School of Engineering

Design, Construction, and Testing of Arduino-Based Lander

Student/Group Name(s):	**Student(s) ID Number:**
Eoghan Collins	23381001
Nelly Frohburg	23360016
Caoilinn O'Brien	23384576
Tadhg Scanlon	23412306
Kento O'Connor	23481984

Class and Year (e.g. 1st Undenominated Engineering):	1st Undenominated Engineering
Subject Code and Name: (e.g. EI140 Fundamentals of Engineering)	EI150
Lecturer Name:	Ethan Delaney, Roshan
Title of Report/Assignment:	Design, Construction, and Testing of Arduino-Based Lander
Submission Deadline:	29/03/2024
Submission Date:	09/04/2024

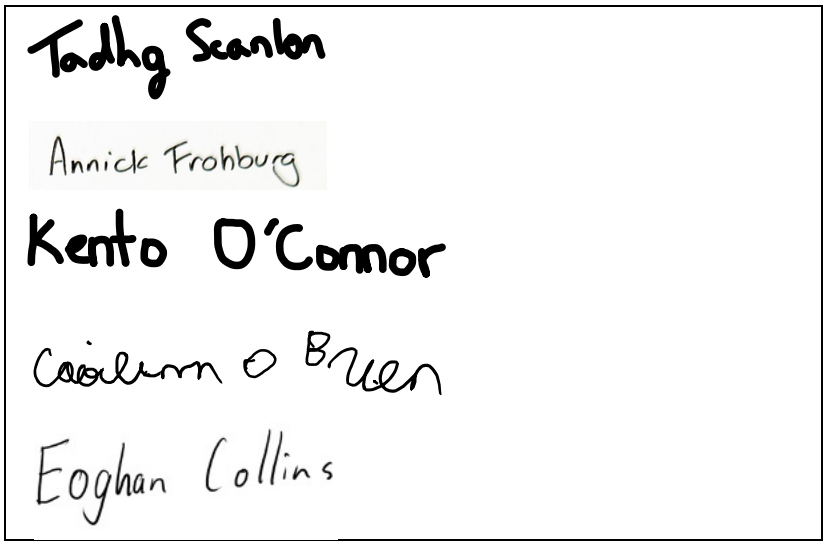
Academic Integrity and Plagiarism

Plagiarism is the act of copying, including or directly quoting from, the work of another without adequate acknowledgement. All work submitted by students for assessment purposes is accepted on the understanding that it is their own work and written in their own words except where explicitly referenced using the correct format. **For example, you must NOT copy information, ideas, portions of text, data, figures, designs, CAD drawings, computer programs, etc. from anywhere without giving a reference to the source.** Sources can include websites, other students' work, books, journal articles, reports, etc. Self-plagiarism or auto-plagiarism is where a student re-uses work previously submitted to another course within the University of in another Institution.

You must ensure that you have read the University Regulations relating to plagiarism, which can be found on the NUI Galway website: http://www.nuigalway.ie/plagiarism/

I have read and understood the University Code of Practice on plagiarism and confirm that the content of this document is my own work and has not been plagiarised.

Students' Signatures



1 Introduction

The final design project of the module EI150 was the Lander project. Students were divided into teams of five. Like the other design projects, the allocated time was four weeks to complete the project. The aim of each team was to design a lander inspired by the Europeans Space Agency's CanSat project.

The design brief outlined that a bottle rocket and capsule were to be built. An Arduino kit had to be situated inside the capsule, protected by padding. The capsule then had to be attached to the bottle rocket which would be launched using a launch mechanism designed by the group. In addition, a parachute had to be designed in order to bring the capsule to a safe landing to protect the Arduino kit from being damaged.

The main aim of the project was to code the Arduino to record different data sets derived from the connected sensors during the launch of the rocket and the subsequent landing of the Arduino. The Arduino kit registers atmospheric pressure, acceleration and temperature using a Barometer, Accelerometer and Thermistor respectively.

The final rocket, capsule and parachute design had to adhere to certain set parameters and constraints. The maximum dimensions for the capsule were 75mm in diameter and 200mm in length. The overall weight of the capsule, parachute and Arduino system could not exceed 250g. In addition, any metal containers were not allowed as they would interfere with the Arduino system. Instructions surrounding the launching mechanism were minimal as teams were left think of their own ideas as to how they want to launch the rocket. However helpful recommendations were offered based on previous projects.

Each group had to supply their own batteries, tools and materials throughout to construct the lander which required careful consideration and organisation. Groups were also expected to meet up outside of the scheduled lab slots in order to complete the project to a high standard within the time constraints.

This report outlines the approach that was took by team 56 to construct the lander while carefully considering the theory behind it and experimenting with different tests to back up results. The report gives an insight into the process of constructing the Lander while also detailing the various challenges faced and how they were overcome.

2 Rocket

The aim of the project was to construct a water rocket which is capable of carrying a load matching the Arduino based telemetry system.

2.1 Underlying Theory of Operation

The principle of the water rocket is based Newton's third law of motion which states that every action has an equal and opposite reaction. A typical water rocket consists of a plastic bottle partially filled with water. The bottle is sealed tightly with a cork with a hole for air to enter it. The bottle is the pressurised with a pump. As air enters the bottle, the pressure inside increases and pushes down on the water inside. When the pressure inside the bottle becomes too high, it overcomes the resistance of the cork and pushes the water out. This creates a reaction force in the opposite direction which causes the rocket to propel upwards. Once the rocket reaches its apex, the weight of the rocket causes it to descend.

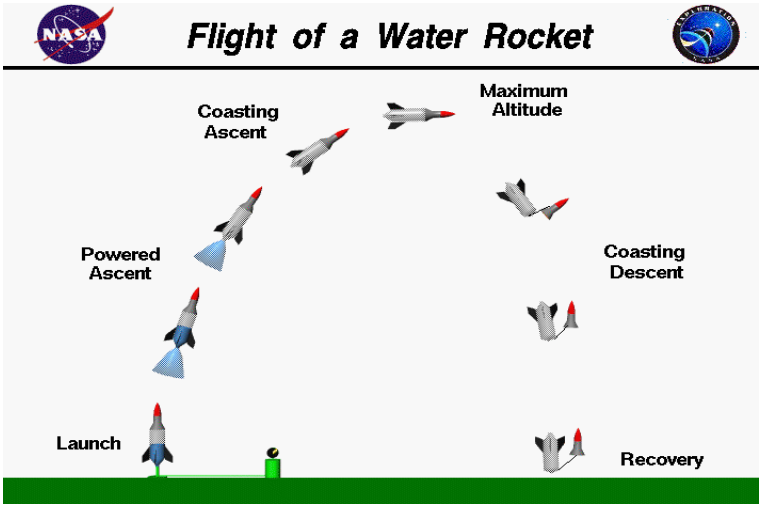


Figure 2.A.1 (Flight of a water rocket)

2.1.1 Rocket design requirements

- Rocket must be free standing. (Fins are a useful way to achieve this)
- Rocket must safely hold the capsule and parachute
- Pressure-based release system (No manual release e.g. release pins)
- (Water-based pneumatic system is recommended)

2.1.2 Rocket Dimensions and Materials

The rocket was built using a 2L PET bottle, a wine cork, duct tape, three fins made from cardboard and a bicycle pump. It was important that a polyethylene terephthalate (PET) bottle was used because of its light weight and durability. Firstly, the cork was too narrow for the bottle, so tape was wrapped around the cork

to ensure a tight fit in the bottle. Then fins were cut out from cardboard and taped on to the bottle. This created an aerodynamic design to allow the rocket to fly in a straight path. The final dimensions of the rocket came out to be 375mm x 290mm (height x width) including fins.

2.1.3 Design of Rocket

Drag is an important factor to account for in the design of the rocket as when a water rocket is launched due to the water leaving the rocket's nozzle and pushing the rocket up into the air the acceleration will decrease because in order for the rocket to continue it will need to force air out of its path ,the force that allows it to happen is called aerodynamic drag. After researching various websites, the team found a water rocket booklet which the team used as a guideline to construct a rocket by the national physics laboratory (NPL)[1] with low drag. According to the booklet its essential to have these feature to provide a low drag.

- The body of the rocket was made from a 2L plastic bottle which proved to be the best one as a long thin rocket tend to have lower drag than short, fat rockets, another crucial thing to have a low drag is to ensure the rocket is smooth.
- It contained four fins which were arranged symmetrically around the back of the rocket and were evenly spaced 90° apart, all the fins were identical in height, width and length (an image of the dimensions are below).Cardboard was recommended as the material to be used to make the fins as its efficient because its thin and light which makes it go through the air faster.

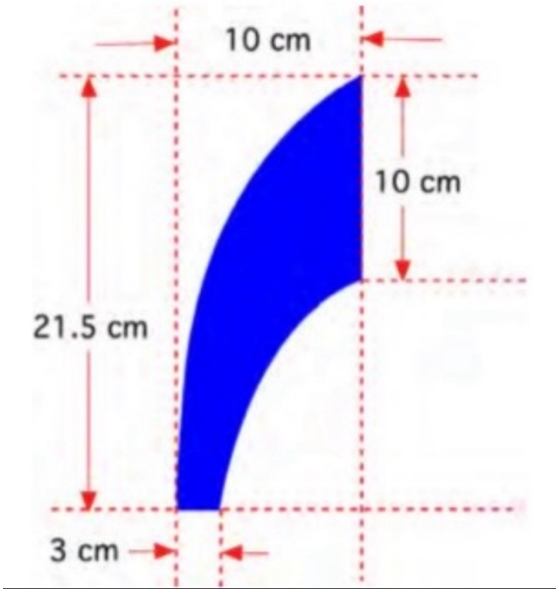


Figure 1: The dimensions of the fins [1]

Another method the team used to find the size of the rocket was using this equation [9]:

$Fd = (½)\ \rho v^2 A Cd$ [9]

Fd is the drag force, ρ is the air density which was taken as 1.22kg/m^3 V is the velocity relative to the object which was taken to be 10m/s, A is the reference area and Cd is the drag coefficient which was taken to be 1.5.This formula helped the team with the decision of the size of the rocket.

The size of the bottle was not only important for the drag it was also essential to find out the pressure that's required for the rocket to launch. The team wanted a rocket that contains a low drag but also doesn't require an insane amount of pressure in order for it to launch the bottle when its 40-50% full. The team found information on a website called the water rocket science guideline for students [2] that a 2L bottle has the facilities to hold 100psi without rupturing some 2l bottles rupture at 130psi or 170psi.This helped the team to settle with a 2L bottle as the body of the rocket.

2.2 Construction of the Rocket

2.1.1 The body

- The rocket was originally made from 750ml plastic water bottle but after a few tests and a lot of research the team decided to change it to a 2L old water bottle as it was more beneficial and made the possibility of reaching the team's goal more likely. Just like what Newtons third law states for every action, there is an equal and opposite reaction, water rockets fall under Newtons third law. 2L bottles tend to work better as larger bottles result in greater launch's as it contains more water which then allows the rocket to expel more water which then in return generates thrust that allows the rocket to launch upwards.
- As previously discussed, the size of the bottle was picked because it had a low drag and for pressure it requires to launch. This played a major role in the final decision of the size of the body.

2.1.2 The fins

- As discussed earlier the fins of the body were made from a cardboard shoe box and the dimensions are shown in figure 1. The NPL guidebook [1] recommended to use the cardboard from a for sale sign, unfortunately no one was able to get their hands on one, instead the fins got their thickness by gluing 2 fins together using a hot glue gun. After the fins cooled down, they were glued to the 2L symmetrically around the rear of the rocket 90° apart.

- Fins are essential when it comes to constructing a rocket as they allow a rocket to stabilize while it's in the air as the rocket wouldn't have any control when it's in the air due aerodynamic forces such as wind and it also creates the centre of pressure at the end of the rocket which helps it manage to stay nose-up while soaring through the air and to ensure a straight trajectory.

- The fins were also designed so that the rocket could free stand without any assistance. The fins had to be designed a second time as the first design did not leave enough room for the needle to enter the bottle. The fins were made longer the second time to account for the extra space required for the pump.

Fig 2.1.2 (Construction of fins)

Fig 2.1.2.1 (Prototype of rocket with fins attached)

2.1.3 The Capsule holder

- The capsule was placed on top of the rocket, A 2L bottles top and bottom was cut off and was taped to rocket. This was done to ensure that the capsule would stay in place before being launched.

Figure 3: The rocket [1]

2.2 The Launch mechanism

It was recommended by a lot of resources to use a pneumatic pump, so the pump used was a bicycle stirrup pump as its better than a hand pump as the work is evenly shared between the two arms. The pump is able to pressure up to 160psi.

A cork was used to seal the bottle, It was chosen because when air pressure is being pumped into the bottle which pushes the water ,the cork still manages to stay in place and holds the water in the bottle due to friction until it reaches a certain pressure then the cork is forcefully ejected out of the bottle then pressure acts on the water causing it to propel forward.

The top of a 1.5L bottle was cut off and is used to elevate the rocket of the ground.

Figure 4: The rocket and its launch mechanism.

2.3 Testing of the rocket

Several tests were carried out in order to ensure that the rocket would work on the day. Here is a recap of some of tests that were carried out and what was learned from it.

2.3.1 mock test 1

For this test the rocket was launched using a bicycle stirrup pump and was launched on grass. The launch wasn't very successful as it barely reached 1m off the ground and a lot of water came out of the opening before take-off. The team decided that there should be something to elevate the rocket and to place it on a smoother surface like a footpath instead of uneven patch of grass to give it a better chance of launching and to wrap tape around the cork to cause more pressure which should prevent water from leaking out of the bottle and maybe cause the rocket to reach a higher height.

2.3.2 mock test 2 (testing the adjustments)

For this test a top of a water bottle was used to elevate the rocket off the ground. The test was carried out on a basketball court due to it having an even surface. The same pump and rocket were used but the only difference was the cork was thickened by tape. This test was very successful it reached 5m.

2.3.3 mock test 3 (testing the parachute and Arduino)

The set up for this test was identical to the second test but this time the parachute was tested. The parachute worked immediately after being launched and brought the capsule safely to the ground, so the test was repeated with the Arduino, and it was successful everything went according to plan and values were also recorded.

2.3.4 Final test

The day of test unfortunately the team’s rocket didn't perform as spectacular compared to the mock tests due to the weather circumstances, but the Arduino was still able to produce values.

3 Capsule and Parachute System

3.1 Capsule

The design for the capsule was a simple one. It involved cutting a 750ml bottle in half, using the bottom for the capsule. The capsule was designed to fit within the given parameters of 75mm in diameter and 200mm in length. The Arduino kit was placed inside the capsule with tissue paper acting as padding. The tissue paper was stuffed in a way that kept the Arduino orientated in a vertical position as this was the direction that the Arduino was calibrated in the code to record the data sets. Strong industrial tape was used to seal the capsule. The pieces of string from the parachute were then attached to the capsule at different intervals along the circumference of the bottle using more tape. Plastic in the shape of a cylinder was cut from a 2L bottle and taped to the top of the body of the rocket. The capsule then was then placed inside this cylinder of plastic. The purpose of this plastic was to ensure that the capsule would not fall off the rocket mid launch but instead, when the rocket would descend to the ground again, the parachute would deploy, and the capsule would slip out of the cylinder of plastic.

3.2 Design of Parachute

The parachute was designed to slow the capsule down after being launched. This allows the capsule to return safely to the ground with minimal damage to the Arduino circuit. The parachute should deploy when the rocket and capsule start their descent back down to the ground. The parachute should deploy when air becomes trapped under the parachute. This causes resistance and slows the parachute and capsule [3]. This is to stop the parachute slowing down the rocket on the ascent and to maximise the height gained. The parachute was designed to be a large flat hexagon.

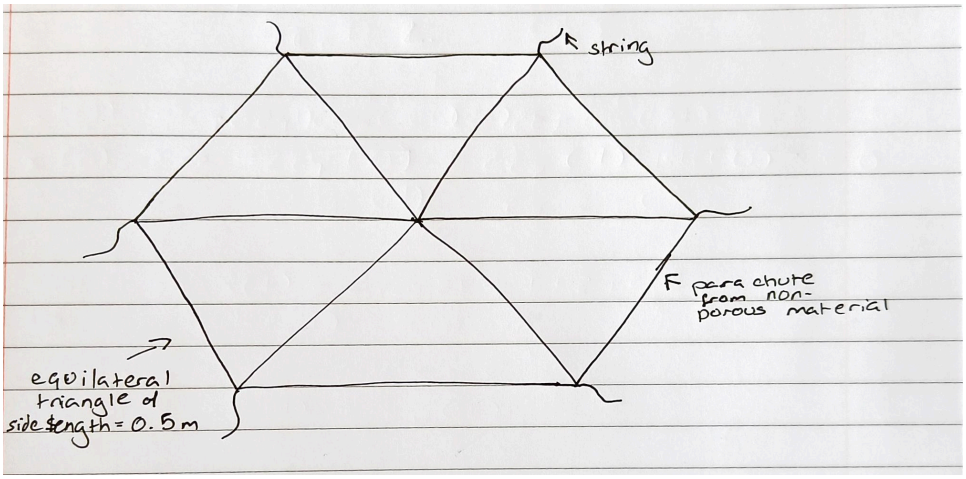


Figure 1: diagram of the parachute design.

The parachute was designed to slow a capsule that weighs 0.25kg. Using equation [3] the size of the parachute of the hexagon was calculated.

$$A = \frac{m \cdot g \cdot 2}{\rho \cdot v^2 \cdot Cd} \qquad [3]$$

A refers to the area of the parachute required. M refers to the mass of the object. G refers to the acceleration due to gravity constant. It was taken to be 9.81m/s^2. ρ is the density of air. It was taken to be 1.22kg/m^3. V refers to the velocity that the object should hit the ground at. It was taken to be 3m/s. Cd refers to the drag coefficient of the shape of the parachute. As the parachute was designed to be a flat sheet it was taken to be 0.75. This gave that the hexagon should have and area of 0.5956m^2. Using equation [4] the side length of the hexagon was calculated.

$$A = \frac{3 \cdot \sqrt{3} \cdot s^2}{2} \qquad [4]$$

A refers to the area of the hexagon. S refers to the side length of the hexagon. Using this calculation the side length of the hexagon needed was found to be 0.479m. This was increased to 0.5m as a factor of safety. This would allow some flaws in the construction while still being able to slow the capsules descent.

3.3 Construction of Parachute

Originally the parachute was designed to be made from plastic from binbags. The binbags were too small to fit in the dimensions that were needed. Therefore, the parachute needed to be made from several binbags attached together. The problem arose when attaching them together. A plan was to connect the panels with either sewing thread or strong tape. The thread caused larger holes in the material than expected. This made the parachute less effective as it let air escape through the holes. Both electrical and duct tape were tested for the designs. Both tapes were hard to work with and difficult to get the panels stuck together properly. The tape also made the parachute too heavy on the seams. This prevented the parachute from opening properly and it caused the parachute to be less effective.

To solve this an umbrella was used. After removing the fabric of the umbrella from the metal contraption inside, it was measured. The dimension of the fabric was a side length of 0.5m so no alteration to the size were necessary.

Figure 2: Image showing measurements to the parachute material

Originally string was sewn onto each vertex. However, during testing this often came undone. Therefore, a new solution was found. Small holes were created at each vertex. This allowed string to be tied securely to the fabric. The length of the string need was found through testing described in section 3.3.

A hook was required for the testing of the parachute. This was created using the closure of the parachute. This strip of material was sewn in a loop to the top of the parachute.

Figure 3: image showing the loop needed for testing.

3.4 Testing for length of parachute strings

3.4.1 Objectives

The objective of this test is to find at what length of string the parachute works best. Working best is defined in this scenario as what slows the capsule down the most. This will affect how the parachute design is finalised for the project.

3.4.2 Method

- Drop the capsule without the parachute three times from a height of 3.5m. This is the control. Record the time for each drop and find the average.
- Attach the parachute with string a length of 1.2m. Drop this from the constant height three times. Record the time for each drop and find the average.
- Shorten the string to 0.7m. Drop this from the constant height three times. Record the time for each drop and find the average.
- Shorten the string to 0.5m. Drop this from the constant height three times. Record the time for each drop and find the average.

