2021

FINAL REPORT

EEE3097S – DESIGN SUBMISSION 4

JESSICA ANDREW (ANDJES006) &
THEODORE PSILLOS (PSLTHE001)

Table of Contents:

(i) Table of Figures:

1. List of Figures:

2. List of Graphs:

3. List of Tables:

A. Admin Documents:

1.1. Individual contributions:

Both of us worked equally on this project and contributed to every submodule, however, there are certain submodules that either member concentrated on. These are listed below:

Name	Contributions	Section Numbers	Page Numbers
Theodore Psillos	Contributed equally on the Paper Design, Sense HAT B research and code and the remaining write ups. Focused on the compression subsystem. As most of the work was shared equally only the compression sections will be given to the right.	C.1.3 C.3.1 D.2.2 D.3.2 D.4.1 E.2.2 E.3.2 E.4.1 F.1.1 F.1.3	Pg 12, 23, 30, 33, 42, 50, 60, 71, 73,74
Jessica Andrew	Contributed equally on the Paper Design, Sense HAT B research and code and the remaining write ups. Focused on the encryption subsystem. As most of the work was shared equally only the encryption sections will be given to the right.	C.1.2 C.3.1 D.2.3 D.3.3 D.4.2 E.2.3 E.3.3 E.4.2 F.1.2 F.1.4	Pg 10, 23, 31, 38, 42, 51, 67, 72, 73, 74

Table 1 - List of Contributions

1.2. Project Management Tool:

Below is a screenshot of the various tasks that we had for our final submission. We were unable to display our tasks on timeline as it required us to use a paid version, however, the dates are still visible.

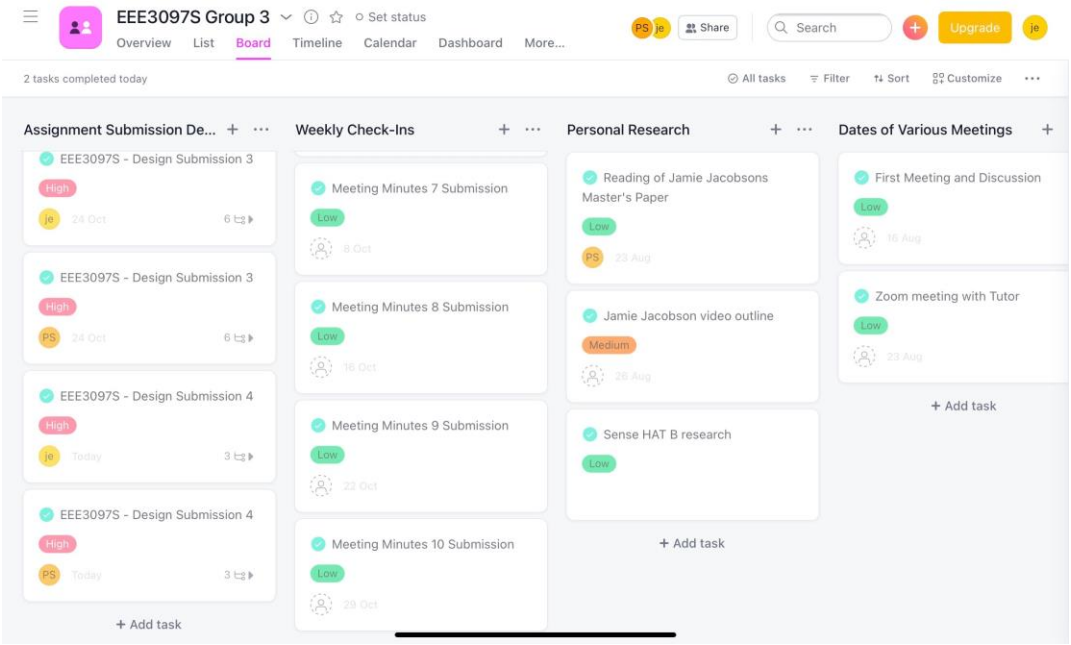


Figure 1 - Timeline of our Tasks

You can also find the link to our project management site on Asana below:

https://app.asana.com/0/1200794780560566/list

1.3. GitHub Link:

We used the GitHub repository more than we did for our paper design submission. This was partially since it is a nice way to collate all the code that we used to test and implement the various encryption and compression techniques. Furthermore, it is incredibly easy to transfer files to the Raspberry Pi by cloning the repository, which allowed us to test the various implementations on the environment that it would need to perform appropriately in. The link to the repo is below and there will be many other links throughout this document that refer directly to code files.

https://github.com/JessicaAndrew/EEE3097S-Group-3.git

1.4. Timeline and Progress:

The project was completed within the time line. With the paper design, progress report 1, progress report 2 and final report all being completed. Additionally, the paper design, progress report 1 and 2 were further edited using the marked and returned versions and then compiled into the final report. The final report and demonstration video were then compiled.

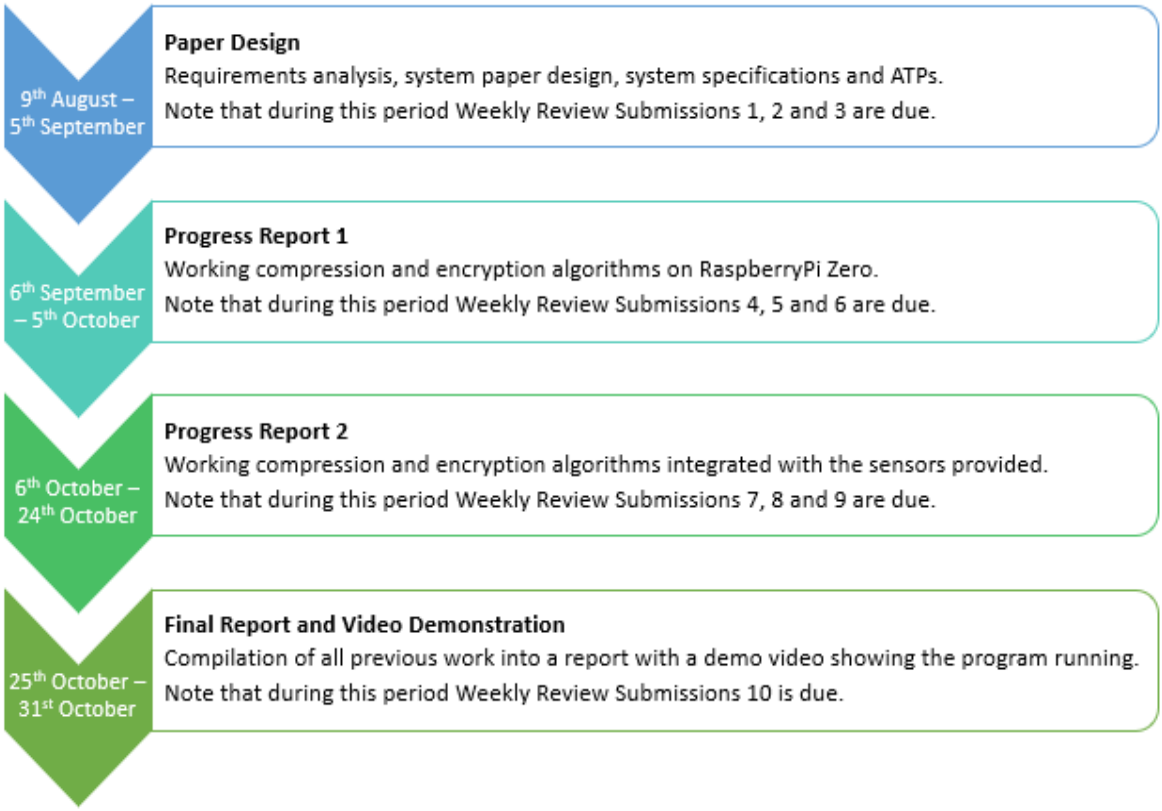


Figure 2 – Timeline of the Project

B. Introduction

An ARM based digital device, otherwise known as a Raspberry Pi Zero, is used to compress, encrypt, and transmit all relevant data packets from an inertial measurement unit (IMU) for further analysis of the environments surrounding the Antarctic region. This Pi Zero design system is one of many subsystems that make up the SHARC BUOY designed by Jamie Jacobson.

The data used within the compression and encryption process was done in two steps. The first step was using data provided to use from the authors of the loken_21 papers from the IMU they used. The second step consisted of using data obtained from the Sense HAT B which was attached to the Raspberry Pi Zero and used to fetch data which was outputted to a csv file for later compression and encryption. The data from both the loken_21 papers and the Sense HAT B was run though a multitude of tests to see how it plots in both the Fourier and time domains. This was to test the effect of sampling rate on the noise seen and to identify whether the Sense HAT B was running as expected (with regards to temperature, pitch, yaw and roll).

Various methods of compression and encryption are compared to determine the most suitable algorithm for this application. First a theoretical comparison of the algorithms was made and then some were chosen for testing in implementation. This were then compared on multiple levels (such as speed and compression ratio) to determine the most suitable algorithm for the type of data outputted by the IMU.

The compression algorithms tested were the bzip2 (bz2), GNU zip (gzip), zip, zlib and Lempel–Ziv–Markov chain algorithm (lzma). All these algorithms were based in some form or another on the zlib algorithm. In light of this, for the set of experiments that were conducted the zlib compression technique was used as the golden measure. These experiments consisted of the compression of files with different sizes (measured in bytes) using the different compression techniques, the measuring of the execution time of the various compression algorithms and comparing the decompressed file to the original file for every technique. The first experiment was used to determine the various compression ratio and percentage compression of each algorithm. While second experiment tested the speed up ratio of each of each algorithm, the last experiment determined whether each technique was lossless or lossy. All these results were fundamental to determining the technique that was used for the designed system and whether they met the various subsystems Acceptance Test Procedures (ATPs).

The encryption algorithms tested were a Reverse Cipher encryption, Data Encryption Standard (DES) and the Advanced Encryption Standard (AES). The reverse cipher was used as a golden measure due it being seen as one of the most primitive types of encryptions. These algorithms were tested on the following factors: execution time (to be used to display battery usage), security level of encryption, whether the encryption is lossless and the RAM usage of the system on the Raspberry Pi Zero. These factors were chosen to encompass the needs of the relative Acceptance Test Procedure (ATPs) and to test the various algorithms for their suitability in this project.

Additionally, the user requirements, functional requirements and specifications of the design are defined with a list of acceptance test procedures highlighted, in which the designs are tested against the design specifications. Finally, the compression and encryptions blocks are combined and tested for correct execution prior to the full implementation of the system on the Raspberry Pi Zero.

C. Requirement Analysis & Paper Design:

1. Requirement Analysis:

1.1. Interpretation of User Requirements:

The following user requirements were derived from the design project scope as well as the thesis on SHARC BUOY's. Each of these requirements will be expanded into functional requirements and thus design specifications in the sections to follow. They are listed in no particular order:

User Requirement ID	Description
UR001	User requires the lowest 25% of the Fourier Coefficients as usable.
UR002	The data must be encrypted.
UR003	The data must be compressed.
UR004	Minimise the power usage and maximise battery life longevity.
UR005	A motion tracking device must be used.
UR006	The design should be adaptable depending on the Antarctic environment that it is in.
UR007	Any programming language suitable for this application may be chosen.
UR008	A Raspberry Pi board must be used for testing implementation.
UR009	Data must be transferred as soon as processing is done.

Table 2 - Interpretation of the User Requirements and their Descriptions

1.2. Comparison of Encryption Algorithms:

The various encryption types start with symmetric versus asymmetric [1]. Symmetric encryption is when there is a single key that encrypts and decrypts the data. It is therefore important that it remains secret on both sides. Asymmetric encryption on the other hand uses multiple keys. There are specific keys to encrypt and decrypt the messages. This type of key is called a cipher [1] which has two publicly known algorithms (one for encrypting the data and one for decrypting the data) and a secret key that is used in this process. Within ciphers there are two types: stream ciphers and block ciphers [1].

Stream ciphers are where each message bit is xor'd with the key's bits to both encrypt and decrypt the data [1]. Due to this operation type the key and message length need to be the same. To make this process more

realistic pseudorandom generators are used by taking in a string (the key) and producing a much larger string in return to do the xor operation. Note for this encryption implementation to remain secure the key can only be used once because if it is used again the secret key can be deciphered [1]. This is made feasible by adding a nonce to the data being sent over that is then added to the secret key which is then used to decrypt the data.

Block ciphers on the other hand take in the input and iteratively encrypt it using a round key which then outputs a same length text [1]. There are two common types of block ciphers: Triple Data Encryption Standard, 3DES [2], (works with inputs of 48 bits) and Advanced Encryption Standard, AES [3], (works with inputs of 128 bits) [1]. The round key works on a key expansion mechanism so that every new round there is a unique key. AES is a substituted permutation network [1] where the process iteratively xor's the key with the current message, then substitutes data based on the given table and finally bits are moved around in an ordered process. This system iteratively repeats this process ten times until a final message is outputted [1]. Decryption of this output is achieved by repeating the above process in reverse. Block ciphers have modes of operation where the Electronic Code Block (ECB) [1] is a commonly used mode. Modes of operation are used to go around the requirement of block ciphers where they only take fixed length inputs. ECB divides the incoming message into 16-byte sections which are then processed individually. This can have a large speed up when done in parallel but if the message repeats itself the encryption is no longer secure. To make this process semantically secure a better operation to use is deterministic counter mode if one-time key modes are wanted [1]. A mode of operation that allows keys to be used multiple times can also be used. Cipher Block Chaining is an example of this. It is a process that links together each 16-byte block by xor'ing the current ciphertext (starts as a random vector) and the current plaintext block and then doing a block cipher encryption process [1]. Decryption is achieved by reversing this process. Another mode of operation is Randomised Counter Mode [1] which is the most secure option of all the ones given. Where an iterated version of the randomised vector is combined with the key until it is if the message. Which is then xor'ed with the message to result in the ciphertext [1]. To decrypt this ciphertext, the entire operation needs to be repeated.

The AES encryption method is easy to implement on an Arduino or a Raspberry Pi as it already has existing libraries [4] for the encryption and decryption functionalities. This library allows for the input of any size message by using Cipher Block Chaining mode of operation.

1.3. Comparison of Compression Algorithms:

The two main branches of compression are lossless and lossy compression [5]. In lossless compression the file size is compressed while the quality of internal data is not lost. Therefore, at the decompression stage all the original data is retained. Lossy compression on the other hand permanently alters the data by removing parts to make it smaller in size [5]. Lossy compression is generally more suited to data such as images while lossless compression is generally better for text.

LZ77 is a compression algorithm [6] that is lossless using the sliding window method. The main idea behind this algorithm is to replace multiple occurrences of the same byte sequences with a reference to its first. This is done by having a dictionary (portion already encoded data) that contains sets of three values. The offset (length between phase start and file start), run length (phrase length) and deviating characters (new phrase markers). When a file is passed through the dictionary updated to reflect the compressed data. The sliding window has search (contains the dictionary) and look-ahead (portion of data about to be encoded) buffers. The sliding window's size is an important aspect in the amount of compression that can occur [7]. The trade-off is between speed of compression and the compression algorithm effectiveness. The longer the sliding window the higher the compression of the file but the longer it will take to compress. In practice the size depends on the data to be compressed and the sliding window can be anything from kilobytes to megabytes in size [7].

The LZR [6] is a modification of the LZ77 algorithm that operates in a linear sequence. If used in a nonlinear fashion it needs significantly more memory than LZ77 as is therefore not optimal.
LZSS [6] is another compression algorithm that is an extension of LZ77. It includes that additional functionality of checking whether a compression substitution decreases the file size to get the smallest size possible. The dictionary also does not include deviating characters which decreases the size of the dictionary.

Deflate is the final compression algorithm [6] that is going to be evaluated. It is a combination of Huffman coding pre-processor and the LZ77/LZSS algorithms. Huffman coding is an encoding method that evaluates frequency of certain characters and then assigns codes to these. The codes are of changing length, this length depends on the frequency of the characters they represent with the more common the characters the shorter the code [7]. Each code can be uniquely decoded to allow for unambiguous decryption. This propterter is ensured by making the codes prefix codes so that the charater's boundaries are clearly set. Huffman coding is done in two phases. First a binary tree is constructed from the original data. The tree is then gone through systematically adding the codes to the characters. Huffman codes have two

implementations, static and dynamic. Static codes are where, for all the input data, the same codes are used. While the dynamic codes are created by examining data sets and determining the typical code sizes. The data sets are created by breaking the input data into blocks of data which is then processed separately [7]. Zlib [7] is a library that is used to implement lossless compression and decompression of data. This library is written in the C language. The Deflate algorithm is used to compress data while the decompression is done using the Inflate algorithm [7]. Zlib has multiple compression levels that range from 0 to 9 [7]. As the level number increases the compression ratio is increased while the speed of compression is decreased. The Robin-Karp algorithm is used for searching the text for matches. The length of matches found can also be optimised so that the code for a match is no longer than the match itself. This algorithm uses a rolling hash format to identify patterns within the text in an efficient way [7]. The Deflate method uses Huffman encoding by allowing for both static and dynamic encoding. It runs both and then chooses the method that provides the best compression [7] (if the compression is the same, the static method will be chosen as its decoding method is quicker). This is done on a block-by-block basis and therefore in an entire data set different blocks could have different encoding types. Zlib [7] also makes use of multiple buffers to control input/output performance and store data blocks.

1.4. IMU (ICM-20649) Data Capturing and Processing:

An inertial measurement unit (IMU) measures and captures the angular rate and acceleration experienced by the object it is attached to. In this application the object would be the SHARC BUOY. A key accelerometer feature of the ICM-20649 is the low power mode operation of the applications processor. Additionally, the digital motion processor (DMU) feature enables an ultra-low operation level of the calibration of the gyroscope, accelerometer and compass while still maintaining peak performance of the sensor data generated [8]. These features mentioned above play an integral role in maximising the longevity of the battery used to power it and the other features of the SHARC BUOY. To simulate best the data transfer between the IMU and PiZero without a real IMU, MathWorks or Aceinna Navigation Studio may be used [9]. The sampled data rate may be calculated as 25% of the actual sample rate of the IMU to ensure a realistic simulation. This is fundamental when testing the various compression and encryption algorithms, which will be discussed further in the acceptance test procedure method.

1.5. Feasibility Analysis:

We will be able to deliver on all the requirements except 1 - the IMU. We do not have access to an IMU; however, we will be able to simulate an IMU to get data that would be expected had the Raspberry Pi Zero been connected in the SHARC BUOY. We both have access to Raspberry Pi Zero Boards and have computers that will make it easy to interface with them. Furthermore, many of the algorithms that are needed for the two submodules, compression and encryption, are easily available on the internet and can be downloaded onto the Pi boards via WiFi.

We are, however, still inexperienced with the Pi boards having just started learning how to use them for over a month and we may run into issues that would require significant research or external help to solve. Additionally, we feel that the filtering of the IMU data so that 25% of the coefficients are extracted may be particularly challenging. From brief research we are aware that it is possible to perform a fast Fourier transform on the IMU output data in our chosen programming language, C. Despite this, we do feel that our lack of C experience may limit us but if there is one thing that this degree has proven to us is our ability to solve problems, we have had no prior knowledge of at all. All in all, we have the resources to meet all user requirements of the design project.

1.6. Possible bottlenecks:

There are multiple trade-offs that have already been mentioned above and are yet to be mentioned. All trade-offs involve time in one way or another. The various trade-offs we expect to encounter are listed in bullet point form below:

- Compression level vs time to compress
    - The more compressed the file, the longer it takes to run the corresponding compression algorithm.
- Choice of encryption algorithm vs data needing to be encrypted
    - Different encryption algorithms have different suitabilities. Some algorithms are better suited to finite length data, while some can handle different length data streams. This choice can affect the time it takes to perform the encryption.
- Processing time vs power used
    - The data needs to be preprocessed to pull out only the needed data, but this processing takes time and the more accurately it is done the longer it takes. There is a trade-off between time to process and depth of processing.
- Data collection vs processing time

- The amount of data that is in each transfer packet needs to be decided. How long does the code wait before it has enough data to effectively process and then transfer. The larger the data collected per packet the longer it will take to process each one and then transmit it. If the wait period is too long, the data that could have been sent may be lost if the buoy only transfers occasionally and is lost before it can transfer a significant amount of data.

- Filtering subsystem
  - This subsystem could provide a challenge in obtaining only the required the lowest 25% of Fourier coefficients as this processing is still not fully understood with regards to how to do it on a Raspberry Pi.

2. Subsystem Design:

There are five main subsystems that have been identified and are relevant to the main Raspberry Pi Zero system. Each of these systems are listed below along with their respective functional requirements and design specifications. It is important to note that the order in which the subsystems are listed is the order in which the submodules interface with the external system as well as the other subsystems. After the list of subsystems, the inter-subsystem interactions and UML diagram will provide a much better, and visual, understanding of how each system and subsystem interface with each other.

2.1. Subsystem and Sub-subsystem Requirements and Specifications:

2.1.1. Extraction & Storing of Data:

This system primarily focuses on the extraction of the data generated by the IMU (a motion tracking device) and the storing of it on the Raspberry Pi Zero. We are constrained to using a 16GB secure digital (SD) card for the simulations that we perform, however, a larger storage space does not necessarily equate to faster processing.

Functional Requirement ID	Description	User Requirement Addressed
FR001	A 6-axis MEMs motion tracking device IMU is to be used	UR005
FR002	A 16GB SD card is used for the storage of data on the PiZero	UR008

Table 3 - Functional Requirements of the Extraction and Storing of Data

Design Specification ID	Description	Functional Requirement Addressed
DS001	An ICM-20649 motion tracking IMU device is used.	FR001

Table 4 - Design Specifications of the Extraction and Storing of Data

2.1.2. Data Filtering:

A primary user requirement is that the lowest 25% Fourier Coefficients is filtered through. This is a fundamental subsystem as it extracts all relevant data from the IMU data output stored in the Pi memory. Furthermore, this reduction in data allows for more efficient compression, encryption and transmission processes. The data IMU output is measured in the time domain and as such a Fourier transform needs to be performed on the dataset to convert it to the frequency domain. With the correct sampling time, this will allow the fast Fourier transform (fft) algorithm to be implemented along with the filtering process of the data's coefficient equivalent. Research has shown that the most appropriate choice in programming language is the C language. It tops the list of all other languages when it comes to energy efficiency, execution time and Mb used [10]. Furthermore, it is highly compatible with the Raspberry Pi Zero.

Functional Requirement ID	Description	User Requirement Addressed
FR003	Filter the data to remove potential aliasing prior to extracting the 25% of coefficients	UR001
FR004	C programming Language is used	UR007

Table 5 - Functional Requirements of the Filtering of Data

Design Specification ID	Description	Functional Requirement Addressed
DS002	The code equivalent of a low pass filter will be applied to the data output of the IMU to extract 25% of the Fourier coefficients of the data.	FR003 & FR004

Table 6 - Design Specifications of the Filtering of Data

2.1.3. Compression of Data:

The compression of data is done so by using one of many compression algorithms that are available on the internet. There have been comparisons made between the various algorithms in which the compression ratio and speed is used to determine the most efficient algorithm for the SHARC BUOY application. The compression of data precedes the encryption of the data for numerous reasons. The most prominent answer is that once data has been encrypted the file generates a random stream of data, which becomes near impossible to try to compress. Additionally, compression depends on finding compressible patterns to reduce the over data size.

The SHARC BUOY project requires the compression of data as the transmission of data from the buoy through existing Antarctic transmission infrastructure [11]. This transmission procedure is limited by the satellite network it uses. This network has high data transfer costs, inconsistent transmission reliability, bandwidth and data structure specifications which all therefore limit aspects of the transmission. Therefore, compression of the data will make it cheaper to transfer, as there is less to transfer, and will increase the likelihood of a complete transfer as the transfer can take place in a shorter period.

Functional Requirement ID	Description	User Requirement Addressed
FR005	Both data sent to the Pi and received from the Pi must be compressed using an appropriate algorithm.	UR003

Table 7 - Functional Requirements of the Compression of Data

Design Specification ID	Description	Functional Requirement Addressed
DS003	The Zlib library which implements a deflate algorithm for compression and an inflate algorithm for inflation. This library is compatible with the Raspberry Pi Zero. The data processed by the Pi Zero must be reduced to the lowest total number of bits, however, without losing any integral information.	FR005

Table 8 - Design Specifications of the Compression of Data

2.1.4. Encryption of Data:

The encryption of data is done using AES encryption. This form of encryption is secure and is widely used and accepted. The AES encryption method is easy to implement on a Raspberry Pi as it already has an existing library for the encryption and decryption functionalities [4]. This library allows for the input of any size message by using Cipher Block Chaining mode of operation. This mode of operation allows for keys to be used multiple times and the safety of the encryption to not be in jeopardy. Therefore, it does not matter if the data set is repeated, the key cannot easily be worked out.

