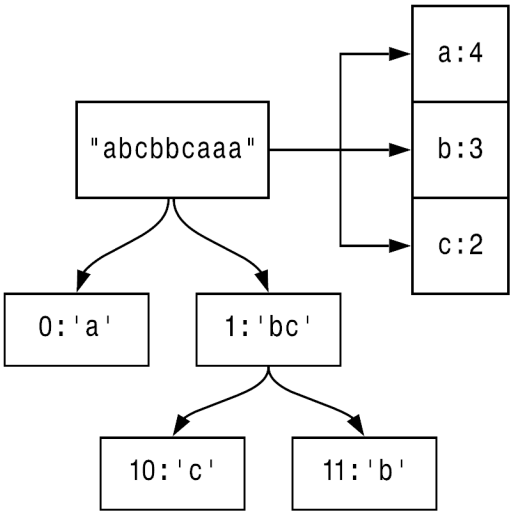

Figure 2: Schematic of Huffman coding on single string

Initially, we assumed that Huffman coding would be fundamental to our compression routine. However, it was found that differential coding

(discussed in Sec 3.2) ended up giving better compression ratios than Huffman. Huffman coding works on strings and so, in our case, the data would have to be encoded as a single string with all the data points essentially "strung" together and then applying the algorithm described in the schematic above. This would then have to be decoded as a string and then converted into 14-signed bit integers which further complicates this algorithm.

Lastly, another crucial drawback of Huffman coding is the complexity of the code and the need for memory space to store the mappings of each character used, which makes it less suitable for the microprocessor to be used on the ESA Comet Interceptor.

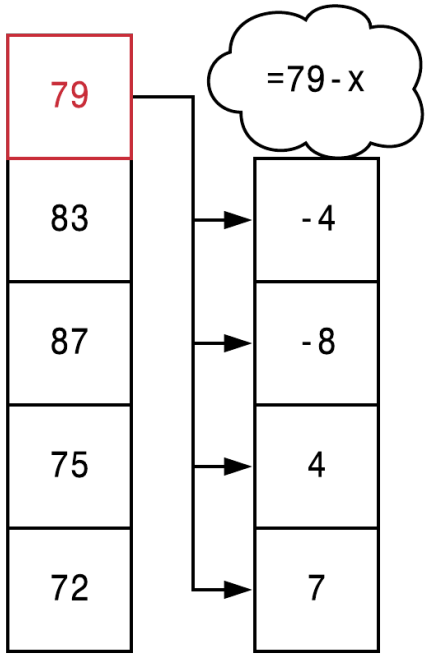

Figure 3: Schematic of Initial Value Coding

3.2 Differential Coding

This compression technique simply uses arithmetic operations, i.e. subtraction in this case, to reduce the number of bits required to store the data. We know that the raw data is in the form of 14-bit signed integers but how many of those bits are actually being used and can we reduce this by removing the redundant bits for each axis of the magnetometer data.

We can separate this technique into types: Initial and Previous. Initial value coding uses the value of the starting number in the data "packet" from which the remaining values are subtracted and hence, stored in fewer bits. Previous value subtracts each number from the next which yields a smaller number to be stored. In both cases, only 1 number from the packet needs to be stored in its original form.

3.2.1 Initial Value Coding (IVC)

From the name itself, it can be inferred that the initial value determines the subsequent outputs for the remaining data set. The following schematic shows how initial value coding works by considering a few arbitrary numbers.

As we can see, all of the remaining numbers are subtracted from 79, i.e. the initial value in this case. The values derived from the subtracting usually require less bits to store and in this project, we would be looking at each axis, i.e. x, y, z- axes, data separately. The aim would be to find the maximum no. of bits that each axis requires and then, encode the values accordingly.

3.2.2 Previous Value Coding (PVC)

Previous value coding differs from IVC in the fact that rather than have a single fixed value from which the remaining data set is subtracted, each value yields a quantity denoted by $x_j - x_i$, i.e. the given value subtracted by the next. The following schematic further aids in the description of this technique using arbitrary numbers.