Figure 4: image showing drop test for the parachute string lengths

3.4.3 Results

	Test 1	**Test 2**	**Test 3**	**Average**

Control	1.19s	1.08s	1.12s	1.13s
1.2m	1.79s	1.80s	1.72s	1.77s
0.7m	1.83s	1.86s	1.85s	1.84
0.5m	1.98s	1.96s	1.89s	1.94

3.4.4

Table 1: data collected from drop test

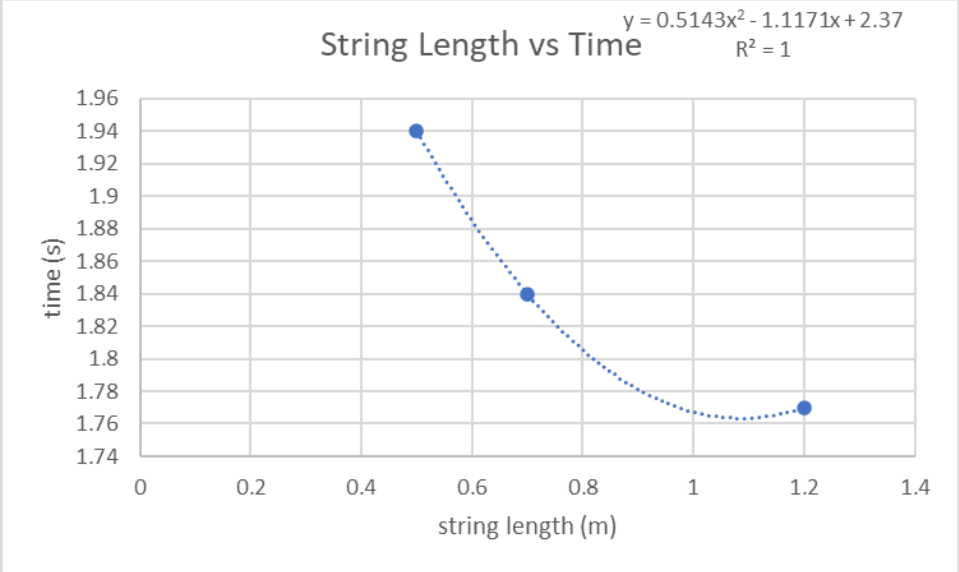


Figure 5: graph showing the influence of string length and time taken to fall.

3.4.5 Discussion

The average time for the parachute to reach the bottom with 0.5m strings was 1.94s. This was the slowest time for all tests. This parachute type slowed the capsule down 71.68% compared to the control.

Inaccuracies may have occurred due to the weather. The tests were completed on the same day. However, after each block of test the strings had to be altered the wind may have changed and skewed the results. To prevent this, tests were done in a timely manner. They were also carried out in a shelter area. Other inaccuracies may have occurred in the timing due to human error. This was prevented as the test was ran multiple times and averages were taken.

3.4.6 Conclusion

To conclude, the parachute worked best with strings of 0.5m. This could be because the longer strings tended to wrap around each other. This prevented the parachute from opening properly. Therefore, the final design for the parachute will include string with the length of 0.5m.

4 Arduino System and Software

The interaction of Arduino Hardware, Arduino Software, Laptop Hardware, and Laptop Software is complex but critical to the project. A block diagram, explaining the overall interaction can be seen in Figure 1 (**Note that Figure numbers start from 1 in this section, due to formatting errors; thus, if Figure 1, is referenced, for example, it is referencing the "Figure 1" specified in this section, unless specified to the contrary**). More detail will be provided about the various aspects in the following subsections.

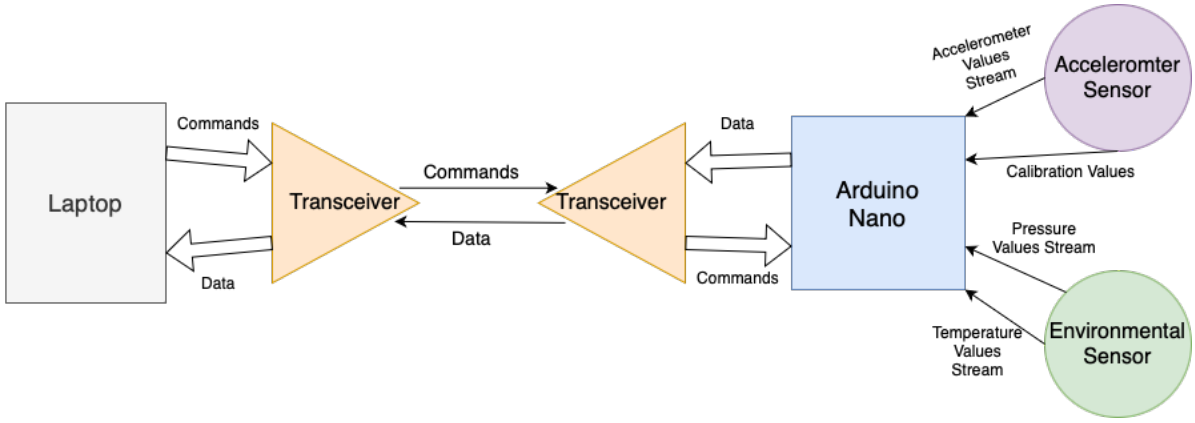


Figure 1: Arduino-Laptop System Block Diagram

4.1 Hardware

The computational hardware for this project was provided by the university, however, in order to improve their ability to use and interact with the hardware, the team made every effort to learn about the hardware. This resulted in a depth of knowledge about the hardware, as detailed throughout this section, which allowed the software to be as developed as it was in the end [see Section 4.2].

4.1.1 Arduino Nano

The Arduino Nano is the cornerstone of this project, allowing precise and in-depth interaction with sensors and transceivers with a micro-computer small and light enough to be fitted to the lightweight, low-thrust rocket detailed in Section 2, permitting data gathering during the motion of the rocket and lander.

4.1.2 Ports

The Arduino Ports are labelled in Figure 3, showing what the team had available to work with. However, as shown in Figure 2, only 2 of the digital pins, 5 of the analogue pins and none of the serial pins were actually usable in the main application of the Arduino, due to the restrictive, if convenient nature of the provided PCB. (It should be noted that the digital ports D2-D12 were accesible via ports in the PCB, but as no more components were deemed necessary, they were left unused.)

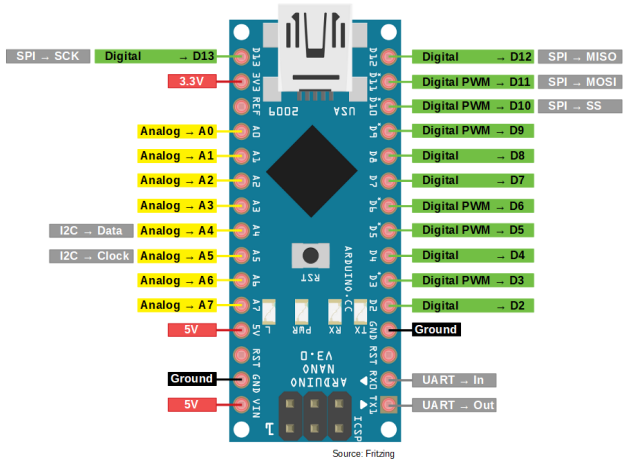


Figure 3: Arduino Nano Diagram showing all analogue and digital ports, as well as some other connections

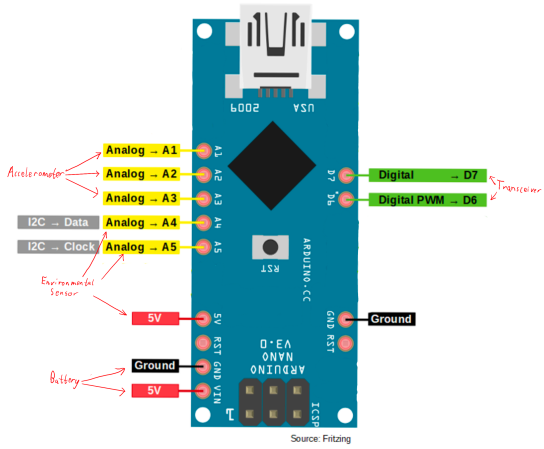


Figure 2: Arduino Nano Diagram Showing only the usable analogue and digital ports in this setup

4.1.2.1 Analog Pins

The analogue pins permitted the integration of the ADXL335 Accelerometer Sensor and BMP280 Environmental sensor into the system. The analogue ports take the voltages supplied by the sensors, using an analogue-to-digital converter to convert the voltages (between 0V and 5V) to a number between 0 and 1023 (a 10-bit number [13]). These output values can then be further processed by the Arduino Nano or transmitted to a connected device for further processing.

4.1.2.2 Digital Pins

In this setup, the Arduino's digital pins are solely use to connect to the HC-12 Wireless Transceiver, permitting data transmission and receipt using the Software Serial module.

4.1.2.3 Serial Pins

Digital pins are used for serial transmission in this use-case [see Section 4.2.3.2]. However, the Arduino nano comes with two digital pins specialised for serial transmission. These are more appropriate to be used for the transmissions carried out in this project. Unfortunately, these pins are configured by default to be used for USB communication with a programming laptop, and as the PCB's permanent nature made changes of setup challenging, the designers opted to leave the serial pins exclusively for USB communication.

This is unfortunate, as Hardware Serial is less computationally intensive and permits higher baud-rates, with greater reliability, allowing faster data transfer, which is extremely important for this project, however, there are ways to achieve faster data transfer without this hardware advantage, as highlighted in Section 4.2.7.

4.1.3 HC-12 Wireless Transceiver

The HC-12 Wireless Transceiver is a crucial component for this project, as it allows for real-time communication between the Arduino and the laptop connected to it.

4.1.3.1 Physics

The physics behind the operation of the HC-12 Wireless Transceiver are similar to those behind the operation of similar devices. The HC-12 is capable of communication in the 433.4-473.0 MHz range, which corresponds to a wavelength of 0.63-0.69m [4] which implies that its communication occurs using electromagnetic microwaves, which is very common for wireless communication technologies, such as in mobile phones.

4.1.3.1.1 Transmission

The transmission of messages by this wireless transceiver is based on the generation and frequency modulation of electromagnetic waves. In short, electromagnetic waves are generated by generating an alternating current and passing it through an antenna, creating an alternating electromagnetic field, as predicted by Maxwell's equations.

While the team was unable to locate specific documentation as to the exact type of data keying employed by the HC-12, Frequency Shift Keying (FSK) is common [5] and is supported by the Si4463 chip that it employs [6]

FSK, in its most basic form, relies on the modulation of electromagnetic frequencies, which is interpreted as data. While there are multiple kinds of FSK, such as QFSK and M-ary FSK which use a number of frequencies to encode multiple bits per frequency, for the sake of brevity, Binary FSK (BFSK) will be explained, as other kinds of FSK are based off this [5]. In BFSK, one frequency is assigned as 0 and another as 1. Every fraction of a second (this is known as the Bit Time [see Equation 5]), the frequency is checked, and a bit is encoded. This means that to transmit a float, for example, which is 4 bytes [7], or 32 bits it would take 32 times the Bit Time.

$$Bit\ Time = \frac{1}{Baud\ Rate} \tag{5}$$

This data is received on the other end, through mechanisms described in Section 4.1.2.1.2, and analysed through protocols like the one explained here.

4.1.3.1.2 Reception

Reception relies on similar processes to transmission, but in reverse. When receiving, electromagnetic waves from the transmission source induce a small alternating current in the antenna, which is then amplified and processed to decode the message encoded using protocols explained in the previous section.

4.1.3.2 Baud Rates

The baud rate of a transmission determines the amount of information that can be sent in a period of time, as shown by equation 5, defining it as the number of bits that can be sent per second. For this project, a higher baud rate is desirable, as, especially during thrust, values can change extremely quickly. The highest baud rate that the HC-12 module can support is 115,200 baud. This does come with some range drop-off (reducing the maximum range to 100m [8]), however this wasn't thought to be a problem, as it was considered unlikely for the rocket to launch that far.

Unfortunately, a more relevant limiting factor was the user of Software Serial for data transmission, as Software Serial begins to become unreliable above baud rates of 9600. Due to this, it was decided to transmit date at a rate of 9600 baud.

4.1.4 CP2102 USB - RS232 / TTL / UART Converter

The HC-12 Wireless Transceiver cannot interface directly with most modern computers, due to its lack of a USB connector [9]. For this reason, the CP2102 USB - RS232 / TTL / UART Converter [see Fig. 4] was employed, to convert the HC-12's connections to a USB connection.

Thus, though not directly impacting any data, or providing much processing, the converter is an essential part of the communication setup for the Arduino.

Figure 4: CP2102 USB - RS232 / TTL / UART Converter

4.1.5 ADXL335 Accelerometer Sensor

The ADXL335 Accelerometer Sensor (henceforth referred to as "the accelerometer") [see Fig. 5] is arguably the most important part of the Arduino-based setup, excepting, perhaps, the Arduino Nano itself.

The accelerometer is critical for detecting the acceleration in 3 axes, providing the majority of the data utilised for understanding the nature of the lander's motion. In this section, the physics behind the accelerometer's operation is explored, along with its application.

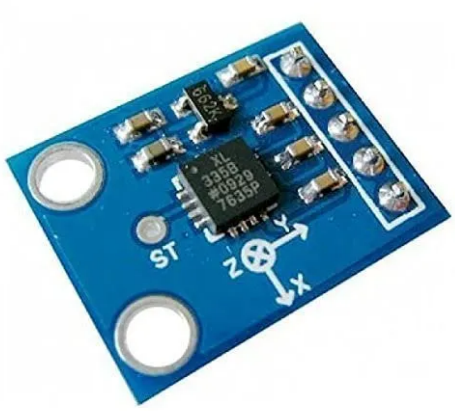


Figure 5: ADXL335 Accelerometer Sensor

4.1.5.1 Physics

Sensors like the accelerometer are considered Microelectromechanical Systems (MEMS) [10], a linking of electronic and mechanical engineering in much the same way that biochemistry is a linking between its two main constituent sciences. MEMS employ electronic and moving parts [10]. This facilitates the measurement of physical phenomena using electronic components.

4.1.5.1.1 Capacitative Sensing

Capacitative sensing is the mechanism by which accelerometers like the ADXL335 functions [20]. The basic idea relies on the Basic Capacitance Formula [see equation 6].

$$C = \frac{\varepsilon A}{d} \tag{6}$$

As can be seen from this formula, capacitance is inversely proportional to distance. Thus, if a force compresses the medium between the capacitor plates (due to acceleration, see equation 7), the capacitance will increase.

$$F = ma \tag{7}$$

Combining this with the current-capacitance-voltage relationship [see equation 8] allows one to correlate an increase in current to an increase in force, by proxy of an increase in capacitance.

$$I = C\frac{dV}{dt} \tag{8}$$

The accelerometer utilises 3 such capacitive sensors [11], permitting acceleration readings on 3 axes, giving the necessary output.

4.1.5.1.2 Signal Processing

Unfortunately, the raw change in current is extremely low, especially within the accelerometer's operating acceleration range of ±3g [see Table 1][11] Because of this, the change in current must be amplified. After this process, the resulting output must be demodulated to recover the data, before the output is amplified again and outputted for the Arduino to read [see Fig. 6].

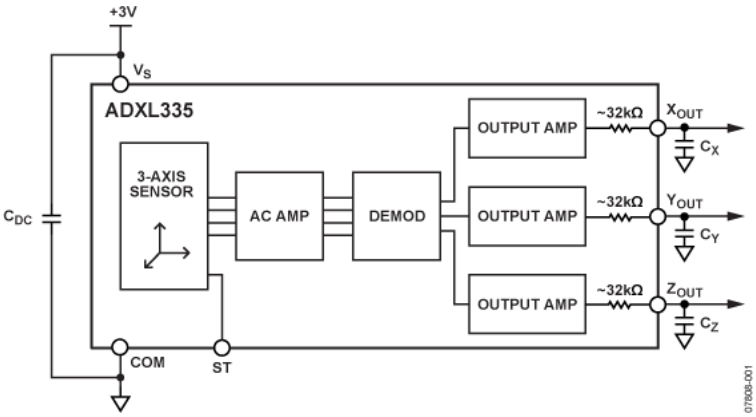


Figure 6: ADXL335 Operational Diagram

4.1.5.2 Specifications

While not critical to the understanding of the accelerometer, I have compiled a table of important accelerometer specifications [11], in order to support points made previously.

Operating Voltage	3 V
Measurement Range	±3g
0g Output Voltage	1.5 V

Table 1: Accelerometer Details

4.1.6 BMP280 Environmental Sensor

The BMP280 Environmental Sensor (henceforth referred to as "the environmental sensor"), while not as critical, in many ways, to the data collection in the lander, is more complex than the accelerometer.

4.1.6.1 Physics

Much like the accelerometer, the environmental sensor relies on MEMS to perform its measurements [see Section 4.1.4.1]. This is true for both temperature sensing and pressure sensing.

4.1.6.1.1 Temperature Sensing

The temperature sensing component of the environmental sensor relies on the use of a thermistor, for which resistance is either inversely proportional or simply proportional to temperature [21]

By applying a known current across the thermistor and measuring the change in voltage, one can calculate the temperature of the sensor, giving a value for temperature.

4.1.6.1.2 Pressure Sensing

Pressure sensing in the environmental sensor relies on the same principle of capacitive sensing discussed in relation to the accelerometer in Section 4.1.4.1.1, so it will not be reiterated here. However, this method of pressure-sensing makes the sensor sensitive to changes in acceleration, rendering its ability to measure atmospheric pressure impaired during moments of acceleration. Humorously, this does result in the team's minimum calculated altitude (measured environmentally) being negative [see Table 2].

Min Altitude (Environmental):	-0.02379501210786771

Table 2: Minimum Environmental Altitude

4.1.6.2 Calibration and Processing

The calibration and processing carried out by the environmental sensor is what increases its complexity beyond that of the accelerometer. As can be seen in Figure 7, the analogue front-end

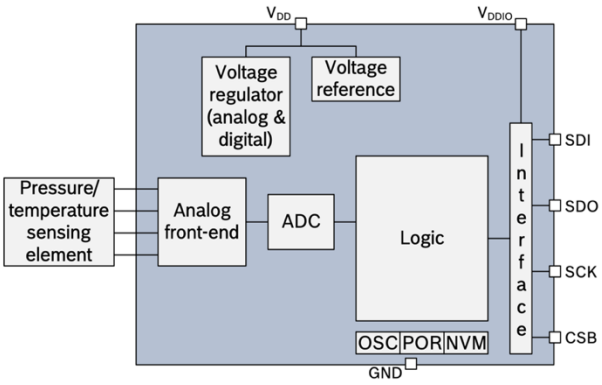


Figure 7: Operational Diagram of Environmental Sensor

of the system, analogous to the entirety of the accelerometer's processing, is only a small part of the system.

After the analogue signal is generated appropriately, presumably through methods similar to those discussed in Section 4.1.4.1.2, it goes through an analogue-to-digital converter, which permits the use of calibration logic, which gives values in the values of hPa and °C. These are then transmitted through the interface to the Arduino, where they are operated upon.

4.1.7 Battery

Using a battery is necessary for the lander, as its motion clearly renders a wired power source impractical. For this project, due to the adapter provided, a 9V alkaline chemical battery was used.

4.1.7.1 Differences Between Battery and USB Power

In theory, the outputted values in the same conditions should be the same for both battery and USB power, however differences were noted in the calibration values gathered when on battery power as opposed to on USB power and even during different battery calibration sessions. Despite this being only anecdotal, without any specific margins of error recorded, due to the prevalence of this issue among the team's peers, smart calibration was implemented [see Section 4.2.6.1.3].

4.1.8 PCB

The PCB provided for this project was critical as it allowed simplistic and consistent connection and interface with the different components of the setup. The provision of the PCB removed a significant amount of complication from manipulating the setup, albeit while removing a certain

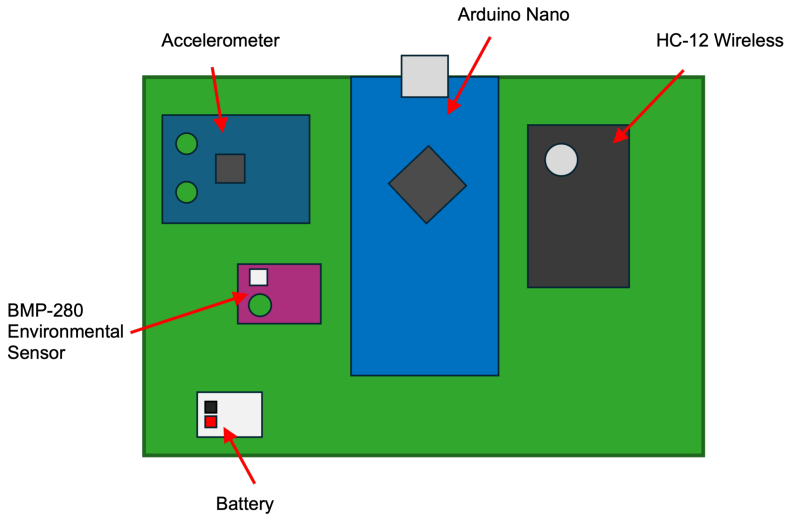


Figure 8: Arduino-PCB Diagram, showing all the components used in the project

degree of freedom. The setup of the PCB can be seen in Figure 8 [see Fig. 2 for an illustration of which ports interact with which components].

4.2 Software

Writing the software was an interesting challenge with plenty of room for improvement, innovation, and optimisation. Throughout the project, the team constantly strived to add to and improve the code, seeking out novel solutions and developing strategies to transmit, obtain, and process data more effectively.

4.2.1 Method

The method employed for interacting with the data was dual-processor oriented, employing close interaction between the Arduino and the connected laptop, using the Arduino almost solely for data collection, calculating only the bare minimum for the operations required for the Arduino and transmission, and calculating the rest with the laptop, where memory and processing speed were not a concern.

Arduino

In order to maximise transmission speed and reduce errors, it was decided to calculate the bare minimum required to facilitate functionality.