Encryption of data to be transmitted is good practice in ensuring safety of data. This data will be encrypted for the transmission between the buoy, the Antarctic transmission tower [11] and then the lab recording the data.

Functional Requirement ID	Description	User Requirement Addressed
FR006	Both data sent to the Pi and received from the Pi must be encrypted using an appropriate algorithm.	UR002

Table 9 - Functional Requirements of the Encryption of Data

Design Specification ID	Description	Functional Requirement Addressed
DS004	The AES encryption method must be implemented due to its compatibility with the Raspberry Pi Zero.	FR006

Table 10 - Design Specifications of the Encryption of Data

2.1.5. Transmission of Data/Packets:

The transmission of data is involved throughout the larger Raspberry Pi subsystem; however, this subsystem primarily focuses on the transmission of the compressed and encrypted data to the iridium network (a global satellite communications company). It must still be mentioned that the I2C (a serial communication protocol) as well as the SPI (Serial Peripheral Interface) are sub-subsystems that are used in the transfer of data from the IMU to the Raspberry Pi and the data from the Pi to the iridium network [11].

Functional Requirement ID	Description	User Requirement Addressed
FR007	The compressed and encrypted data is transmitted to the iridium network.	UR009

Table 11 - Functional Requirements of the Transmission of Data

Design Specification ID	Description	Functional Requirement Addressed
DS005	The iridium modem that is in the larger SHARC BUOY system is used to transmit the data to the iridium network.	FR007

Table 12 - Design Specifications of the Transmission of Data

2.1.6. Raspberry Pi Zero:

All these various interfaces occur or interact with the system that is defined by the Raspberry Pi Zero.

Functional Requirement ID	Description	User Requirement Addressed
FR008	A Raspberry Pi Zero is to be used	UR008

Table 13 - Functional Requirement of the Raspberry Pi Zero

2.2. Inter-Subsystem and Inter-Sub-subsystems Interactions:

As mentioned in submodule 6, the Raspberry Pi Zero, acts as the main system in this design, however, it is a subsystem within the larger SHARK BUOY system. The IMU (or ICM-20649) is a submodule within the SHARK BUOY system, and it interacts with the Raspberry Pi Zero system by transferring the data measured to the Pi's SD card. This transfer of data occurs in the first subsystem module which exists as a sub-subsystem of the SHARK BUOY system.

Secondly, within the Pi's system another subsystem is the data filtering in which the Pi's processor is used to compute the Fourier transform and extract 25% of the Fourier coefficients [12] from the IMU data. Once the data has been filtered, the Pi then compresses the data using deflate and inflate algorithms provided by the Zlib library [7]. This is the third submodule mentioned above and is known as the compression of data. Following this, the data is then encrypted in the fourth submodule known as encryption of data. The AES data encryption technique [3] is used. Finally, the data processing is complete, and it may be transmitted using the iridium modem and to be uploaded to the iridium network.

It must be noted that the subsystems of the Raspberry Pi Zero system interact with at most 2 other submodules and the processes occur in a sequential mode. The active lifespan of each submodule listed is dependent on submodule 1 and when the IMU has produced sufficient data to be sampled and processed. The process that was explained above is graphically depicted in the block diagram below:

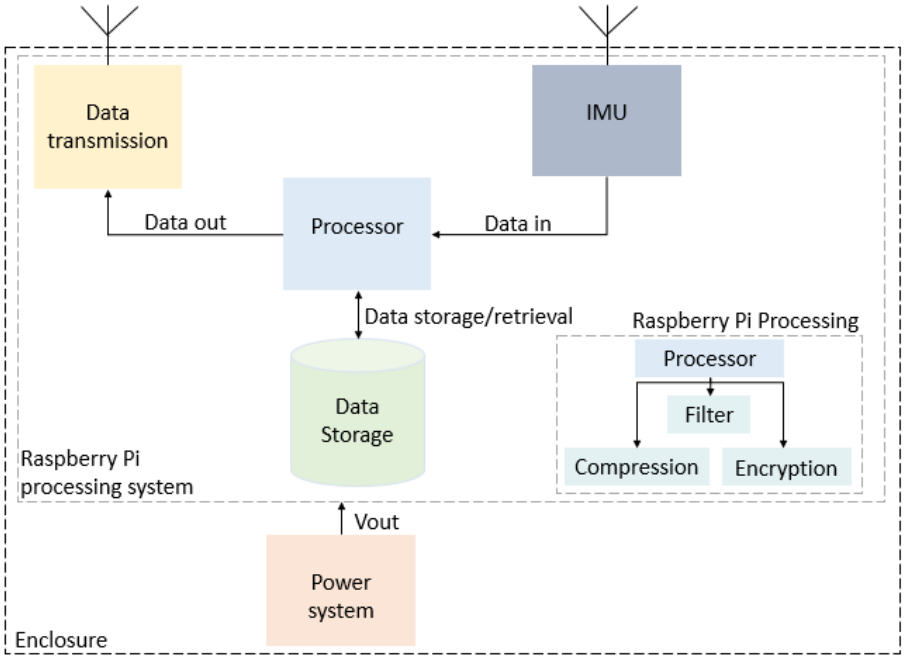


Figure 3 - Block Diagram Showing Interlinked Subsystems [13]

2.3. UML Diagram:

The UML Diagram shows the flow of the overall Buoy Subsystems and how each block interacts with the other blocks within the systems.

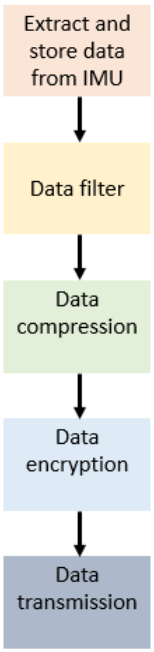


Figure 4 - UML Diagram Showing the Different Subsystems [14]

2.4. Use Case Diagram:

The use case diagram below focuses on the two main buoy subsystem blocks which are namely the compression and encryption blocks and how they interact with the IMU, its data and the data transmission system.

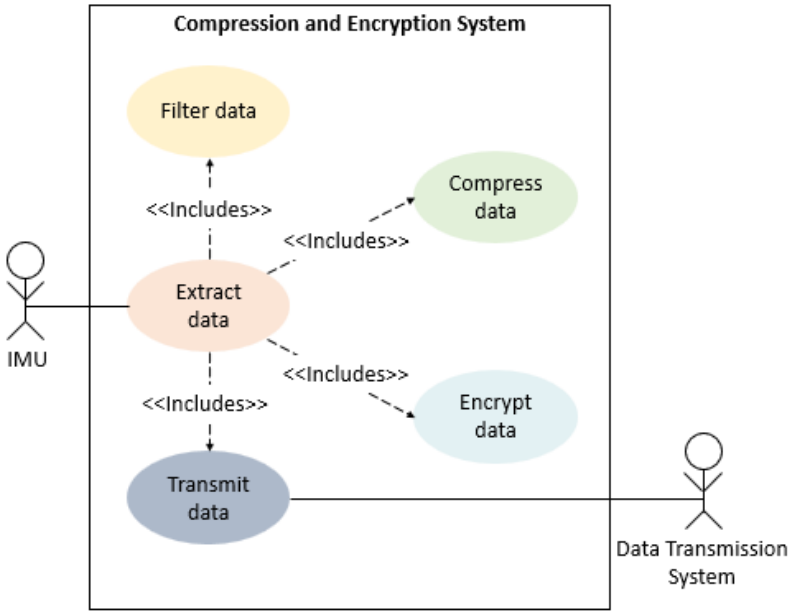


Figure 5 - Use Case Diagram Showing how the Compression and Encryption Blocks Interact with the IMU

3. Paper Design Acceptance Test Procedures:

The various acceptance test procedures (ATPs) that are used to evaluate the performance of the design and whether it meets the requirements is analysed in the following subsections below:

3.1. Figures of Merit:

Subsystem	Figures of Merit
1.) Extraction & Storing Data	Test to show that the software running on the Raspberry Pi translates to similar **running speeds, efficiency and correct processing** when done using the IMU.
2.) Data Filtering	Test to show the **speed of data filtering**. This includes both the processing time to calculate the 25% value of Fourier coefficients in order to effectively filter the data and the actual filtering of the data packet. Test to determine how **accurately the data is filtered** by calculating 5% above and below the desired percentage of coefficients (25%).
3.) Compression of Data	The compression of the data is **efficient** and uses little battery power to process the data. Upon compression, little to **no data is lost**, with 25% of the Fourier coefficients being extractable. The **compressed file size** should fall between 40% to 60% of the original file size. The **compression time** must be as low as possible to reduce the amount of battery power used, however, it must not be too low that some of the above ATP's such as compressed file size are not met. The **compression ratio** of the original file size and compressed file size must be less than 0.9.
4.) Encryption of Data	The **execution time** of the algorithm must take less than 10 seconds for every 16Kb of data thereby minimising the battery power used. The encryption of the data must be **secure** and should not be able to be decrypted easily unless you have access to the unique key. A **lossless encryption** must occur to ensure that upon decryption, the file contains the same data as the file before it was encrypted. Less than 5Kb of Random Access Memory (**RAM**) must be used in the encryption process.
5.) Transmission of Data/Packets	Test to show the **speed of transmission** using different data packet sizes. Another test would determine whether there is any **data lost (packet loss) during transmission**
6.) Raspberry Pi Zero	The Raspberry Pi system must be able to withstand the harsh Antarctic conditions. Furthermore, the device should be able to operate within a 10% **safety margin of the environmental conditions** that ensures all data processing and encryption can still be done accurately.

Table 14 - Figures of Merit of the Overall System

3.2. Experiment Design of ATPs:

3.2.1. Experiment design to test the Compression ATPs:

Various replica IMU datasets are available to be used in the testing of the figures of merit for the compression subsystem. Alternatively, data sets may be generated using the IMU simulation model on MatLab. The ZLIB compression algorithm will be used to compress all the various data files as well as decompression using the ZLIB algorithm. The final compressed file size of each data set will be compared to the original data file size to determine the compression file size and compression ratio. During the compression process, the clock libraries in C will be used to measure the execution time, which will allow for a benchmark to be set for the average speed of compression. Furthermore, the battery percentage shrinkage may be calculated for each compression to determine the amount of power used. Lastly, the Fourier transform on the original dataset and decompressed dataset may be performed to compare the 25% Fourier coefficients of each. This will determine whether there was any data lost and how efficient the compression process was as a python program will be written to simulate this process.

3.2.2. Experiment design to test the Encryption ATPs:

On the other hand, the various replica IMU datasets will also be used in the testing of the figures of merit for the encryption subsystem. The AES encryption algorithm will be used to encrypt all the various data files as well as the decryption of them. The security of the encryption algorithm can be tested by trying to decrypt the dataset by using different keys. If any key besides the unique key for the encrypted algorithm decrypts the data, the encryption was not effective. Linux based tools can be used to monitor the change in the RAM usage. The RAM usage for the encryption time can be compared to typical RAM usage to assess how efficient it is. Similarly to the execution measurement time of the compression subsystem, the C clock functions may be used to test the execution time of the encryption algorithm. Lastly, the decrypted data will be compared to the original data to ensure that it is readable by a human and that it is the same as the data before it was encrypted.

3.1. Acceptable Performance Definition:

The system needs to compress the data significantly to be able to be efficiently transferred by the Antarctic transmission network. However, no data must be lost in the compression and therefore lossless compression needs to be used.

The processing done on the buoy needs to require as small amount of battery power as possible. The buoy is planned to operate for at least a month with minimal recharging of the battery possible. This therefore means that if the processing pulls too much battery power the operation time of the buoy will be decreased as the system designer making the battery subsystem needs to be able to incorporate the needed battery size for the processing on the buoy, which is of a small size.

3.1.1. Compression subsystem:

ATP Name	ATP Description
Efficiency acceptance criteria	The compression of the data is efficient and uses only 1% battery power to process the data.
Data loss acceptance criteria	Little to no data is lost within the upper 75% of Fourier coefficients, and within the lower 25% of Fourier coefficients being 100% not lost or changed.
Compression file size acceptance criteria	The compressed file size should be between 40% to 60% of the original file size.
Compression time acceptance criteria	The time taken to compress the data (the running of the compression code) must be as low as possible, less than 10 seconds per 16Kb of data.
Compression ratio acceptance criteria	Ratio of the original file size and compressed file size must be less than 0.9.

Table 15 - Paper Design ATPs for the Compression Block

3.1.2. Encryption subsystem:

ATP Description	ATP Description
Execution time acceptance criteria	The execution time of the algorithm must take less than 10 seconds for every 16Kb of data.
Lossless encryption acceptance criteria	Decrypted data must contain the same data as the file before it was encrypted (specifically the lowest 25% of Fourier coefficients).
RAM usage encryption acceptance criteria	Less than 5Kb of Random Access Memory (RAM) must be used in the encryption process.
Data encryption acceptance criteria	Data encryption must be secure, therefore a unique key needs to be used in the encryption process.

Table 16 - Paper Design ATPs for the Encryption Block

D. Validation Using Simulated/Old Data

1. Data

A simulation-based validation was needed to best produce a replica environment of having an actual IMU device without physically having one. These would allow for the results to be realistic and tangible so that they could be used to investigate the various impacts that each block's algorithms had when the IMU data was produced.

1.1. Data Used:

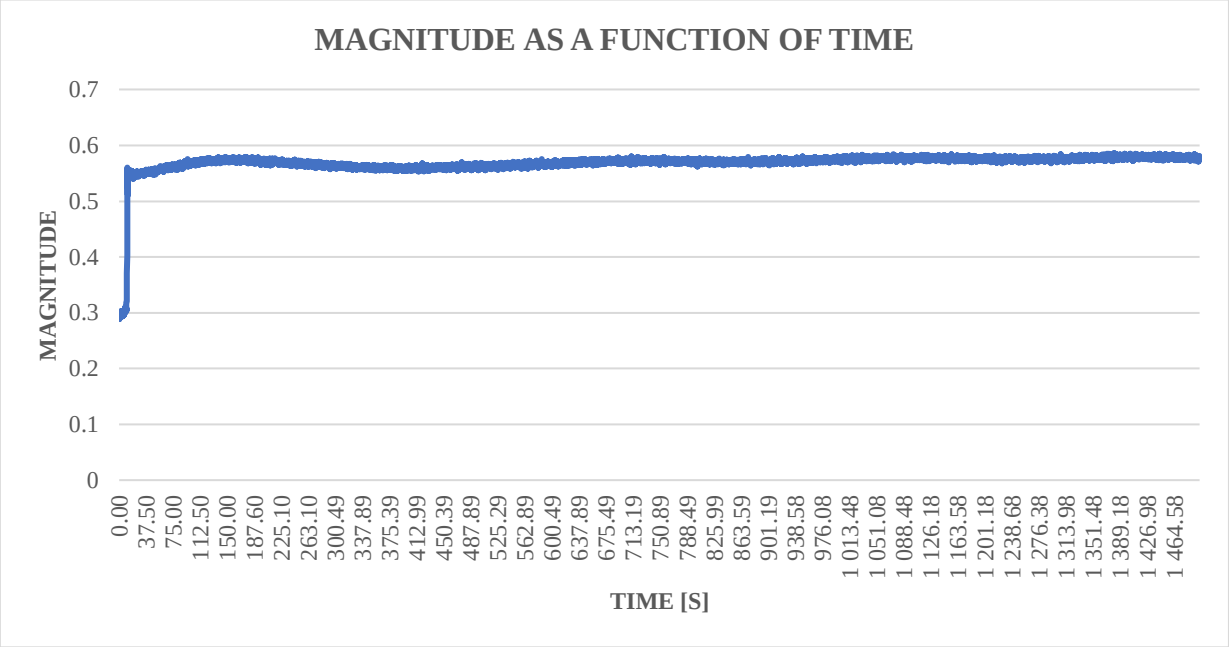
The data used for testing both the compression and encryption methods were the nine raw data sets from the IMU from the authors of the loken_21 paper from February 2021. This data was used as it comes from the same device, the IMU, as the data that will be produced by the buoy. Therefore, this gives an accurate set of the type of data that will need to be compressed and encrypted. This data was in the form of a csv file. The different compression and encryption algorithms work differently (more or less effectively) on different types of data therefore it is important to make sure that the data used in the testing phase accurately represents the data to be used in the real buoy.

1.2. Data Justification:

The data that was provided to us in week 5 are replica sets of data from an IMU used back in 2021. This data format, which is a csv file, is assumed to be the same, however, this will be confirmed in our next assignment when we will have the Sense Hat and an IMU at our disposal. This will be used to retest all the ATPs and compare the results to the ones we obtained in this practical assignment. The data used is justifiable due to it coming from the same measurement device that is applicable in our application and that will be used in our next assignment. The decompressed and decrypted data needs to be in the same file format that the original IMU file supplied to the pi was in. This ties in with the ATP where both the compression and encryption blocks were required to be lossless and have decrypted and decompressed files that match the original file.

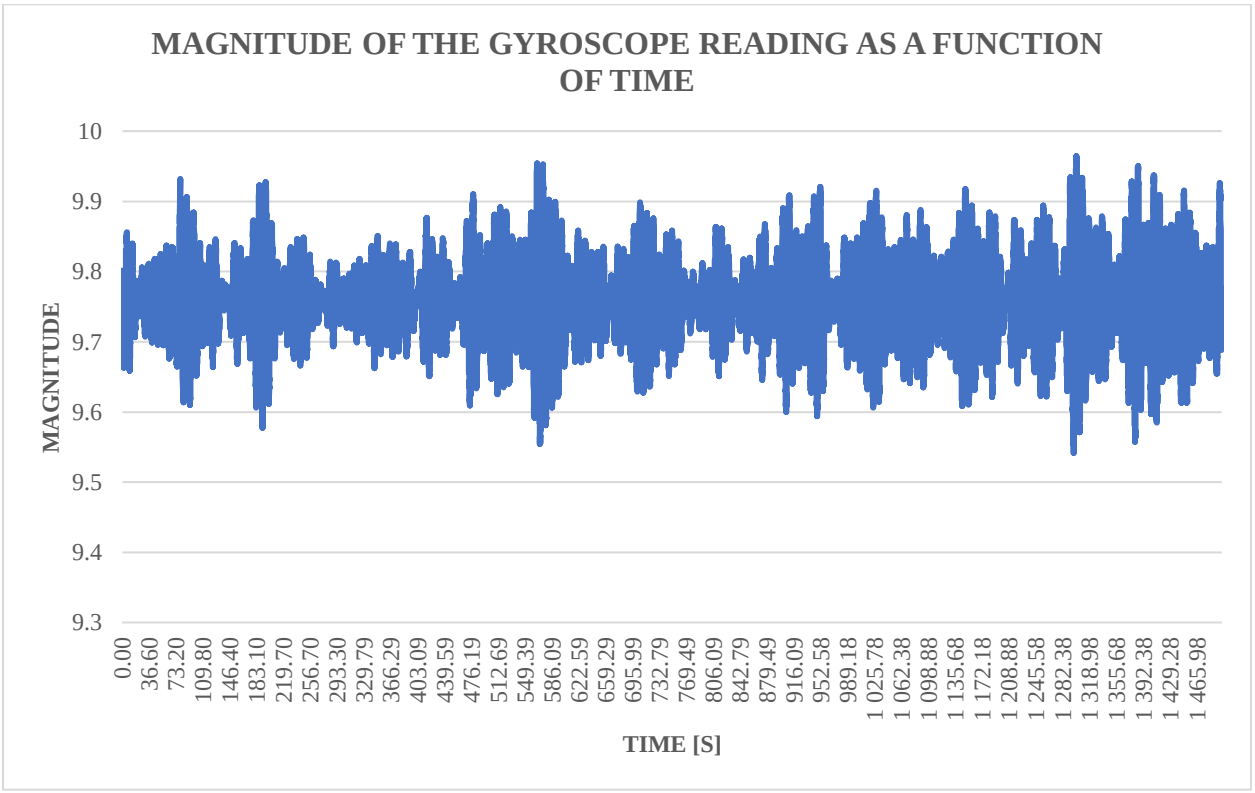
1.3. Initial Data Analysis:

In the five plots below the magnitude, gyroscope, accelerometer, temperature, and pressure were plotted in the time domain. These plots are using the data from "2018-09-19-03_57_11_VN100.csv" as it can be assumed that all nine files contain similar data. As the rest of the project's focus is on the compression and encryption of the data set, therefore the data within is not critical if it fits the expected data that the IMU on the buoy will output. The reason for this is as both the compression and encryption methods used are lossless and therefore the specification that the minimum 25% of Fourier coefficients cannot be lost is met.

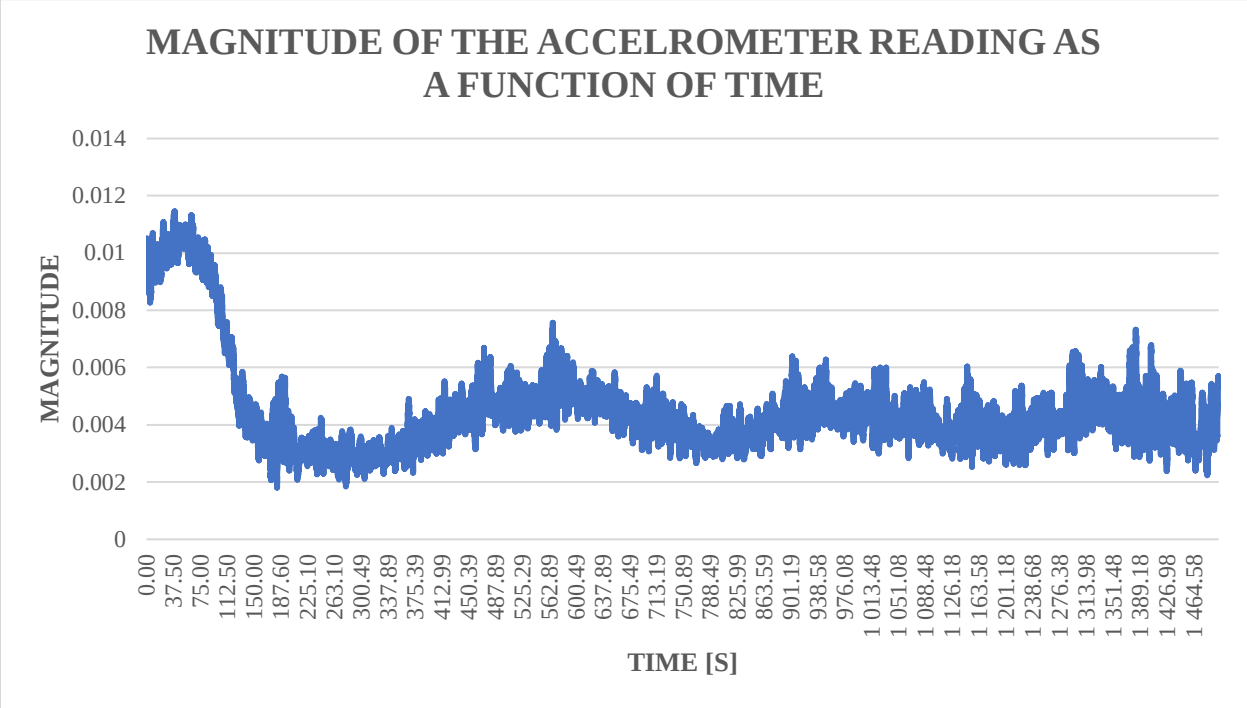


Graph 1 - Magnitude as a Function of Time

Above and below are the graphs of magnitude and gyroscope reading as a function of time.