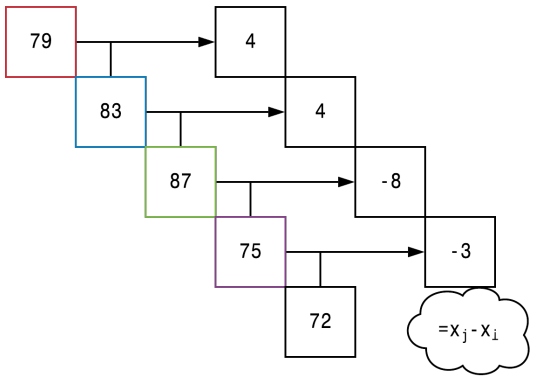

Figure 4: Schematic of Previous Value Coding

From an initial inspection, the results should show that previous value coding yields better results for our data in particular. This is because we are looking at a time series data and so for IVC, the difference from the initial value would gradually grow larger as we increase the sample size of the data. With PVC, the difference remains small throughout since we only consider the following value. It should also be mentioned here that there aren't any large fluctuations in the data, hence allowing us to make this deduction and further discuss this in Sec IV.

3.3 Golomb Encoding

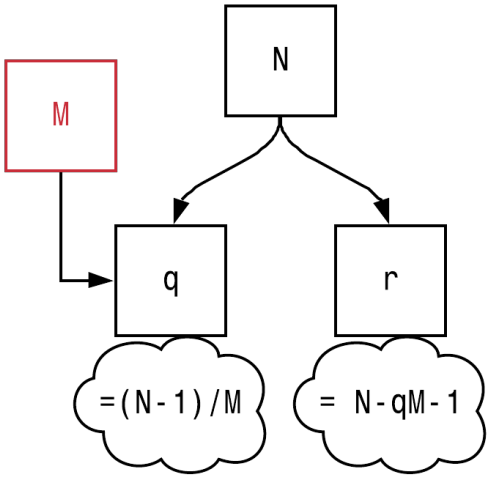

Figure 5: Schematic of Golomb Encoding

Golomb encoding allows for a **tunable** parameter, i.e. M-value in this case, to divide a number into a quotient and remainder. The following schematic describes the process of Golomb encoding on an arbitrary number.

From the figure above, we can see how the input parameter M affects the overall encoding. The value of q is stored as Unary coding (0-0, 1-10, 2-110, etc.) while the remainder r is stored as truncated binary code. The aim for this technique is to find the optimum M-values that minimises the number of bits required to store data for each of the axes.

It should be noted that in the Bepicolombo mission, Golomb-rice coding, i.e. a specific type of Golomb encoding, was used which means that the M-value was taken to be a power of 2. However, we found that there was a negligible difference and so, larger increments had to be taken.

IV Results

This section looks at the results obtained by trialling the compression algorithms on the solar wind data provided to us. It should be noted that, in this project, we are dealing with a large amount of data so instead of compressing the entire data set, we though it best to take only a small sample size from the data and see how well the compression techniques work. For some of these techniques, we look at how the sample size affects the compression factor. It was sometimes not possible to conduct this on all given the large number of iterations involved in the code.

Furthermore, the number of bits for both the compressed and uncompressed files were found using the built-in `sys` library in Python. This was used consistently throughout the project. The actual numbers here may vary if other functions that return the file size are used. However, the compression factor will remain constant and so this is a good measure of the compression of a given algorithm.

4.1 `zlib`, `lzma` and `snappy`

This section has been included to discuss the initial results obtained by implementing existing Python libraries on the data. Prior to looking at individual algorithms, we looked at existing libraries that mostly used a combination of algorithms to compress the data.

`zlib` works on provding a compression level ranging from 0 to 9, with 0 $\rightarrow$ no compression and 9 $\rightarrow$ highest compression. Tests were done measuring the time taken and the compressed file size for each level.

4.2 Huffman Coding

4.3 Differential Coding

Following from Huffman coding, we found that this technique was not quite suitable on its own and did not achieve the target compression ratio. Moving on to Differential coding, we will see the results both initial and previous value coding compression on our data.

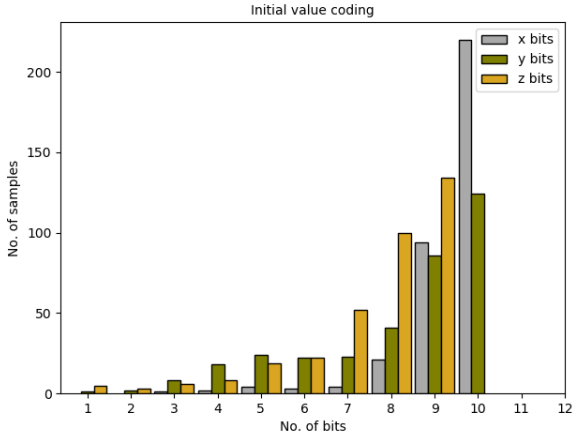

Figure 6: Initial value coding on a sample size of $N = 350$

In the above figure, we find that this particular sample uses a large number of bits to store B_x data which lead to an overall poor compression factor. For the B_y and B_z data, the maximum number of bits required to store them is 10 and 9 respectively. This is only for a single random chunk from the data but we will show later how the maximum number of bits for each axis changes when we consider different chunks from the data set.