This consisted of the following:

- Acceleration (and relevant smoothing) (m/s^2)
- Pressure and temperature (and relevant smoothing) (hPa, °C)
- Data construction for transmission
- Mode detection

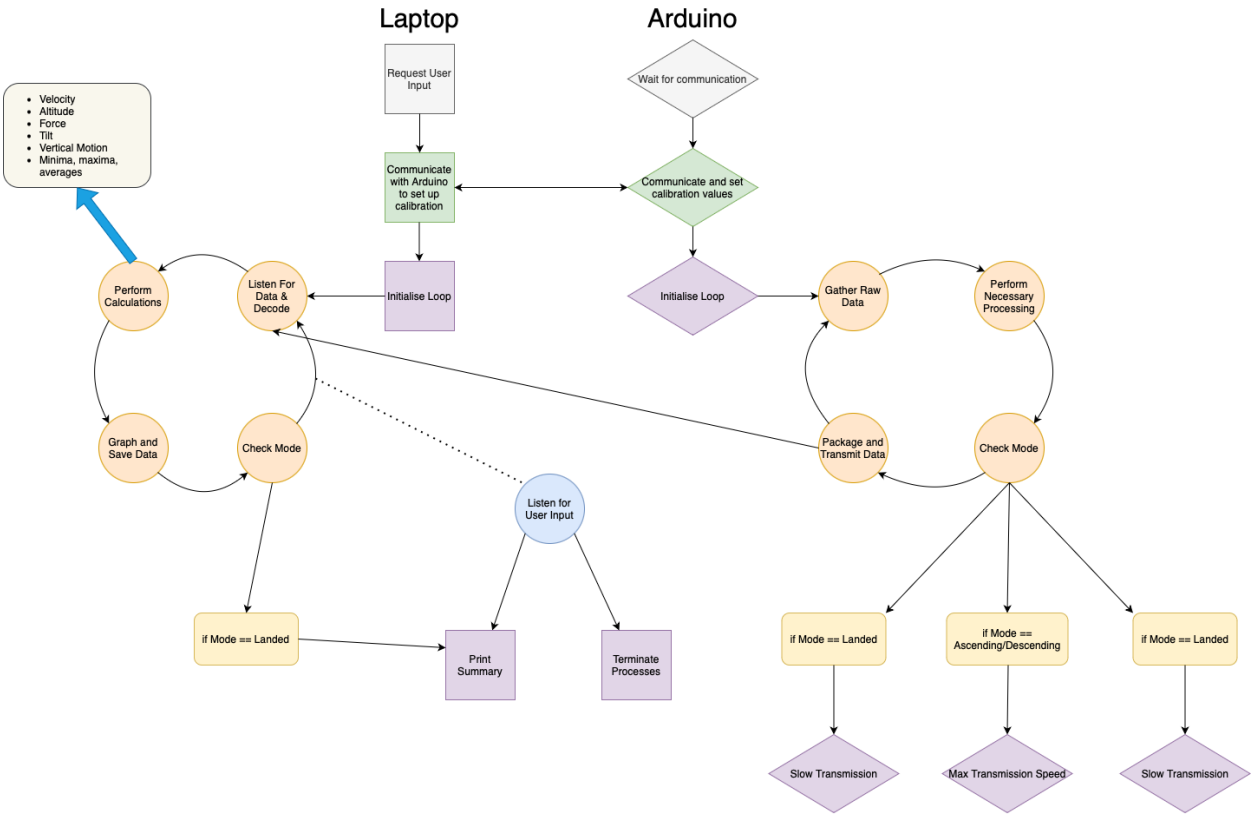
Laptop

Everything aside from the four processes specified above was calculated on the laptop [see Table 3]. The laptop values were calculated in python, using the data transmitted by the arduino.

Values	Units
Vertical Acceleration	m/s^2
Velocity (Including Vertical)	m/s
Force (Including Vertical)	N
Altitude (calculated by accelerometer and environmental sensor)	m
Tilt	°
Mode change times	s
Averages, maxima, minima	All

Table 3: Values calculated by the Python data-analysis code, alongside the units used for them

The flowchart seen in Figure ? illustrates the logic by which all of the relevant processes in this method occur, during normal operation. This logic was used during both the launch test and drop test, although the mode-checking was disabled during the drop test, for simplicity, and a few areas had to be added or improved before the launch test [see Section 4.2.15]. This is elaborated on in greater detail throughout Section 4.2.



4.2.2 Libraries

In order to make the data-processing system in Python, a number of Libraries were installed and imported to support certain operations.

4.2.2.1 PySerial

PySerial was imported in order to facilitate interaction with the serial port connected to the HC-12 Transceiver.

The main functions that were used were Serial, in_waiting, readline, and write, as demonstrated in table 4.

```
seri = serial.Serial(serial_port, baud_rate, timeout=timeout)
. . .
if seri.in_waiting:
. . .
line = seri.readline().decode('utf-8', errors='ignore').rstrip()
. . .
seri.write(message.encode())
```

Table 4: PySerial Example Code

4.2.2.2 Threading

Threading allows different processes to be executed concurrently, by executing them on different threads. This is important for graphing and receiving user-input without putting every call into the same main function. See Table 5 for an example use.

```
serial_thread = threading.Thread(target=read_and_process_serial_data, daemon=True)
    serial_thread.start()
```

Table 5: Threading Example Code

The same process was used for the handle_user_input(), as the graphing function had to run on the main thread, due to restrictions imposed by MacOS [12].

4.2.2.3 Matplotlib

Matplotlib was essential for this project, as the Serial Plotter program suggested did not work on MacOS and the majority of the team's programming and program execution was to take place on the chief programmer's laptop.

Matplotlib allows complex graphs to be constructed in python and displayed. The program has great flexibility, as demonstrated in Figure 9.

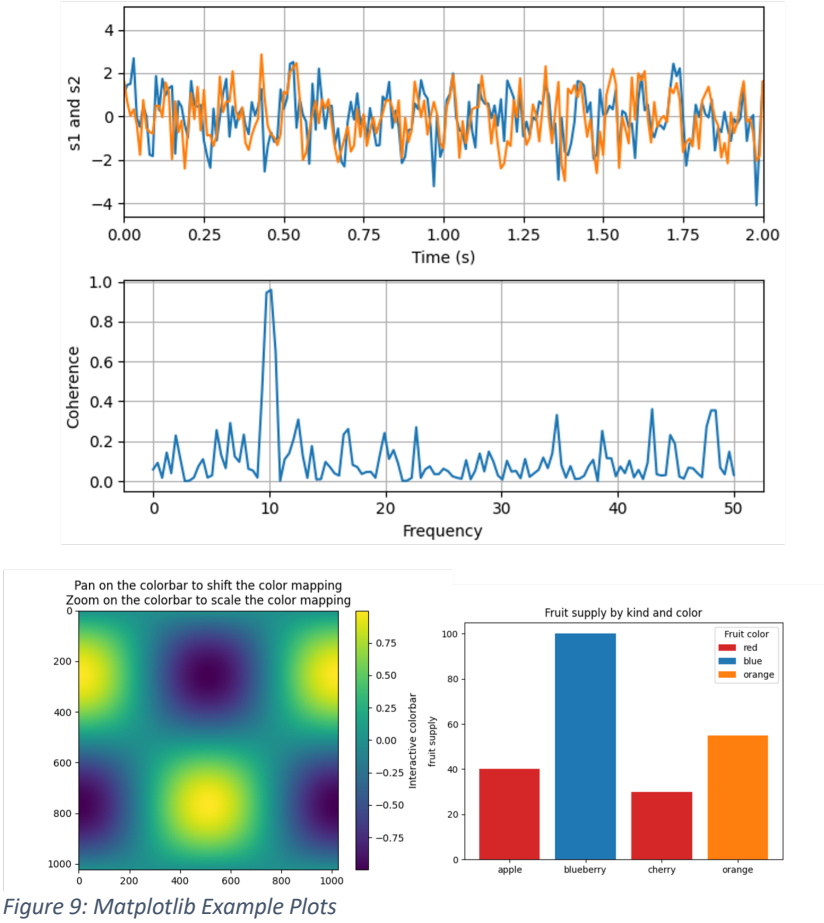


Figure 9: Matplotlib Example Plots

While requiring some interesting applications of the library [see Section 4.2.11], Matplotlib permitted the design and application of a custom Serial Plotter which worked on MacOS, albeit after some extensive error handling.

4.2.2.4 Openpyxl

Openpyxl is a Python library that increases the ease with which .xlsx files can be modified by the programming language. Using Openpyxl was one of the simplest parts of the project, simply requiring the specification of the file, sheet, and cell to append data [see Table 6].

```
workbookPath = '../TransmissionStorage.xlsx'
workbook = openpyxl.load_workbook(workbookPath)
sheet = workbook['MainSheet']
. . .
sheet.cell(row=rawValueRow+len(time), column=rawAyColumn).value=accelY[-1]
. . .
sheet['E3']=descendingTime
```

Table 6: Openpyxl Example Code

4.2.2.5 Math

Math is a very commonly used Python library that simply permits the use of more advanced mathematical operations than are provided by default in Python. In the team's case, the library was used to provided trigonometric functions, in order to calculate the tilt of the rocket [see Table 7 for an example].

```
tilt = math.asin(clamped_value)
...
yComponent = math.cos(Xtilt[-1]) * math.cos(Ztilt[-1])
vectorMagnitude = math.sqrt(xComponent**2+yComponent**2+zComponent**2)
```

Table 7: Math (library) Example Code

4.2.2.6 Pickle

Pickle is another extremely simple Python library, which is simply used to permit the storage of Python variables past runtime. This allows for later graphing and utilisation of data if necessary, providing greater flexibility, without having to deal with the complications that come with extracting data from a text file or excel file. An example of this library's use can be seen in Table 8.

```
def saveLists():#Saving data for later use
    dataToPickle = {
        "realtime": realtime,
        #Other variables omitted for brevity
    }
    with open('graphingLists.pkl', 'wb') as file:
        pickle.dump(dataToPickle, file)
```

Table 8: Pickle Example Code

4.2.2.7 Other Libraries

It should also be noted that there was also some minor use of other libraries, but as they are extremely simple and add what is essentially very base functionality to Python, their discussion has been omitted for brevity (time and struct are examples of these libraries).

4.2.3 Arduino Setup

4.2.3.1 Libraries

The libraries imported for the Arduino were required for functionality. The imports are specified here, but will be specified more briefly than the Python Libraries described in Section 4.2.2 [see Table 9].

```
#include <SoftwareSerial.h> //Permits the use of digital pins as Serial Pins
#include <Wire.h> //Permits interaction with devices using the I2C protcol. Used for the BMP280 Environmental. Provides functions used such as:
                  //begin(), write(), available(), read()
#include <SPI.h> //Not used for any particular device, but maintained for compatibility
#include <Adafruit_Sensor.h> //Supports the Adafruit_BMP280 library and allows standard interaction with Adafruit Sensors
#include <Adafruit_BMP280.h> //Library specifically for the BMP280 Environmental Sensor, allowing easy interaction
```

Table 9: Arduino Library Imports

4.2.3.2 Hardware Interactions

The hardware interactions in this project form the bulk of the PCB's use, allowing the team to gather data for the Python script to further analyse. For this reason a number of hardware-to-software interactions are specified early in the Arduino script [see Table 10 for an example].

```
// HC-12 Wireless Transceiver
SoftwareSerial HC12(7, 6); // HC-12 TX Pin, HC-12 RX Pin
// Bosch Environmental Sensor (BME/BMP280)
Adafruit_BMP280 env_sensor; // use I2C interface
Adafruit_Sensor *env_temp = env_sensor.getTemperatureSensor();
Adafruit_Sensor *env_pressure = env_sensor.getPressureSensor();
```

Table 10: Arduino Hardware Setup Example

These early definitions and setups simplify matters later on, allowing serial communications to be called for the HC-12 with simple calls, such as those seen in Table 11.

```
for (int i = 0; i < sizeof(dataStream); i++) {
    //This just writes the data to the HC12
if(HC12.available()){
    HC12.write(dataStream[i]);  }}
```

Table 11: HC-12 Example Calls

They also retrieve the abstract sensor interfaces for the environmental sensor, so they can be interacted using the Adafruit_Sensor library [see Table 12].

```
sensors_event_t temp_event, pressure_event;
  env_temp->getEvent(&temp_event);
  env_pressure->getEvent(&pressure_event);
  // Store value into variables: temperature (Celcius), pressure (hPa)
  TempNow = temp_event.temperature;
  PressNow = pressure_event.pressure;
```

Table 12: Environmental Sensor Example Calls

This early setup of hardware interactions will be essential for the effective operation of the lander code, allowing the team to successfully obtain and analyse data from the rocket launch and subsequent descent.

4.2.4 Data Gathering

4.2.4.1 Raw

4.2.4.1.1 Accelerometer

The raw data is obtained from the accelerometer as exemplified in Table 13. As can be seen, the analogue inputs are obtained as shorts from the analogue sensors. Technically this is unnecessary and results in some suboptimality, as the analogue outputs are always 10-bit integers [13], and shorts consist of 2 bytes (16-bits) [7]. However, this is to support custom data organisation [see Section 4.2.7.3]

```
void readFromAccelerometer(){
  short Xraw = analogRead(A1);
// Other axes omitted for brevity
//Used for transmission
  rawAx = (int8_t)(Xraw-calib0X);
  AccelNowX = mapAccelX((float)Xraw);
  smoothAccelReading(&AccelNowX);
}
```

Table 13: Accelerometer Data-Gathering Sample Code

As can be seen from Table 13, a mapping function was also used on the acceleration values that would be used for mode calculations. This was implemented as the team was unsatisfied with the variance from 0g caused by utilising Arduino's built-in map function with the positive and negative g acceleration values. An example of the mapping function is discussed in Section 4.2.9.3. This function was found to produce values closer to 0g when the sensor should have been outputting such values [see Table 14].

Built-In Map Function Average (After Smoothing)	Custom Map Function Average (After Smoothing)
2.12 ms^{-2}	0.37 ms^{-2}

Table 14: Comparison of the Average Divergence from 0m/s^2 when using the built-in map function and the custom map function

4.2.4.1.2 Environmental Sensor

The data-gathering for the environmental sensor is very similar to that of the accelerometer, with some notable differences.

The most notable differences are in the receipt of data, which is handled using the Adafruit_Sensor Library [see Section 4.2.3.1 and 4.2.3.2], and in the encryption of data for transmission. The data gathered from the sensors is stored in 16-bit integers (shorts were not used simply for consistency). However, it should be noted that pressure and temperature are transmitted as differences from certain values. For temperature it is the difference from the initial temperature, and for pressure it is the difference from ground pressure.

This allows temperature to be transmitted as a 10-bit integer that permits an 8 degree per transmission variance, and for pressure to be transmitted as a 14-bit integer that allows for variation of 163.84hPa from ground pressure, which was considered to be enough, as atmospheric pressure only varies between 100hPa at sea level [14] and varies by only 5.99hPa across an altitude variation of 50m, according to the barometric formula [see Equation 9].

$$P = P_0\left(1 - \frac{L \cdot h}{T_0}\right)^{\frac{g \cdot M}{R \cdot L}} \tag{9}$$

Pressure was compared to sea-level pressure as pressure was seen to be more sensitive to changes, and a small error could throw off all subsequent calculations. This resulted in a larger required data size, but the team was prepared to make that sacrifice in the interest of accuracy.

An example of the code for this section is shown in Table 15, showing only temperature in the interest of brevity.

```
void readEnvironmental(){
  // Read Environmental Sensor
  sensors_event_t temp_event;
  env_temp->getEvent(&temp_event);
  // Store value into variables: temperature (Celcius), pressure (hPa)
  TempNow = temp_event.temperature;
  smoothEnvReading(&TempNow);
  tempChange = (int16_t)(TempNow-initialTemp);
```

Table 15: Environmental Data Gathering Sample Code

4.2.4.2 Smoothing

The smoothing for both the environmental sensor and accelerometer is extremely similar, so the accelerometer-reading smoothing will be explained here, as it can stand in for the environmental smoothing as well, and as such, including the environmental smoothing in this section would serve no purpose.

The smoothing uses a principle of averaging, where a buffer of previous values is stored. The new value obtained from the sensor is then inserted into the buffer, replacing a previous value. After this, the values are averaged, returning a smoothed value. A visualisation of this can be seen in Figure 10.

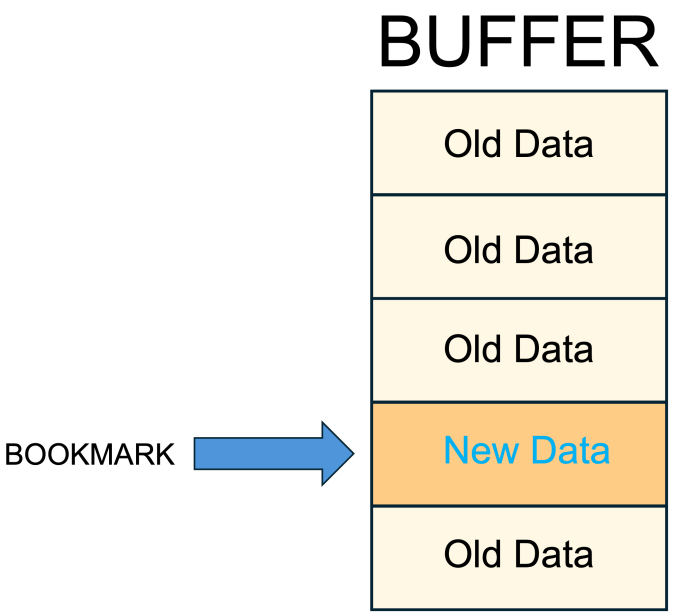


Figure 10: Data Buffer Visualisation. For data smoothing, a buffer is stored of old values, which are periodically replaced by new values, then averaged. This ensures a more gradual change in values that is less susceptible to noise.

It might be thought that this would result in slow or inaccurate results, however, due to the extremely rapid speed at which the code runs [see Section 4.2.7.3], this generally does not matter hugely, and simply results in lessening the impact of noise in results.

A snippet of the code that serves this function is shown in Table 16.

```
void smoothAccelReading(float *a_x /*Pointer to the input variable, which allows it to be changed without needing to return a value or use global variables*/){
  float SumX = 0;
  AccelBufferX[currentBookmarkA] = *a_x; //Updates one of the data values to the new value
 currentBookmarkA++; //Cycles which value is getting updated

  if (currentBookmarkA == WINDOW_SIZE) {currentBookmarkA=0;}
  for(int i = 0; i<WINDOW_SIZE; i++){ //Gets the sum of the buffers
    SumX+=AccelBufferX[i];    }
  *a_x = SumX/WINDOW_SIZE; //Performs the averaging calculation
}
```

Table 16: Data Smoothing Sample Code (Note: two other variables are passed to the function, hence the reason for using pointers instead of returning the value)

As can be seen from Table 16's code, pointers to the input data (e.g. accelNowX) are used to allow its manipulation within the function. This was implemented to circumvent restrictions

which ensure only one value can be returned in a function in C. This could alternatively have been achieved using structs, but this was considered simpler for the team's purposes.

4.2.5 Main Function – Python

While there is more to discuss about the Arduino code, to proceed, further investigation of the Python code is required in order to provide context. For this reason, the main Python function will be explored.

```
if __name__ == "__main__":
    setupFunc()
    #Start the serial data handling and input handling separate threads
    serial_thread       =       threading.Thread(target=read_and_process_serial_data,
daemon=True)
    serial_thread.start()
    serial_thread2 = threading.Thread(target=handle_user_input, daemon=True)
    serial_thread2.start()
    # Start the graphing on the main thread – required because of MacOS
    start_graphing()
    time.sleep(5)
    graphAfter()
```

Table 17: "Main Function" - This code is run when the Python program is run. Then it calls a number of other functions, which, in turn, call more functions

As can be seen from Table 17, the primary purpose of the main function is to start threads and call functions. However, it is the content of these functions that matters.

The "setupFunc()" function initiates communication with the Arduino, facilitating calibration and initiation of data transmission at the appropriate time. Once this function has completed, three threads are run:

1. serial_thread: This runs the "read_and_process_serial_data()", which, as the name would imply, reads and processes data from the serial port, which is received by the HC-12 Transceiver
2. serial_thread2: While not having a descriptive name, this thread runs the "handle_user_input()" function, which allows for user input during the operation of the function, permitting the user to terminate all or part of the function and allowing important data to be printed at the user's discretion.
3. Main thread: The graphing is run on the main thread as MacOS doesn't allow graphical processes to be run on secondary threads [12]. This displays a live feed of the data incoming, in several important metrics.

After the graphing function concludes, there is a brief wait, before the "graphAfter()" function is called. This produces summary graphs for the metrics displayed by the main graphing function, as Matplotlib makes it difficult to implement a customisable scrolling graph and this was seen as a simpler alternative.

4.2.6 Setup Function

The setup function “setupFunc()” called in the code snippet displayed in Table 17 is vital to initiate communication between the Arduino and laptop. Its functionality and methodology is explained in this section.

4.2.6.1 Communication

The majority of the Setup Function involves two-way communication with the Arduino Nano, via the HC-12 Transceiver. This allows for flexibility in the setup and ensures that the Arduino and laptop are properly synced to ensure the effectiveness of mutual calibration, graphing, and binary transmission.