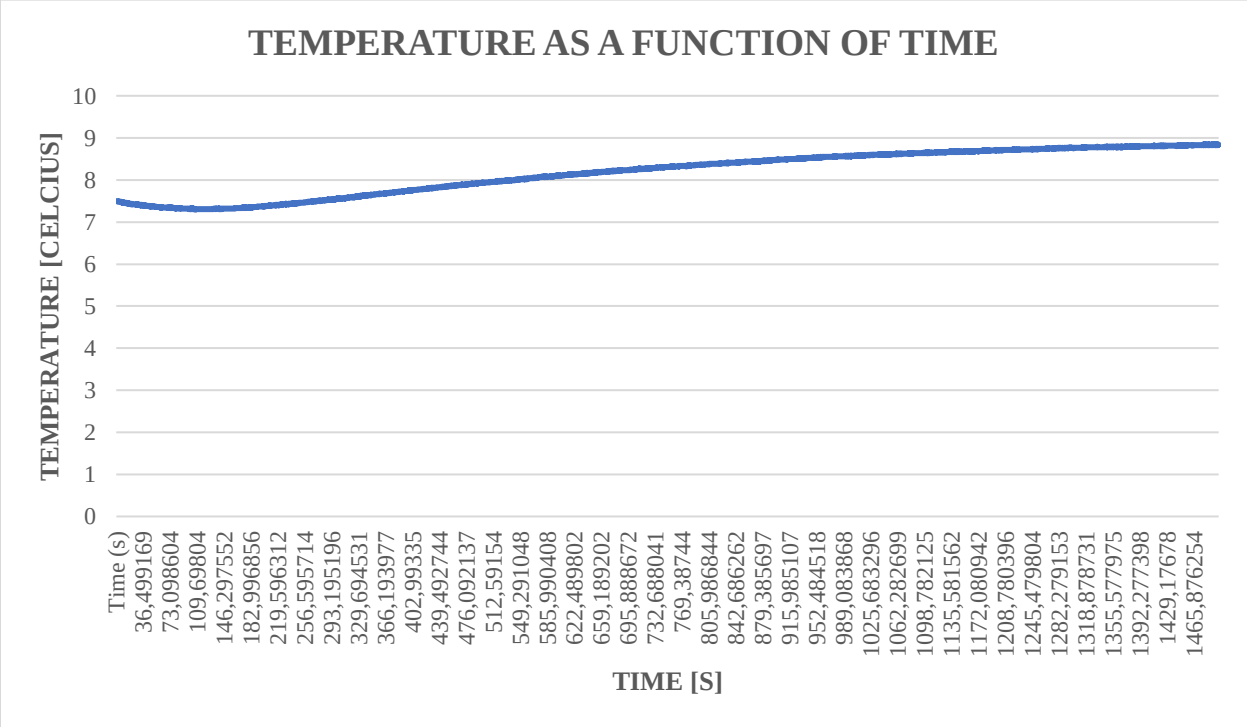


Graph 2 - Magnitude of the Gyroscope as a Function of Time

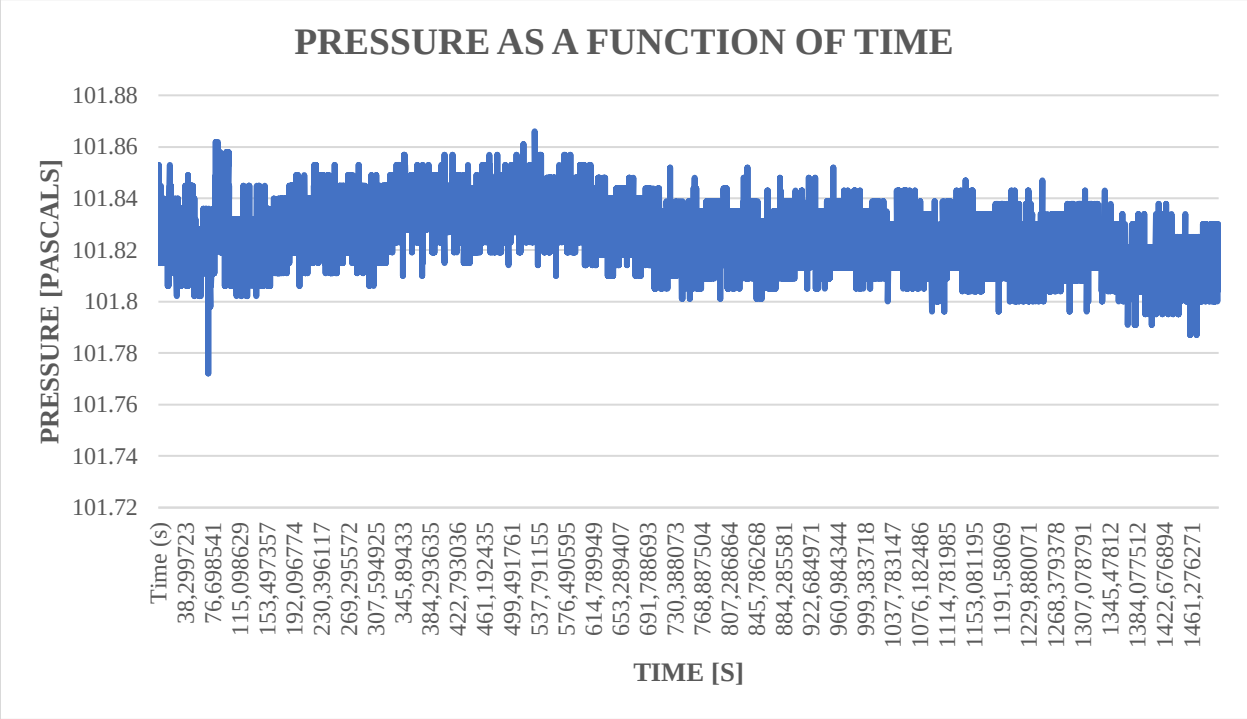


Graph 3 - Magnitude of the Accelerometer as a Function of Time

Above and below are the graphs of accelerometer and temperature readings as a function of time.

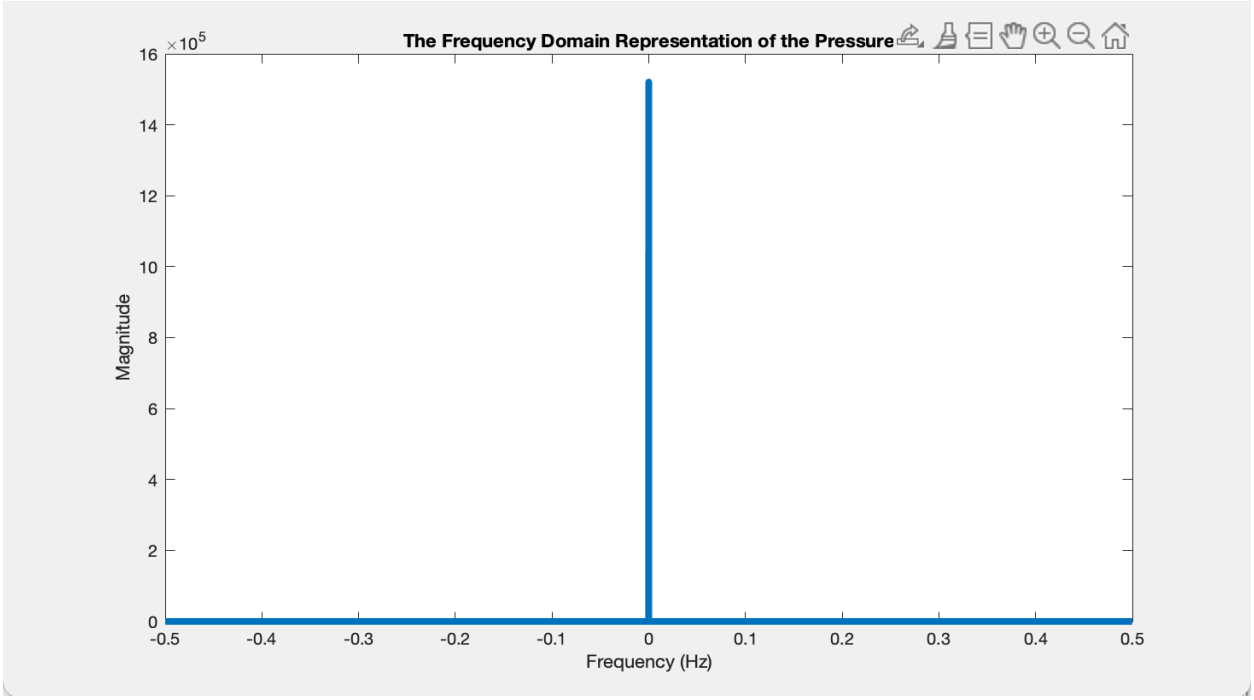


Graph 4 - Temperature as a Function of Time

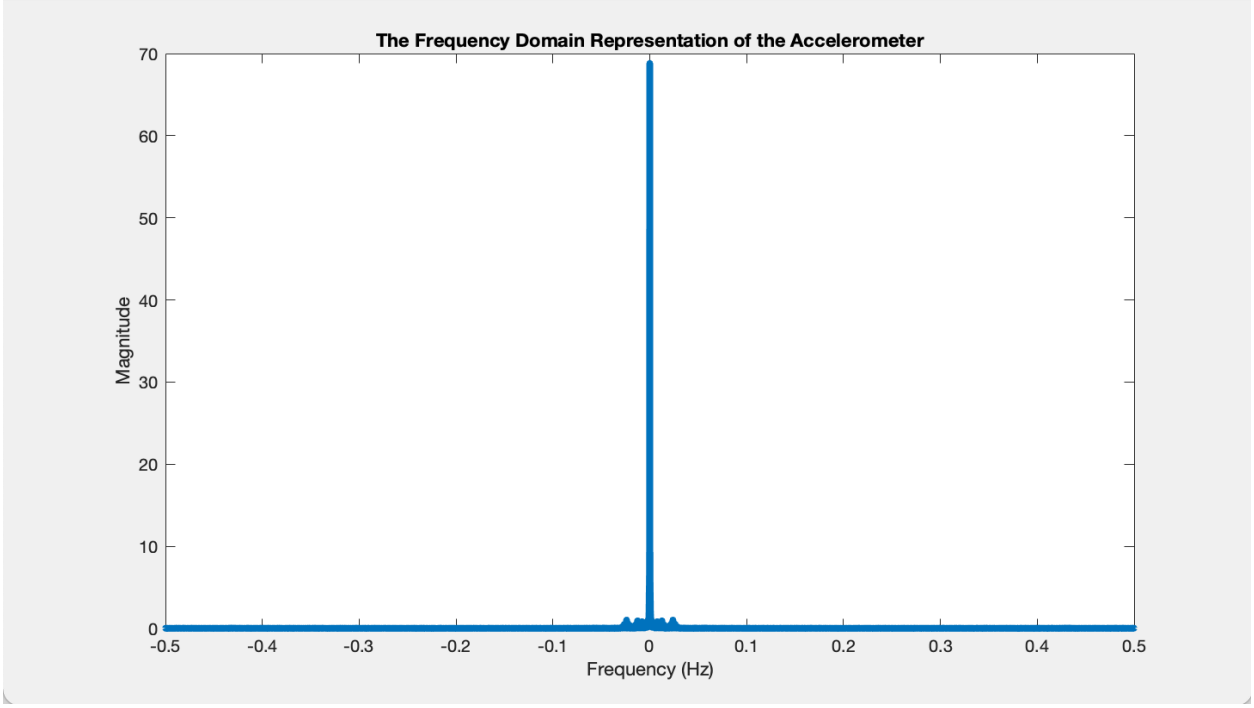


Graph 5 - Pressure as a Function of Time

Below can be seen the Fourier domain plots of the Pressure, Accelerometer, and the Gyroscope that were determined using MathWorks. For all three Fourier Transfers there was one major pulse on the origin [13].

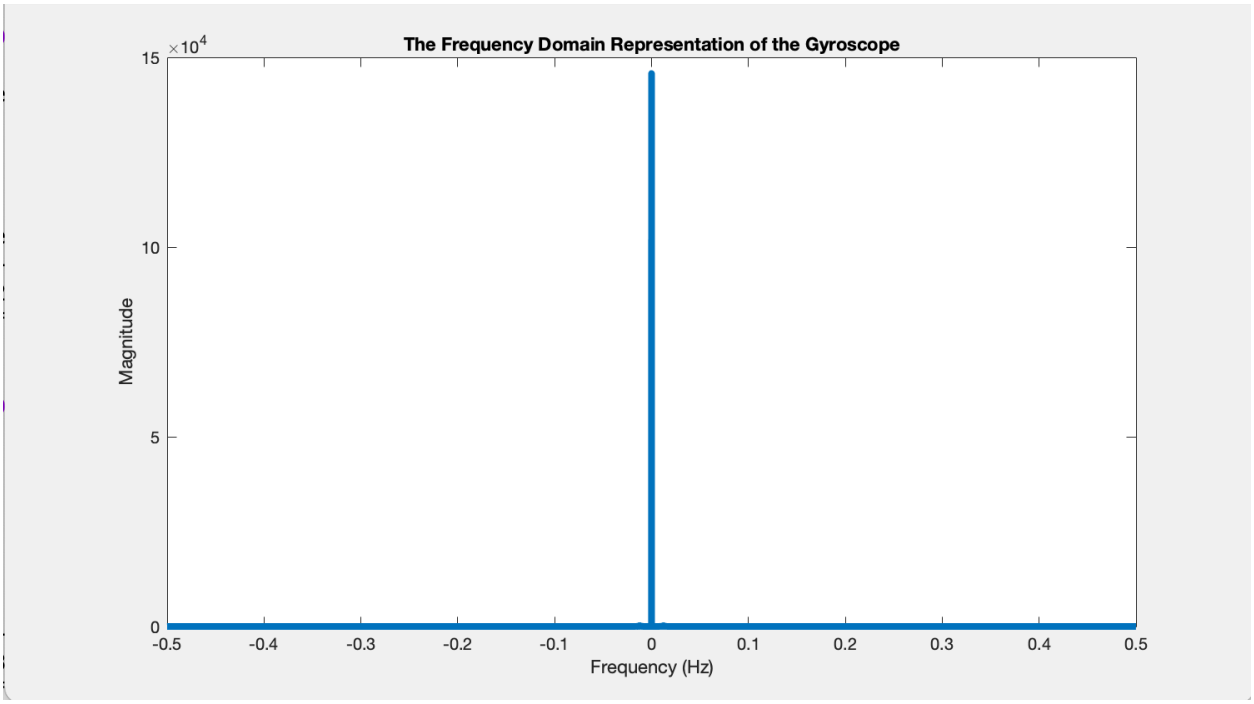


Graph 6 - Frequency Domain Representation of the Pressure



Graph 7 - Frequency Domain Representation of the Accelerometer

Above and below are the Frequency domain representations of the Accelerometer and Gyroscope magnitudes.



Graph 8 - Frequency Representation of the Gyroscope

2. Experiment Setup:

2.1. Simulations/Experiments to Check the Overall Functionality of the System:

The overall system is a combination of the compression and encryption into one Python file that does both blocks one after the other. This set of simulations are done once the best compression and encryption algorithms have been found and developed in the sub-blocks below. When combined the compression and encryption are combined into one runnable python file and the decryption and decompression are combined into another runnable python file. A check python code is used to both shallow and deep check the output of the final decompression against the input original file. Both the compression and the encryption are lossless and therefore the input and output should be identical.

2.2. Experiments for Compression Block:

The compression block ran a variety of tests to obtain values that could be compared to the various ATPs that were set for this block in the paper design. All tests used the data 9 data sets that were provided in the IMU file named 2018-09-19_IMU. The main object of the compression block is to reduce the number of bytes per file while retaining all information inside it. There were 5 “lossless” techniques that were implemented and were namely bzip2 (bz2), GNU zip (gzip), zip, zlib and Lempel–Ziv–Markov chain algorithm (lzma). Before running the various compression techniques, the various sizes of the files were obtained that would be later used in the compression ratio and compression percentage calculations for each file. With 5 different python files, the various compression techniques were run. Following this, the python time library was used to measure the time of execution for each comparison for all nine csv files. Lastly, 2 comparison tester files were used to compare the decompressed file size and bytes (comparison.py) and then the actual file contents (comparison2.py) to the original csv file that was compressed. It must be mentioned that the psutil python library was used to measure the RAM usage to determine the amount of power used, however, after many tests it would only show either 0 or 100 usage. As will be discussed in the later sections, this proved inadequate when determining whether a compression technique met the power usage ATP. It must be noted that the gzip compression method, which is a widely implemented compression technique, was taken as the Golden measure. Furthermore, each technique was implemented with the fastest compression technique that the method had available in its library.

2.3. Experiments for Encryption Block:

The encryption block ran a variety of tests to indicate areas of interest for the various ATPs. The first test that was run was to take the data set "2018-09-19-03_57_11_VN100.csv" and remove most of the data until only 118 bytes were left. This was done to decrease the run times of the algorithms that take much longer. Then three encryption algorithms were run in order to test them against each other and find the most suitable one. The three tested were a Reverse Cipher algorithm, Triple Data Encryption Standard (3DES), and Advanced Encryption Standard (AES). They were all compared on the following attributes: execution speed, security level, lossless, and RAM usage. This was done using various built-in Python libraries (time for execution speed and psutil for RAM usage). Then all the data was tabulated, and the best algorithm was chosen. Note that the Reverse Cipher was used as a Golden Measure to compare the other two algorithms too as it is widely seen as a very primitive encryption algorithm as it is very easy to break but the simplest to implement.

Then the best algorithm was tested on 9 different data sets to see the speed for the different data sets and to check if it is always lossless.

Note that for all the tests the data was also decrypted in order to check if it is lossless. The encrypted file was run through decryption algorithms and then a checking code was applied to both the data sets to check if the input file to the encryption algorithm and the output file to the decryption algorithm were the same.

The following installs were needed to run these three algorithms on the Raspberry Pi Zero [14]:

- sudo pip3 install cryptography
- sudo pip3 install psutil
- sudo pip3 install pyDES
- sudo pip3 install pycrypto

Note this code was adapted using [15].

2.4. Type of Data Each Block is Expected to get in and out:

The IMU data is in the form of a csv file.

The first algorithm run is the compression of the data. This algorithm runs on the inputted csv files from the given IMU data sets and it outputs a compressed file with the file extension bz2. Then the encryption algorithm takes in this bz2 file and encrypts it to an encrypted bz2 file.

The decryption algorithm takes in the encrypted bz2 file and outputs a decrypted bz2 file. Then the decompression algorithm decompresses the data to a File type.

3. Results:

3.1. Results of Simulations/Experiments to Check the Overall Functionality of the System:

The output of the overall function produces the following command line outputs:

```
pi@raspberrypi:~/EEE3097S $ python3 CompressEncrypt.py
Uncompressed size: 9140290
Compressed size: 3142764
Time: 14.026906251907349
Enter the password:test
Total run time:  0.5941636562347412
Total RAM usage:  0.0
pi@raspberrypi:~/EEE3097S $ python3 DecryptDecompress.py
Password: test
pi@raspberrypi:~/EEE3097S $ python3 TestLostData.py
The result will return True if the files match and False if they do not:
Shallow match:  True
Deep match:  True
pi@raspberrypi:~/EEE3097S $
```

Figure 6 - The Terminal Command Output Line After the CompressionEncryption Code has Been Executed

The output data seen above is for the file "2018-09-19-11_55_21_VN100.csv". This run shows a compression execution time of 14.026906251907349s and an encryption run time of 0.5941636562347412s. Additionally the compression compressed the file from 9140290 bytes to 3142764 bytes. This is a compression 34.38% of its original size, therefore a compression of 65.62%. Additionally, when the checks are run both the shallow and the deep check return that no data was lost. The combined file may be found in our GitHub repository that contains the CompressEncrypt.py, DecryptDecompress.py and TestLostData.py python files.

3.2. Results of the Experiments for Compression Block:

As discussed in the paper design submission, it was found that the zlib library, which is a lossless compression technique, should be used. As such, multiple tests were conducted to measure the compression ratio, percentage of compression, execution speed and whether it was indeed lossless or not. In order to do so, many other techniques were used for this experiment namely the gzip, zip, lzma and bz2 on top of the zlib. It must be noted that all of the above techniques are lossless and implement zlib in their own way [16].

The experiments were first tested on a MacBook Pro with a 3.1 GHz Dual-Core Intel Core i5 Processor with an 8 GB 2133 MHz LPDDR3 memory. In light of this, it must be mentioned that the results obtained will differ slightly to that of the results on the Raspberry Pi but the principle regarding which technique is fastest or compresses the most still holds across different devices [17].

As can be seen on the following page, the table shows the file size (in bytes) of each of the original data files that were provided to us. It then shows the compressed file size (in bytes) for each compression technique for every data file [18]. The averages of each compression technique were then calculated using Excel and its many functions. These averages were then used to calculate the compression ratio and percentage of compression for each compression method. It must be noted that percentage of compression refers to the percentage that the original file was decreased *to* as opposed to the percentage that the original file was decreased *by*. For greater clarity this is represented by the mathematical equation shown below:

$$\%\ Decreased\ \boldsymbol{by} = 1 - \%Decreased\ \boldsymbol{to}$$

Please refer to the next page that contains all the tables and appropriate figures pertaining to the compression testing for various techniques.

Below is the table that was mentioned in the previous page that contains the compression ratio and compression file size percentages for each compression technique used:

File Number	Original File Size (bytes)	Compression Method				
		bz2 (bytes)	gzip (bytes)	zip (bytes)	zlib (bytes)	lzma (bytes)
03-57-11	9 071 107	3 219 776	3 773 567	3 771 908	3 773 582	2 906 228
04-22-21	9 067 582	3 228 369	3 785 763	3 784 099	3 785 778	2 925 932
06-28-11	9 083 530	3 240 603	3 788 868	3 786 962	3 788 883	2 940 956
06-53-21	9 080 938	3 240 103	3 786 676	3 784 855	3 786 691	2 938 020
08-59-11	9 141 903	3 199 690	3 751 931	3 750 100	3 751 946	2 887 792
09-24-21	9 152 735	3 157 902	3 736 857	3 734 926	3 736 872	2 828 036
09-49-31	9 132 735	3 196 629	3 751 928	3 750 160	3 751 943	2 890 732
11-55-21	9 140 290	3 142 764	3 705 331	3 703 474	3 705 346	2 837 152
12-20-31	9 158 715	3 113 702	3 688 701	3 686 825	3 688 716	2 795 076
Average	9 114 393	3 193 282	3 752 180	3 750 368	3 752 195	2 883 325
Compression Ratio	-	0,35036	0,41168	0,41148	0,41168	0,316349
Compression File Size	-	35,04%	41,17%	41,15%	41,17%	31,63%

Table 17 - Compression Ratio and Percentage of Compression for Each Compression Technique

	Compression Method				
Python File	**bz2**	**gzip**	**zip**	**zlib**	**lzma**
Comparison1.py	Yes	Yes	Yes	Yes	Yes
Comparison2.py	Yes	Yes	Yes	Yes	Yes

Table 18 - Lossless or Lossy Determination for each Compression Technique

Table 18 on the previous page shows the result when each compression technique's decompressed file was compared to the original file it compressed to check whether the file and all data inside it was preserved. Following this, table 19 below shows the execution time in seconds that each compression technique took to compress each file. Furthermore, it shows the average execution time for every compression method [19].

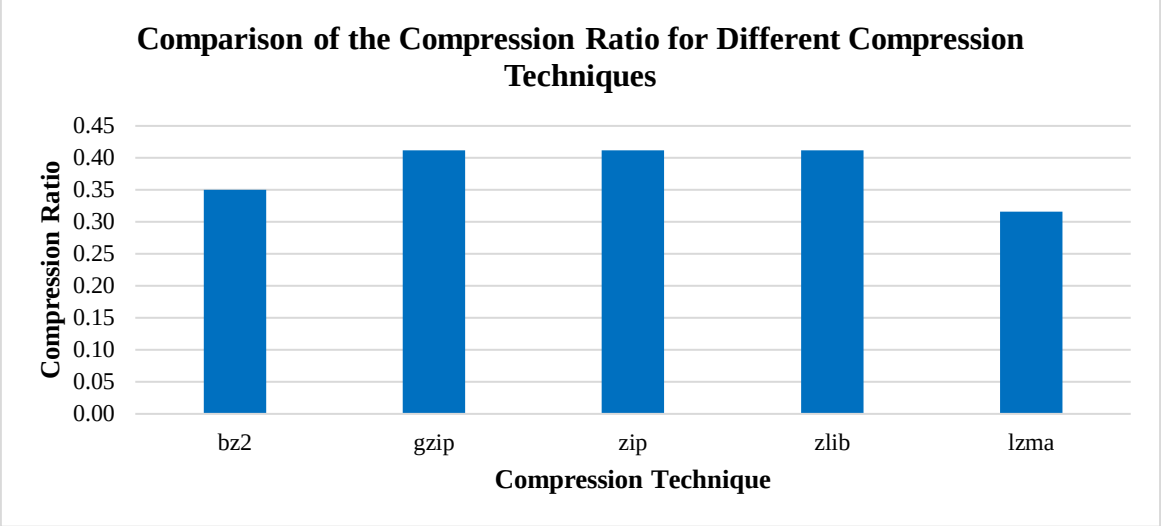
File No.	bz2 execution time	gzip execution time	zip execution time	zlib execution time	lzma execution time
03-57-11	0,78124	1,12090	0,69275	1,12354	14,08882
04-22-21	0,79607	1,17159	0,65859	1,12696	12,77642
06-28-11	0,82492	1,16016	0,63266	1,11275	13,84217
06-53-21	0,81212	1,20697	0,62665	1,13913	13,23417
08-59-11	0,80498	1,14653	0,63219	1,14743	14,17993
09-24-21	0,81810	1,16661	0,62536	1,14235	13,33591
09-49-31	0,83596	1,18343	0,68979	1,10756	14,52019
11-55-21	0,82085	1,16828	0,64312	1,14121	14,06942
12-20-31	0,80430	1,15239	0,61728	1,13923	13,89345
Average	**0,81095**	**1,16410**	**0,64649**	**1,13113**	**13,77116**

Table 19 - The Average Execution Time of Every Compression Technique

When referring to table 17 and the graph below, it is evident that the compression technique lzma had the best compression ratio of 0.316349 and the best percentage of compression, which was 31.63%. This is a stark difference to the golden measure which was the zlib compression method as it averaged a better compression ratio by roughly 0.10 and a better compression percentage of 10%. Coming in second was the bz2 compression method which was not far off the best compression performance as it had a compression ratio of 0,35036 and a compression percentage of 35.04%. This too is much improved in comparison to the golden ratio that only had a compression ratio of 0.41168 and a compression percentage of 41,17%. The gzip and zipfile compression techniques performed similarly to the golden measure [20]. The graph below depicts the graphical representation of the compression ratios for each compression technique in table 17.