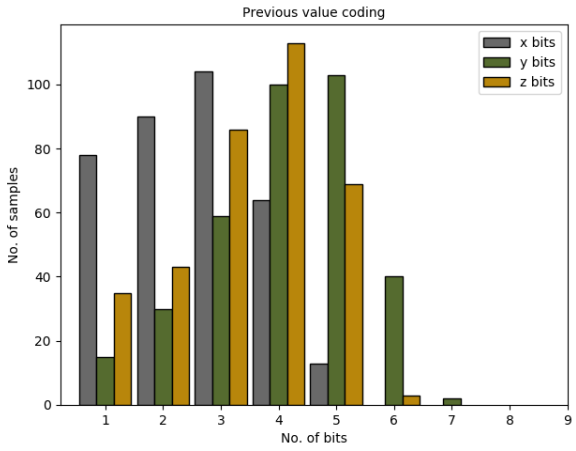

Figure 7: Previous value coding on same sample data as Fig 6

From the above figures, it can be inferred that the previous value differential coding requires nearly half as many bits as the initial value coding. We had expected this result since the data doesn't fluctuate much over short time intervals and so the differences between each value is always small and hence, require fewer bits.

In the diagram below, 12000 trials are shown where each trial is a chunk of 350 samples that are chosen randomly from the entire data set. The number of bits shown in this figure are the **maximum** number of bits for each axis in a single trial. The histogram represents how often a given number of bits is the required number for each axis.

It should be mentioned here that we are only looking at the maximum number of bits in each trial. If we were to compress each data point separately, we would need to store the information of the number of bits required along with them and it would significantly complicate the compression process and make it difficult to identify from the compressed packet the number of bits to decode for a given value.

For initial value coding, the algorithm compresses both the y- and z- components from 14-bits to $\sim$ 10 to 11 bits and sometimes even 12. In contrast, the x-component varies significantly across different trials, ranging from 5 to 10 bits.

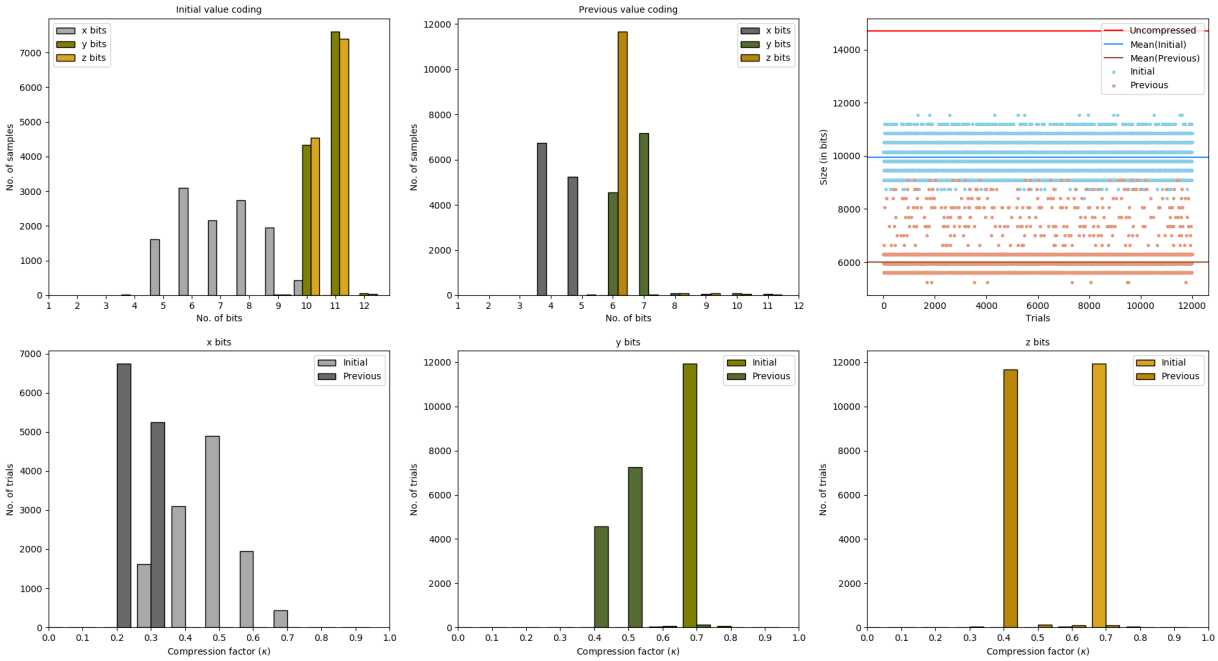

Figure 8: Compression results across 12000 trials for both differential coding techniques

This is reflected in the plots of the no. of trials versus κ. For the x-component, $\kappa \sim 0.3$ to 0.6 which reflects very high compression whereas the y- and z-components only have $\kappa \sim 0.7$.