The initial point of the setup function is shown in Table 18, with different input choices resulting in different procedures.

```
while True:
        ChoiceNum = input("Use old calibration values (1), recalibrate (2), or read pre-initialised data-stream (3):   ")
```

Table 18: Code initiating Arduino-Laptop communication. The choice selected here (1, 2, or 3), determines what functions are executed thereafter

While not called here, the function “calibrationStep()” is ubiquitous to all the following procedures, so it will be described here for simplicity [see Table 19]

```
def calibrationStep(instruction, message, expectedMessage, success_message):
        while True:
            numberIn = int(input(instruction))
            if numberIn == 1:
                if message == "C6":
                    findCalibValues()# This causes the laptop to listen for calibration values after sending the last calibration request

                seri.write(message.encode())
                response = waitForMessage(message, expectedMessage)
                if response == "SUCCESS":
                    print(success_message)
                    return True
                elif response == "RESET":
                    return False
            elif numberIn == 404:
                response = waitForReset()
                if response == "RESET":
                    return False
            else:
```

```
                print("Input not recognised. Try again.")
```

Table 19: "calibrationStep()" function; this function takes a number of inputs, to allow it to call functions to send messages, wait for responses, receive calibration values, or reset calibration

As can be seen here, the calibration step function requests an input. If the input is “1”, to confirm, it will send the specified message to the Arduino, before calling the “waitForMessage()” function to stop the function moving on without the Arduino.

As well, if the “waitForMessage()” function returns “SUCCESS”, the function returns True, allowing steps to progress.

However, if the function returns “RESET”, calibration is reset as the Arduino and laptop have moved out of sync somewhere. This failsafe prevents a litany of errors that could occur without it.

There is also another failsafe which enables manual resetting of calibration if the user inputs “404”. This allows the user to continue their use, without having to manually rest both the Arduino and the Python script.

Finally, chronologically, if not semantically, the “calibrationStep()” function checks for a “C6” message. This is sent in the last calibration message. This causes the “findCalibValues()” function to be called, which allows the laptop to receive the calibration values calculated by the Arduino, and thereby interpret the raw, 10-bit acceleration values it receives [see Section 4.2.7.3 for the reason why raw values are transmitted].

The “waitForMessage()” function should also be described here, to clarify its effects. As such, the main body of the function is displayed in Table 20.

```
def waitForMessage(message, expected_message, firsttimeout = 1, timeout_duration=5,
reset_message="R", reset_ack="R"):
        start_time = time.time()
        secondtime = time.time()
        while True:
            if seri.in_waiting:
                line = seri.readline().decode('utf-8', errors='ignore').rstrip()
                if expected_message in line:
                    return "SUCCESS"
                elif reset_ack in line:  # Handle receiving reset acknowledgment
                    print("Reset acknowledged. Starting calibration over.")
                    return "RESET"
# Error handling omitted from this snippet
```

Table 20: Main body of the "waitForMessage()" function

This function allows for flexibility in the messages that can be transmitted and acknowledgements that can be listened for.

The similar function called “waitForReset()” is based off the same code, but listens exclusively for the “RESET” command, in the event of a manual reset.

4.2.6.1.1 Arduino Code

While a lot of the project's processing is done on the laptop for the purposes of efficiency and resource-management, in the calibration and setup stage, the Arduino performs a significant amount of the work. The Arduino side of the programming will be discussed in subsequent sections, but there are a few elements that are common to all the Arduino calibration interaction.

```
void checkInput(){
  if(HC12.available()){
    char inChar = (char)HC12.read();
    inputString+=inChar;
    receiveInput();
  }
}
```

Table 21: The "checkInput()" function, which listens for messages sent to the HC-12 transceiver. It should be noted that this function only works before modes are initiated, or in the pre-launch, or landed modes, due to the half-duplex nature of the HC-12 transceiver [15]

Firstly, there is the "checkInput()" function [see Table 21], which is called at the start of every loop before rapid data transmission begins. This function checks the HC-12 buffer, and if there is data there, adds it to a string and reads it using the "receiveInput()" function [see Table 22].

```
void receiveInput(){
  unsigned long currentTime = millis(); // Get the current time in milliseconds
  // Check if more than a second (1000 milliseconds) has passed since the last command
  if (currentTime - lastCommandTime > 1000) {
    if (inputString.indexOf("C") != -1){ // Check if "C" is contained within the inputString
      if (inputString.indexOf(String(calibNum))!=-1){
      Calibrate(calibNum);
      calibNum++;
      inputString = ""; // Clear the input string after processing the command
      }
// Other command reads and error-handling omitted for brevity
```

Table 22: The "receiveInput()" function, which allows the Arduino to act on the data found by the "checkInput()" function

The "receiveInput()" function first ensures that an acceptable amount of time has passed since the last command in order to prevent accidental double-commands, and misreads which were common for a time (this delay could likely be shortened, but the setup is not time-sensitive).

Next, the function checks if the received commands match any commands for which the function has an action assigned. This comes with a failsafe for the calibration, in which if the calibration number doesn't line up, the function assumes a desync. It checks a few times, to reduce the chance of this desync being a simple transmission error, then calls the "resetCalibration()" function to resync the laptop and Arduino.

4.2.6.1.2 Old Calibration

This option for calibration is selected by inputting “1” when asked for a choice, as seen in Table 23.

What it does is sends a signal to the Arduino that tells it to use the old calibration values, which were pre-calculated. This reduces accuracy, not accounting for changes in orientation or battery degradation, or any other sources of error, but is useful for rapid testing, which was critical during development of the code.

The laptop then doesn’t update its own calibration values, which are set to the old values by default, but simply begins the data streaming.

The full laptop code for the old calibration option is shown below in Table 23.

```
if ChoiceNum == "1":
            if calibrationStep("Enter 1 to confirm choice:  ", "OLD", "X",
                               "Calibration  Complete,  Data  Transmission  Has
Begun"):
                break
            else:
                continue
```

Table 23: Main body of the code executed upon the “Old Calibration” user input

The Arduino then takes the input, using the function shown in Table 22 and executes the “useOldCalibrations()” [see Table 24] function to finalise setup. This sets all calibration values to precalculated values, then begins the main logic loop by setting the variable beginner to 1 [see Appendix A1.1].

```
void useOldCalibrations() {
  calib0X=oldCalib0X;
  calibPosX=oldCalibPosX;
  calibNegX=oldCalibNegX;
//Other axes omitted for brevity
  provideSlopes();
  transmit("X");
  calibNum = 7;
  delay(100);
  beginner = 1;
}
```

Table 24: Code executed when the Arduino receives the command to use its pre-calculated ("Old") calibration values

4.2.6.1.3 Recalibration

The recalibration option, selectable from the initial input, provides, in theory, more accurate, temporally relevant calibration values. It achieves this by communicating with the Arduino, indicating when the Arduino should calibrate and listening for its acknowledgement of the completed step. A visualisation of this approach can be seen in Figure 11.

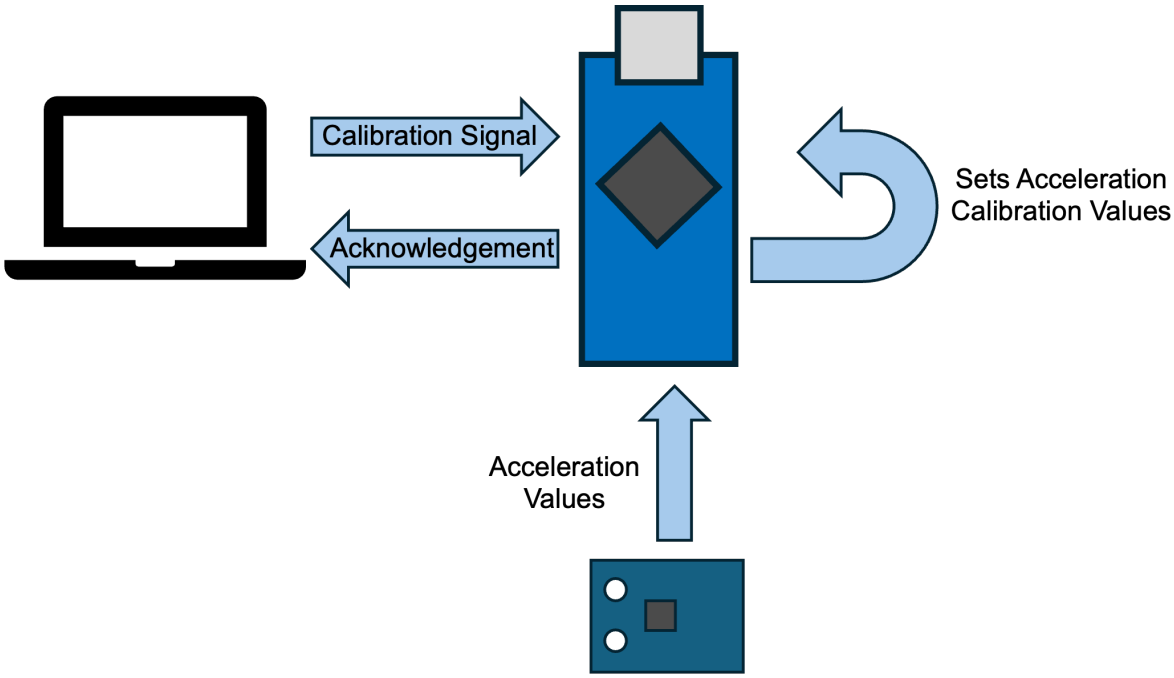


Figure 11: Diagram showing the communication between the Arduino Nano, accelerometer, and laptop. It should be noted that upon the last value being calibrated, the Arduino transmits the calibration values alongside its acknowledgement

```
elif ChoiceNum == "2":


            if not calibrationStep("Please orient Arduino on the Z-axis. Enter '1' to confirm Z-axis calibration (Note: at any point you can input '404' to restart calibration):   ", "C1","C", "Z Calibration 1 Complete. Please Invert Arduino."):
                continue  # Restart calibration if RESET is acknowledged
# Most calibration checks omitted from this snippet due to similarity
            if not calibrationStep("Enter '1' to begin inverted Y-axis calibration: ","C6","C", "Y Calibration 2 Complete. Good job!"): #This transmission is important, as it triggers the listening for calibration values
            if calibrationStep("Enter '1' to begin data transmission:   ","C7","C",
                               "Data Transmission Has Begun"):
                break #exit the loop if all previous steps have been executed sequentially without a "false" being returned
            else:
```

```
                    continue
```

Table 25: Sample of code executed upon the "Recalibration" user input

As can be seen from Table 12, the code requests various orientations, then, when the user is satisfied with the orientation, they confirm that the Arduino should calibrate. This action is then carried out by the “calibrationStep()” function that was discussed previously.

As well as this, in the final confirmation, which begins data transmission, the laptop first receives the calibration values due to the condition triggered by the message of “C6” [see Table 19].

The receipt of the calibration values is detailed in Table 26.

```
def findCalibValues():
    global calibPosX, calib0X, calibNegX, calibPosY, calib0Y, calibNegY, calibPosZ,
calib0Z, calibNegZ
    line = seri.readline().decode('utf-8').strip()
    # Check if the line is not empty
    if line:
        # Split the line into values
        values = line.split(",")
        # Ensure that there are enough values in 'values' to unpack successfully
        if len(values) == 11:
            identifier, calibPosX, calib0X, calibNegX, calibPosY, calib0Y, calibNegY,
calibPosZ, calib0Z, calibNegZ, calibTemp = values
# Error handling omitted for brevity
```

Table 26: Main body of the calibration values receipt code

In essence, this function listens for a string-based input of calibration values, then, if there is no error, it writes the calibration values to global variables. Otherwise, it calls for a reset.

This is important, as it permits the transmission of acceleration values as raw, 10-bit numbers, which reduces the size of the transmissions, allowing them to be faster.

On the Arduino end of the process, things are much simpler, with the main function being called being the elementary “Calibrate()” function [see Table 27]

```
void Calibrate(int calibMode){
  float xValue = analogRead(A1);
//Other axes omitted for brevity
  if(calibMode == 1){
    calibPosZ = zValue;
    calib0X = xValue;
    calib0Y = yValue;
    transmit("C");
    Serial.println("Received");
  }
```

```
//Other calibMode conditions omitted due to their similarities
  else if(calibMode == 7){
      calib0X = calib0X/4;
      calib0Y = calib0Y/4;
      calib0Z = calib0Z/4;
      provideSlopes();
      readEnvironmental();

      String transmitter = "CALIBVALUES";
      transmitter = transmitter + "," + slopePosX + "," + calib0X + "," + slopeNegX
+ "," + slopePosY + "," + calib0Y + "," + slopeNegY + "," + slopePosZ + "," + calib0Z
+ "," + slopePosZ + "," + tempNow;
      transmit(transmitter);
      transmit("C");
      beginner =1;
  }
//Error handling omitted for brevity
```

Table 27: Arduino Calibration Code Sample. While longer than the calibration-receipt on the laptop, it is significantly shorter than the rest of the calibration-interaction laptop code, and also simpler upon inspection

Here one can see the responses to the Python commands. Here, depending on how many calibrations have already occurred, different sensors are calibrated from the raw sensor values gathered directly using an “analogRead()”.

Then, finally, the calibration values are transmitted. As during the calibration there is plenty of time to transmit data, it is sent as a string for simplicity.

4.2.6.1.4 Existing Data Stream

The option to read from a pre-initialised data stream is the simplest option that can be selected from the input request, however, it proved useful in testing, and could also be used if connection with the Arduino is lost mid-transmission.

It simply asks for confirmation, then exits the setup function, allowing the rest of the functions to proceed uninterrupted [see Table 29]. If this option is selected when there is no pre-initialised data stream, no error will be thrown and the user will simply have to terminate the script and initialise it again.

```
elif ChoiceNum == "3":
            confirmNum = input("Enter 1 to confirm choice:  ")
            if confirmNum == "1":
                break
            else:
                continue
```

Table 28: Main body of code which initialises listening to a pre-existing data-stream

4.2.6.2 Error Handling

While the inclusion of a while loop with continue statements does prevent a number of possible errors from occurring, testing unveiled a few more issues with the team's setup which needed to be rectified with further error handling.

The crux of this error handling is an addition to the waitForMessage() function described in Table 29.

This addition makes it so that if there is no response from the Arduino for a duration of 1 second, the original message is sent again. If no response is received for 5 seconds, a reset signal begins to be broadcast alongside this. This makes it so that, exceptional circumstances aside, any errors in communication will be resolved.

```
def waitForMessage(message, expected_message, firsttimeout = 1, timeout_duration=5, reset_message="R", reset_ack="R"):
        start_time = time.time()
        secondtime = time.time()

        # Main logic omitted. Previously omitted logic begins here:
                if (current_time - start_time) > timeout_duration:
                    seri.write(reset_message.encode())  # Send RESET command
                    print("No response, sending RESET.")
                    start_time = current_time  # Reset the start time for the next interval
                    timeout_duration = 2  # Set the new timeout duration for RESET handling
                elif ((current_time-secondtime) > firsttimeout) and ((current_time-start_time) < timeout_duration):
                    print("No response, retrying")
                    seri.write(message.encode())
                    secondtime = current_time
```

Table 29: Main body of "waitForMessage()" function error handling

This behaviour was added after it was noticed that the Arduino wouldn't always respond to the first message sent. While there is little supporting evidence for any particular hypothesis, the team's prevailing theories are that due to the HC-12's half-duplex nature [15] (which also prevents it from receiving commands when transmitting data rapidly), it may have been transmitting when the message was initially broadcast, or alternatively that some of the message was corrupted during transmission. Either way, retransmission proved to be effective in mitigating this issue.

It should also be noted that there is a delay included in the reading in order to prevent accidental double-commanding, that these issues were present even without this delay.

The handling of this error was especially crucial, as the use of this technique was required several times on launch day. In Table 30, a snippet of the terminal output on the day can be seen. As shown, it includes the retrying and reset measures, although it appears only the retry measure was acted upon by the Arduino. All in all, this is very curious behaviour, but due to the team's ingenuity, it was able to be remedied.

```
Enter '1' to confirm inverted Z-axis calibration:   1
No response, retrying
No response, retrying
No response, retrying
No response, retrying
No response, sending RESET.
No response, retrying
No response, retrying
Z Calibration 2 Complete. Please orient Arduino on X-axis.
Enter '1' to confirm X-axis calibration:   1
```

Table 30: Launch day terminal sample output

4.2.7 Data Transmission

For this project, the team transitioned through 3 different methods of data transmission. These data transmission methods grew iteratively more complex and more efficient. This increase in efficiency was important to allow faster transmission, which was important as due to the restrictions of Software Serial, attempting to use baud rates higher than 9600 would become unreliable [see Section 4.1.2.2].

Because of this, the team wanted to use their existing bandwidth more efficiently, allowing more rapid data-processing. The development of the team's data transmission strategies is documented in this Section.

4.2.7.1 String Method

The "String Method" was included with the project's template code [16]. This method was very simple and depended on Arduino's built-in string datatype, as well as the print() function in Software Serial.

```
dataToSend = "";
        dataToSend = dataToSend + time_now + "," + mode + "," + AccelNowX + "," +
AccelNowY + "," + AccelNowZ + "," + PressNow + "," + TempNow + "\n";
```

Table 31: Example of how data is packaged in the string-based communication method. From here, the string would simply be sent with "HC12.print()"

The String Method was implemented using code similar to that found in Table 31. As can be expected, the String Method was very inefficient, as each character in a string takes up a byte, or 8 bits. This is far more data than is necessary to send, and also necessitates the existence of a delimiter (","), as well as a terminator ("\n"). The one redeeming feature of the String Method was that it sent varying amounts of data depending on the length of the various elements in the

string. Unfortunately, this came at the trade-off of requiring the delimiters, which could have been implemented in any other method, although it was decided by the team that the added overhead outweighed the potential benefits.

For these reasons, once the code was working as expected, the String Method was swiftly abandoned in favour of more efficient methods.

4.2.7.2 Float Binary Encoding Method

The second method explored for data transmission was transmitting the floats as binary, rather than as strings. This was more efficient as long as the strings replaced by each float were longer than 3 characters, as each float consisted of 4 bytes, and each character consisted of 1, plus a required delimiter. In most cases this was more efficient, although it was noted that ideally, the mode wouldn't be transmitted as a float, as it would only ever be one of 4 numbers, requiring only 2 bits, rather than 32, in order to display it.

Due to this increase in efficiency, a version of the Arduino and Python code was written to use binary-encoded floats, instead of Strings. A snippet of the float encoding code can be seen in Table 32.

```
void transmitBinaryFloats(float* floatArray, size_t numFloats){
  for (size_t i = 0; i<numFloats; i++){
    HC12.write((byte*)(void*)&floatArray[i], sizeof(float));
  }

  byte terminator[4] = {0xFF, 0xFF, 0xFF, 0xFF};
  HC12.write(terminator, 4);
}
```

Table 32: Float Binary Encoding Transmission Method Sample Code

This function was called in the main loop, after first calling the function "readPackageAndTransmit()", which simply adds the relevant floats to the float array that is then passed to the transmission function. Due to this function's simplicity, it is not detailed here, but only relegated to Appendix A1.1 with the remainder of the code.

The effect of the new function was great. In theory, it should result in a 46.7% decrease in transmission time [see Equation 10].

$$\begin{gathered}\text{String (a sample string taken from results)} \\ = 35451{,}2, -2.032491207, -1.243833122, -11.27009392, -1.243833122 \backslash n \\ = 60\text{B} = 480\text{b}\end{gathered}$$

$$\begin{gathered}\text{Float Data Size} = 4 \cdot 8\ (\text{time, mode, accel X, Y, Z, pressure, terminator, deliminator}) = 32\text{B} \\ = 256\text{b} \\ \text{Percentage Size Decrease} = \text{Theoretical Time Decrease} = \ 100 \cdot \left(1 - \frac{256}{480}\right) = \\ 46.667\%\ \text{decrease}\end{gathered} \tag{10}$$

In practice, it resulted in a 42.3% decrease in transmission time [see Table 33], according to calculations made using data in the project excel files [see Section 4.2.11]. This float-based method was used for the drop test [see Section 5.1].

Average String Transmission Time	Average Float Transmission Time	Percentage Difference
0.0982	0.0567	42.3%

Table 33: Display of practical difference between float and string transmission, in terms of time

This suggested that the hypothesis of the transmission bottlecap was correct, and that by compressing the data, faster transmission, and thereby greater accuracy was possible. In this case, the Arduino is able to transmit 1.875 times the amount of information that was possible before.

4.2.7.3 Custom Binary Encoding Method

The final method for data transmission that was developed was one in which custom binary datatypes were encoded. This has been touched upon in previous sections, but the idea will be clarified in Table 34.

Variable	Data Size	Reasoning
Time	12 bits	If one uses the change in time, rather than the actual time, with 12 bits, one can get up to 4 seconds with 3 decimal places of accuracy *It was considered to use the Python time module for this, but the readings were too far off between transmissions [see Appendix A2]*
Mode	1 bit	By simply transmitting a 1 instead of a 0 each time the mode changes, one can send only 1 bit to send all the required info
Acceleration (Y)	10 bits	The raw values provided by the accelerometer are 10 bits in size [13]
Acceleration (X,Z)	8 bits	These values cannot exceed ±g, theoretically so only 8 bits are needed.
Pressure	14 bits	[See Section 4.2.4.1.2]
Temperature	10 bits	[See Section 4.2.4.1.2]

Terminator	12 bits	A large terminator is required to prevent the possibility of it appearing in the data. After storing the rest of the data, the choices are between a 4 bit terminator and a 12 bit terminator, without using bit field structs. A 4 bit terminator theoretically has a $2^{-4} = \frac{1}{16}$ chance of randomly occuring, which was too common for the team, so a 12 bit terminator was decided upon, with a 2^{-12} chance of randomly occurring.
TOTAL	59 bits	

Table 34: Illustration of how small the data transmitted can be, in terms of file-size

With this data-compression strategy, the team was able to achieve a data-size of 59 bits, just 23% of the float data size. In theory, this should result in a transmission time decrease of 77% compared to the float strategy, and in practice, it does result in a 71.4% decrease in transmission time [see Table 35].