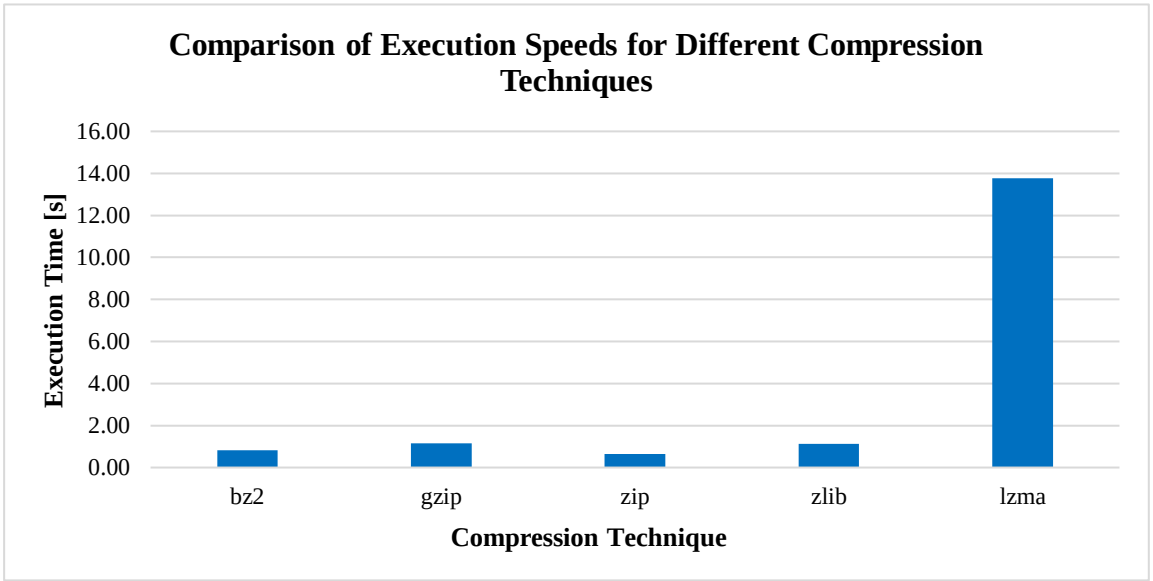
Table 18 are the results when the compressed file for every compression technique was compared to the original uncompressed file using the python files comapison.py and comparison2.py. These two python files performed a shallow and deep match where the shallow match compares the files size and date modified while the deep match compares the actual contents of both files. If either does not evaluate then

a false is returned, however, because the compression techniques used above are lossless, they all evaluated to true.



Graph 9 - Comparison of the Compression Ratio for Different Compression Techniques

Lastly, the time python library was used to time the execution speeds of each compression technique. What was interesting is that while lzma compressed the original file the most, it had the slowest execution time almost averaging 14 seconds. Be mindful that this execution time was determined when the techniques were run on the MacBook Pro and thus this compression method would be even slower if it were implemented on the Raspberry Pi. On the other hand, the zipfile compression method had the fastest execution time averaging 0,64649 seconds. Coming a close second was the bz2 compression method that averaged an execution time of 0,81095. Both of these were significantly faster than the golden measure benchmark of 1,13113 seconds which was the execution time the zlib compression technique. These execution times were plotted in the graph below:



Graph 10 - Comparison of Execution Speeds for Different Compression Techniques

While there isn't a clear winner, the compression technique that will be used may be chosen based on the user's requirements or preferences as all techniques meet the ATPs (this will be discussed more in-depth later). For example, the user may want to prioritize execution time over amount of compression. In this case the zipfile compression method would be recommended. On the other hand, the user may prefer to have the file compressed to the smallest possible size without worrying about execution time. As such, the lzma compression technique would be the most suitable technique for this case. All in all, we decided to choose the compression technique that performed the best in both the execution and compression categories. Upon analysis of the data is evident that this would be the bz2 compression technique as it placed 2nd best in both testing categories. It is easy to implement anyone of these techniques as all the code that was used to gain these results are at the respective links of the bz2, gzip, zip, zlib and lzma python files.

Please refer to the next page that contains all the tables and appropriate figures pertaining to the encryption testing for various techniques.

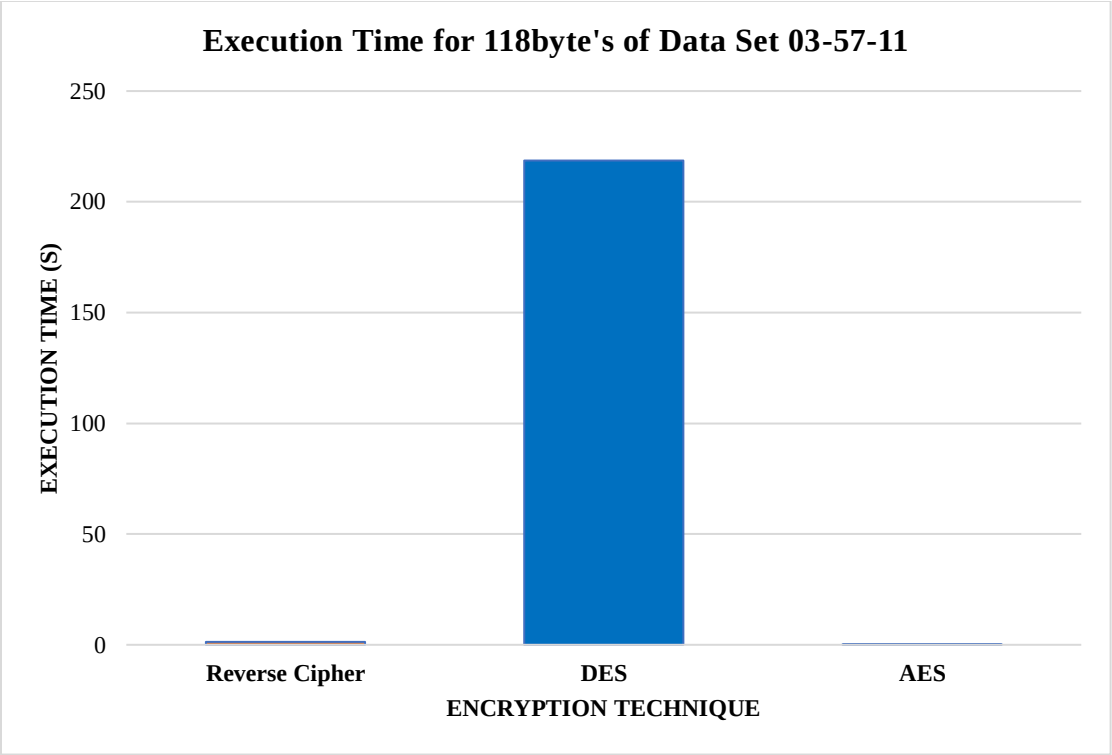
3.3. Results of the Experiments for Encryption Block:

The following table below describes the results of the encryption tests for various implementations of encryption techniques:

Encryption Method	Data Set	Execution time (s) (data set 1)	Execution Time (s) for 118byte data set	Execution time (s) (16Kb)	Secure level (1-5(very secure))	Lossless (decrypt the same)	RAM usage (%)
Reverse Cipher	03-57-11	98,9970	1,31862	13,4233	1	Y	100
DES	03-57-11	-	218,648	29,6471	3	Y	100
AES	03-57-11	1,60836	0,02142	0,00290	5	Y	100
AES	04-22-21	1,62087	0,02159	0,00293	5	Y	100
AES	06-28-11	1,61755	0,02155	0,00292	5	Y	100
AES	06-53-21	1,56896	0,02090	0,00283	5	Y	100
AES	08-59-11	1,60182	0,02134	0,00289	5	Y	100
AES	09-24-21	1,57968	0,02104	0,00285	5	Y	100
AES	09-49-31	1,58706	0,02114	0,00287	5	Y	100
AES	11-55-21	1,57989	0,02104	0,00285	5	Y	100
AES	12-20-31	1,60335	0,02136	0,00290	5	Y	100

Table 20 - The Encryption Test Results for the Reverse Cipher, DES and AES Encryption Techniques

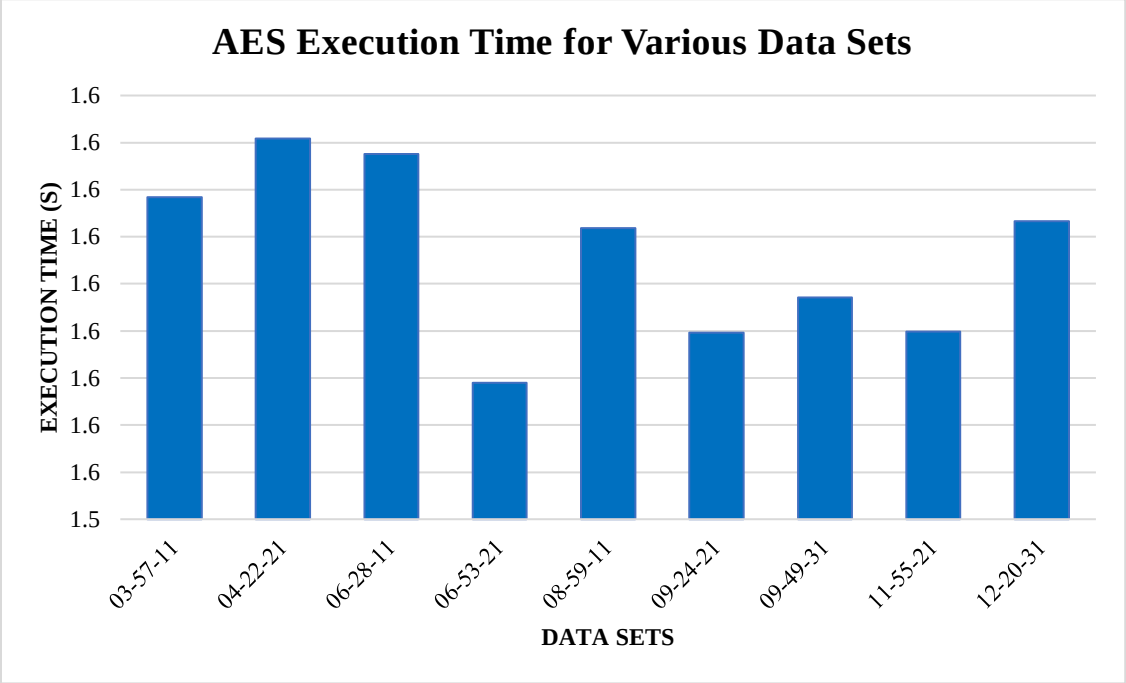
This data shows that all three methods are lossless and only AES is seen as secure. The RAM usage for each run toggled between 0% and 100% and was therefore seen as not a reliable measure and was discarded. The security level of the Reverse Cipher is abysmal [14] and was therefore used more as a golden measure to compare the other two methods too as it is commonly seen as the simplest form of encryption. When the run time for the 118-byte data set is looked at both the Reverse Cipher and AES methods run quickly where the DES runs much slower. This can be seen in the figure below which compares the different algorithms run time for one data set. The python files for the various encryption techniques may be found at AES, DES and Reverse Cipher.



Graph 11 - Encryption Execution Time for Various Methods

Therefore, DES was then eliminated as an option due to its slow running speed which would use up too much of the buoy's battery. Therefore, this left AES which is a widely accepted encryption standard with a good security level, is lossless for data encryption and has a running speed on average 0,002883205s per 16Kb of data which is much better than the wanted 10s/16Kb. AES was then tested on nine different data sets to test that not only is it lossless for all of them but also comes up with similar execution speeds for all of them. This comparison can be seen in the figure below where it can be seen that all the data sets had encryption execution times between 1.57s and 1.62s. This is quite a fast speed when it is seen that the data sets range in size between 8856Kb and 8945Kb. This is quite a fast speed when it is seen that the data sets range in size between 8856Kb and 8945Kb.

Note that the AES method requires a shared key to be used. This means that the user has a password that is used to encrypt and decrypt the data. Having a shared key is not a security risk in this instance as the same person sets the password on the buoy and then retrieves and decrypts the data so no transfer of passwords is necessary and therefore it cannot be broken. Obviously, this depends on the password's length and randomness, the longer and more random the better the security level.



Graph 12 - AES Execution Time for Different Datasets

This method was implemented in Python due to its compatibility with the Raspberry Pi Zero, the abundance of available libraries for the encryption standards and to easily tie in with the compression algorithm which is also written in Python. Note that code was written to first run the encryption, then decryption algorithms and then test the output of the decryption algorithm against the input data to the encryption algorithm. This produced a terminal output that looked like the figure below.

```
pi@raspberrypi:~/EEE3097S $ python3 AES-5.py
Enter the password:test
Total run time:  1.6216495037078857
Total RAM usage:  100.0
Password: test
pi@raspberrypi:~/EEE3097S $ python3 TestLostData.py
The result will return True if the files match and False if they do not:
 True
pi@raspberrypi:~/EEE3097S $ python3 AES-5.py
Enter the password:test
Total run time:  1.6495518684387207
Total RAM usage:  100.0
Password: test
pi@raspberrypi:~/EEE3097S $ python3 TestLostData.py
The result will return True if the files match and False if they do not:
 True
pi@raspberrypi:~/EEE3097S $ python3 AES-5.py
Enter the password:test
Total run time:  1.6293063163757324
Total RAM usage:  100.0
Password: test
pi@raspberrypi:~/EEE3097S $ python3 TestLostData.py
The result will return True if the files match and False if they do not:
 True
```

Figure 7 - Screenshot of the Terminal Commands when Conducting Different Tests

3.4. Effect of Different Data on the System:

Adding white-Gaussian noise to the raw data set will not change the effectiveness of the compression or the encryption. This is because both were implemented to be lossless and therefore no data is lost and the bottom 25% of Fourier coefficients are not lost. The issue of the noise will only therefore become relevant when the data is being used once decrypted. Making the data be under-sampled, therefore less data to transfer, would have the effect of speeding up both the compression and the encryption blocks as they are both dependent on the size of file to be transferred.

Please refer to the next page that contains all the tables and appropriate figures pertaining to the ATPs for the compression and encryption blocks.

4. Simulated Data Acceptance Test Procedures:

4.1. Compression ATPs:

ATP Name	ATP Description	ATP met in Design	ATP in Design Comment
Efficiency acceptance criteria	The compression of the data is efficient and uses only 1% battery power to process the data.	No	This was not easy to test on the Raspberry Pi Zero as it either returned a result of 0% or 100%. Therefore, it will be removed or adapted as a specification.
Data loss acceptance criteria	Little to no data is lost within the upper 75% of Fourier coefficients, and within the lower 25% of Fourier coefficients being 100% not lost or changed.	Yes	For the 25% Fourier coefficients to be obtained, the decompressed file just needs to be compared to the original file where the contents and size are checked. Considering lossless compression techniques being used, all methods met this ATP.
Compression file size acceptance criteria	The compressed file size should be between 40% to 60% of the original file size.	Yes	As seen in the results section, all methods fell within this range with some failing below. Considering this, the ATP will be adjusted, however, it still passes the test procedure as the lower the file size is compressed the better. The file size was compressed to 35.04% of the original file size or compressed by 64.96%.
Compression time acceptance criteria	The time taken to compress the data (the running of the compression code) must be as low as possible, less than 10 seconds per 16Kb of data.	Yes	The bz2 compression technique executed in less than 1 second for a file size close to 10Mb. Considering this, the ATP needs to be adjusted to reflect a better representation of what the execution speed should be. It had an execution speed of 0.8109
Compression ratio acceptance criteria	Ratio of the original file size and compressed file size must be less than 0.9.	Yes	Similarly, to the compressed file size, the compression ratio ATP needs to be adjusted as the compression technique was significantly faster (or smaller) when compared to the ATP. It had a ratio of 0.35036.

Table 21 - Simulated Data ATPs for the Compression Block

The change in the compression block specifications were as follows:

Original Specifications	New Specification
The compression of the data is efficient and uses only 1% battery power to process the data.	This specification is going to be tied to the execution time and compression ratio as this relates to the battery power consumption and CPU usage.

Table 22 - Change in Compression Block Specifications

4.2. Encryption ATPs:

ATP Name	ATP Description	ATP met in Design	ATP in Design Comment
Execution time acceptance criteria	The execution time of the algorithm must take less than 10 seconds for every 16Kb of data.	Yes	The ATP was met and improved by a very large factor (a speed of 0.002883205s/16Kb was achieved) therefore this specification can be edited to decrease the time per 16Kb of data significantly.
Lossless encryption acceptance criteria	Decrypted data must contain the same data as the file before it was encrypted (specifically the lowest 25% of Fourier coefficients).	Yes	AES method of encryption was lossless.
RAM usage encryption acceptance criteria	Less than 5Kb of Random-Access Memory (RAM) must be used in the encryption process.	No	This was not easy to test on the Raspberry Pi Zero as it either returned a result of 0% or 100%. Therefore, it will be removed or adapted as a specification.
Data encryption acceptance criteria	Data encryption must be secure, therefore a key need to be used in the encryption process.	Yes	The AES encryption method uses the same key to encrypt and decrypt the data. If this key is not leaked, then the data will remain very secure.

Table 23 - Simulated Data ATPs for the Encryption Block

The change in the encryption block specifications were as follows

Original Specifications	New Specification
Less than 5Kb of Random-Access Memory (RAM) must be used in the encryption process.	The use of Random-Access Memory (RAM) must allow for functionality for all needed devices to continue (not create errors in the program) from filling up the RAM when just the encryption algorithm is being run.

Table 24 - Change in Encryption Block Specifications

Please refer to the next page for the next section of the report.

E. Validation Using a Sense Hat (B) IMU

1. IMU Module:

A Hardware-based validation was needed to further explore the encryption and compression algorithms produced by a physical device that is similar to the IMU used in the actual buoy. It provides a greater understanding with regards to the hardware limitations that may arise within the system. Additionally, it provides a greater understanding regarding the sensors and how certain motion affects the data produced and the size of it. These would allow for the results to be realistic and tangible so that they could be used to investigate the various impacts that each block's algorithms had when IMU data was produced.

1.1. Comparison between the ICM-20649 & Sense Hat (B):

The IMU and the Sense Hat (B) have some similar sensors. The table below shows a list of the various sensors for each. The N/A means Not Applicable and was used to show that the IMU used in the Buoy does not have all the sensors that the Sense Hat (B) can provide .

The table on the next page page clearly shows that the Waveshare Sense Hat (B) has a far greater range of sensors than the IMU (ICM-20649) that will be used in the thesis. The 3 main sensors that are shared between both devices are the gyroscope, accelerometer and temperature sensor. Both the gyroscope and the accelerometer have 3 values that correspond to a 3-axis cartesian plane and are namely the x-,y- and z-axes. The main difference is that the Sense Hat (B) has an IMU of its own, which is the ICM-20948 [23]. The ICM-20649 boasts a much faster degrees per second rotational speed than the Sense Hat (B) means that the IMU that will be used will provide more accurate data than the Sense Hat (B) for both the gyroscope and accelerometer. On the other hand, the main differences between the temperature sensors is that the IMU has a digital sensor while the Sense Hat (B) has an analogue sensor. The main difference between digital and analogue (like the MCP9700) sensors is that analogue sensors use a transfer function to determine the temperature while a digital sensor does not require a program to know the transfer function prior to determining the temperature [24]. As a result, digital sensors are often more accurate as they do not need to be recalibrated to take into account DC and gain offsets like analogue sensors. All in all, even though the data that is produced may be more accurate using one device than the other, the size of the data produced will be approximately the same. As we are most concerned with the compression and encryption blocks of this larger subsystem, the same data sizes are far more important than the accuracy of the data itself [25].