These values are certainly very impressive given that the Bepicolombo mission achieved $\kappa \sim 0.6$ which was our target compression factor for this report. The algorithms written for this report were tailored towards space magnetometer data and relied heavily on the input data and hence, could be the reason we achieved high compression.

Compared to a single trial, the results for multiple trials are more or less consistent with those in Figs 6 and 7 which shows that for $N = 350$, choosing a random sample from the entire data set does not significantly affect the final compression factor. In this light, it should also be mentioned that the

Owing to the large number of iterations and time taken to produce the results, only a sample size of $N = 350$ was investigated. However, the code does allow for the value of N to be changed easily and this can also be considered in further extensions of this project. We also find that the results are in agreement with the fact that one component of the magnetometer will be aligned with that of the comet.

4.4 Golomb Coding

The final compression algorithm to be investigated is Golomb coding which, as discussed previously, stores the remainder and quotient from dividing the input data with a chosen M-value. The results that follow investigate the "correct" choice of M for our data.

In Fig 9, we find that there exists a minimum/optimum point for the graph of d_c v/s M-value where $d_c \rightarrow$ compressed file size (in bits). We obtain a different M_{opt} value for each axis and with a simple change to code, we can use separate M-values for each axis further improving the possible compression factors. The following table summarizes the figure to show the optimum M-value for each axis.

B-field axis	M_{opt} value
x	2
y	15
z	7

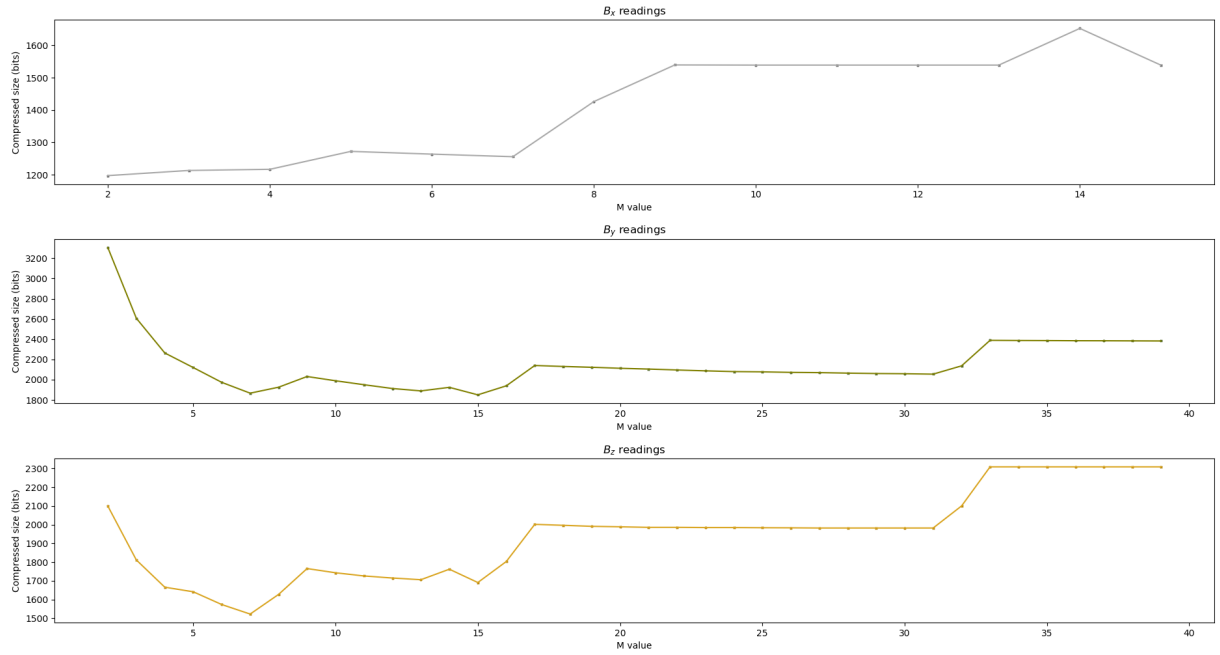

Figure 9: Graphs of the number of bits required to store a sample size of $N = 350$ for different M-values for each of the magnetic field axes