Average String Transmission Time	Average Float Transmission Time	Average Custom Transmission Time	Percentage Difference (Custom vs Float)	Percentage Difference (Custom vs String)
0.0982	0.0567	0.0162	71.4%	83.5%

Table 35: Illustration of the practical speed differences between the string-method, float-method, and custom-method

With this new data compression technique, the transmission time is only 16.5% of the original transmission time. This allowed the team to make 6 times the transmissions that they were originally capable of, greatly increasing accuracy.

A snippet of the custom binary system, including transmission, is shown in Table 36. This method of encoding does require bit manipulation within uint8_t datatypes, and if the team were to attempt this again, with the benefit of hindsight, using bit field structs would have been seriously considered.

```
void transmitOptimisedBinary(){
  uint8_t dataStream[9] = {0};
  // timeChange: 12 bits
    dataStream[0] = timeChange >> 4; // Top 8 bits
```

```
    dataStream[1] = (timeChange & 0xF) << 4; // Bottom 4 bits of timeChange
// Intermediate bit manipulation excluded for brevity
    // Terminator: 12 bits
    dataStream[7] |= (terminator >> 8) & 0xF; // Top 4 bits of terminator
    dataStream[8] = terminator & 0xFF // Bottom 8 bits of terminator
  // Transmit the data stream
  for (int i = 0; i < sizeof(dataStream); i++) {
    //This just writes the data to the HC12
    HC12.write((byte*)(void*)dataStream[i], sizeof(uint8_t));
  }
}
```

Table 36: Custom binary encoding sample code

4.2.7.4 Possible More Efficient Methods

While the team was satisfied with their current methods and the deadline for the test was fast approaching, since the previous method was finalised, there have been some considerations about methods with possibilities to compress the data even further.

One method suggested was to use batch-transmission, whereby a number of packets might be sent at once, with a single terminator, in order to reduce the overhead caused by the terminator's transmission. This could possibly be combined with Run-Length-Encoding [17] to reduce the transmission cost even further, although it was unclear how much values would repeat, making the impact of this unclear.

Another method considered, although much more complicated, was Huffman Coding. This would have analysed the data that the code transmits, determining the frequency of various elements, before creating a Huffman Tree, which reduces the cost of sending more common data [18]. However this is just a high-level overview and it was unclear just how much this would impact the data-transmission speed.

All in all, it was decided that leaps and bounds had already been made in regards to data compression, and that any further compression attempts would yield significantly diminishing returns, in this case at least.

4.2.8 Reading Serial Data

Once the data has been transmitted by the Arduino, it needs to be interpreted and used by the Python script. To this end, it must be decoded [see Table 37]. This simply involves using the terminator to identify a string and then reversing the processes that occur in Table 37, which can be done simply in Python. After this point, there are values that can be used to calculate items of relevance, (altitude, velocity, etc.).

```
        while True:
           if ser.in_waiting > 0:
            buffer += ser.read(ser.in_waiting)
```

```
            terminator_pos = buffer.find(terminator)
            while len(buffer>=packet_size):
                while terminator_pos != -1:
                    # Ensure there is a complete packet before the terminator
                    if terminator_pos >= packet_size:
                        packet_start_pos = terminator_pos - packet_size
                        packet = buffer[packet_start_pos:terminator_pos]

                        try:
                            dataList = list(struct.unpack('<9B', packet)) #This
unpacks the data, reading it as a series of unsigned integers
                            try:
                                realtime.append((float((dataList[0] << 4) |
(dataList[1] >> 4)))/1000)
                            except ValueError:
                                if realtime:  # Check if not empty to avoid IndexError
                                    realtime.append(realtime[-1])
# The rest of the function is omitted for brevity, but follows the pattern of the code
provided. See Appendix A1.2.1 for the full code.
```

Table 37: Binary decoding sample code

4.2.9 Calculation Functions

Once the data has been obtained from the Arduino and decoded, operations must be performed on the data, in order to glean meaningful insights. To this end, a number of calculations are carried out. These range from extremely simple (value comparisons) to relatively complex (trigonometry and vector algebra [see Section 4.2.9.2]). In the description of these calculations, the actual code has generally been excluded for conciseness, as it is simply a replication of the mathematics, however the full code can be read in appendix A1.2.

4.2.9.1 Transmission Time

This is one of the simplest calculations, simply involving subtracting the previous time from the current time to find the time between transmissions [see Equation 11]

$$\Delta t = t_1 - t_0 \tag{11}$$

4.2.9.2 Tilt

"Tilt" in this scenario refers to the angle at which the rocket is at, when compared to the vertical. This can be seen in Figure 12, where tilt is labelled as "θ".

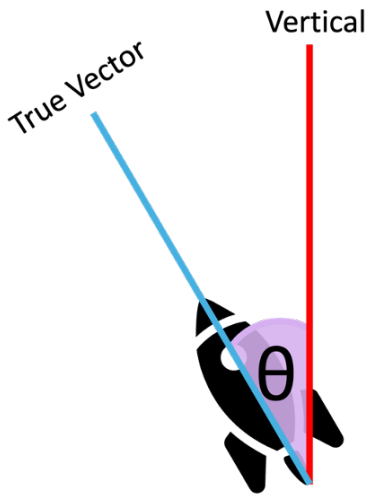


Figure 12: Illustration of what is meant by "tilt", the angle at which the tangent to the rocket's path differs from the vertical

4.2.9.2.1 Reason

Tilt was calculated due to concerns about the verticality of the rocket's trajectory. As the rocket could not be guaranteed, and in fact was unlikely, to travel a perfectly vertical path, simply taking the vertical accelerometer reading (the y-axis reading in this case) as the basis for vertical motion was unfeasible.

4.2.9.2.2 Mathematics

It was originally conceived to calculate tilt using the percentage of gravitational acceleration read by the vertical accelerometer. However, the flaw with this plan was that during moments of irregular acceleration (launch, initial parachute deployment), this would be unreliable.

Because of this, an alternative was suggested: as the x and z axis accelerometers should, in theory, not be able to experience acceleration other than gravitational acceleration due to tilt (this ignores wind acceleration, but that was thought to be a lesser source of error than rocket-tilt), their readings could be used to find their components of the tilt, and thereby obtain the absolute tilt of the rocket.

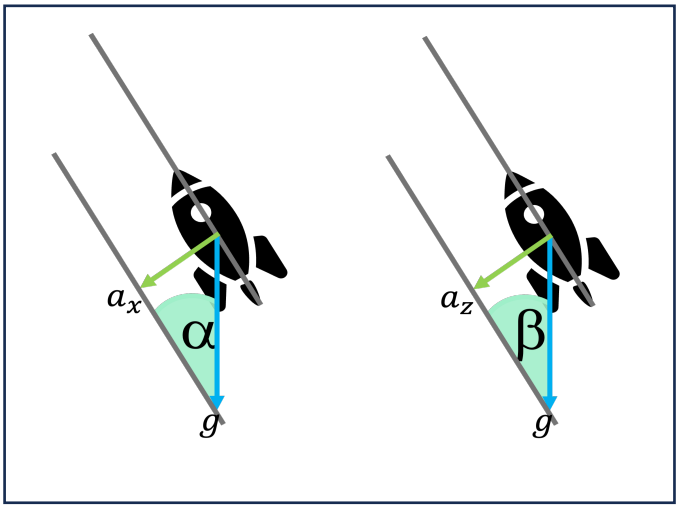


Figure 13: Illustration of a method to obtain rocket tilt in 2 axes based off accelerometer readings

$$\alpha = \sin\left(\frac{a_x}{g}\right)$$

$$\beta = \sin\left(\frac{a_z}{g}\right) \tag{12}$$

Equation 12: Formula for tilt calculation, á la Figure 13

The obtention of the x and z components of tilt is relatively simple, as can be seen in Figure 13 and thereby Equation 12. However, calculation grows slightly more complex when considering how the absolute tilt might be obtained from these.

It was decided to take the x and z components as components as a vector, and use them to find the y-component to thereby find the magnitude of the vector. A visualisation of this can be seen in Figure 14, along with the required calculations in Equation 13.

$$\begin{aligned} \text{X Component} &= \alpha \\ \text{Z Component} &= \beta \\ \text{Y Component} &= \gamma = \cos(\alpha) \cdot \cos(\beta) \\ \text{Vector Magnitude} = \text{Absolute Tilt} &= \theta = \sqrt{\alpha^2 + \beta^2 + \gamma^2} \end{aligned} \tag{13}$$

Figure 14: Seen here is a projection of the angle vectors, generated in Matplotlib. Shown is the product (black) of the X (red) and Z (purple) vectors [top left], Y(blue) and Z vectors [top right], and thereby all three vectors [bottom]

4.2.9.3 Acceleration

Both onboard the Arduino and on the laptop, acceleration is calculated from raw accelerometer values. This is achieved using a custom mapping function, an example of which is provided in Table 38.

```
float mapAccelX(float rawX) {
  float realAcceleration;
  if (rawX >= calib0X) {
    // If the rawValue is between 0 and g, use the slope for that segment
    realAcceleration = slopePosX*(rawX-calib0X);
  } else {
    // If the rawValue is between -g and 0, use the slope for that segment
    realAcceleration = slopeNegX*(rawX-calib0X);
  }
  return realAcceleration;
}
```

Table 38: Acceleration mapping sample code

While this function, which uses 3 datapoints to calculate the equation of 2 lines, may seem redundant, at least for the Arduino, due to its built-in "map()" function, Figure 39 concisely shows why it is important. In essence, while the built-in function generally produces similar values to the custom mapping function, especially close to g values, it loses some accuracy around 0g, which is one of the most crucial values to get correct.

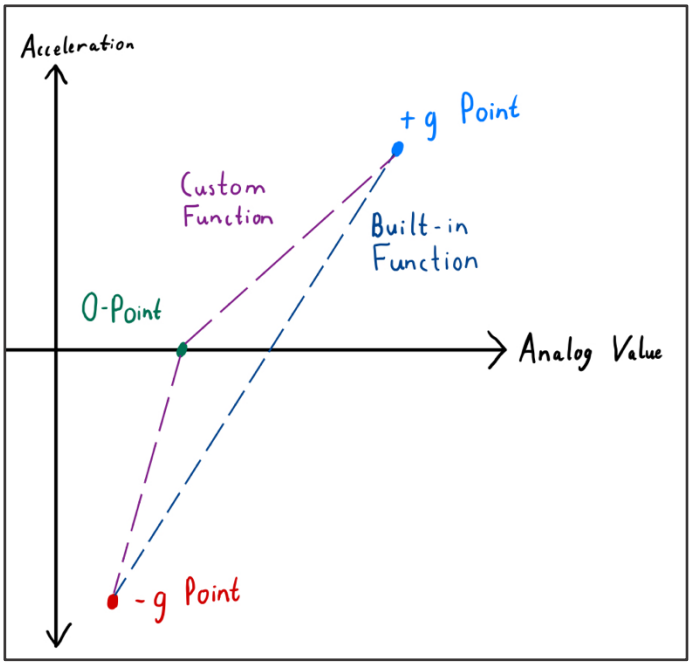


Table 39: Illustration of why custom mapping is useful (exaggerated for visual assistance)

The effectiveness of this approach is shown in Table 14, found in Section 4.2.4.1.1, which proves its utility in this application, showing how it is closer to outputting a 0 when it should be, as opposed to the built-in "map()" function.

The reason acceleration is calculated on the Arduino as well as the laptop, despite the transmitting of the raw data, is that the acceleration values are needed for the mode-changing logic, as detailed in Section 4.2.10.

4.2.9.3.1 Vertical Acceleration

Vertical acceleration is only calculated on the laptop, as the team decided that the computational burden of calculating tilt on the Arduino wasn't worthwhile, especially when considering the failsafes on the laptop-end, including the ability to trigger prints of the data at the user's discretion [see Section 4.2.14].

On the laptop, as tilt (θ) has previously been calculated [see Section 4.2.9.2], calculating the vertical acceleration is a simple matter of plugging the y-acceleration into Equation 14.

$$a_{vertical} = \mathrm{a_y} \cdot \cos(\theta) \tag{14}$$

4.2.9.4 Velocity

Velocity is calculated using a very simple variation of the equations of motion [see Equation 15]. There is little more to discuss here, except that the various axes are of course calculated with their respective accelerations, and vertical velocity is calculated using the vertical acceleration.

$$v = u + a \cdot \Delta t \tag{15}$$

4.2.9.5 Force

Force is another extremely simple variable to calculate. It simply relies on Newton's 2nd Law [see Equation 16], with the mass varying due to an "if" statement, which checks whether the capsule is ascending or descending. This is due to the fact that the capsule detaches from the rocket during descent, reducing the effective force.

$$F = ma \tag{16}$$

4.2.9.6 Altitude

To calculate altitude, the team used two different methods, one based off the environmental sensor, and another based off the accelerometer.

The environmental sensor method utilised the hypsometric formula, which relates altitude to pressure and temperature. Different versions of the hypsometric formula exist [22] but the one used for this project can be seen in equation 17.

$$h = \left(\left(\frac{P}{P_0}\right)^{\frac{1}{-5.255877}} - 1\right) \cdot \frac{T+273.15}{0.0065} \tag{17}$$

The accelerometer-based method once again utilised the equations of motion. As the time between calculations was so small, acceleration was assumed to be zero between each step, resulting in Equation 17, which was iterated many times in order to get what should be an accurate result for altitude.

$$h = h_{prev} + v \cdot \Delta t \tag{18}$$

4.2.9.7 Maxima, Minima, Averages

A requirement for the project was for it to be able to display the maximum, minimum, and average for a number of values. The methodology for these is quite simple.

For calculating the maximum and minimum values for a list, the team used the built-in “max()” and “min()” functions in Python.

Similarly, while there was no “average()” function built into python, one was easily constructed, simply by combining the “sum()” function and “len()” function to calculate the mean value of the list.

4.2.10 Mode Checking

Another requirement for the project was the activation of different “mode” states for the Arduino, although naturally these were replicated in the Python script. The required states are displayed in Table 40, and cause different behaviours also displayed in the table.

Mode	**Condition**	**Explanation**	**Effect**
Pre-Launch (1)	Activated by default	The Arduino will always be activated before its launch.	Transmits once very 5 seconds.
Ascending (2)	Previous mode is Pre-Launch	Ascent must come after pre-launch.	Transmits as fast as possible.
	Vertical Acceleration > g	During the moment of launch, vertical acceleration must exceed gravitational acceleration, in order to lift off the ground.	Sets launch time.

Descending (3)	Previous mode is Ascending Vertical Acceleration < g	Descent must come after ascent. In the moment that the parachute activates, there is a jolt upwards as the parachute decelerates the capsule. This results in a brief moment where vertical acceleration is less than gravitational acceleration.	Transmits as fast as possible Sets descent-begin time.
Landed (4)	Previous Mode is descending Vertical Acceleration ≈ g No recent change in pressure More than 3 seconds since descent began **OR** 20 seconds passed since ascent began	Landing must come after descent. If the capsule is on the ground, acceleration should be roughly equal to gravity and pressure will not have changed recently as there has been no recent change in altitude. It was not expected for the lander to be in the air for longer than 10 seconds, so a 20 second timer was introduced as a failsafe, in case the landing mode failed to activate.	Transmits once every 10 seconds. Python script prints maxima, minima, and averages.

Table 40: This table explains the logic behind the mode-checking function

The code for the mode-checking was simple, once the logic was worked out, simply saving certain time values and previous pressures to provide the values for comparison.

4.2.10.1 Testing

The mode-checking logic was tested, initially simply by manual acceleration of the PCB, then later during rocket-testing.

In the manual tests, the mode-checking logic worked mostly fine, although some artificial downward acceleration was required to trigger the descent mode, after which the landing mode generally activated nearly-instantly.

After this, the condition that descent must have begun more than 3 seconds prior was introduced to prevent accidental activations of the landing mode.

The rocket-based testing was unfortunately limited [see Section 2.3.2], however in the test conducted, the mode-checking logic worked effectively. Unfortunately, due to the unusual mode of flight experienced in Section 5.2, the mode-checking only reached the Ascending Mode. The user-input failsafe [see Section 4.2.14] was then utilised to print the data without having to wait the full 20 seconds for the landed mode to activate.

4.2.11 *Excel*

In order to better store and display the data, an excel sheet was created, which could then be modified using Openpyxl to display the relevant data. This was done to allow more simplistic viewing of the data, as opposed to in a text file, and to leave open the possibility for using Excel's tools for calculation and graphing.

4.2.11.1 Excel Sheet

A snapshot of the Excel Sheet can be seen in Figure ?. While time-consuming to create, the design of the Excel Sheet is simple, merely providing cells for storing various gathered data.

It should be noted that there is more to the sheet, in further columns, but as they are only for storing bulk data (similarly to the "Time", "Mode", "A_x", etc. columns) they have not been displayed in Figure 15.

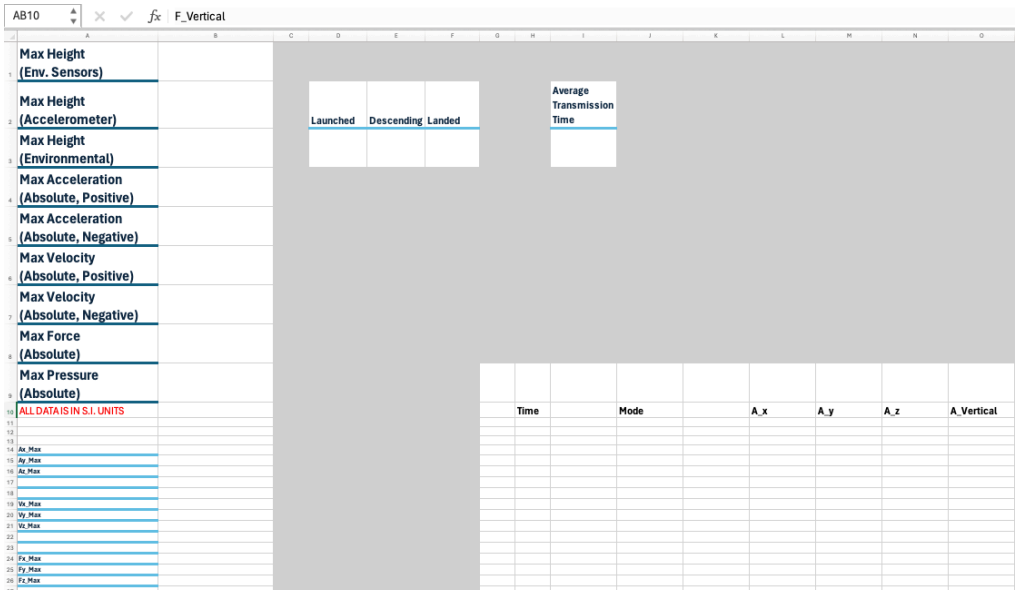


Figure 15: Excel data-storage sheet snapshot

4.2.11.2 Openpyxl

Similarly to the design of the Excel sheet, the creation of the code to interact with it was time-consuming, but not complex. As explained in Section 1.2.2.4, cells in an Excel Sheet can be modified using this library, simply by specifying either the row and column or the cell name.

Because of this, the overall design of the excel appendment functions does not differ much from what is seem in Section 1.2.2.4, simply adding variables, and incrementing down rows in the case of the bulk data columns.

4.2.12 *Live Graphing*

It was a requirement for this project, for graphs to be generated from the data received, in real-time. The suggested solution for this was Serial Plot, an application that permits the live-graphing of data directly from a serial port, without any setup bar sending the data in a string [19].

Unfortunately, there was no version of this software available for Mac [see Fig. 16], and the majority of the programming was being conducted on Mac. As well as that, serial-plotting applications like this limited the data transmission to string-methods, which would eliminate the vast improvements in transmission speed achieved in Section 4.2.7. Finally, these serial-plotting applications generally use device-time, rather than time-data broadcasted by the Arduino, which is an issue as demonstrated in Appendix A2. For these reasons, it was decided to build a custom serial plotter using Matplotlib and Python.

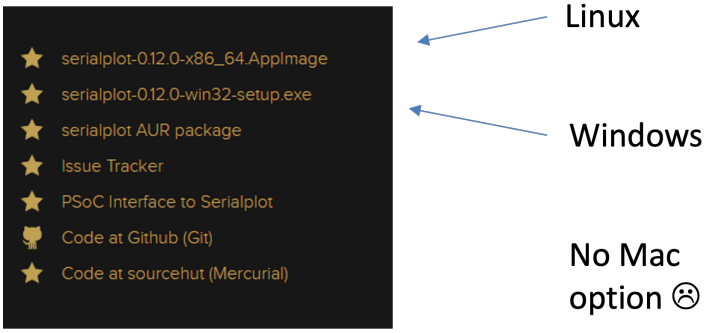


Figure 16: Illustration of the lack of a Mac Version of the Serial Plot software

4.2.12.1 Graph Generation

Of course, the most important part of the serial plotter in this case is its ability to generate a graph of recent data. The serial plotter did not need to save data, automatically generate new channels, etc, as the channels could be hard-coded, and all the data was already being saved to both a .xlsx file [see Section 4.2.11] and a .pkl file [see Section 4.2.13.1].