IMU (ICM-20649) Sensors [21]	Sense Hat (B) Sensors [22]
Gyroscope • 3-axis • Programmable FSR with ±100/500/2000/4000 °/sec	**Gyroscope** • 3-axis • 16-bit resolution • ±250/500/1000/2000 °/sec
Accelerometer • 3-axis • Programmable FSR with ±4/8/16/30g	**Accelerometer** • 3-axis • 16-bit resolution • ±2/4/8/16g
Digital temperature sensor	**Temperature sensor** • Range: -30°C to 100°C • Accuracy: ±0.2°C
N/A	**Humidity sensor** • Range: 0-100% rH • Accuracy: ±2% rH
N/A	**Colour sensor** • 4 channels RGBC • 16-bits per channel resolution
N/A	**Barometer** • 24-bit resolution for pressure • 16-bit resolution for temperature • Accuracy: ±0.025hPa • Speed: 1-75 Hz
N/A	**Magnetometer** • 16-bit resolution • Range: ±4900 μT
N/A	**ADC** • 12-bit resolution

Table 25 - Comparison between the ICM-20649 & Sense Hat (B):

1.2. Extrapolation of Sense Hat (B):

As mentioned in the previous paragraph, in order to replicate the data that will be produced on the IMU, only the sensors that are common between both devices will be used. These are the Gyroscope, Accelerometer and Temperature Sensor. Furthermore, the data was formatted to take the same shape that the raw IMU data files provided to us took. An example of what this format looks like is shown in the image below taken from the files:

2018-09-19-03_57_11_VN100

computer utc start 2018-09-19-03:57:21.506044																													
UTC computer time	MagX	MagY	MagZ	AccX	AccY	AccZ	GyroX	GyroY	GyroZ	Temp	Pres	Yaw	Pitch	Roll	DCM1	DCM2	DCM3	DCM4	DCM5	DCM6	DCM7	DCM8	DCM9	MagNED1	MagNED2	MagNED3	AccNED1	AccNED2	ACCNED3
2018-09-19-03:57:21.717926 -0.14322271943092346 0.2232052981853485 0.123428575694561 0.15357686579227448 -0.30159956216812134 -9.795899391174316 0.00508350133895874 -0.0013912274735048413 -0.008807962760329247 7.502488136291504 101.84500122070312 51.88072204589844 0.95980566																													
2018-09-19-03:57:21.817726 -0.1410953253507614 0.21974442899227142 0.1235906854271887 0.15984362363815308 -0.30425748229026794 -9.779479026794434 0.005224071908742189 -0.0011848110007122 16 -0.008935784921050072 7.5031352043151855 101.8280029296875 51.85698699951172 0.963294																													
2018-09-19-03:57:21.917707 -0.14538313448429108 0.22439616918563843 0.12579116225242615 0.1569005399942398 -0.29826948046684265 -9.782461166381836 0.004827750381082296 -0.0009488037903793156 -0.008939079008996487 7.502196788787842 101.8530044555664 51.82896041870117 0.967599																													
2018-09-19-03:57:22.017733 -0.14438782632350922 0.2198091596364975 0.12465985864400864 0.15925079584121704 -0.30477917194366455 -9.774477005004883 0.005078970920294523 -0.0010242330608888946 -0.008752092719078064 7.49957799911499 101.8320083618164 51.80281066894531 0.97143024																													
2018-09-19-03:57:22.117717 -0.14319761097431183 0.2232208251953125 0.1271720826625824 0.15469375252723694 -0.30872249603271484 -9.755866050720215 0.005107069853693247 -0.0009864562889561057 -0.009002655744552612 7.50026798248291 101.84000396728516 51.77418518066406 0.9753627																													
2018-09-19-03:57:22.217686 -0.14326296746730804 0.2209295630455017 0.12470496445894241 0.16080111265182495 -0.30884605646133423 -9.749424934387207 0.004865021910518408 -0.0011699099559 3369 -0.008859817869961262 7.5013580322265625 101.8280029296875 51.748191833496094 0.978881																													
2018-09-19-03:57:22.317733 -0.14428161084651947 0.22437554597854614 0.12585045397281647 0.15797586739063263 -0.3115788400173187 -9.74105453491211 0.004976013209670782 -0.0012970133684575558 -0.009132187813520432 7.499946594238281 101.8280029296875 51.72221374511719 0.9813843																													
2018-09-19-03:57:22.417797 -0.1465568244457245 0.2209947258234024 0.1257745325565338 0.16294534504413605 -0.3130553364753723 -9.739278793334961 0.004936043173074722 -0.0012547707883641124 -0.009054646827280521 7.499579906463623 101.8280029296875 51.69611740112305 0.984293580																													
2018-09-19-03:57:22.517726 -0.14545537531375885 0.22097420692443848 0.12583346664905548 0.16190342605113983 -0.3162236213684082 -9.728490829467773 0.004644147586077452 -0.0015776463551446795 -0.008987781591713428 7.499178409576416 101.8280029296875 51.66779708862305 0.985924																													
2018-09-19-03:57:22.617713 -0.1487094908952713 0.22331589460372925 0.1256270706653595 0.16106994450092316 -0.31497976183891296 -9.719582557678223 0.004643251188099384 -0.0008518678951077163 -0.009389842860400677 7.4979424476623535 101.8280029296875 51.64188003540039 0.991598																													
2018-09-19-03:57:22.717685 -0.14646583795547485 0.22555047273635864 0.12446887791156769 0.16358499228954315 -0.3218514621257782 -9.71753215789795 0.004382750950753689 -0.0008635037811473012 -0.009203701280057[illegible] 7.4980082511901855 101.8320083618164 51.613956451416016 0.997149																													

Figure 8 - Raw Data from a Replica of the IMU

The image below is the output file of the Sense Hat (B) file:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Time, MagX, MagY, MagZ, AccX, AccY, AccZ, GyroX, GyroY, GyroZ, Temp, Pres, Yaw, Pitch, Roll													
2	2021-10-16 17:18:19.627698,-57.5,66.25,40.75,92,1371,16863,16,0,1,22.71,1021.01,172.58280052189218,8.362924024401662,4.075740591265154													
3	2021-10-16 17:18:19.967374,-55.125,66.25,41.5,74,1331,16889,5,-1,0,22.26,1021.01,166.00379230952868,16.129432631129916,7.594950519553109													
4	2021-10-16 17:18:20.301478,-57.75,64.875,41.0,65,1297,16931,0,1,1,22.25,1021.01,160.42937664551732,22.42992009260376,10.191643540663632													
5	2021-10-16 17:18:20.640499,-57.75,65.625,40.875,69,1351,16909,1,1,-1,22.27,1021.0,155.56384168020887,26.813578934246646,12.17477906113748													
6	2021-10-16 17:18:20.975155,-56.625,64.375,41.0,85,1352,16903,2,0,2,22.27,1021.01,151.26402543212416,29.514502722237133,13.50277079652751													
7	2021-10-16 17:18:21.316464,-56.375,65.875,40.5,24,1380,16887,6,1,2,22.28,1020.98,147.06212859408657,31.068635919598794,14.308368693758316													
8	2021-10-16 17:18:21.654284,-57.875,64.125,40.125,39,1368,16894,5,1,0,22.28,1021.02,142.90805735597135,31.97828617589782,14.453299672463334													

Figure 9 - Format of the Data Generated by the Waveshare Sense Hat (B)

Additionally, the raw data file is saved as a csv file and so we ensured that when writing our programs the output data was saved to a csv file as well. The size of the data set would depend on the length of time between each sampling of the data from the IMU. To accommodate for this variable change in file size length, different sampling times will be used. All these factors mentioned above will be crucial when simulating the compression and encryption blocks again as not only will we obtain live data and output the data to files, we will have a more accurate idea of the file sizes that are produced by the IMU (ICM-20649) and how well the compression and encryption methods we chose run on the Raspberry Pi Zero. This testing can be extrapolated easily as the same Raspberry Pi Zero and a very similar IMU are used to get a real time idea of how the IMU should operate [26].

1.3. Validation Tests:

There are various validation tests that can be conducted to ensure that the Sense Hat (B) is working as expected. The first test would be to check that the temperature sensor is working. One can test this by using any phone's temperature sensor via the weather app and comparing the temperature values produced to the values that the Sense Hat (B) produces. Any values that are incredibly high, such as 40°C and above, or incredibly low, such as 0°C or lower, show that the sensor is not working properly. Note that although the

Sense Hat (B) can produce temperatures that range between -30°C to 100°C, it would be abnormal to get values outside of the 0°C to 40°C range as we current test environment is Cape Town, which varies between that range depending on the time of day and season [27].

Secondly, the Pitch, Yaw and Roll measurements can be determined by moving the Sense Hat (B) in the appropriate axis direction. The image below is a great representation of the direction of each axis. Even though the image is of a Raspberry Pi Sense Hat, the principle still holds as the axes are the same across the different devices. Additionally, we also used the pressure sensor on the Sense Hat (B) even though the actual IMU will not have a pressure sensor. This is purely to test the overall functionality of the system.

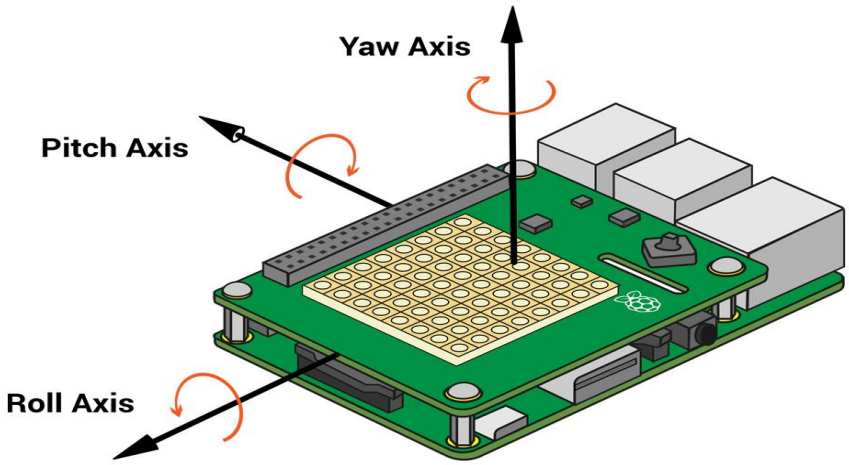


Figure 10 - The Roll, Pitch & Yaw Axes of a Raspberry Pi Sense Hat

We also considered seeing the effect of moving the Sense Hat (B) in the Mag X, Y and Z directions, however, after initial analysis of the data, this test would not be feasible due to the small movements that were only possible as a result of the short Raspberry Pi Zero USB cable connected to the laptop [28].

2. Experiment Setup:

2.1. Simulations/Experiments to Check the Overall Functionality of the System:

2.1.1. Test 1:

The first set of tests run will be to validate the working of the IMU. As described in the Validation Tests section, there will be several tests that will be run to validate the data produced by the Sense Hat (B). The first will be the generation of temperature data by seeing the change in temperature when the IMU is placed in direct sunlight on a warm day. This will further be compared to temperature that is determined by various apps on mobile phone devices. Secondly, the pitch, yaw and roll data will be manipulated by rotating the

IMU in the directions that correspond to the image shown in the previous page. As the data produced is related to the degrees in which the sensor is rotated around the respective axis, it is expected that once rotated a full 360° the data will "reset". As such spikes that vary between two values is the expected output.

In addition to the temperature sensor values being verified so will the pressure data produced. Although we were unable to change altitudes to cause a significant change in the pressure values, the values may be validated easily. If the pressure in pascals produced is the same as the current pressure produced by the phone sensor then it is validated and if not then the sensor is non-operational. Finally, the acceleration data was tested by moving the Sense Hat around, while changing the speed of each movement. Generally, a majority of the sensors were tested by shaking the Sense Hat to see the effect that it had on the data. It must be mentioned that other tests could have been implemented such as bringing a magnet close to the IMU, however, the validation tests focused primarily on sensors relevant to the IMU on the buoy.

2.1.2. Test 2:

The second set of tests run on the overall system was to check the impact of changing the output data set size on the run time of the Sense Hat. This is expected to be linear due to the waiting element in the code but was tested for validation. Additionally this test provided the needed data sets for the following tests. The time in which data was pulled from the Sense Hat was varied in order to produce the different file sizes and lengths that were used in the various tests mentioned.

2.1.3. Test 3:

Another test run was to check changing the wait time between each sample taken. This data was then transformed into the Fourier domain to check for noise. The pressure sensor and the magnitude of the accelerometer sensor were used to perform a Fourier domain analysis on the data by varying the sampling rate.

2.1.4. Test 4:

The fourth test run for the overall system was putting the various data set sizes through the compression and encryption combined algorithm to test the different speed up ratios or runtimes of the each of these data set sizes.

2.1.5. Test 5:

The final overall system test run was using the compressed and encrypted files created in test three and thus compare the final file size to the original file size.

Note that the overall system is a combination of the compression and encryption into one Python file that does both blocks one after the other. This set of simulations are done once the best compression and encryption algorithms have been found and developed in the sub-blocks below. When combined the compression and encryption are combined into one runnable python file and the decryption and decompression are combined into another runnable python file. A check python code is used to both shallow and deep check the output of the final decompression against the input original file. Both the compression and the encryption are lossless and therefore the input and output should be identical.

2.2. Experiments for Compression Block:

The compression block ran a variety of tests to obtain values that could be compared to the various ATPs that were set for this block in the paper design and thus re-evaluated in Section D. All tests used the ten files that were produced by the Waveshare Sense Hat (B) where each file had a different file size. The main object of the compression block is to reduce the number of bytes per file while retaining all information inside it. There were 5 "lossless" techniques that were implemented and were namely bzip2 (bz2), GNU zip (gzip), zip, zlib and Lempel–Ziv–Markov chain algorithm (lzma). Before running the various compression techniques, the various sizes of the files were obtained that would be later used in the compression ratio and compression percentage calculations for each file. With five different python files, the various compression techniques were run. Following this, the python time library was used to measure the time of execution for each comparison for all ten csv files. Lastly, two comparison tester files were used to compare the decompressed file size and bytes (comparison.py) and then the actual file contents (comparison2.py) to the original csv file that was compressed. It must be mentioned that the psutil python library was used to measure the RAM usage to determine the amount of power used, however, after many tests it would only show either 0 or 100 usage. As will be discussed in the later sections, this proved inadequate when determining whether a compression technique met the power usage ATP. It must be noted that the gzip compression method, which is a widely implemented compression technique, was taken as the Golden measure. Furthermore, each technique was implemented with the fastest compression technique that the method had available in its library.

When comparing the set of tests carried out in Section E, they differ slightly to that of the tests run in Section D. The main difference is that the file sizes used in Section D were significantly large (roughly 9 Mb) and all roughly the same size. In this section a varying length of sizes, that are significantly smaller than Section D, were used. Furthermore, all tests where run ON the Raspberry Pi Zero board with the Sense Hat (B) attached to it. In Section D, all tests were run on a MacBook pro. As such if trying to compare the

results of both reports, there will be obvious changes in the compression ratios, percentages of compression and execution speeds, however, it is expected that the overall performance of every compression technique should be the same as in Section D.

2.3. Experiments for Encryption Block:

The experiment run on the encryption block was to test the speed of encrypting different data set sizes for the different encryption algorithms. This was done using the different data sets obtained in test two for the overall system.

The three encryption algorithms tested are:

- Reverse Cipher algorithm
- Triple Data Encryption Standard (3DES)
- Advanced Encryption Standard (AES)

The testing was done using various built-in Python libraries (time for execution speed). This test was done in order to reinforce the findings of the last paper that AES is the fastest algorithm. Note that it is also the most secure which is a needed attribute [29].

Note that the Reverse Cipher was used as a Golden Measure to compare the other two algorithms too as it is widely seen as a very primitive encryption algorithm as it is very easy to break but the simplest to implement.

Note that for all the tests the data was also decrypted in order to check for losslessness. The encrypted file was run through decryption algorithms and then a checking code was applied to both the data sets to check if the input file to the encryption algorithm and the output file to the decryption algorithm were the same.

2.4. Type of Data Each Block is Expected to get in and out:

The IMU data is in the form of a csv file and should include time, magnitude, acceleration, gyroscope, temperature, pressure, yaw, pitch and roll. The chosen data from the Sense HAT was data common to the ICM-20649 that will actually be used on the buoy.The first algorithm run is the compression of the data. This algorithm runs on the inputted csv files from the data sets, produced from the Sense HAT run, and it outputs a compressed file with the file extension bz2. Then the encryption algorithm takes in this bz2 file

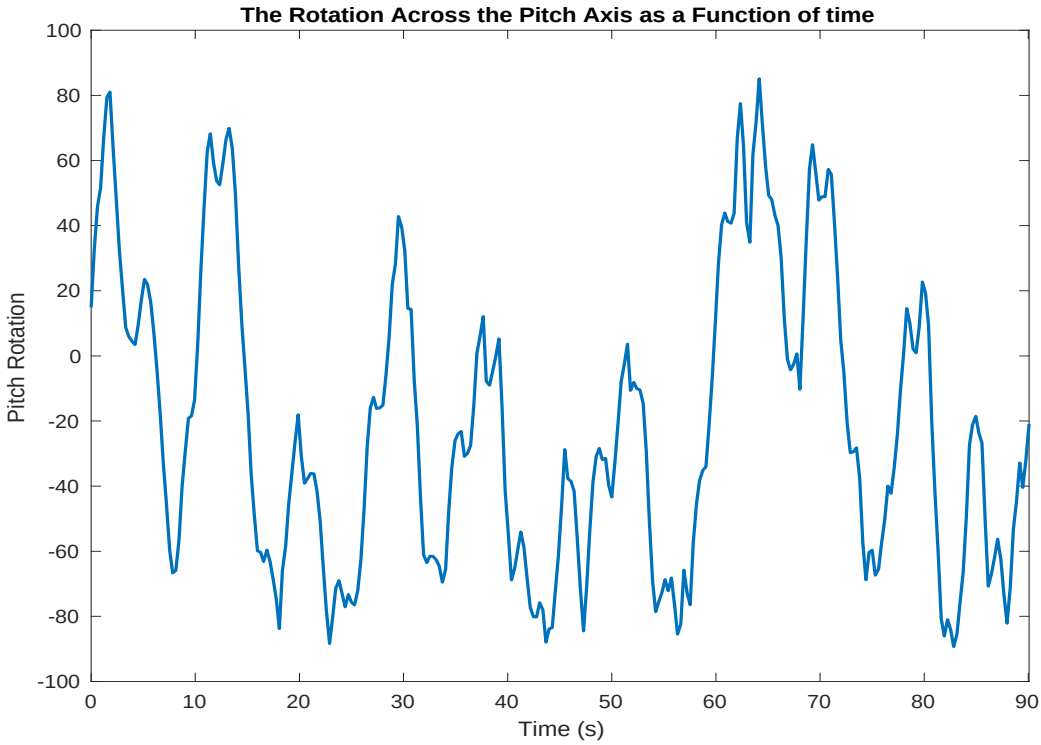
and encrypts it to an encrypted bz2 file. The decryption algorithm takes in the encrypted bz2 file and outputs a decrypted bz2 file. Then the decompression algorithm decompresses the data to a File type.

3. Results:

3.1. Results of Simulations/Experiments to Check the Overall Functionality of the System:

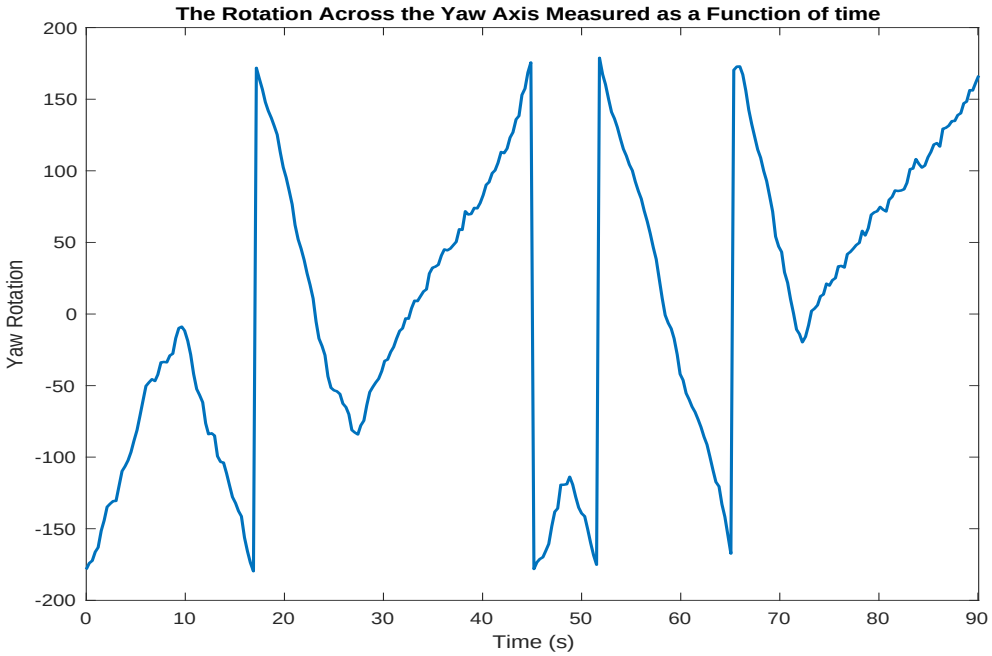
3.1.1. Results of Test 1 *(Validate IMU data)*:

Note that for all the results generated, the corresponding csv files were imported into MatLab where the various plotting functions were used to generate the results and plots shown below. Please see Appendix A for snippets of the code. The application of the FFT (Fast Fourier Transform) will be shown in test 3 where the pressure and accelerometer Fourier domain expressions will be derived and plotted. The results of the Pitch, Yaw & Roll axes as well as the Temperature and Pressure time domain expression of the data is displayed in the images below:



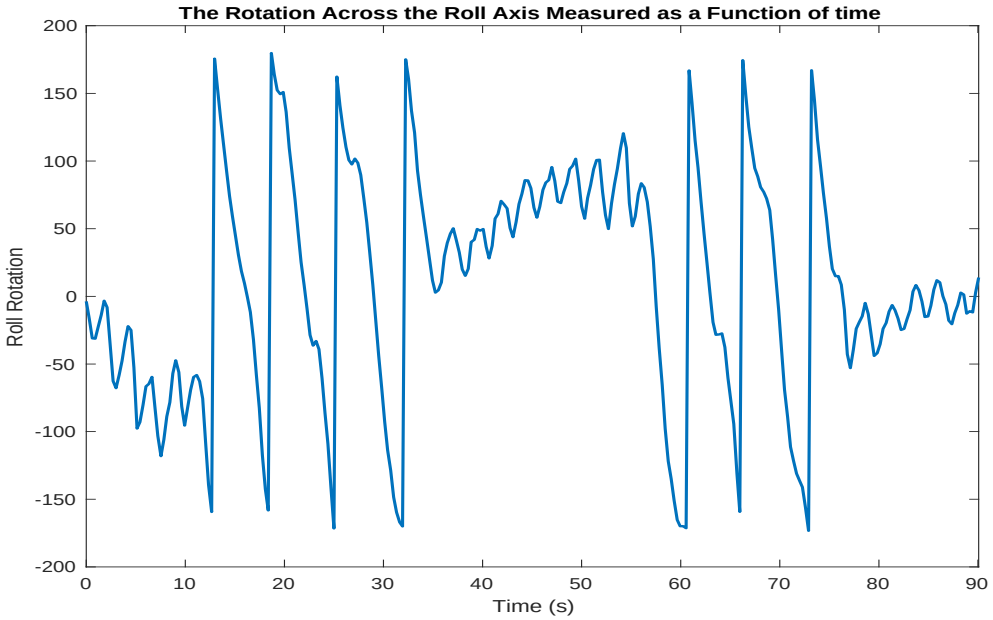
Graph 13 – Rotation Across the Pitch Axis as a Function of Time

As expected, there are spikes that vary between two values as the Sense Hat was rotated around the Pitch axis. Furthermore, the values range from positive to negative values which too was expected.



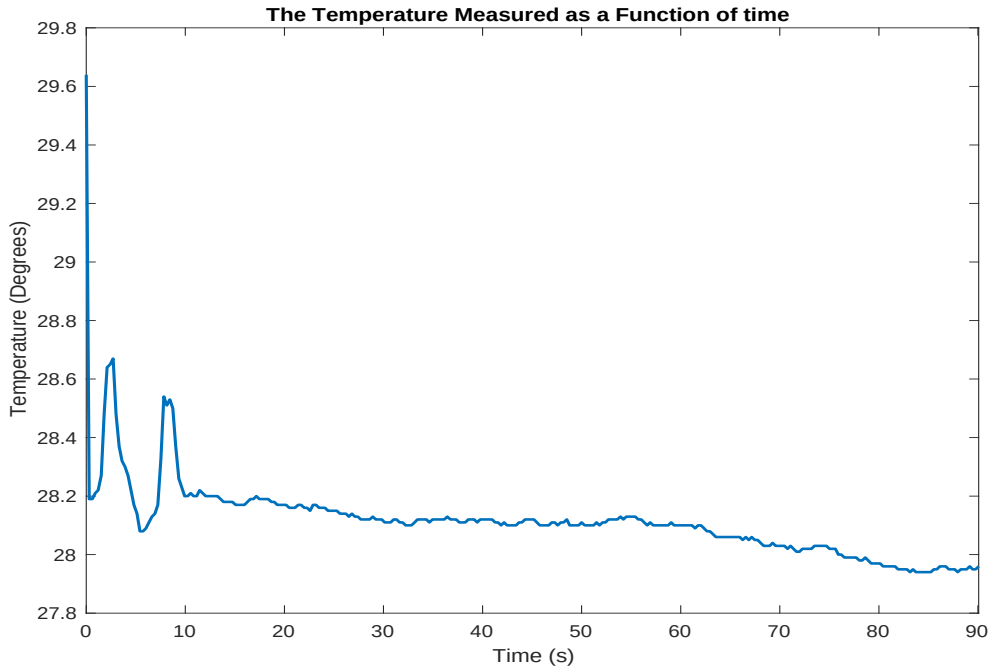
Graph 14 - Rotation Across the Yaw Axis as a Function of Time

The spikes were more accentuated than that of the Pitch axis, however, they still varied between two values as the Sense Hat was rotated around the Yaw axis. Furthermore, the values range from positive to negative values which too was expected.



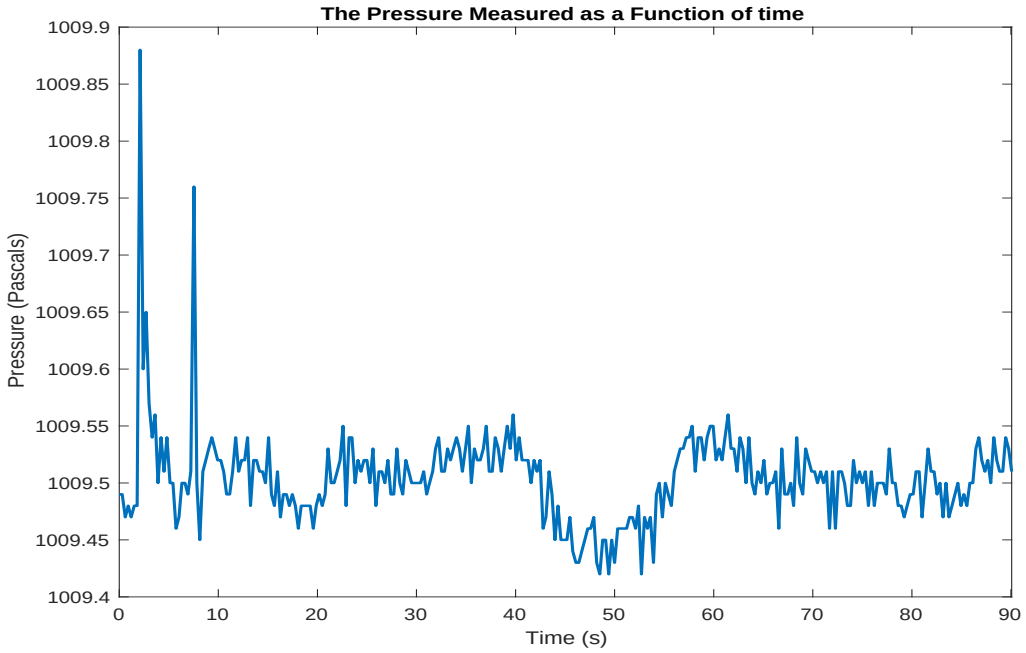
Graph 15 - Rotation Across the Roll Axis as a Function of Time

Graph 15 on the previous page shows that there are spikes that vary between two values as the sense hat was rotated around the Roll axis.