Table 1: Optimum M-value for each axis

The satellite for the Comet Interceptor mission is to be oriented such that one of its axes is aligned with the comet so it remains more or less constant, i.e. we see this with the x-axis data. Consequently, the other 2 axes measure data such that one axis lags the other by a phase of $\frac{\pi}{2}$. This implies that we should obtain similar M-values for both the y- and z- axis but instead we find that $M_{\text{opt}} = 15$ for y and $M_{\text{opt}} = 7$ for z. This difference is due to the absolute values of the readings, as shown in Fig 1. The y-axis data has been shifted upwards by some factor which results in a different M-value. If we were to shift this back down, we would obtain a similar M-value for both the y- and z- axes since their data evolves almost identically with time, accounting for the phase lag.

This result highlights the level of dependence on the raw data and how even the actual values of the data can affect the overall compression factors for each algorithm. Ultimately, the data received from the Comet Interceptor mission should not show this shift but the aim of this project is also to identify potential issues with the data and compression algorithms and see how they can affect the results.

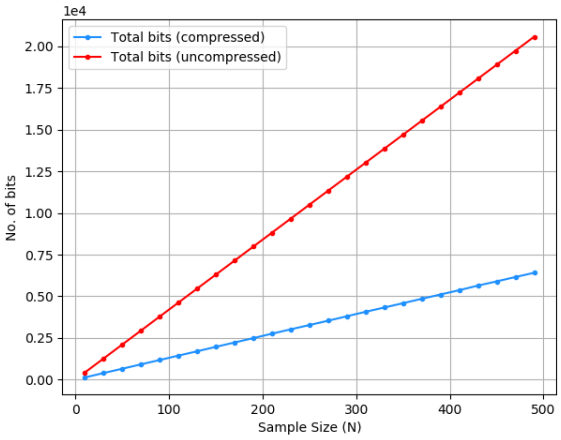

Figure 10: Graph of total no. of bits v/s sample size with $N \in [10, 490]$

The figure above shows how the total compressed and uncompressed bit size evolved with N, i.e. sample size, in a range of 10 to 490 samples. It shows that both the compressed and uncompressed bits increase linearly with sample size

(although with different gradients). This may look trivial, however, from the plot, it was found that calculating κ at each sample size yielded almost the exact same compression factor, i.e. $\kappa \sim 0.3$! This implies that κ is independent of N allowing any sample size to be taken for Golomb coding. In this case, only limits to the transmission of the on-board instrument will determine an optimum value of N which should be tested prior to the mission.

Finally, the results discussed in this report only look at one data file out of three. The algorithms were also implemented on the other two files and both showed very similar compression factors to those calculated here.

V Potential Extensions

In this section, we will discuss future work that can use this project as a platform to carry out further research. Firstly, it might be useful to investigate the effect of a combination of algorithms. As we have seen in this project, Huffman coding is not suitable for this type of data. However, the Bepicolombo mission used all 3 techniques discussed in this report in tandem with each other to achieve a ~ 0.6 compression factor. Hence, it would make sense to consider a combination of these techniques.

Another possible extension for this project would be to simulate the data received from the actual mission by modelling the mission and the conditions under which the data would be collected. This would then allow work to be conducted using data that closely resembles actual magnetometer data rather than use solar wind data.

Lastly, this project was done under a time constraint so we felt that with more time, we would like to have tested the time taken for the compression, using these simplified algorithms in Python, for each technique. This would then allow us to extract the optimum parameters for compression, e.g. M-value in Golomb encoding, that minimises the time taken to compress and therefore, improves the latency.

VI Conclusion

For the actual mission, the algorithm would be written in C on a microprocessor and so, it is also important to consider the complexity of the code due to limitations of the equipment. As we see, Golomb encoding does offer extremely favourable compression factors, however, due to its complexity, it might be worth to use Differential coding, i.e. PVC, instead for its simplicity alone.

VII Bibliography

1. D. Fischer, G. Berghofer, W. Magnes and T. L. Zhang, "*A lossless compression method for data from a spaceborne magnetometer*" 2008 6th International Symposium on Communication Systems, Networks and Digital Signal Processing, Graz, 2008, pp. 326-330. DOI: 10.1109/CSNDSP.2008.4610829

VIII Appendix