Thankfully, Matplotlib takes care of most of the complex graphical work. The main challenge here was understanding how to effectively use Matplotlib to this end, and, of course, preventing errors, which unfortunately plagued the drop test [see Section 5.1].

Firstly, it had to be decided what would be graphed. The team decided that there would be 5 pages, each containing 3 plots, designated as seen in Table 41.

```
fig1, axs1 = plt.subplots(3, num='Acceleration Data')
    fig2, axs2 = plt.subplots(3, num='Environmental Data')
    fig3, axs3 = plt.subplots(3, num='Velocity Data')
    fig4, axs4 = plt.subplots(3, num='Force Data')
    fig5, axs5 = plt.subplots(3, num='Vertical Data')
```

Table 41: Plot windows setup code snippet

Details of what exactly these plots contained is generally self-explanatory, with acceleration, velocity, and force, simply containing the three axes gathered. However, for details on ‘Environmental Data’ and ‘Vertical Data’, see Figure 17.

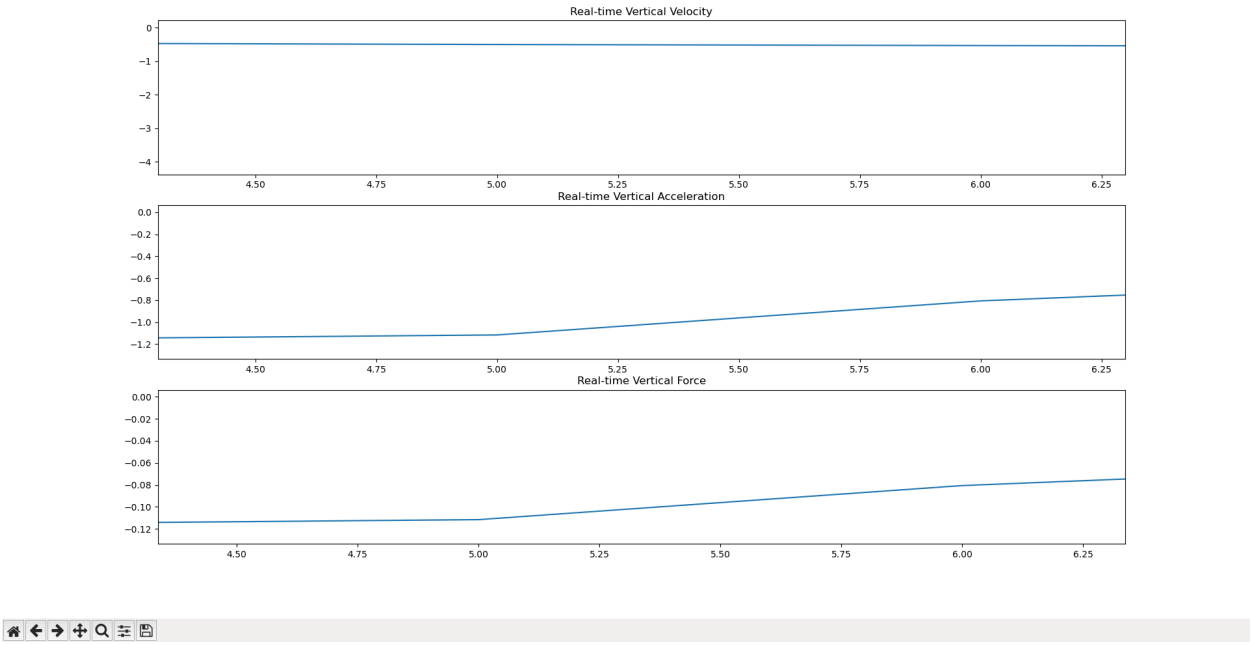


Figure 17: Examples on what “Environmental Data” [top] and “Vertical Data” [bottom] contain

Once the graph windows with their subplots had been created, next, data had to be added to the graphs. This was done every 50 milliseconds, as doing it faster caused some unreliability in testing.

For each of the graph windows, each of the subplots would be iterated through, giving them their titles and updating the data, ensuring the data on both axes was equal to prevent Matplotlib errors. Then at the end, the limits would be updated to give an animated effect described in Section 4.2.12.2. An example of this methodology can be seen in Table 42.

```
def animate_accel(i):
        try:
            for j, ax in enumerate(axs1):
                ax.clear()
                if j == 0:
                    equalized_accelX    =    equalize_before_plot(accelX,    "accelX")
#Ensures list lengths are equal to "realtime", which they are plotted against, to
avoid Matplotlib errors
                    ax.plot(range(len(realtime)), equalized_accelX, label='Accel X')
#Calls the Matplotlib plot() function
                    ax.set_title('Real-time Acceleration X')
                elif j == 1:
                    equalized_accelY = equalize_before_plot(accelY, "accelY")
                    ax.plot(range(len(realtime)), equalized_accelY, label='Accel Y')
                    ax.set_title('Real-time Acceleration Y')
                elif j == 2:
                    equalized_accelZ = equalize_before_plot(accelZ, "accelZ")
                    ax.plot(range(len(realtime)), equalized_accelZ, label='Accel Z')
                    ax.set_title('Real-time Acceleration Z')
                update_limits(ax, range(len(realtime)), realtime[-1], scroll_offset)
#Updates the axes to give an animated effect
        except Exception as e:
            print(f"Error in animate_accel: {e}") #Prevents the graphing from ceasing
should an error occur
```

Table 42: Example graph setup code

4.2.12.2 Animation

The animation of the graph was another important aspect to replicate Serial Plot's functionality. Simply using Matplotlib to graph everything would result in the receipt of a constantly growing summary graph, that would very quickly stop being able to accurately represent short-term changes [see Fig. 18], as opposed to the more real-time-oriented graphs demonstrated in Figure 19, which more closely mimics Serial Plot's methods. (Note: Matplotlib does have a zoom feature, but this doesn't work well with constantly updating plots).

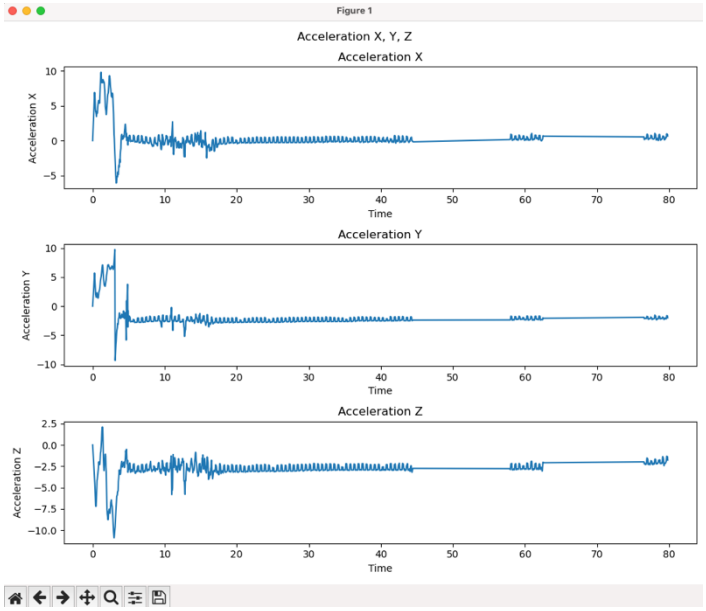


Figure 18: Example showcasing the need for a narrow-window animation

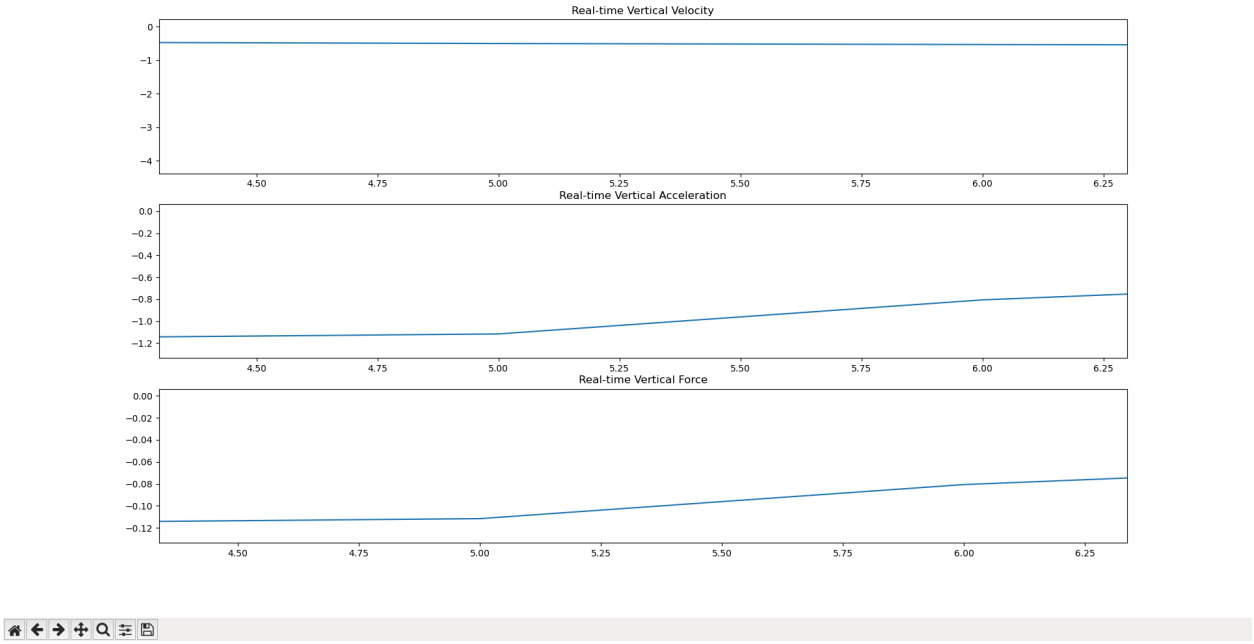


Figure 19: Narrow-window, animated graph example. Less information is visible here, but more relevant information.

To conduct the animation, a window size had to be decided, and then a limit placed on the x and y axes, to prevent Matplotlib from displaying anything outside these time limits [see Table]

```
def update_limits(ax, time_data, timecheck, window_size=60):  # Adjust window_size as needed
```

```
    if timecheck > 0:
        right_limit = timecheck
        left_limit = max(0, right_limit - window_size)
        # Ensure there's a minimal difference between left_limit and right_limit
        if left_limit == right_limit:
            left_limit -= 1  # Decrement left_limit
            right_limit += 1  # Increment right_limit to ensure they are not identical
        ax.set_xlim(left_limit, right_limit)
        ax.figure.canvas.draw()
```

Table 43: Matplotlib animation sample code

This function was called during each graphing call, to ensure the fluid motion of the graph.

4.2.13 Post-Graphing

The post-graphing processes that will be described here are what permitted the display of graphs in any sections where they appear, due to corruption of the folder containing original screenshots.

These processes also enabled overview graphs and left open the possibility of later data-analysis of things not explicitly saved. This was utilised in 4.2.7 to determine the average time taken for transmission, due to an error preventing this data being saved in Excel.

4.2.13.1 Pickling

As described in Section 4.2.1, the Python Pickle library allows the saving of Python variables after runtime. For this reason, every variable calculated or received was saved, in a list of lists, to a pickle file. This method is very simple, simply requiring the lists to be saved to be put into an enclosing list, be given a title, and be saved using the “pickle.dump()” function.

4.2.13.2 Graphing

Of course, the pickle files are what permitted the generation of previously viewed graphs, but before that functionality was considered, a function was created that would simply create the overview graphs originally not desired, in order to provide a better overall grasp of the journey taken by the Arduino.

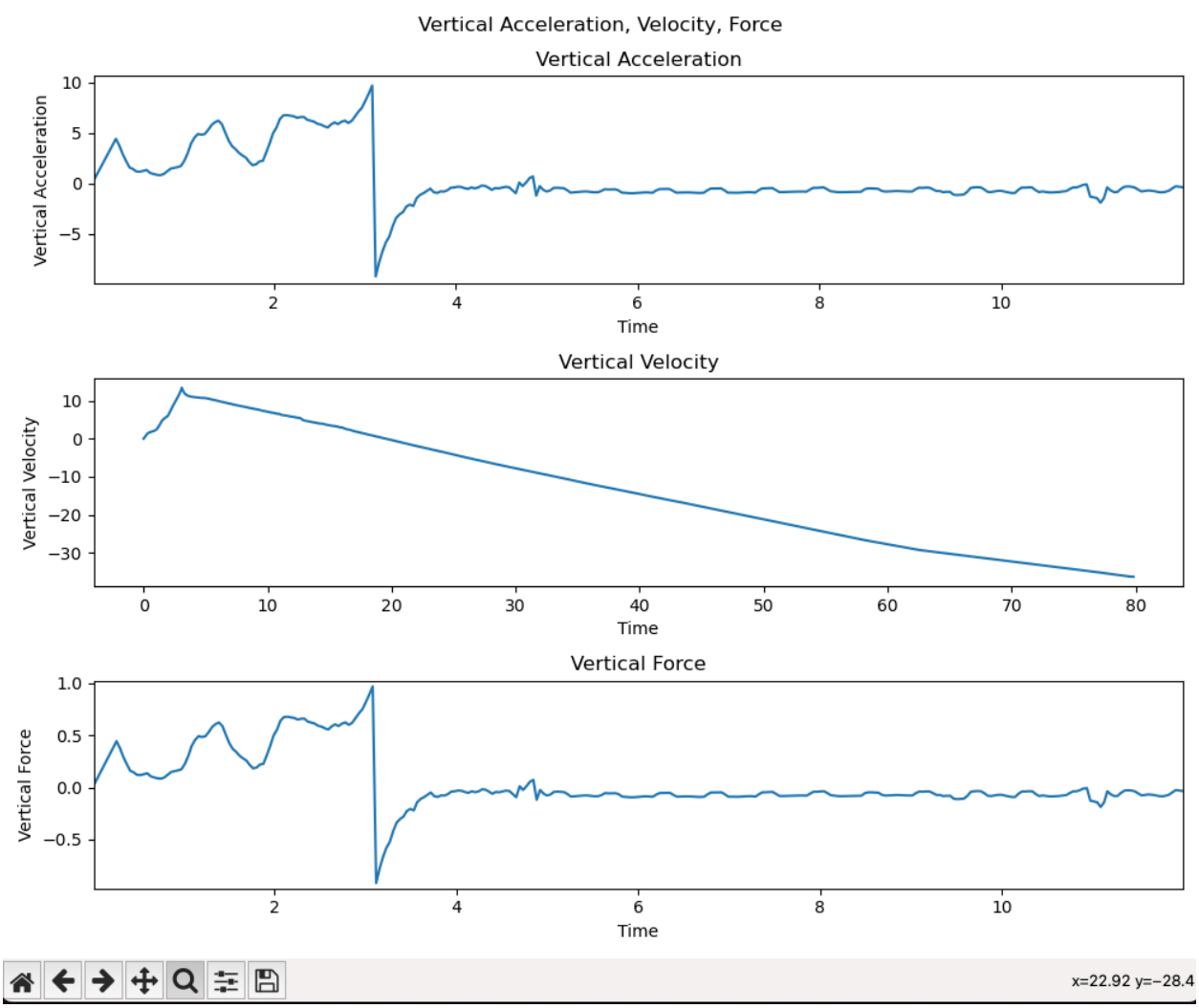


Figure 20: Summary graph for 1st official launch test, zoomed using Matplotlib's tools, to reach the point at which acceleration and force mostly occurred (not zoomed at velocity, as that is erroneously constantly changing)

An example of this graph can be seen in Figure 20. This data is from the 1st official launch test and in the acceleration and force sections, accurately displays the effects of launching and then the Arduino flipping, before coming to rest on its side [see Section 5.2]. The vertical velocity, however, hints at an issue in the code, whereby vertical velocity can still decrease if altitude is zero.

4.2.14 Live Commands

The final piece of the Python program is its ability to accept user commands during runtime.

This was added as a failsafe, primarily in case the mode changes didn't activate, but also in case certain elements of the function impeded the progress of others. Most of the commands went unused, however, the "print" command was used after the unsuccessful 2nd launch, to still permit the printing of data.

As the method by which these commands were accepted was a simple "while True" loop checking for input on an independent thread, it will not be shown in this section, though it can be seen in Appendix A1.2.1.

4.2.15 Drop Test Setup

4.2.15.1 Code

The code, both on the Arduino and the Laptop were very similar to the finished products on the day of the drop test [see Section 5.1 for drop-test details]. In terms of calculation, the only things omitted were the mode-checking, tilt, and averages [see Sections 4.2.9.2, 4.2.9.7, 4.2.10].

Unfortunately, one major difference was seen in error-handling, specifically in the graphing-section. Due to this, small errors in the data transmitted and incidental desyncs caused the graphing-system to crash, which unfortunately stopped the entire Python program.

As such, little meaningful data was gathered in the drop test, although a stronger resolve arose to create a more robust system throughout the code. A resolve which resulted in the finished product working well, in spite of hardware issues [see Section 5.2].

4.2.15.2 Sampling Rate

For the drop test, no artificial sampling rates were imposed. The cap for the speed of data obstinance was simply the rate at which data could be transmitted. This was dependent on the speed of the code and transmission, which, at the time of the drop-test, due to the float-based transmission system then in effect [see Section 4.2.7.3], was approximately a sample once every 0.0567s. This equates to a sampling rate of approximately 17.63Hz.

As well as this, there are certain physical limitations in sensors which could reduce sampling rates further, such as the speed of the Arduino's analogue-to-digital converters, the speed at which the Arduino can switch between different analogue signals, the physical response time of the sensor. However, in practice, these effects are much smaller than the impact of the transmission time and likely even the speed at which the code runs.

4.2.16 Main Test

The code used in the main test utilised everything described in Section 4.2, the full code of which can be found in Appendix A1.

4.2.16.1.1 Main Test

In the main test, there were also no artificial sampling rates. In this case, due to the improved, custom binary-encoding method, the effective sampling rate was 61.73Hz.

5 Evaluation Tests

5.1 Drop Test

5.1.1 Objective

This test was run to ensure the parachute worked with capsule. It also allowed for the testing of the code and Arduino. This test made sure that the code was written properly, and correct readings were being obtained.

5.1.2 Method

- A 5m pole was set up. This pole had a pulley system to hoist the parachute up.
- The parachute and capsule with Arduino were clipped to the pulley system.
- The parachute system was then hoisted up to the top of the pole.
- The parachute was released and fell to the ground.
- Readings were taken with the Arduino.

5.1.3 Results

Unfortunately, due to errors in the data-processing code, the only data obtained was before the actual drop. This data was extremely limited and therefore discarded.

5.1.4 Discussion

The parachute slowed the capsule down. Due to time constraints the test was only preformed once. Unfortunately, due to errors caused by issues in the data received by the Python code, the data gathered was extremely limited and not useful.

5.1.5 Conclusion

This test showed that the parachute worked well and that further error handling had to be added to the Python code.

5.2 Rocket Launch Test

5.2.1 Objective

The objective of this test was to gather data with the Arduino about how the rocket flew. The rocket should fly straight up. The capsule and parachute should then deploy. The Arduino should calculate the acceleration, pressure and temperature it experiences during the flight.

5.2.2 Method

- Fill water bottle rocket with water.
- Insert cork into the opening of the bottle.
- Place bottle rocket on the stand.
- Attach pump by putting needle inside of the cork in the bottle.
- Place the capsule with the Arduino and parachute on top of the rocket.
- Start pumping air into the rocket using the pump.
- When the rocket reaches full pressurisation, it will fly into the air.

- Using a computer gather the data collected by the Arduino.

5.2.3 Results

The rocket flew sideways instead of straight up as intended. This caused a very short flight. There was also not enough air trapped by the parachute. Therefore, the parachute did not open to slow the descent of the capsule.