Graph 16 - Temperature Measure as a Function of Time

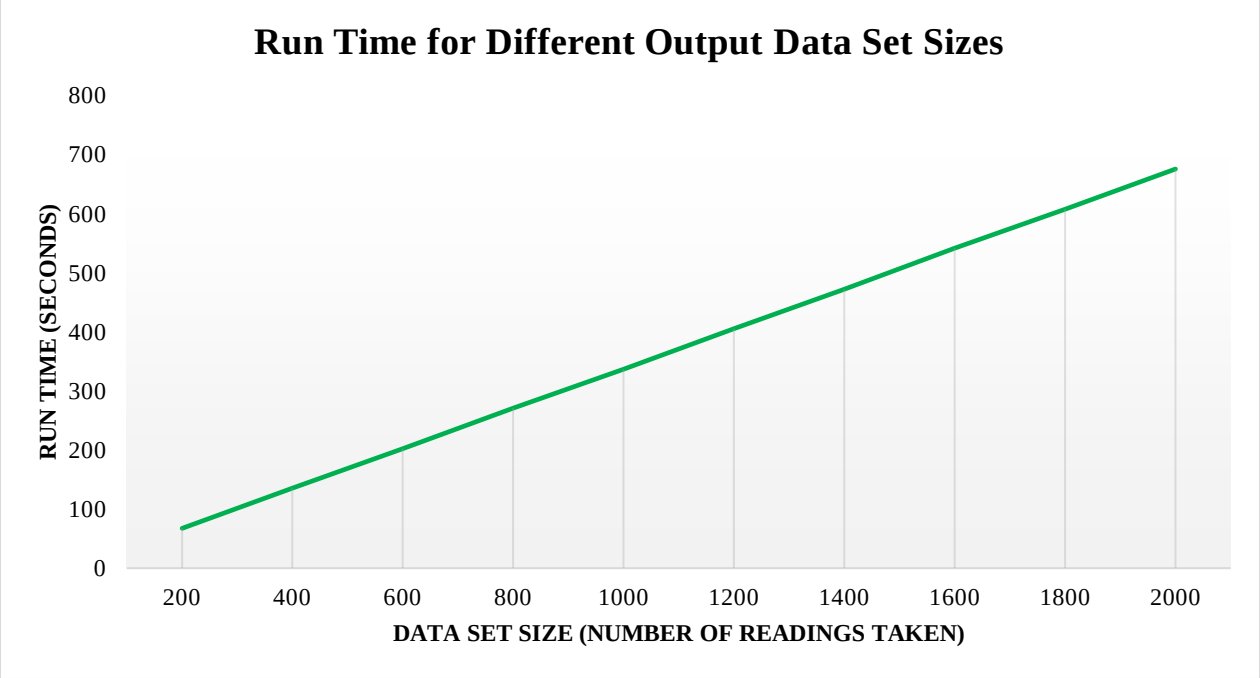
Above, the temperature produced averaged at roughly 28°C which corresponded to the weather temperature app found in Apples App Store. Originally there was a spike in the temperature that can be attributed to the change in temperature of the Raspberry Pi's CPU and other internal hardware when placed in the sun.



Graph 17 - The Pressure Measured as a Function of Time

3.1.2. Results of Test 2 *(Changing output data set size on Sense Hat runtime)*:

The amount of data (number of sets of readings taken) was changed in order to validate the relationship between runtime and output data set size. This can be seen in the graph below.



Graph 18 - Run Time for Different Output Data Set Sizes

As was expected from the way the code was implemented (with a specified wait time between samples) the relationship was perfectly linear. This reinforced that the code was executing correctly and was not producing any weird results. Additionally this test produced the needed data sets to be used through the rest of the project.

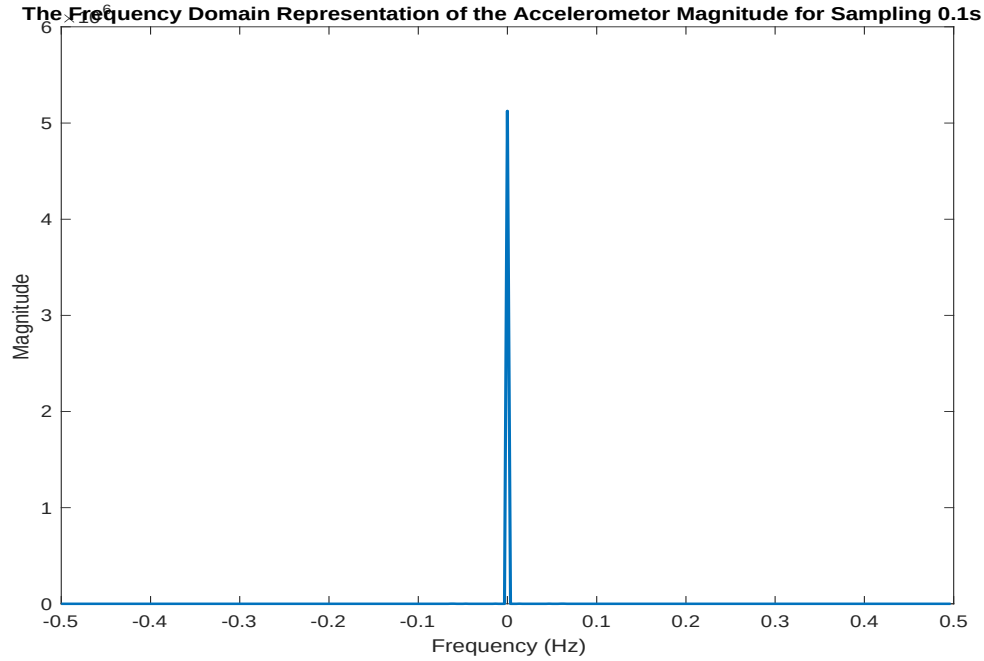
The Raspberry Pi Zero terminal looked like the following when the file containing the code to use the Sense HAT was called.

```
pi@raspberrypi:~/EEE3097S $ python3 SenseHatCode.py

Sense HAT Test Program ...

Time: 202.2595341205597
```

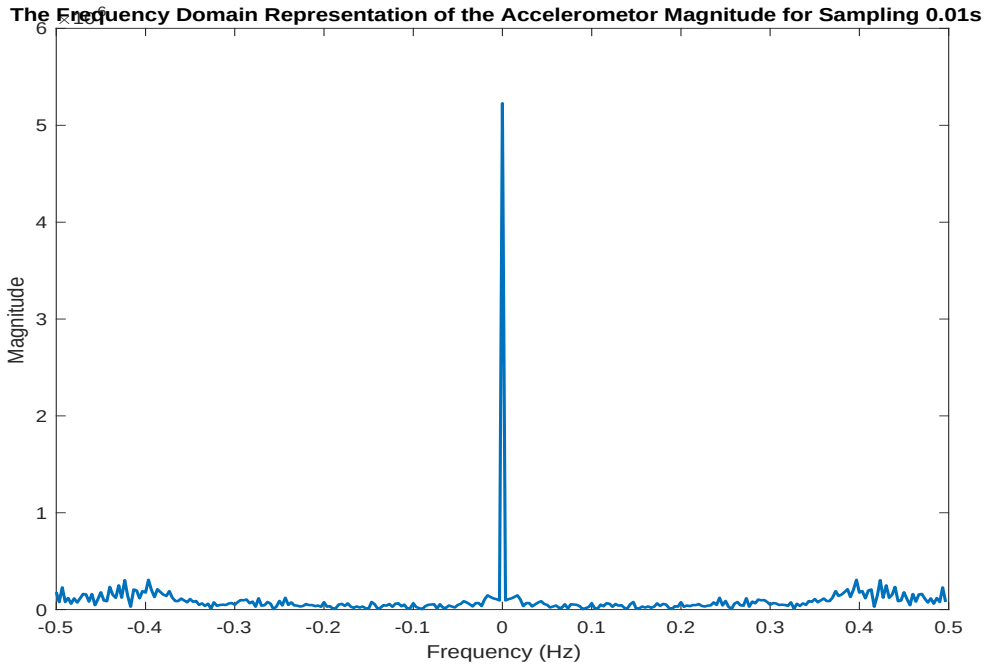
Figure 11 - Screenshot of the Time taken to Obtain a Longer File Size

3.1.3. Result of Test 3 *(Effect of sample rate on noise of Fourier Coefficients)*:

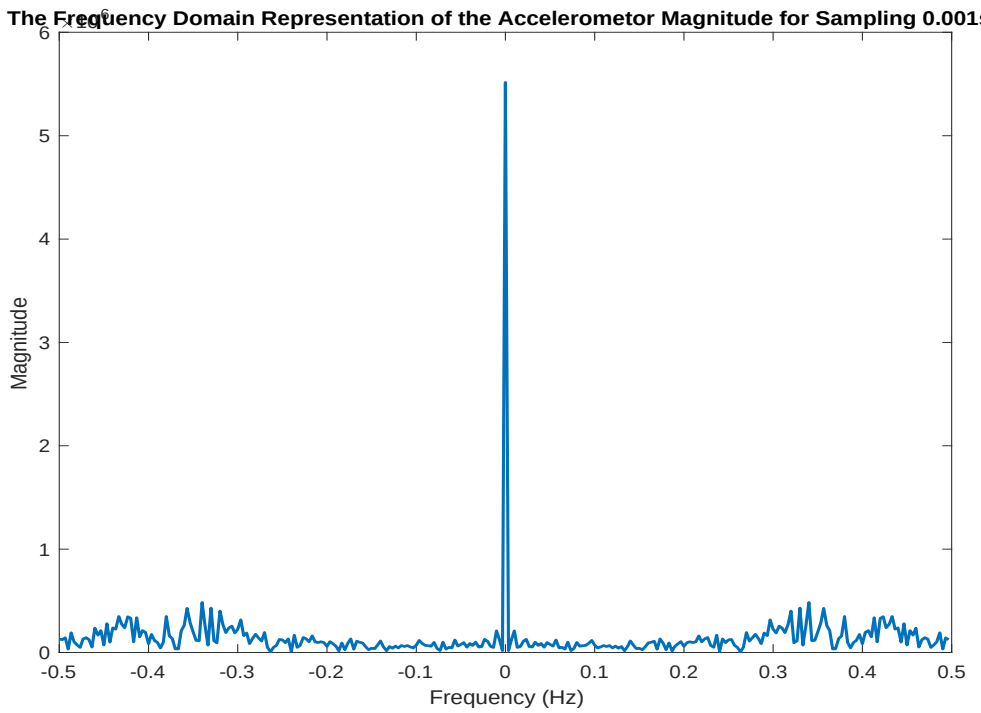
As mentioned in Test 3, the Fourier domain expressions were obtained for Accelerometer readings. Both will be shown below, however, the Accelerometer graphs will be shown for sampling rates of 0.1, 0.01 and 0.001 seconds.



Graph 19 - The Frequency Domain Representation of the Accelerometer Magnitude for Sampling 0.1s



Graph 20 - The Frequency Domain Representation of the Accelerometer Magnitude for Sampling 0.01s



Graph 21 - The Frequency Domain Representation of the Accelerometer Magnitude for Sampling 0.001s

Graphs 19 to 21 display the frequency domain expression of the Accelerometer Magnitude for different sampling rates. In this context, the sampling rate refers to the number of data that is written per second. The smaller the sampling rate the more data is written to the file per second and thus the larger the file will be. Although this may seem counter intuitive as a smaller sampling rate will result in a much larger file, it does offer the possibility of more accurate data. However, as seen by the graphs above, the smaller the sampling rate becomes, the more noise is introduced in the signal. Hence, when one chooses a particular sampling rate, they must consider the trade-offs that are concerned with noise and data accuracy.

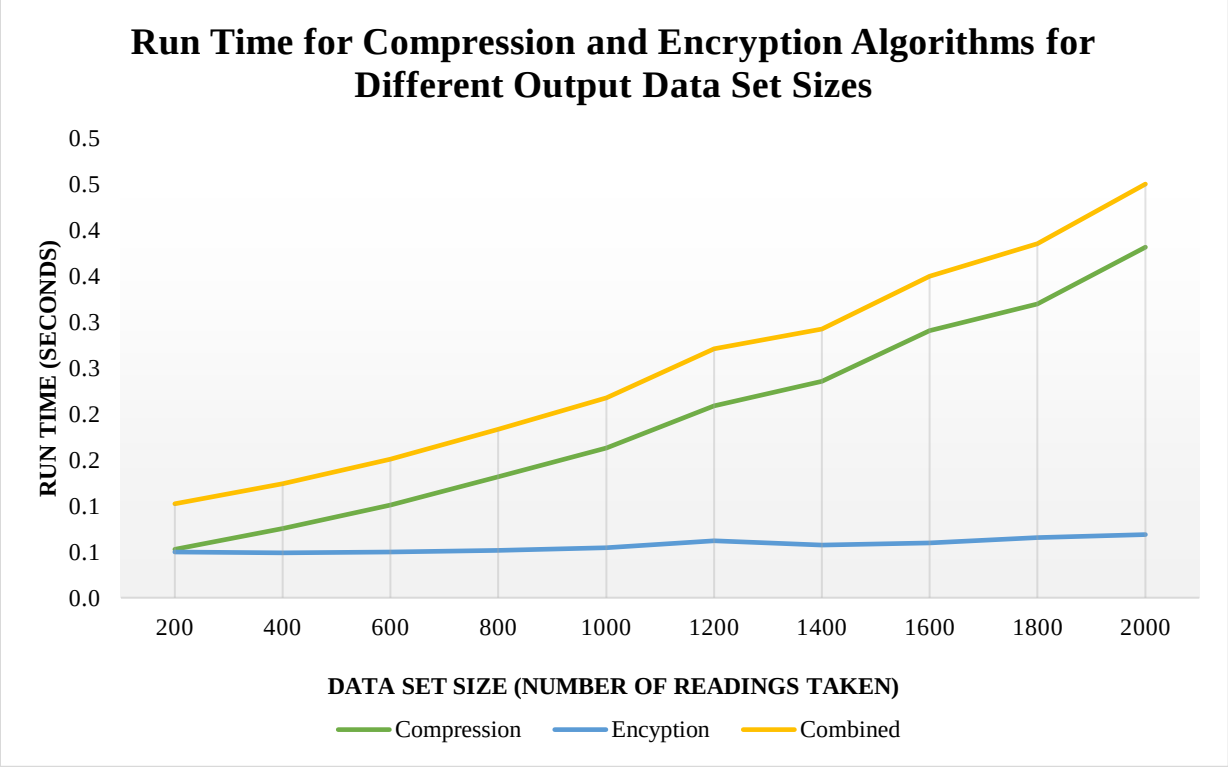
3.1.4. Result of test 4 *(Run time for different data set sizes through compression and encryption combined algorithm)*:

This run produced the following plot output in the Raspberry Pi Zero terminal. Showing, among other things, the compression and encryption time for the run.

```
pi@raspberrypi:~/EEE3097S $ python3 CompressEncrypt.py
Uncompressed size: 139674
Compressed size: 42282
Total compression time: 0.1633129119873047
Enter the password:testing
Total encyption time:  0.05501604080200195
Total RAM usage:  100.0
```

Figure 12 – Command Line Results of Test 3

The different run times for the different data file sizes was then graphed to indicate the relationship between runtime and data set size. This was plotted for both the encryption and compression algorithms separately and then together as an overall system runtime. This can be seen in the graph below.

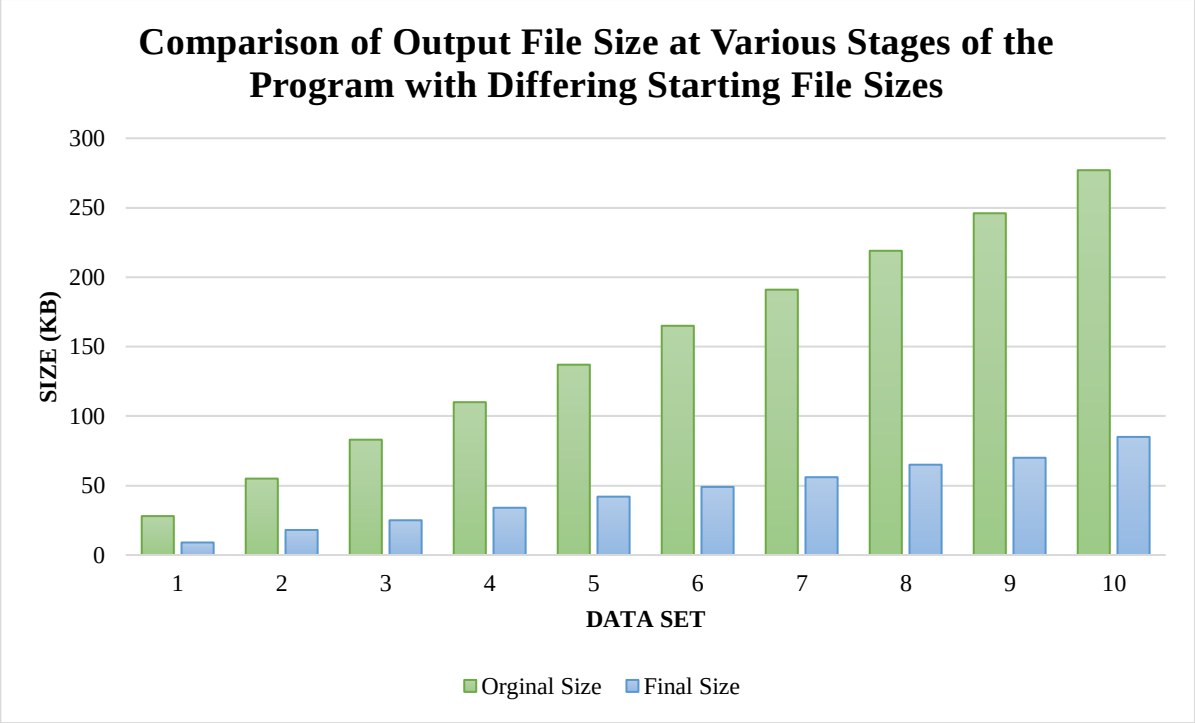


Graph 22 - Run Time for Compression and Encryption Algorithms for Different Output Data Set Sizes

The compression run time has a polynomial, of order 2, trendline (as can be seen). This therefore shows that as the data set size increases the run time will increase at a faster rate. This therefore means that choosing a data set size is not a trivial choice as making it too big will mean that its runtime is greatly increased. As can be seen in the graph above the encryption run time only has a very slight upwards trend. This is because it is encrypting the already compressed file and therefore the files are relatively small compared to the size they started out as. Finally the overall combined time once again was a polynomial, of order 2, trendline. This is as it follows the shape of the compression curve with only a slight increase at each point from the almost constant encryption time.

3.1.5. Result of Test 5 *(Final output file size compared to input file size after compression and encryption combined algorithm)*:

The final test for the overall system consisted of testing the ratio of encryption for the various file sizes. The comparison between the starting file size and the compressed version of that particular file can be seen in the graph below:



Graph 23 - Comparison of Output File Size at Various Stages of the Program with Differing Starting File Sizes

The graph above shows that as the original file's sizes increase linearly, so does the final file size (after both compression and encryption), just at a much slower rate. This indicated that increasing the file size by a certain factor does not increase the compressed size by the same factor. This is as the constant size change in the original data sets leads to a constant size increase in the compressed file which is just much smaller. This means that increasing the file size does not increase the compressed file size as much so it could be worth it to increase the file size if the compressed file does not increase by a substantial amount. The point where this relationship changes must be found in order to not go over the threshold.

The ratio of compression can be seen in the table below.

Original Size (kB)	Final Size (kB)	Ratio of Size
28	9	3,11
55	18	3,06
83	25	3,32
110	34	3,24
137	42	3,26
165	49	3,37
191	56	3,41
219	65	3,37
246	70	3,51
277	85	3,26

Table 26 - Ratio of Compression Results of Test 5

As can be seen in the table above the ratio of compression between the original file and the final compressed file is quite constant at around 3.2. The following section will provide a more in-depth discussion regarding the analysis of all compression algorithms for different data sizes and their corresponding results.

3.2. Results of the Experiments for Compression Block:

As discussed in the paper design and Section D submission, it was found that the zlib library, which is a lossless compression technique, should be used. As such, multiple tests were conducted to measure the compression ratio, percentage of compression, execution speed and whether it was indeed lossless or not. To do so, many other techniques were used for this experiment namely the gzip, zip, lzma and bz2 on top of the zlib. It must be noted that all the above techniques are lossless and implement zlib in their own way [16].

The main difference that the tests conducted in this section were done so on a Raspberry Pi Zero as opposed to a MacBook Pro like in Section D. The specs of the Raspberry Pi Zero are as follows: A ARM11 1Ghz CPU, 512MB RAM, 5V Power and 2.4GHz 802.11n Wireless LAN. This is a stark difference to the MacBook Pro that has a 3.1 GHz Dual-Core Intel Core i5 Processor with an 8 GB 2133 MHz LPDDR3 memory. As such the results obtained will differ significantly to that of Section D not only because different file sizes are being used but also because the tests are being run on a completely different hardware device. The results will be more relevant as the Raspberry Pi Zero will be used in the buoy and not a MacBook Pro.

As can be seen on the following page, the table shows the file size (in bytes) of each of the original data files that were provided to us. It then shows the compressed file size (in bytes) for each compression technique for every data file [18]. The averages of each compression technique were then calculated using Excel and its many functions. These averages were then used to calculate the compression ratio and percentage of compression for each compression method. It must be noted that percentage of compression refers to the percentage that the original file was decreased *to* as opposed to the percentage that the original file was decreased *by*. For greater clarity this is represented by the mathematical equation shown below:

$$\%\ Decreased\ \boldsymbol{by} = 1 - \%Decreased\ \boldsymbol{to}$$

Please refer to the next page that contains all the tables and appropriate figures pertaining to the compression testing for various techniques.

Below is the table that was mentioned in the previous page that contains the compression ratio and compression file size percentages for each compression technique used:

File Number	Original File Size (bytes)	Compression Method				
		bz2 (bytes)	gzip (bytes)	zip (bytes)	zlib (bytes)	lzma (bytes)
200.csv	28 374	9 040	10 981	11 158	10 965	9 300
400.csv	56 049	18 257	21 842	22 186	21 826	18 644
600.csv	84 179	25 338	31 460	32 064	31 444	26 976
800.csv	111 868	34 241	42 346	43 066	42 330	36 284
1000.csv	139 674	42 282	52 544	53 523	52 528	44 948
1200.csv	168 328	49 486	62 483	63 695	62 467	53 576
1400.csv	194 742	56 699	71 933	73 158	71 917	61 800
1600.csv	223 496	65 754	82 496	83 950	82 480	71 048
1800.csv	251 876	70 927	90 970	92 463	90 954	77 160
2000.csv	283 524	86 876	105 655	107 177	105 639	90 592
Average (bytes)	**154 211**	**45 890**	**57 271**	**58 244**	**57 255**	**49 033**
Compression Ratio		**0,2976**	**0,3714**	**0,3777**	**0,3713**	**0,3180**
Compression File Size		**29,76%**	**37,14%**	**37,77%**	**37,13%**	**31,80%**

Table 27 - Compression Ratio and Percentage of Compression for Each Compression Technique

	Compression Method				
Python File	**bz2**	**gzip**	**zip**	**zlib**	**lzma**
Comparison1.py	Yes	Yes	Yes	Yes	Yes
Comparison2.py	Yes	Yes	Yes	Yes	Yes

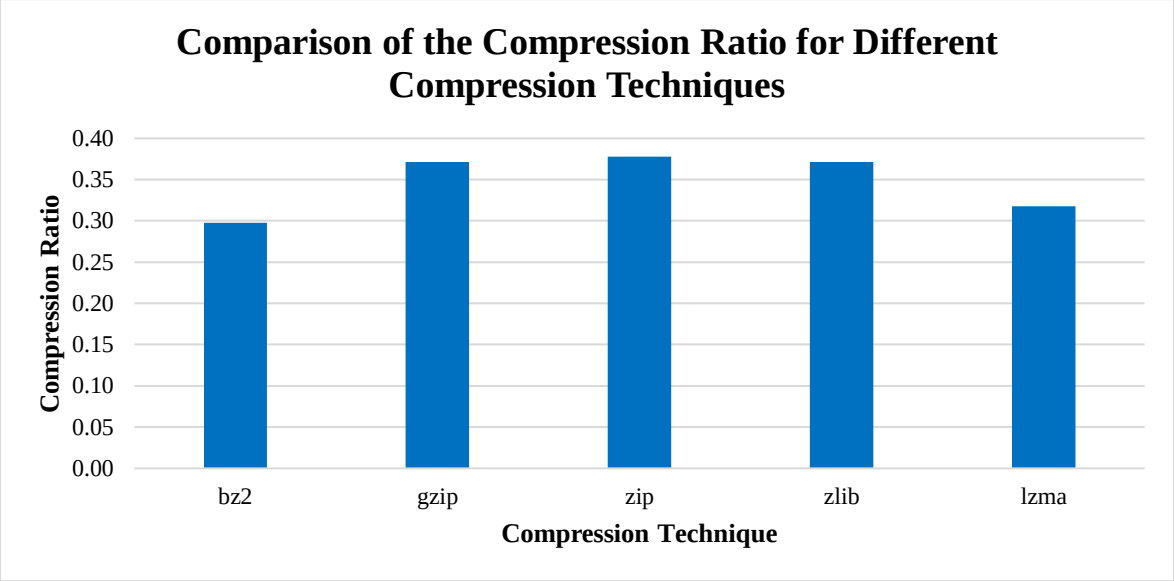
Table 28 - Lossless or Lossy Determination for each Compression Technique

Table 28 on the previous page shows the result when each compression technique's decompressed file was compared to the original file it compressed to check whether the file and all data inside it was preserved. Following this, table 29 below shows the execution time in seconds that each compression technique took to compress each file. Furthermore, it shows the average execution time for every compression method [19].

File No.	bz2 execution time	gzip execution time	zip execution time	zlib execution time	lzma execution time
200.csv	0,0504	0,0310	0,0285	0,0283	1,1770
400.csv	0,0736	0,0882	0,0722	0,0902	2,3515
600.csv	0,1012	0,1705	0,0982	0,1631	3,4643
800.csv	0,1313	0,2176	0,1387	0,2174	4,6472
1000.csv	0,1632	0,2814	0,1685	0,2741	5,6215
1200.csv	0,1982	0,3683	0,2177	0,3713	6,6321
1400.csv	0,2383	0,3852	0,2449	0,3764	7,9520
1600.csv	0,2756	0,4576	0,2742	0,4428	9,0145
1800.csv	0,3151	0,6457	0,3601	0,5265	10,4507
2000.csv	0,3618	0,6615	0,4252	0,6220	11,8453
Average	**0,19087**	**0,33070**	**0,20284**	**0,31120**	**6,31560**

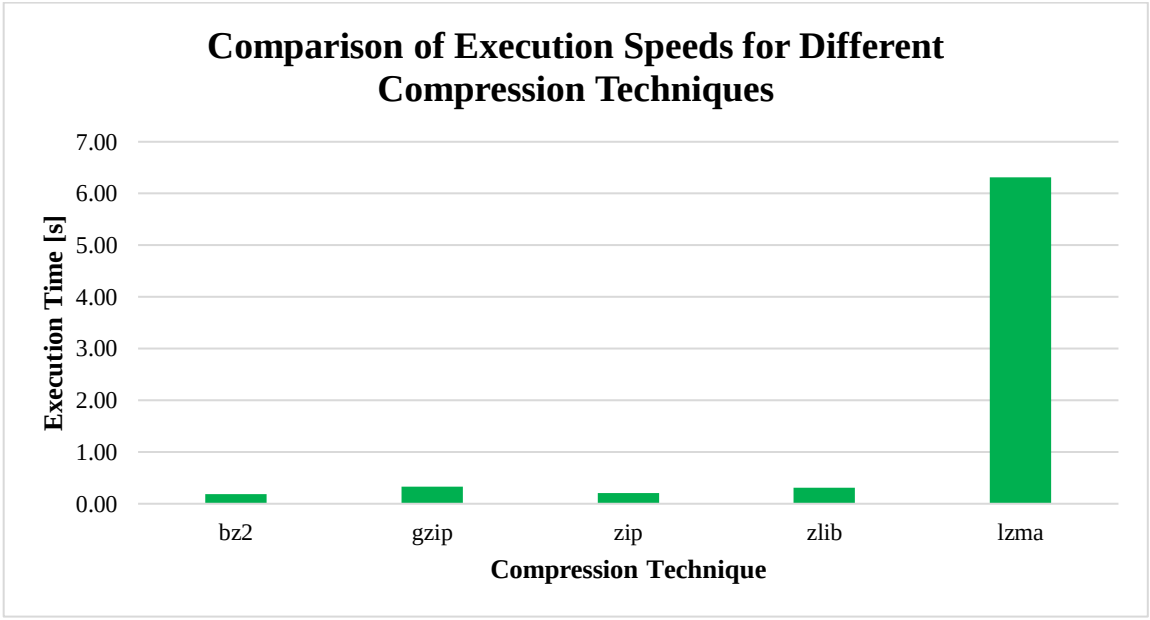
Table 29 - The Average Execution Time of Every Compression Technique

When referring to table 27 and the graph below, it is evident that the bz2 compression technique was the fastest with a compression ratio of 0.2976 and best percentage of compression of 29.76%. This compression technique was the one we decided to use after progress one's results and analysis. In comparison to the golden measure of zlib it is faster by more than 8% showing far superior performance than the other benchmarks. The lzma technique came second, then gzip and zlib with zipfile being the only compression technique that performed slower than the golden measure of zlib. The graph below depicts the graphical representation of the compression ratios for each compression technique in table 27.



Graph 24 - Comparison of the Compression Ratio for Different Compression Techniques

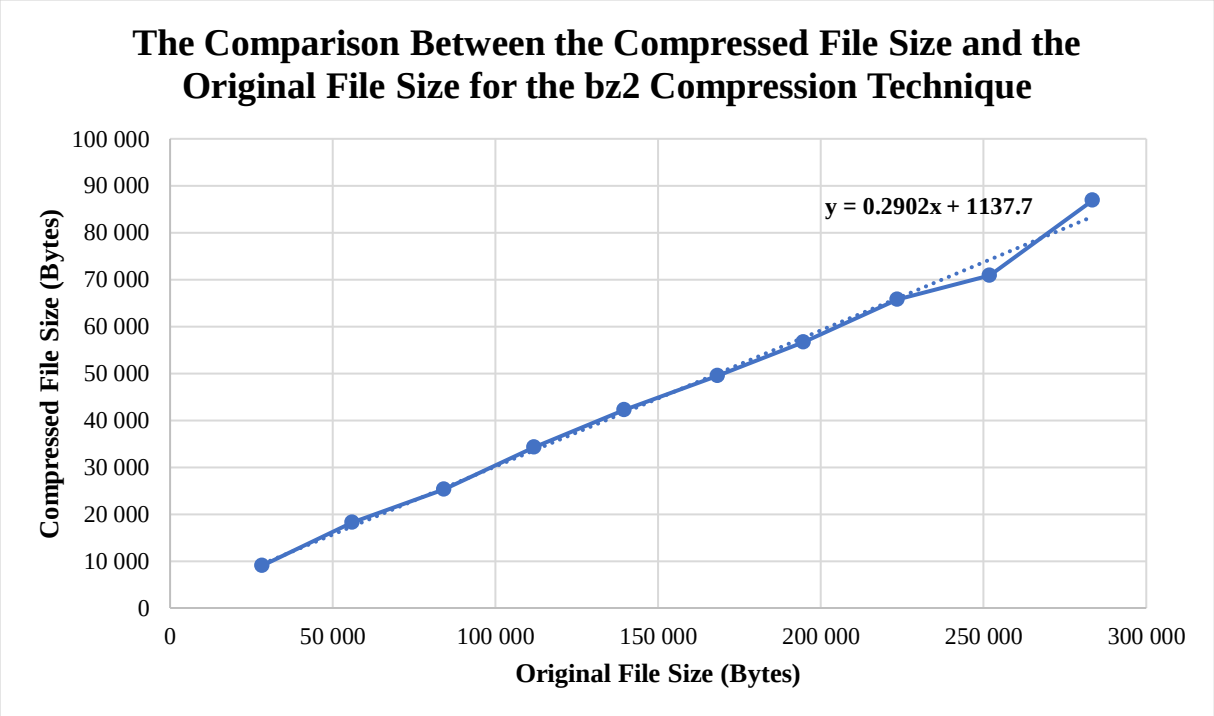
Table 28 are the results when the compressed file for every compression technique was compared to the original uncompressed file using the python files comapison.py and comparison2.py. These two python files performed a shallow and deep match where the shallow match compares the files size and date modified while the deep match compares the actual contents of both files. If either does not evaluate then a false is returned, however, because the compression techniques used above are lossless, they all evaluated to true.



Graph 25 - Comparison of Execution Speeds for Different Compression Techniques

Lastly, the time python library was used to time the execution speeds of each compression technique. What was interesting is that while lzma compressed the original file the second most, it had the slowest execution time almost averaging 6.5 seconds. On the other hand, the bz2 compression method had the fastest execution time averaging 0.19087 seconds. The bz2 and the zip execution speeds were the only compression techniques to perform better than the golden measure which was 0.3112. These execution speeds can be seen in table 3 and in graph 26.

Unlike Section D where there wasn't a clear winner, the tests conducted in this section showed that the bz2 compression technique provides both best compression ratio and the fastest compression time. Although, these values were determined for smaller data set sizes than in Section D, they support the original decision made to choose the bz2 compression technique. Furthermore, most techniques, if not all, met the user's requirements and thus the ATPs, however, this will be discussed more in-depth in the section to follow. It is easy to implement any one of these techniques as all the code that was used to gain the results can be found at the respective links of the bz2, gzip, zip, zlib and lzma python files.



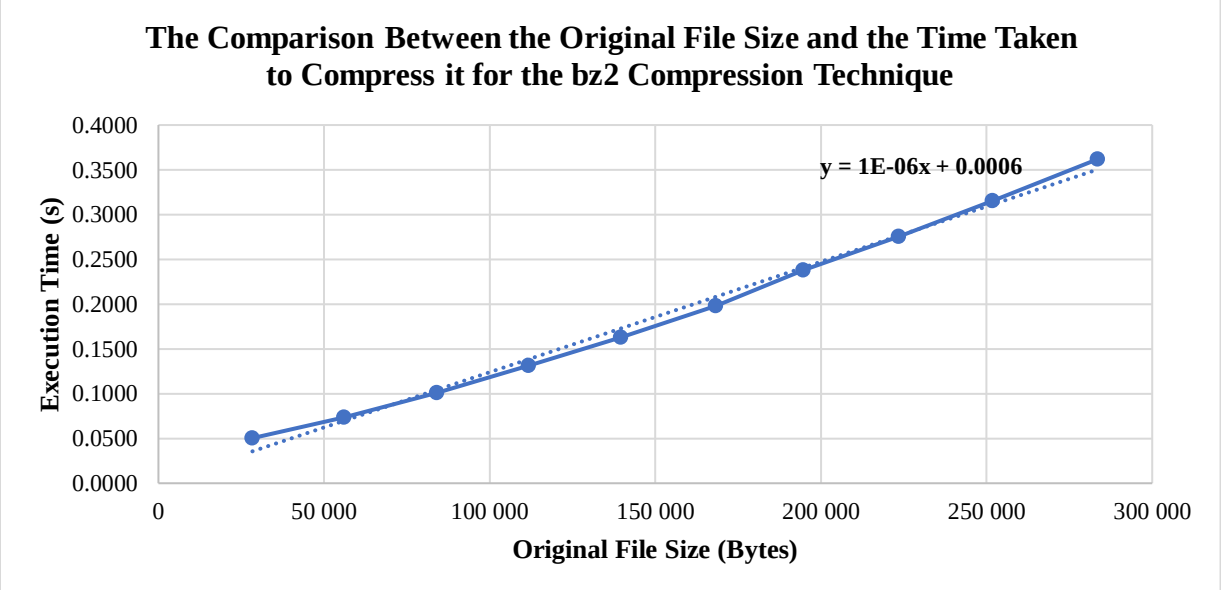
Graph 26 - The Comparison Between the Compressed File Size and the Original File Size for the bz2 Compression Technique

With bz2 being the clear compression technique, Excel's line of best fit function was used to find the equation that best represents the relationship between the compressed file size and the original file size. Naturally, the file size has a direct relationship with the compressed file size and the graph 27 depicts an almost near linear relationship between the two variables. This provides useful information in that the compressed size of a random file may be determined. As the user, they can then determine the most appropriate compressed file size that is produced by a chosen original file size. This formula is shown below where y represents the compressed file size in bytes and x represents the original file size, also in bytes.

$$y = 0.2902x + 113.7 \quad \textbf{(1)}$$

For example, the compressed file size for an original size of 9MB is expected to be around the region of 2.6 MB when the bz2 compression technique is used.

In addition to determine the compressed file size the average execution time for a particular file size may be determined when run on the Raspberry Pi Zero. Again, with bz2 being the clear compression technique, Excel's line of best fit function was used to find the equation that best represents this relationship. Like the above relationship there is a direct relationship between the original file size and the execution time when the file is being compressed. The graph on the following page depicts the relationship between these two variables with a straight line being used to describe the relationship of the two variables.



Graph 27 - The Comparison Between the Original File Size and the Time Taken to Compress it for the bz2 Compression Technique

This formula is shown below where y represents the expected execution time in seconds and x represents the original file size in bytes.

$$y = 10^{-6}x + 0.0006 \quad \textbf{(2)}$$

For example, the execution time for an original size of 9MB is expected to be around the region of 10 seconds when the bz2 compression technique is used. In comparison to the combined compression and encryption algorithm execution time of 14 seconds this is a decent approximation.

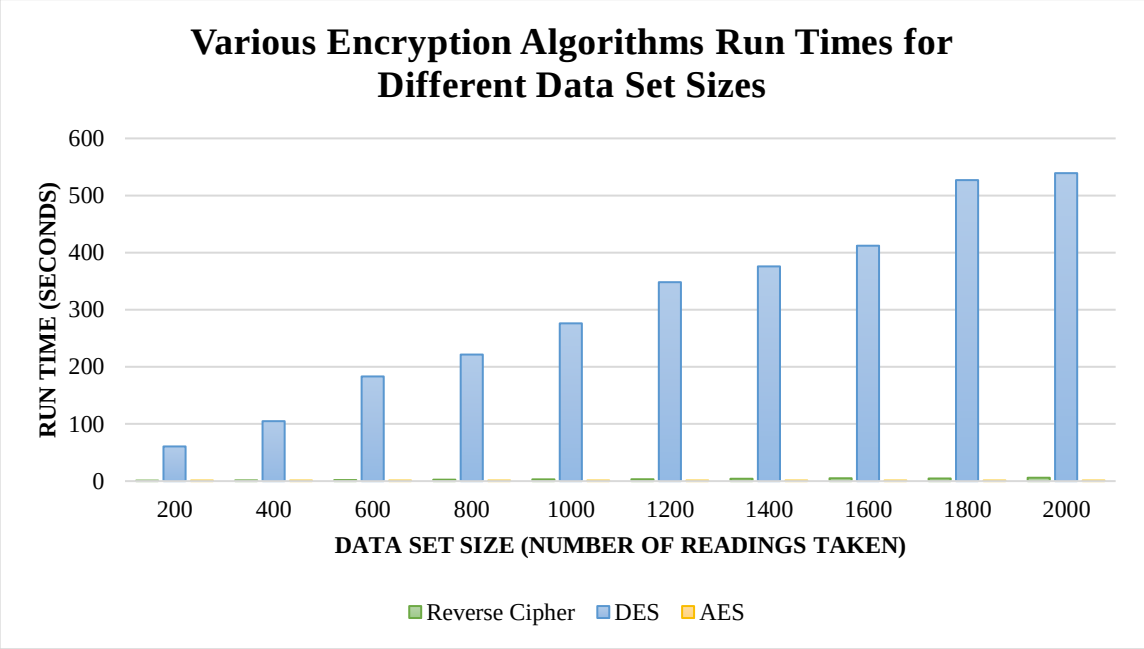
3.3. Results of the Experiments for Encryption Block:

All the following encryption tests were run on the Raspberry Pi Zero, and the encryption tests run previously using different data sets were also run solely on the Raspberry Pi Zero.

Data Set Size	Reverse Cipher (s)	DES (s)	AES (s)
200	0,5037	60,7009	0,0528
400	1,2924	104,9444	0,0613
600	1,8193	183,2440	0,0645
800	2,5437	221,5583	0,0779
1000	3,0833	276,1812	0,0843
1200	3,2716	348,2573	0,0949
1400	4,1523	375,8778	0,0997
1600	5,0013	412,0925	0,1008
1800	4,6116	526,9367	0,1044
2000	5,9221	539,1343	0,1111

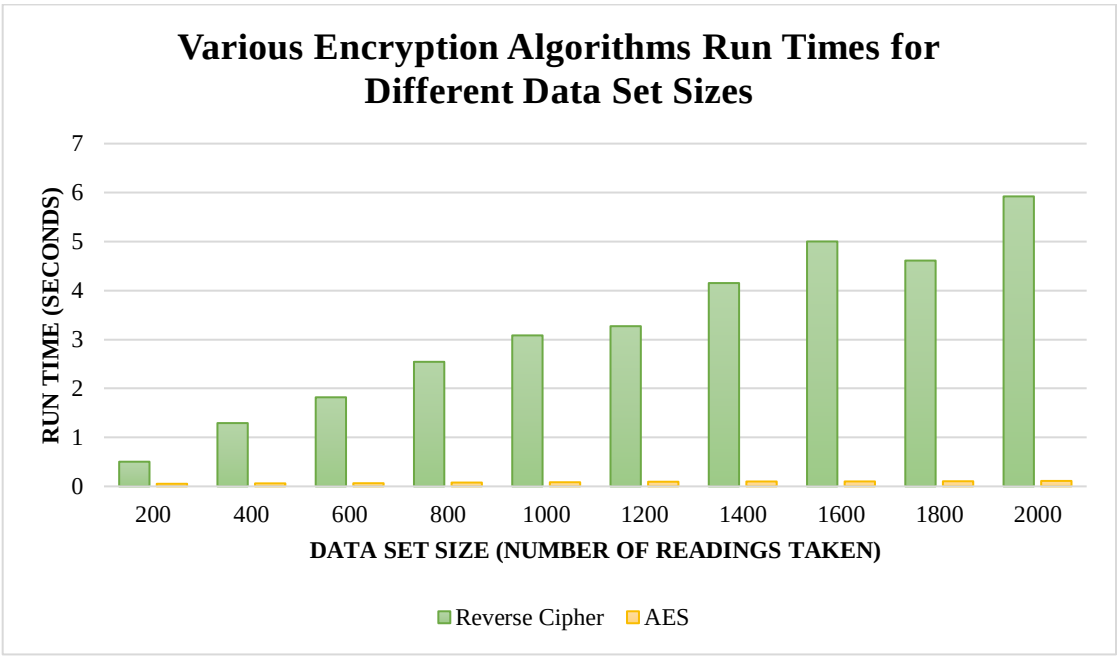
Table 30 - Results of the Encryption Execution Time Tests

As before the three methods are lossless but only AES is seen as secure. The security level of the Reverse Cipher was abysmal and was therefore used more as a golden measure to compare the other two methods too as it is commonly seen as the simplest form of encryption. When the run time for the 10 different data sets (produced from the above run) is looked at both the Reverse Cipher and AES methods run quickly where the DES runs much slower. This can be seen in the figure below which compares the different algorithms run time for the data sets.



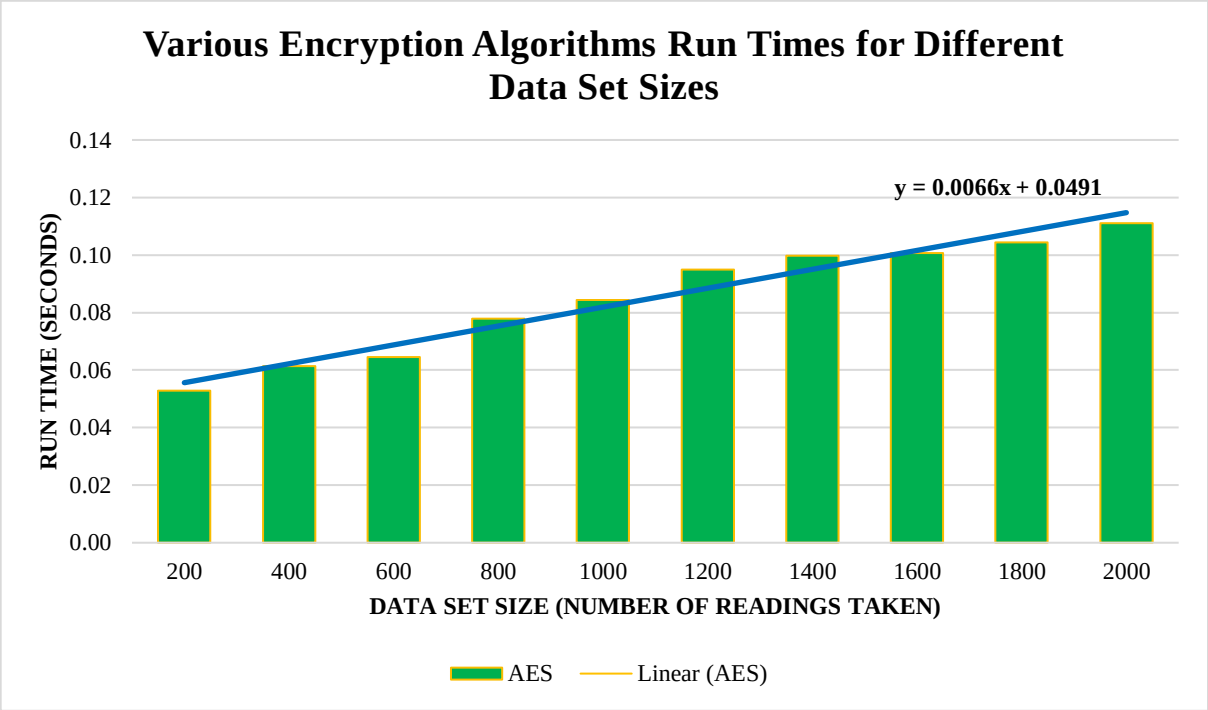
Graph 28 - Various Encryption Algorithms Run Times for Different Data Set Sizes

Therefore DES was then eliminated as an option due to its slow running speed which would use up too much of the buoys battery. Note that these results are consistent with the observed results with the previous data sets.



Graph 29 - Various Encryption Algorithms Run Times for Different Data Set Sizes

Therefore this left AES which is a widely accepted encryption standard with a good security level, is lossless for data encryption and has a running speed on average 0,006419s per 16Kb of data which is much better than the wanted 0.01s/16Kb. As can be seen in the graph above the run times for the different data sets for AES and Reverse Cipher were graphed without the data for DES so that they could be better seen. Then a graph of just AES was plotted for further detail.



Graph 30 - Various Encryption Algorithms Run Times for Different Data Set Sizes

The graph above shows that the linear increase in data set size corresponds to a linear increase in the encryption algorithm run time. This relationship governs that increasing the data set size will increase the run time in accordance with the trendline shown (y=0.0066x+0.0491). Therefore a compromise must be found with the run time of the algorithm compared to how long to get the IMU to gather data.