Despite this, the code worked, thanks to its failsafes, and was able to provide a summary of the launch, which can be seen in table ?, which shows the raw terminal output. To view the code that provided this, see the "printData()" function in Appendix A1.2.1.

```
Max Acceleration X:   9.782463073730469
Max Acceleration Y:   9.71680842101574
Max Acceleration Z:   2.110269546508789
Min Acceleration X:   -6.086047172546387
Min Acceleration Y:   -9.344438908100129
Min Acceleration Z:   -10.893394470214844
Average Acceleration X:   0.28927652844094
Average Acceleration Y:  -1.969066529571769
Average Acceleration Z:  -2.92325482536786
Maximum Vertical Acceleration:   9.71680842101574
Minimum Vertical Acceleration:   9.71680842101574
Average Vertical Acceleration:   -0.42896355797000035
Max Velocity X:  19.02224913454056
Max Velocity Y:  14.39580235083748
Max Velocity Z:  5.032728522837164
Min Velocity X:  0 // It is unknown what caused this, as the graphs suggest this should have been ~-80, which is also incorrect, but would have been more interesting
Min Velocity Y:  0
Min Velocity Z:  0
Average Velocity X: 15.018264451966315
Average Velocity Y: 14.305827702292305
Average Velocity Z: 14.30311700498527
Maximum Vertical Velocity:   13.396214050746492
Minimum Vertical Velocity:   -27.78487245541345
Average Vertical Velocity:   -3.612004272300895
Max Pressure:   967.6016845703125
Min Pressure:   0 // This was caused by initialising Pressure to 0 at the start
Average Pressure:   966.7947407957996
Max Temperature:   17.43000030517578
```

```
Min Temperature:   0 // This was caused by initialising Temperature to 0 at the start
Average Temperature:   966.7947407957996
Max Altitude (Environmental):   0.021497259709150732
Max Altitude (Accelerometer):   12.32003810305
Min Altitude (Environmental):   -0.02379501210786771
Min Altitude (Accelerometer):   0
Average Altitude (Environmental):   0.0007529734665711697
Average Altitude (Accelerometer):   6.6294570563264
Max Force X:   0.978246307373047
Min Force X:   -0.6086047172546387
Average Force X:   0.027755631902396515
Max Force Y:   1.9526808421015742
Min Force Y:   -0.14243703174591069
Average Force Y:  0.2808498287084614
Max Force Z:   0.21102695465087892
Min Force Z:   -1.0893394470214843
Average Force Z:  -0.21136325572123413
Maximum Vertical Force:   0.971680842101574
Minimum Vertical Force:   -0.919287527598282
Average Vertical Force:   -0.042896355797000046
Max Tilt X:   1.495851809453058
Min Tilt X:   -0.6692426331604691
Average Tilt X:   0.03470622796499617
Max Tilt Z:   0.21680868711632983
Min Tilt Z:   -1.5707963267948966
Avereage Tilt Z:   -0.308562375743586
Max Absolute Tilt:   1.5707963267948966
Min Absolute Tilt:   0
Average Absolute Tilt:
Launch Time: 27.5439
Descent Began:   0
Landing Time:   47.2895
Final Time:   59.969
Average Transmission Time:   0.01859724473257698
```

5.2.4 Discussion

The rocket flew sideways due to the stand being unstable. The stand was built of weak plastic that bent and deformed under the weight of the rocket on top. The cork was also too worn through. This caused water to leak from the bottle. This prevented the max amount of pressure to build up in the bottle. This caused the rocket to experience less propulsion. Therefore, the rocket did not take off properly. This means the rocket under preformed.

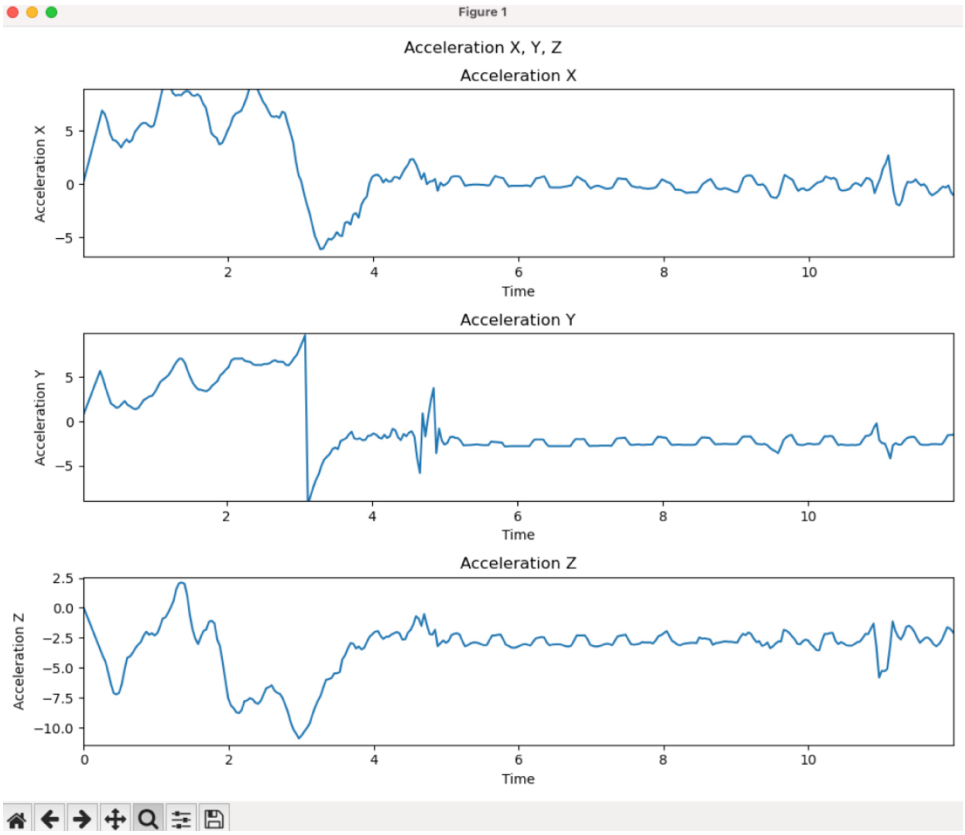
5.2.5 Conclusion

If the rocket had a stronger stand which kept it up and prevented it from falling, the worn cork was replaced by a new cork before the test to prevent pressure from being released from the rocket and switching to a foot pump as it might pump more pressure at a faster rate into the rocket. All these aspects could contribute to the rocket performing better on the day of the test.

6 Data Analysis

6.1 Analysis of Sensor Data

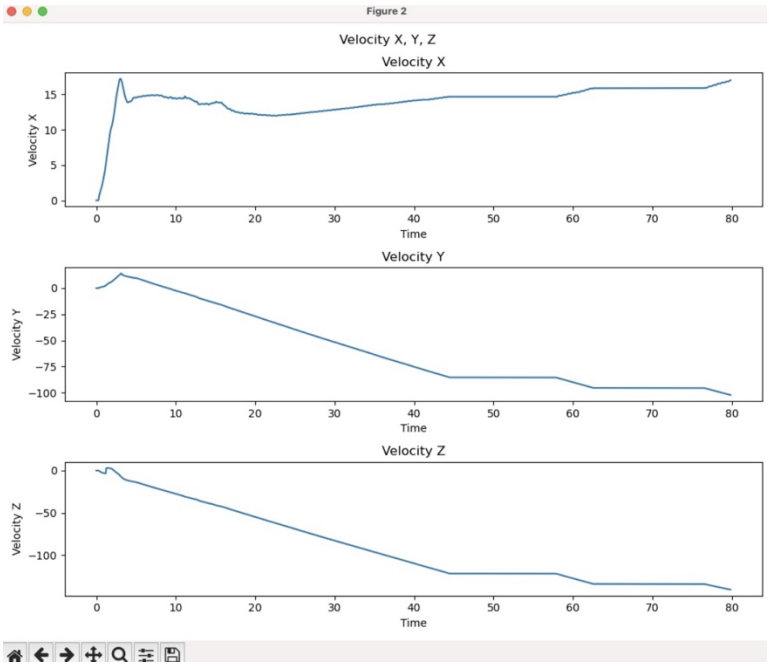
The live sensor data plots below show the acceleration of the rocket throughout the flight. There are many peaks in the graphs which were caused by a failed launch and the poor weather conditions, resulting in severe rotation of the capsule, before it settled down at around 5 seconds [see the data stabilisation in Fig 6.1]. The failed launch meant that the rocket did not reach its intended apex or go in a straight path which led to the Arduino receiving scattered data. The strong wind on the day of the launch further interfered with the flight path and caused an irregular reading leading to an uneven graph.



(Fig 6.1)

6.2 Analysis of Sensor Data

The live sensor data plots below show the velocity of the rocket throughout the flight. It is quite clear by the graph that these values are inaccurate. This is due to the unexpected flight path of the rocket. The lander did not go very high and moved much more laterally than intended. It also landed sideways which resulted in the velocity values being inaccurate after roughly 4 seconds. The uneven velocities whilst descending also suggest that the parachute did not deploy as this would have caused the velocity to be very stable.



(Fig 6.2)

6.3 Methodology for calculations

The methodology for calculating the minimum and maximum launch acceleration and velocity were calculated at the end of the flight, using Python's built-in "min()", "max()", "sum()" and "len()" functions, as can be seen in Table 6.3.

```
    print("Max Acceleration X:  ", max(accelX))
    print("Min Acceleration X:  ", min(accelX))
    print("Average Acceleration X:  ", sum(accelX)/len(accelX))
```

Table 44: Sample code showing the methodology behind the calculation of maxima, minima, and averages, as explained in Section 4.2.9.7. Full code can be viewed in the "printData()" function in Appendix A1.2.1

The methodology for other calculations is discussed in Section 4.2.9, and the code can be found in Appendix A1.2.1, under accurately named functions.

7 Conclusion

Unfortunately, on the day of the test the rocket underperformed. For the rocket to perform better a few adjustments need to be made such as changing the cork, the cork was worn and there was no spare cork to replace it, so the cork was taped to make it thicker to help to reduce the amount of water and pressure that was released. If the cork worked optimally, it would have contributed to the rocket reaching the height that was desired. Another adjustment that is crucial is to make a stronger stand for the rocket, the stand was made from plastic and worked for test that were carried out on a calm day but due to the wind on the day of the testing it was knocked several times and the weight of the rocket falling on it caused it to bend and loose its shape, with a stronger and more elevated stand the rocket could have performed better. The parachute and capsule did not leave the rocket sometimes due to the capsule holder being too long which made it stuck inside the holder, If the walls of the capsule holder were smaller it would make it easier for the capsule to be launched into the air. Lastly switching pumps might have helped with the rocket reaching its optimal height, the pump used was a pneumatic bicycle stirrup pump, stirrup pumps do distribute the work evenly between both arms, but they are still tiring to use especially when you are using it for over a minute which causes you to slow down leading to less pressure been forced into the rocket, If a foot pump was used instead it might have worked more efficiently and allow more pressure to be entered into rocket in a shorter length of time. With a better rocket launch the Arduino would have provided with better results.

8 Individual Contributions and Project Management

8.1.1 Nelly Frohburg

Design parachute from research online and equations. Gathered materials for the parachute. Constructed the parachute in line with the design made. Ran test to find best length of strings and to prove the parachute functioned. Gathered data from this test and constructed a graph using excel. Altered the parachute design based on the results. Wrote out sections 3.2-4 in relation to how the parachute was constructed. Wrote out section 5 based on the official tests that were ran during the project. Attended all scheduled labs. Discussed how the lander should function. Solved problems that arose spontaneously such as how to construct the base of the rocket so that it would hold upright.

8.1.2 Kento O Connor

Worked on designing and constructing the rocket. This included adjusting fin sizes, securing the fins to the rocket and modified the cork to fit securely into the nozzle. Performed multiple tests of launching the rocket at home. Kento attended all scheduled lab meetings and met up with teammates outside of lab hours. Completed the rocket part of the report with Caoilinn and the data analysis section of the report.

8.1.3 Caoilinn O Brien

Worked with my teammates and discussed with them the design of the rocket. Worked with Tadhg to create a holder for the capsule. After a lot of research constructed the rocket and launch mechanism. Carried out several tests of the rocket before attaching the capsule and parachute to it and recorded it for the report. Tested the rocket with the parachute and capsule and worked with Nelly to construct a stronger base to hold up the rocket when the original base broke on the day of testing. Contributed to the weekly reports. Attended all scheduled labs and meetings arranged outside of lab hours. Wrote section 2, 5.2.6 and 7 of the report.

8.1.4 Tadhg Scanlon

My primary task for the project was to construct the capsule to hold the Arduino kit. I researched different ideas and materials before starting but ultimately concluded that a 750ml water bottle would be the best option. I set about getting the bottle and constructing the capsule as soon as possible as it was required for the drop test. I sourced tissue paper to act as padding and bought strong industrial tape to cover the top of the capsule and to attach the parachute. On the day of the drop test I aligned the Arduino in an upright position and helped with attaching the parachute to the capsule. I had the idea of testing the parachute and capsule before the drop test itself which proved beneficial as it was observed that one string wasn't attached right. I also proposed the solution on how to hold the capsule on top of the rocket while being launched. I attached a cylinder of plastic to the top of the rocket that the capsule could sit into. On the final testing day, I believe I took on a primary leadership role. The team first encountered problems with the rocket stand. First, they taped cardboard to the base of the rocket, but this wasn't strong enough. I sourced a flat piece of aluminium and taped the rocket stand to this, providing a strong and

secure base. I was the launch operator on the day which involved using a pump to create pressure. Throughout, I added as much as possible into the weekly report, sent texts into our WhatsApp group chat to organise meet up times, completed the Introduction and Capsule part of this report and continuously checked for and edited mistakes in the report.

8.1.5 Eoghan Collins

- Attended all scheduled labs, created all shared documents, submitted all weekly reports
- Created the WhatsApp groupchat for communication
- Managed which tasks were given to which people during the project
- Devised the mechanics and maths behind the Arduino code
- Programmed the Arduino, including data obtention, data processing, calibration, mode-switching, data-packaging, and transmission
- Devised the mechanics and maths behind the Python code
- Wrote the Python code for data decoding, calculation, data analysis, data storage, serial plotting, user-commands, calibration, and any other code
- Tested the Arduino and Python code extensively, adding debugging and error handling throughout the process
- Sourced the aluminium balancing base used for the rocket, designing for its use
- Recorded test launches and drop-test
- Wrote Section 4
- Provided all the graphics and citations for Section 4
- Fixed report formatting, to use proper headings, fonts, etc. although others' subsequent changes resulted in formatting errors
- Created the video, after sourcing all media, excluding some recordings provided by teammates
- Provided the data for Section 5.2.3
- Provided all graphs for Section 6

8.2 Individual Reflections

8.2.1 Nelly Frohburg

During this project I learnt how important communication is. It was difficult to properly explain what I wanted to do for my part of the project. But I found when I took the time to explain it the project went a lot smoother. It was difficult to keep the sections of the project separate as we had to build one entity. As a group we did not really think about how each part would work together. I think this made for some hectic and last-minute adjustments that needed to be made. To avoid this in the future I think it is wise to attempt to build parts together so everyone will know how they work. This might also lead to better ideas during building.

During the project I learnt to understand code more. I learnt how much can be possible with code that is quite simple. But also, with small modifications how complex the code can get. This gave me greater knowledge into electrical engineering. I learnt how parachute work. How they are deployed. I learnt how much they can slow objects down. I learnt how to build pressure rockets out of recycled materials. This gave me greater knowledge into mechanical engineering.

8.2.2 Kento O Connor

The most valuable lesson I have learned from this project is the importance of communication. The project felt quite isolated at times, and this led to some unorganised parts. We did not work as a team early on and my role clashed with another teammate. The project helped me to learn that it is important to take initiative and reach out if there is anything you are unsure with. However, as the deadline grew closer, we began to work more as a team, and this led to improved results. The clashed role ended up being a blessing as we learned from our mistakes and took the best part of our designs to create a much-improved rocket. Overall, I enjoyed this project as it gave me an insight to the real world where engineers from different fields come together to create a final product.

8.2.3 Caoilinn O Brien

This project helped me to learn how important it is to communicate and cooperate with your teammates. As a team we were extremely lucky that we all got along whenever we meet but we still isolated ourselves from each other. We all got consumed by our roles and did not work together until the very last minute. The one thing I will take from this it is crucial to communicate with your teammates and do not be afraid to ask for help when you are stuck on something as your teammates want to help you.

I also learned that it is important to be organized. When constructing the rocket, I never took into consideration the weather and over testing. Due to intense winds on the day of testing the rocket stand broke and after the many tests carried out the cork was worn, and I was not organized enough to have a spare cork. As a team we should have arranged to meet and test the rocket earlier instead of the day of testing.

8.2.4 Tadhg Scanlon

Despite the various challenges and adversities faced throughout this project, I would say that I enjoyed it overall. I can confidently say that I gained valuable experience to develop my team working skills and organisation skills on top of the greater understanding that I now have on Arduino coding and computer systems. I also enjoyed learning about the physics behind the parachute falling through the air and the different factors that must be considered. On the test day I was quite disappointed with the height that the rocket was launched to. We were lucky that the rocket even got off the ground so that data was obtained. If I were to do it again, I would do more tests with launching the rocket. Overall, I enjoyed the practical element that this project offers along with the refreshing idea of making a video which adds to the excitement. I believe that we worked well as a team, communicated effectively, assigned jobs evenly and reached deadlines.

8.2.5 Eoghan Collins

I deeply enjoyed this project as my role as chief programmer allowed me to experience electronic and computer engineering, after a fashion, which I deeply enjoyed, and has filled me with confidence for my course choices.

I learned a lot about software-hardware interactions, Python, Arduino, and how to research for programming. I believe that all of these skills will stand to me in the future.

One issue that was faced in this project was a disconnect between different aspects of the project and the work done for them. I feel that as I had the programming for the project to keep me busy, I didn't interact with the other facets of the project as deeply as I could have, although the programming was undoubtedly crucial. I believe that the project could have benefitted from more working alongside each other, at least for the hardware sections, so that the different components could have meshed more easily, and perhaps the rocket could have been tested earlier, or possibly a back-up rocket could have been constructed, so that the final launch might not have been quite so underwhelming.

All that said, however, I am very proud with how I and my team performed and conducted ourselves during this project, and I feel that together we have embodied the principles of engineering in this report.

9 References

[1] https://www.npl.co.uk/skills-learning/outreach/water-rockets/wr_booklet_print.pdf

[2] Water-rockets science for hobbyist, students, and teachers of …

[3] Strickland JH, Higuchi H. Parachute aerodynamics - An assessment of prediction capability. Journal of Aircraft. 1996 Mar;33(2):241–52.

[4] Robert Rozee, 2016, HC-12 Wireless Serial Port Communication Module User Manual, Version 2.3A, Universidad Politécnica de Madrid, pp. 2

[5] Simon Haykin, "Communication Systems," 5th Edition, Wiley

[6] Silicon Labs, "Si4463 Radio IC Datasheet.", pp. 24

[7] Oracle Corporation. "Data Types and Sizes." Oracle.com, 2010, docs.oracle.com/cd/E19253-01/817-6223/chp-typeopexpr-2/index.html.

[8] George, Roshan, and Delaney, Ethan, Galway Lander – Lecture 2 Sensors & Rocket Design, 2024, University of Galway, pp. 33

[9] Robert Rozee, 2016, HC-12 Wireless Serial Port Communication Module User Manual, Version 2.3A, Universidad Politécnica de Madrid

[10] S. Gilli, Alexander. "What Is MEMS (Micro-Electromechanical System)? Definition from WhatIs.com." IoT Agenda, May 2019, www.techtarget.com/iotagenda/definition/micro-electromechanical-systems-MEMS#:~:text=A%20MEMS%20(micro%2Delectromechanical%20system.

[11] Analog Devices, 2009, ADXL335 Manual

[12] Apple Inc. "Diagnosing Memory, Thread, and Crash Issues Early." Apple Developer Documentation, 2024, developer.apple.com/documentation/xcode/diagnosing-memory-thread-and-crash-issues-early. Accessed 5 Apr. 2024.

[13] Arduino. “Arduino Reference.” Arduino.cc, 2019, www.arduino.cc/reference/en/language/functions/analog-io/analogread/.

[14] Swedish Meteorological and Hydrological Institute . “Air Pressure and Sea Level | SMHI.” Www.smhi.se, Published 10 Aug. 2010, Updated 23 Apr. 2014, www.smhi.se/en/theme/air-pressure-and-sea-level-1.12266#:~:text=The%20average%20sea%20level%20during. Accessed 5 Apr. 2024.

[15] Hughes, Mark. “Understanding and Implementing the HC-12 Wireless Transceiver Module - Projects.” Www.allaboutcircuits.com, 2 Nov. 2016, www.allaboutcircuits.com/projects/understanding-and-implementing-the-hc-12-wireless-transceiver-module/.

[16] George, Roshan, and Delaney, Ethan, “Full Lander Program Template”, University of Galway, 2024

[17] "Data recording method". Google Patents. 8 August 1983.

[18] Huffman, Ken (1991). "Profile: David A. Huffman: Encoding the "Neatness" of Ones and Zeroes". Scientific American: 54–58.

[19] Yavuz Özderya, Hasan . “SerialPlot - Realtime Plotting Software.” Hackaday.io, hackaday.io/project/5334-serialplot-realtime-plotting-software. Accessed 5 Apr. 2024.

[20] Büsching, F.; Kulau, U.; Gietzelt, M.; Wolf, L. Comparison and Validation of Capacitive Accelerometers for Health Care Applications. Comput. Methods Programs Biomed. 2012, 106, 79–88.

[21] "PTC thermistor vs. NTC thermistor for measuring the temperature of a liquid". Electrical Engineering Stack Exchange.

[22] "Hypsometric equation - AMS Glossary". American Meteorological Society.