This method was implemented in Python due to its compatibility with the Raspberry Pi Zero, the abundance of available libraries for the encryption standards and to easily tie in with the compression algorithm which is also written in Python. Note that code was written to first run the encryption, then decryption algorithms and then test the output of the decryption algorithm against the input data to the encryption algorithm. This produced a terminal output that looked like the figure on the next page.

```
pi@raspberrypi:~/EEE3097S $ python3 AES.py
Enter the password:testing
Total run time:  0.06633830070495605
Total RAM usage:  100.0
Password: testing
pi@raspberrypi:~/EEE3097S $ python3 TestLostData.py
The result will return True if the files match and False if they do not:
Shallow match:  True
Deep match:  True
```

Figure 13 - AES Encryption Terminal Output

All the tests run were on comparatively smaller data sets due to time constraints in waiting for the Sense HAT to fetch the data. But as a range of values was determined the predicted values for larger datasets can be determined through data extrapolation. Note that all these findings are consistent with those from the previous set of tests in Section D where the given IMU data sets were used. This shows that the AES algorithm works as expected for the expected data (it is lossless, most secure and faster encryption algorithm tested). Furthermore, these algorithms may be found in our GitHub at the following AES, DES and Reverse Cipher links.

3.4. Effect of Different Data on the System:

Adding white-Gaussian noise to the raw data set will not change the effectiveness of the compression or the encryption. This is because both were implemented to be lossless and therefore no data is lost and the bottom 25% of Fourier coefficients are not lost. The issue of the noise will only therefore become relevant when the data is being used once decrypted. Making the data be under-sampled, therefore less data to transfer, would have the effect of speeding up both the compression and the encryption blocks as they are both dependent on the size of file to be transferred.

The idea of noise comes in when the Sense HAT pulls data very quickly therefore generating noise in the data. This was tested by changing the sampling rate and graphing the Fourier transforms of the data to view the noise.

4. Acceptance Test Procedures:

ATPs for the Sense Hat B validation are explained and shown above. Please refer to section ‘Validation Tests’ within the ‘IMU Module’ section for these.

4.1. Sense Hat (B) Compression ATPs:

ATP Name	ATP Description in Section D	New ATP Description	ATP met in Design	ATP in Design Comment
Data loss acceptance criteria	Little to no data is lost within the upper 75% of Fourier coefficients, and within the lower 25% of Fourier coefficients being 100% not lost or changed.	The comparison1.py and comparison2.py python files return true when the original file and decompressed file are tested for any data loss.	Yes	This ATP was met as all compression techniques returned a result of True when compared to their original files. This is as expected as all compression techniques are lossless and as such no data should be lost
Compression file size acceptance criteria	The compressed file size should be between 40% to 60% of the original file size.	The compressed file size should be between 25% to 50% of the original file size	Yes	As shown in the results section all the compression techniques met this ATP design. Furthermore, the bz2 compression technique fell on the lower side of the specified range. This shows its superiority as it outperformed all other compression techniques in this ATP.
Compression time acceptance criteria	The time taken to compress the data (the running of the compression code) must be as low as possible, less than 10 seconds per 16Kb of data.	The time taken to compress the data (running the compression code) must be less than 1 second for every 16KB of data	Yes	All techniques besides lzma compression met this ATP. Despite lzma's ability to compress the original file very small, it fails to compress the data within 1 second for every 16KB of data. However, the chosen compression technique bz2 fell well within this ATP as it had an execution time of 0.19 seconds.
Compression ratio acceptance criteria	Ratio of the original file size and compressed file size must be less than 0.9.	The ratio of the original file size and compressed file size must be less than 0.4	Yes	Similar to the compressed file size, was met by all compression techniques with some outperforming the others. As mentioned in the results section, the bz2 compression technique had the best ratio of 0.2976 which is far less than the specified ATP requirement of 0.4

Table 31 - Analysis of Compression ATPs

4.2. Sense Hat (B) Encryption ATPs:

ATP Name	ATP Description in Section D	New ATP Description	ATP met in Design	ATP in Design Comment
Execution time acceptance criteria	The execution time of the algorithm must take less than 10 seconds for every 16Kb of data.	The execution time of the algorithm must take less than 0.01 second for every 16Kb of data.	Yes	The ATP was met and (a speed of 0.006419s/16Kb was achieved) therefore showing a slight improvement on the new ATP.
Lossless encryption acceptance criteria	Decrypted data must contain the same data as the file before it was encrypted (specifically the lowest 25% of Fourier coefficients).	Decrypted data must contain the same data as the file before it was encrypted (specifically the lowest 25% of Fourier coefficients).	Yes	The AES method of encryption was lossless. This was tested using the TestLostData.py code that tested (both shallow and deep) whether the original data matched the decrypted file.
RAM usage encryption acceptance criteria	Less than 5Kb of Random Access Memory (RAM) must be used in the encryption process.	The use of Random Access Memory (RAM) must allow for functionality for all needed devices to continue (not create errors in the program) from filling up the RAM when just the encryption algorithm is being run.	Yes	This was tested on the Raspberry Pi Zero by allowing the code to run and if it completed successfully it was determined that the RAM usage did not stall the Raspberry Pi Zero and therefore met the ATP.
Data encryption acceptance criteria	Data encryption must be secure, therefore a key needs to be used in the encryption process.	Data encryption must be secure, therefore a key needs to be used in the encryption process.	Yes	The AES encryption method uses the same key to encrypt and decrypt the data. If this key is not leaked then the data will remain very secure. The key can be made as random as needed to make it more secure. This process could be done using a random generator.

Table 32 - Analysis of Encryption ATPs

F. Consolidation of ATPs & Future Plans:

1. Consolidation of ATPs:

1.1. Final Compression Acceptance Test Procedures:

The final acceptance test procedures for the compression block are shown below. After multiple submissions this is the most accurate and best possible ATP description. The python code and overall performance of the module met all the ATP designs specified with there being no need to make any adjustments to the original specifications. The final ATP table is shown in table 14 below.

ATP Name	Final ATP Description	ATP met in Design
Data loss acceptance criteria	The comparison1.py and comparison2.py python files return true when the original file and decompressed file are tested for any data loss.	Yes
Compression file size acceptance criteria	The compressed file size should be between 25% to 50% of the original file size	Yes
Compression time acceptance criteria	The time taken to compress the data (running the compression code) must be less than 1 second for every 16KB of data	Yes
Compression ratio acceptance criteria	The ratio of the original file size and compressed file size must be less than 0.4	Yes

Table 33 - Final Compression ATP Table

1.2. Final Encryption Acceptance Test Procedures:

The final acceptance test procedures for the encryption block are shown below. After multiple submissions this is the most accurate and best possible ATP description. The python code and overall performance of the module met all the ATP designs specified with there being no need to make any adjustments to the original specifications.

ATP Name	New ATP Description	ATP met in Design
Execution time acceptance criteria	The execution time of the algorithm must take less than 0.01 second for every 16Kb of data.	Yes
Lossless encryption acceptance criteria	Decrypted data must contain the same data as the file before it was encrypted (specifically the lowest 25% of Fourier coefficients).	Yes
RAM usage encryption acceptance criteria	The use of Random Access Memory (RAM) must allow for functionality for all needed devices to continue (not create errors in the program) from filling up the RAM when just the encryption algorithm is being run.	Yes
Data encryption acceptance criteria	Data encryption must be secure, therefore a key needs to be used in the encryption process.	Yes

Table 34 - Final Encryption ATP Table

1.3. Change in Compression Specifications:

The only specification that was changed for the compression block related to the measurement of the battery power.

Original Specifications	New Specification
The compression of the data is efficient and uses only 1% battery power to process the data.	This specification is going to be tied to the execution time and compression ratio as this relates to the battery power consumption and CPU usage.

Table 35 - Final Change in Compression Block Specifications

1.4. Change in Encryption Specifications:

The only specification that was changed for the compression block related to the measurement of the battery power.

Original Specifications	New Specification
Less than 5Kb of Random-Access Memory (RAM) must be used in the encryption process.	The use of Random-Access Memory (RAM) must allow for functionality for all needed devices to continue (not create errors in the program) from filling up the RAM when just the encryption algorithm is being run.

Table 36 - Final Change in Encryption Block Specifications

2. Future Plans

With the final report being submitted we are incredibly happy with the final produce that we managed to produce. That being said there are some improvements that can be to our final product and also future plans that may be recommended to the buoy team. The only aspect that our product is lacking in is the automation department. Unfortunately due to time constraints and other factors we were unable to design a process that is fully autonomous and only requires a start at the push of a button. Currently our design has three main steps. The running and pulling data from the Sense Hat B, the compressing and encrypting of the data produced by the Sense Hat B and finally the decompression and de-encryption of the data. Although the decompression and de-encryption of data is often done from a remote location to that of the buoys location, the pulling of data and compressing and encrypting of it are done on the Raspberry Pi Zero and hence the buoy. If the this process may can be made automate so that a certain trigger or interrupt within the larger buoy system invokes the process the project would be 100% implementable. That being said, this was not a requirement of the project and so we are extremely happy with the overall efficiency of our code.

Furthermore, research that was conducted in our paper design and requirements analysis showed that the fastest and most efficient programming language in terms of battery power consumption and execution time was the C language. Unfortunately we were not as well rehearsed in C language as we were in python and as such opted to use python for our coding. The same research showed that the python programming language is less efficient and slower than the C language. As such, it would be strongly recommended that one rewrites the python code in C language, wherever possible. Not only will this speed up the compression and encryption of data but it will also use less battery power, which was one of the major user requirements of the buoy system.

Additionally, the data transfer system was not modelled and for this code to fulfil its final goal of receiving, compressing and encrypting data which is then transferred using the iridium network (a global satellite communications company) that is located in Antarctic, it must also incorporate transfer system code.

Finally, the encryption algorithm takes in a command line parameter for the needed password. This password can be stored in the buoys memory and called whenever needed.

G. Conclusion

An ARM based digital device, otherwise known as a Raspberry Pi Zero, is used to compress, encrypt, and transmit all relevant data packets from an inertial measurement unit (IMU) for further analysis of the environments surrounding the Antarctic region. This Pi Zero design system is one of many subsystems that make up the SHARC BUOY designed by Jamie Jacobson.

The data used within the compression and encryption process was done in two steps. The first step was using data provided to use from the authors of the loken_21 papers from the IMU they used. The second step consisted of using data obtained from the Sense HAT B which was attached to the Raspberry Pi Zero and used to fetch data which was outputted to a csv file for later compression and encryption. The data from both the loken_21 papers and the Sense HAT B was run though a multitude of tests to see how it plots in both the Fourier and time domains. This was to test the effect of sampling rate on the noise seen and to identify whether the Sense HAT B was running as expected (with regards to temperature, pitch, yaw and roll).

These tests on the showed that the Sense HAT B was working was expected with regards to its temperature, pitch, yaw and roll readings. Additionally, it can be seen that when the sample rate was increased this led to an increase in noise seen on the Fourier transforms of the various data sets. This leads to the conclusion that increasing the sample rate to much would several degrade the quality of the data recorded but this must be balanced with the requirements of obtaining the full set of the lower 25% of Fourier coefficients.

Various methods of compression and encryption are compared to determine the most suitable algorithm for this application. First a theoretical comparison of the algorithms was made and then some were chosen for testing in implementation. This were then compared on multiple levels (such as speed and compression ratio) to determine the most suitable algorithm for the type of data outputted by the IMU.

The compression algorithms tested were the bzip2 (bz2), GNU zip (gzip), zip, zlib and Lempel–Ziv–Markov chain algorithm (lzma). All these algorithms were based in some form or another on the zlib algorithm. In light of this, for the set of experiments that were conducted the zlib compression technique was used as the golden measure. These experiments consisted of the compression of files with different sizes (measured in bytes) using the different compression techniques, the measuring of the execution time of the various compression algorithms and comparing the decompressed file to the original file for every technique. For the experiments tested using simulated or old data, the algorithm that performed the best out of all three experiments was the bz2 as it had the second-best compression ratio of 0,35036 while it had the second-best execution time of 0,81095. This experiment was tested on a MacBook Pro

with a 3.1 GHz Dual-Core Intel Core i5 Processor with an 8 GB 2133 MHz LPDDR3 memory and so results differed to that of the Raspberry Pi Zero, however, the principle regarding the best performing compression algorithm on average is still applicable. For the experiments tested using a different IMU or the Sense HAT B, the algorithm that performed the best out of all three experiments was the bz2 as it had the second-best compression ratio of 0,35036 while it had the second-best execution time of 0,19087. Unlike the previous experiment, this experiment used the Raspberry Pi Zero with a ARM11 1Ghz CPU, 512MB RAM, 5V Power and 2.4GHz 802.11n Wireless LAN.

The first experiment was used to determine the various compression ratio and percentage compression of each algorithm. While second experiment tested the speed up ratio of each of each algorithm, the last experiment determined whether each technique was lossless or lossy. All these results were fundamental to determining the technique that was used for the designed system and whether they met the various subsystems Acceptance Test Procedures (ATPs).

The encryption algorithms tested were a Reverse Cipher encryption, Data Encryption Standard (DES) and the Advanced Encryption Standard (AES). The reverse cipher was used as a golden measure due it being seen as one of the most primitive types of encryptions. These algorithms were tested on the following factors: execution time (to be used to display battery usage), security level of encryption, whether the encryption is lossless and the RAM usage of the system on the Raspberry Pi Zero. These factors were chosen to encompass the needs of the relative Acceptance Test Procedure (ATPs) and to test the various algorithms for their suitability in this project.

All the encryption tests were run on the Raspberry Pi Zero. As can be seen for both the loken_21 papers and the Sense HAT data sets all three methods of encryption (reverse cipher, DES and AES) were lossless. The security of the reverse cipher was abysmal and was used as a golden measure to compare the other two algorithms too. DES had a better security level but was still not that secure while AES was seen as quite secure and is widely accepted as a sufficient encryption algorithm. The run time of the reverse cipher and AES was quite close and relatively fast when encrypting both data set types (when compared per byte). While the DES method took a significantly longer time for both data set types and was therefore discarded on the basis that it would use to much battery power which is a scarce resource on the buoy. Therefore, this left AES which is a widely accepted encryption standard with a good security level, is lossless for data encryption and has a running speed on average 0,002883205s per 16Kb of data for loken_21 papers data sets and 0,006419s per 16Kb of data for the Sense HAT B data sets. As can be seen these values are quite close in value and show that the results of both test sets are consistent with each other. A linear increase in data set size corresponds to a linear increase in the encryption algorithm's run time. This relationship governs that increasing the data set size will increase

the run time in accordance with the equation y = 0.0066x+0.0491 (where y is run time and x is data set size/200).

Additionally, the user requirements, functional requirements and specifications of the design are defined with a list of acceptance test procedures highlighted, in which the designs are tested against the design specifications. The ATPs were amended during the process but the basic specifications defined in the start were all met. These were mostly around using as little battery power as possible, providing secure encryption and significant compression that greatly reduced the file size for transfer.

Finally, the compression and encryptions blocks are combined and tested for correct execution prior to the full implementation of the system on the Raspberry Pi Zero. This allowed for the testing of how the compression and encryption will interact and whether this will still produce lossless output. Within the experimentation it was shown that combining them into one did lead to successful compression and encryption. Once these files were decrypted and decompressed it was shown that when compared to the original file it produced a lossless output as needed and expected from the previous results.

H. References:

[1] E. Williams, “Cryptography 101: Symmetric Encryption,” 30 March 2020. [Online]. Available: https://medium.com/@emilywilliams_43022/cryptography-101-symmetric-encryption-444aac6bb7a3. [Accessed 4 September 2021].

[2] J. Lake, “What is 3DES encryption and how does DES work?,” Comparitech, 20 February 2019. [Online]. Available: https://www.comparitech.com/blog/information-security/3des-encryption/. [Accessed 4 September 2021].

[3] M. Cobb, “Advanced Encryption Standard (AES),” Search Security, April 2020. [Online]. Available: https://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard. [Accessed 4 September 2021].

[4] Doxygen, “AES Encryption Library for Arduino and Raspberry Pi,” 10 February 2015. [Online]. Available: http://spaniakos.github.io/AES/. [Accessed 4 September 2021].

[5] BBC, “Encoding images,” BBC, [Online]. Available: https://www.bbc.co.uk/bitesize/guides/zqyrq6f/revision/4#:~:text=Compression%20can%20be%20lossy%20or,Lossy%20compression%20permanently%20removes%20data. [Accessed 4 September 2021].

[6] L. F. Buenavida, “Crunch Time: 10 Best Compression Algorithms,” DZone, 28 May 2020. [Online]. Available: https://dzone.com/articles/crunch-time-10-best-compression-algorithms. [Accessed 4 September 2021].

[7] E. Chen, “Understanding zlib,” January 2019. [Online]. Available: https://www.euccas.me/zlib/. [Accessed 4 September 2021].

[8] TDK InvenSense, “ICM-20649 Datasheet,” TDK InvenSense, San Jose, 2021.

[9] Aceinna, “Aceinna,” Aceinna, [Online]. Available: https://developers.aceinna.com. [Accessed 5 September 2021].

[10] M. C. F. R. R. R. J.́. C. J.̃. P. F. J.̃. S. Rui Pereira, “Ranking Programming Languages by Energy Efficiency,” Preprint, Covilhã, 2021.

[11] J. Jacobson, “SHARC BUOY: Remote Monitoring of Ice Floes in Southern,” University of Cape Town, Cape Town, 2020.

[12] EEE3097S, *Design Requirements: EEE3097S,* Cape Town: University of Cape Town, 2021.

[13] MathWorks, “Fourier Transforms,” MathWorks, [Online]. Available: https://www.mathworks.com/help/matlab/math/fourier-transforms.html. [Accessed 4 October 2021].

[14] Tutorials Point, “Cryptography with Python - Quick Guide,” Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/cryptography_with_python/cryptography_with_python_quick_guide.htm. [Accessed 2 October 2021].

[15] K. Juno, “Encrypt File with AES in Python,” 12 December 2018. [Online]. Available: https://kentjuno.com/learning/python/encrypt-file-with-aes-in-python/?__cf_chl_managed_tk__=pmd_4.cGcq2gD9UtQYVErpRiH1izX4Kt0M8jeatU03SUhTY-1633191688-0-gqNtZGzNAuWjcnBszRM9. [Accessed 2 October 2021].

[16] M. Heinz, “All The Ways to Compress and Archive Files in Python,” towards data science, 13 September 2021. [Online]. Available: https://towardsdatascience.com/all-the-ways-to-compress-and-archive-files-in-python-e8076ccedb4b. [Accessed 4 October 2021].

[17] S. Robinson, “Python zlib Library Tutorial,” Stack Abuse, 2017. [Online]. Available: https://stackabuse.com/python-zlib-library-tutorial/. [Accessed 4 October 2021].

[18] zlib, “Usage Example,” zlib, 12 December 2012. [Online]. Available: https://www.zlib.net/zlib_how.html. [Accessed 4 October 2021].

[19] E. Chen, “Understanding zlib,” Hugo, 5 January 2019. [Online]. Available: https://www.euccas.me/zlib/. [Accessed 4 October 2021].

[20] L. Collin, “A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA,” 31 May 2005. [Online]. Available: https://tukaani.org/lzma/benchmarks.html. [Accessed 4 October 2021].

[21] InvenSense, “ICM-20649,” InvenSense, 13 December 2016. [Online]. Available: https://product.tdk.com/system/files/dam/doc/product/sensor/mortion-inertial/imu/data_sheet/ds-000192-icm-20649-v1.0.pdf. [Accessed 24 October 2021].

[22] InvenSense, “ICM-20948,” InvenSense, 6 August 2017. [Online]. Available: https://invensense.tdk.com/wp-content/uploads/2016/06/DS-000189-ICM-20948-v1.3.pdf. [Accessed 24 October 2021].

[23] Waveshare, “Sense Hat (B),” Waveshare, 21 July 2021. [Online]. Available: https://www.waveshare.com/wiki/Sense_HAT_(B). [Accessed 24 October 2021].

[24] Apollo-Soyuz, “Movement,” GitHub, 9 June 2017. [Online]. Available: https://github.com/raspberrypilearning/astro-pi-guide/blob/master/sensors/movement.md. [Accessed 24 October 2021].

[25] Waveshare, “ Sense HAT (B) for Raspberry Pi, Multi Powerful Sensors,” Waveshare, 2021. [Online]. Available: https://www.waveshare.com/sense-hat-b.htm. [Accessed 24 October 2021].

[26] Digi-Key, “ICM-20649,” Digi-key, 2021. [Online]. Available: https://www.digikey.co.za/en/products/detail/tdk-invensense/ICM-20649/8540792. [Accessed 24 October 2021].

[27] Barley_Li, “Analog vs Digital sensors - Which One is Better?,” Digi-Key, 9 June 2019. [Online]. Available: https://forum.digikey.com/t/analog-vs-digital-sensors-which-one-is-better/3651. [Accessed 24 October 2021].

[28] RaspberryPi, “Sense HAT data logger,” RaspberryPi, [Online]. Available: https://projects.raspberrypi.org/en/projects/sense-hat-data-logger/2. [Accessed 24 October 2021].

[29] w3schools, “Python Datetime,” w3schools, 2021. [Online]. Available: https://www.w3schools.com/python/python_datetime.asp. [Accessed 24 October 2021].

[30] J. N. Jacobson, “SHARC Buoy,” University of Cape Town, Cape Town, 2021.

[31] J. Wyngaard, *Modular Design & UML,* Cape Town: University of Cape Town, 2021.

[32] P. Porpoise, ““'utf-8' codec can't decode byte 0xb5 in position 11: invalid start byte” Code Answer’s,” Grepper, 23 September 2020. [Online]. Available: https://www.codegrepper.com/code-examples/python/%27utf-8%27+codec+can%27t+decode+byte+0xb5+in+position+11%3A+invalid+start+byte. [Accessed 2 October 2021].

[33] Programmer All, “Python implements DES encryption algorithm and 3DES encryption algorithm,” Programmer All, [Online]. Available: https://programmerall.com/article/53631125586/. [Accessed 2 October 2021].

[34] patildhanu4111999, “How to Convert Bytes to String in Python ?,” Geeks for Geeks, 11 December 2020. [Online]. Available: https://www.geeksforgeeks.org/how-to-convert-bytes-to-string-in-python/. [Accessed 2 October 2021].

[35] Python Software Foundation, “time — Time access and conversions,” Python Software Foundation, [Online]. Available: https://docs.python.org/3/library/time.html#time.time. [Accessed 2 October 2021].

[36] J. Cage, “How to get current CPU and RAM usage in Python?,” Stack Overflow, 18 March 2010. [Online]. Available: https://stackoverflow.com/questions/276052/how-to-get-current-cpu-and-ram-usage-in-python. [Accessed 2 October 2021].

[37] PhJulien, “CPU, RAM and disk monitoring using python,” 5 November 2012. [Online]. Available: https://www.raspberrypi.org/forums/viewtopic.php?t=22180. [Accessed 2 October 2021].

[38] P. Goyal, “How to get current CPU and RAM usage in Python?,” Geeks for Geeks, 24 January 2021. [Online]. Available: https://www.geeksforgeeks.org/how-to-get-current-cpu-and-ram-usage-in-python/. [Accessed 2 October 2021].

[39] manjeet_04, “Python | Convert String to bytes,” Geeks for Geeks, 22 May 2019. [Online]. Available: https://www.geeksforgeeks.org/python-convert-string-to-bytes/. [Accessed 2 October 2021].

[40] S. Heydari, “Executing Shell Commands with Python,” Stack Abuse, 19 September 2021. [Online]. Available: https://stackabuse.com/executing-shell-commands-with-python/. [Accessed 3 October 2021].

[41] Programiz, “Python File I/O,” Parewa Labs Pvt. Ltd., [Online]. Available: https://www.programiz.com/python-programming/file-operation. [Accessed 4 October 2021].

[42] D. Cross, “Winning the Data Compression Game,” towards data science, 29 April 2020. [Online]. Available: https://towardsdatascience.com/winning-the-data-compression-game-af145363ae49. [Accessed 4 October 2021].

[43] GeeksforGeeks, “How to compare two text files in python?,” 4 January 2021. [Online]. Available: https://www.geeksforgeeks.org/how-to-compare-two-text-files-in-python/. [Accessed 4 October 2021].

[44] GeeksforGeeks, “How to get current CPU and RAM usage in Python?,” 24 January 2021. [Online]. Available: https://www.geeksforgeeks.org/how-to-get-current-cpu-and-ram-usage-in-python/. [Accessed 4 October 2021].

[45] codecademy, “Strings,” codecademy, 2021. [Online]. Available: https://www.codecademy.com/learn/learn-python-3/modules/learn-python3-strings/cheatsheet. [Accessed 31 October 2021].

[46] tutorialspoint, “tutorialspoint,” tutorialspoint, 2021. [Online]. Available: https://www.tutorialspoint.com/python/python_command_line_arguments.htm. [Accessed 31 October 2021].