10 Appendices

A1 Code

A1.1 Arduino Code

Were the team reusing the code, they would compress the variable declarations, use header files for some functions, and create structs to house the binary datatypes.

```
// FOR CALIBRATION: PLACE IT FLAT – 1G, PLACE IT FLAT UPSIDE DOWN – –1G

//=============================================================================
// Import necessary libraries
//=============================================================================

#include <SoftwareSerial.h>  //Permits the use of digital pins as Serial Pins
#include <Wire.h>  //Permits interaction with devices using the I2C protocol. Used for the BMP280 Environmental. Provides functions used such as:
                   //begin(), write(), available(), read()
#include <Adafruit_BMP280.h>  //Library specifically for the BMP280 Environmental Sensor, allowing easy interaction
#include <Adafruit_Sensor.h>  //Supports the Adafruit_BMP280 library and allows standard interaction with Adafruit Sensors
#include  <SPI.h>   //Not used for any particular device, but maintained for compatibility
#include <math.h>  //Used for advanced math operations – Unused, but kept for flexibility

//=============================================================================
// Constants – These never change
//=============================================================================

// N.B. Every I2C device has a unique address/chip ID
#define I2CADDR (0x76)
#define CHIPID (0x60)  // 0x60 for BME and 0x58 for BMP.

// Moving Average Filter / Buffer / Track values
#define WINDOW_SIZE 5

const int ForceMode =
    0;  // Set to 1 to force the arduino to use a particular mode
```

```
const int ForceModeMode =
    3;  // Change number to set the mode that is being forced

#define GROUND_PRESSURE (1013.25)
float g = 9.81;

//============================================================================
// Initialising variables and peripherals
//============================================================================

// HC-12 Wireless Transceiver
SoftwareSerial HC12(7, 6);  // HC-12 TX Pin, HC-12 RX Pin

// Bosch Environmental Sensor (BME/BMP280)
Adafruit_BMP280 env_sensor;  // use I2C interface
Adafruit_Sensor *env_temp = env_sensor.getTemperatureSensor();
Adafruit_Sensor *env_pressure = env_sensor.getPressureSensor();

// time variables
unsigned long time_start = 0, time_prev = 0, time_diff = 0,
              pre_launch_last_transmit_time = 0, landed_last_transmit_time = 0;

// mode variable
int prev_mode = 0, mode = 0;

float floatArray[7] = {0};

// Custom Variables - Added by Student
float AccelBufferX[WINDOW_SIZE] = {0};
float AccelBufferY[WINDOW_SIZE] = {0};
float AccelBufferZ[WINDOW_SIZE] = {0};
int currentBookmarkA = 0;
float tempBuffer[WINDOW_SIZE] = {0};
float pressBuffer[WINDOW_SIZE] = {0};
int currentBookmarkE = 0;
```

```
float calibPosX = 981;
float calibNegX = -981;
float calib0X = 0;
float calibPosY = 981;
float calibNegY = -981;
float calib0Y = 0;
float calibPosZ = 981;
float calibNegZ = -981;
float calib0Z = 0;

float oldCalibPosX = 410;
float oldCalibNegX = 274;
float oldCalib0X = 335;
float oldCalibPosY = 404;
float oldCalibNegY = 268;
float oldCalib0Y = 341;
float oldCalibPosZ = 408;
float oldCalibNegZ = 272;
float oldCalib0Z = 339;

float slopeNegX = 0;
float slopePosX = 0;
float slopeNegY = 0;
float slopePosY = 0;
float slopeNegZ = 0;
float slopePosZ = 0;

String inputString = "";

int calibNum = 1;
int beginner = 0;
unsigned long subtractTime = 0;
int checkCount = 0;
int launchTime = 0;

float AccelNowX = 0;
```

```
float AccelNowY = 0;
float AccelNowZ = 0;
float AccelNowAbsolute = 0;

float SumX = 0;
float SumY = 0;
float SumZ = 0;
float sumPress = 0;
float sumTemp = 0;

float TempNow = 0;
float PressNow = 0;
float pastPress = 0;

uint16_t timeChange = 0xFFF;
bool modeChange = false;
uint16_t rawAy = 0x3FF;      // 10 bits
int8_t rawAx = 0xFF;         // 8 bits
int8_t rawAz = 0xFF;         // 8 bits
int16_t PressDiff = 0x3FFF;  // 14 bits
int16_t tempChange = 0x3FF;  // 10 bits
uint16_t terminator =
    0xFFFF;  // 8 bits, could be used as a packet end or separator
float initialTemp = -2000;

float dataToSend[9] = {0.00};
int dataToSendSize = 7;

int landable = 0;

void setup() {
  // Code has started, start timestamp.
  time_start = millis();

  // Setup serial connections to the HC-12/ PC
  // Start serial streams between Arduino and HC-12/ PC
```

```
  HC12.begin(9600);
  Serial.begin(9600);

  // Give console time to get running
  while (!Serial) {
    Serial.println(F("Waiting for serial connection..."));
  }

  // Setup for the Environmental Sensor
  if (!env_sensor.begin(I2CADDR, CHIPID)) {
    Serial.println(
        F("Could not find a valid BME/BMP280 (Environmental) sensor, check "
          "CHIPID/libraries/wiring!"));
    while (1) delay(10);
  }

  // Default Adafruit_BMP280 settings from datasheet/
  env_sensor.setSampling(
      Adafruit_BMP280::MODE_NORMAL,     /* Operating Mode. */
      Adafruit_BMP280::SAMPLING_X2,     /* Temp. oversampling */
      Adafruit_BMP280::SAMPLING_X16,    /* Pressure oversampling */
      Adafruit_BMP280::FILTER_X16,      /* Filtering. */
      Adafruit_BMP280::STANDBY_MS_500); /* Standby time. */

  // initialise times
  pre_launch_last_transmit_time = time_start;
  landed_last_transmit_time = time_start;
}

float mapAccelX(float rawX) {
  float realAcceleration;
  if (rawX >= calib0X) {
    // If the rawValue is between 0 and g, use the slope for that segment
    realAcceleration = slopePosX * (rawX - calib0X);
  } else {
    // If the rawValue is between -g and 0, use the slope for that segment
```

```
    realAcceleration = slopeNegX * (rawX - calib0X);
  }

  return realAcceleration;
}

float mapAccelY(float rawY) {
  float realAcceleration;

  if (rawY >= calib0Y) {
    // If the rawValue is between 0 and g, use the slope for that segment
    realAcceleration = slopePosY * (rawY - calib0Y);

  } else {
    // If the rawValue is between -g and 0, use the slope for that segment
    realAcceleration = slopeNegY * (rawY - calib0Y);
  }
  return realAcceleration;
}

float mapAccelZ(float rawZ) {
  float realAcceleration;

  if (rawZ >= calib0Z) {
    // If the rawValue is between 0 and g, use the slope for that segment
    realAcceleration = slopePosZ * (rawZ - calib0Z);
  } else {
    // If the rawValue is between -g and 0, use the slope for that segment
    realAcceleration = slopeNegZ * (rawZ - calib0Z);
  }
  return realAcceleration;
}
void smoothAccelReading(
    float *a_x /*Pointer to the input variable, which allows it to be changed
                 without needing to return a value or use global variables*/
    ,
```

```
    float *a_y, float *a_z) {
  float SumX = 0;
  float SumY = 0;
  float SumZ = 0;

  AccelBufferX[currentBookmarkA] =
      *a_x;  // Updates one of the data values to the new value
  AccelBufferY[currentBookmarkA] = *a_y;
  AccelBufferZ[currentBookmarkA] = *a_z;
  currentBookmarkA++;  // Cycles which value is getting updated

  if (currentBookmarkA == WINDOW_SIZE) {
    currentBookmarkA = 0;
  }
  for (int i = 0; i < WINDOW_SIZE; i++) {  // Gets the sum of the buffers
    SumX += AccelBufferX[i];
    SumY += AccelBufferY[i];
    SumZ += AccelBufferZ[i];
  }
  *a_x = SumX / WINDOW_SIZE;  // Performs the averaging calculation
  *a_y = SumY / WINDOW_SIZE;
  *a_z = SumZ / WINDOW_SIZE;
}

void smoothEnvReading(float *temp, float *press) {
  float sumTemp = 0;
  float sumPress = 0;
  tempBuffer[currentBookmarkE] = *temp;
  pressBuffer[currentBookmarkE] = *press;
  currentBookmarkE++;
  if (currentBookmarkE == WINDOW_SIZE) {
    currentBookmarkE = 0;
  }

  for (int i = 0; i < WINDOW_SIZE; i++) {
    sumTemp += tempBuffer[i];
```

```
    sumPress += pressBuffer[i];
  }

  *temp = sumTemp / WINDOW_SIZE;
  *press = sumPress / WINDOW_SIZE;
}

void readFromAccelerometer() {
  short Xraw = analogRead(A1);
  short Yraw = analogRead(A2);
  short Zraw = analogRead(A3);

  // Used for transmission
  rawAx = (int8_t)(Xraw - calib0X);
  rawAy = (uint16_t)(Yraw);
  rawAz = (int8_t)(Zraw - calib0Z);

  // Used for mode calculation
  AccelNowX = mapAccelX((float)Xraw);
  AccelNowY = mapAccelY((float)Yraw);
  AccelNowZ = mapAccelZ((float)Zraw);
  smoothAccelReading(&AccelNowX, &AccelNowY, &AccelNowZ);
}

void readEnvironmental() {
  // Read Environmental Sensor
  sensors_event_t temp_event, pressure_event;

  env_temp->getEvent(&temp_event);
  env_pressure->getEvent(&pressure_event);

  if (initialTemp == -2000) {
    initialTemp = temp_event.temperature;
  }

  // Store value into variables: temperature (Celcius), pressure (hPa)
```

```
  TempNow = temp_event.temperature;
  PressNow = pressure_event.pressure;
  smoothEnvReading(&TempNow, &PressNow);
  tempChange = (int16_t)((TempNow - initialTemp) * 1000);
  PressDiff = (int16_t)((pressure_event.pressure - GROUND_PRESSURE) * 1000);

  if ((PressNow - pastPress) < 0.05) {
    landable =
        1;  // This variable decides whether the mode can be set to "LANDED"
  } else {
    landable = 0;
  }
  pastPress = PressNow;
}

int detectMode(float acceleration) {
  if (!ForceMode /*Only activate the function if we're not currently forcing a
specific mode for testing*/) {
    // MODE CODES:
    // 0 - UNASSIGNED
    // 1 - PRE-LAUNCH
    // 2 - ASCENDING
    // 3 - DESCENDING
    // 4 - LANDED
    if (mode == 0 && acceleration < (g * 1.1)) {
      mode = 1;
    } else if (mode == 1 && acceleration > (g * 1.2)) {
      mode = 2;
      prev_mode = 1;
      modeChange = true;
      launchTime = millis()-subtractTime;
    } else if (mode == 2 && acceleration < (g * 0.8)) {
      mode = 3;
      prev_mode = 2;
      modeChange = true;
    } else if (mode == 3 && acceleration > (g * 0.95) && (landable == 1)) {
      mode = 4;
```

```
      prev_mode = 3;
      modeChange = true;
    } else if (mode != 1 && (millis()-launchTime)>20000) {
      mode = 4;
      prev_mode = 3;
      modeChange = true;

    // else {HC12.println("No Mode Change");}
  }

  else {
    mode = ForceModeMode;
  }
}

void transmit(String dataSending) {
  // Send to HC-12 for wireless transmission
  HC12.println(dataSending);
}

void provideSlopes() {
  slopeNegX = g / (calib0X - calibNegX);
  slopePosX = g / (calibPosX - calib0X);

  slopeNegY = g / (calib0Z - calibNegY);
  slopePosY = g / (calibPosZ - calib0Y);

  slopeNegZ = g / (calib0Z - calibNegZ);
  slopePosZ = g / (calibPosZ - calib0Z);
}

void Calibrate(int calibMode) {
  float xValue = analogRead(A1);
  float yValue = analogRead(A2);
  float zValue = analogRead(A3);
```

```
  if (calibMode == 1) {
    calibPosZ = zValue;
    calib0X = xValue;
    calib0Y = yValue;
    transmit("C");
    Serial.println("Received");
  } else if (calibMode == 2) {
    calibNegZ = zValue;
    calib0X += xValue;
    calib0Y += yValue;
    transmit("C");
    Serial.println("Received");
  }
  if (calibMode == 3) {
    calibPosX = xValue;
    calib0Z = zValue;
    calib0Y += yValue;
    transmit("C");
    Serial.println("Received");
  } else if (calibMode == 4) {
    calibNegX = xValue;
    calib0Z += zValue;
    calib0Y += yValue;
    transmit("C");
    Serial.println("Received");
  } else if (calibMode == 5) {
    calibPosY = yValue;
    calib0X += xValue;
    calib0Z += zValue;
    transmit("C");
    Serial.println("Received");
  } else if (calibMode == 6) {
    calibNegY = yValue;
    calib0X += xValue;
    calib0Z += zValue;
    transmit("C");
```

```
    Serial.println("Received");
  } else if (calibMode == 7) {
    calib0X = calib0X / 4;
    calib0Y = calib0Y / 4;
    calib0Z = calib0Z / 4;
    provideSlopes();
    delay(100);

    Serial.println("Received");
    String transmitter = "CALIBVALUES";
    transmitter = transmitter + "," + slopePosX + "," + calib0X + "," +
                  slopeNegX + "," + slopePosY + "," + calib0Y + "," +
                  slopeNegY + "," + slopePosZ + "," + calib0Z + "," +
                  slopePosZ + "," + TempNow;
    transmit(transmitter);
    transmit("C");
    beginner = 1;
  } else if ((calibMode > 7) || (calibMode < 1)) {
    Serial.println("ERROR IN CALIBRATION MODE");
  }
}

unsigned long lastCommandTime =
    0;  // Initialize a global variable to track the last command execution time

String removeSubstring(String original, String toRemove) {
  // Find the start position of the substring to remove
  int startPos = original.indexOf(toRemove);

  // Check if the substring was found
  if (startPos != -1) {
    // Find the end position of the substring
    int endPos = startPos + toRemove.length();

    // Construct the new string without the unwanted substring
    String before = original.substring(0, startPos);
```

```
    String after = original.substring(endPos);
    return before + after;
  }

  // If the substring was not found, return the original string
  return original;
}

void useOldCalibrations() {
  calib0X = oldCalib0X;
  calibPosX = oldCalibPosX;
  calibNegX = oldCalibNegX;
  calib0Y = oldCalib0Y;
  calibPosY = oldCalibPosY;
  calibNegY = oldCalibNegY;
  calib0Z = oldCalib0Z;
  calibPosZ = oldCalibPosZ;
  calibNegZ = oldCalibNegZ;
  provideSlopes();
  String transmitter = "CALIBVALUES";
  transmitter = transmitter + "," + slopePosX + "," + calib0X + "," +
                slopeNegX + "," + slopePosY + "," + calib0Y + "," + slopeNegY +
                "," + slopePosZ + "," + calib0Z + "," + slopePosZ + "," +
                initialTemp;
  transmit(transmitter);
  Serial.println("Sending X");
  transmit("X");
  calibNum = 7;
  delay(100);
  beginner = 1;
}

void resetCalibration(unsigned long cTime) {
  Serial.println("RESETTING");
  beginner = 0;
  calibNum = 1;
```

```
  transmit("R");
  inputString = "";  // Clear the input string after processing the command
  lastCommandTime = cTime;  // Update the last command time to the current time
}

void receiveInput() {
  unsigned long currentTime = millis();  // Get the current time in milliseconds

  // Check if more than a second (1000 milliseconds) has passed since the last
  // command
  if (currentTime - lastCommandTime > 1000) {
    if (inputString.indexOf("C") !=
        -1) {  // Check if "C" is contained within the inputString
      if (inputString.indexOf(String(calibNum)) != -1) {
        Calibrate(calibNum);
        calibNum++;
        inputString =
            "";  // Clear the input string after processing the command
        lastCommandTime =
            currentTime;  // Update the last command time to the current time
        checkCount = 0;
      } else if (checkCount > 7) {
        resetCalibration(currentTime);
        checkCount = 0;
      } else {
        checkCount++;
      }
    } else if (inputString.indexOf("R") !=
               -1) {  // Check if "RESET" is contained within the inputString
      resetCalibration(currentTime);
    } else if (inputString.indexOf("OLD") != -1) {
      // Serial.println("Old read");
      useOldCalibrations();
    } else if (inputString.length() > 15) {
      inputString = "";
    }
```

```
  }
}

void checkInput() {
  if (HC12.available()) {
    char inChar = (char)HC12.read();
    inputString += inChar;
    receiveInput();
    // Serial.println(inputString);
  }
}

void transmitOptimisedBinary() {
  uint8_t dataStream[9] = {0};
  // So, I was originally transmitting the floats as binary, but I realised that
  // for a lot of things I don't need all that data. So I had a look at what I
  // felt I needed and only provided for that
  // Pack the data into the byte stream
  // timeChange: 12 bits - allows for 4 seconds between transmission at 3
  // decimal accuracy
  dataStream[0] = timeChange >> 4;          // Top 8 bits
  dataStream[1] = (timeChange & 0xF) << 4;  // Bottom 4 bits of timeChange

  // Mode: 1 bit - just says if the mode changed or not, rawAy: 10 bits - that's
  // the size of the values provided by the accelerometer
  dataStream[1] |= modeChange << 3;     // 1 bit from modeChange
  dataStream[1] |= (rawAy >> 7) & 0x7;  // Top 3 bits of rawAy
  dataStream[2] = (rawAy & 0x7F) << 1;  // Remaining 7 bits from rawAy

  // rawAx: 8 bits - these values cannot go above or below their gravity
  // calibrations in theory, so this gives the difference from their 0g value
  dataStream[2] |= (rawAx >> 7);        // Top 1 bit of rawAx
  dataStream[3] = (rawAx & 0x7F) << 1;  // Bottom 7 bits of rawAx

  // rawAz: 8 bits
  dataStream[3] |= (rawAz >> 7);        // Top 1 bit of rawAz
```

```
  dataStream[4] = (rawAz & 0x7F) << 1;  // Bottom 7 bits of rawAz

  // tempChange: 10 bits - required for 4 degrees change to 3 decimals
  dataStream[4] |= (tempChange >> 8) & 0x1;  // Top 2 bits of tempChange
  dataStream[5] = (tempChange >> 0) & 0xFF;  // Middle 8 bits of tempChange

  // PressDiff: 14 bits - required for 60 hPa change to 3 decimals
  dataStream[5] |= (PressDiff >> 12) & 0x3;  // Top 2 bits of PressDiff
  dataStream[6] = (PressDiff >> 4) & 0xFF;   // Middle 8 bits of PressDiff
  dataStream[7] = (PressDiff & 0xF) << 4;    // Bottom 4 bits of PressDiff

  // Terminator: 12 bits
  dataStream[7] |=
      (terminator >> 8) &
      0xF;  // Shift the terminator right by 8 bits to get the top 4 bits, then
            // mask with 0xF to ensure only the top 4 bits are used.

  // Store the remaining 8 bits of the terminator in dataStream[8]
  dataStream[8] = terminator & 0xFF

                  // Transmit the data stream
                  for (int i = 0; i < sizeof(dataStream); i++) {
    // This just writes the data to the HC12
    HC12.write((byte *)(void *)dataStream[i], sizeof(uint8_t));
  }
}

void transmitBinaryFloats(float *floatArray, size_t numFloats) {
  for (size_t i = 0; i < numFloats; i++) {
    HC12.write((byte *)(void *)&floatArray[i], sizeof(float));
  }

  // byte terminator[4] = {0xFF, 0xFF, 0xFF, 0xFF};
  // HC12.write(terminator, 4);
}
```

```
void readPackageAndTransmit(unsigned long time_now) {
  Serial.println(time_now);
  readEnvironmental();
  floatArray[0] = time_now;
  floatArray[1] = (float)mode;
  floatArray[2] = AccelNowX;
  floatArray[3] = AccelNowY;
  floatArray[4] = AccelNowZ;
  floatArray[5] = PressNow;
  floatArray[6] = TempNow;
  transmitBinaryFloats(floatArray, 7);
}

void loop() {
  if (beginner == 0) {
    checkInput();
    subtractTime = millis();
  } else {
    // delay(1000);
    Serial.println(millis());
    unsigned long time_now = millis() - subtractTime;
    readFromAccelerometer();
    detectMode(AccelNowY);
    // Serial.println(AccelNowX);

    switch (mode) {
      case 1:
        if ((time_now - pre_launch_last_transmit_time) > 5000) {
          pre_launch_last_transmit_time = time_now;
          readPackageAndTransmit(time_now);
          checkInput();
        }
        break;

      case 2:
        readPackageAndTransmit(time_now);
```

```
        break;

      case 3:
        readPackageAndTransmit(time_now);
        break;

      case 4:
        if ((time_now - landed_last_transmit_time) > 10000) {
          readPackageAndTransmit(time_now);
          checkInput();
        }
        break;

      default:
        // Serial.println("ERROR: No Mode Initialised");
        // HC12.println("ERROR: No Mode Initialised");
        // HC12.print("Acceleration (vertical) Value:   ");
        // HC12.println(AccelNowZ);
        break;
    }
  }
}
```

A1.2 Python

This code could be much prettier and better organised, but it does a lot and does it well. Were the team reusing this code, they would divide it into various function files and look into condensing the parts that use many lines without providing much benefit. The team would also use a Vector3 struct for many of the values.

A1.2.1 Main File (“dataAnalysis.py”)

```
import serial
import threading
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time
import openpyxl
from excelFunctions import appendExcel
from readValues import graphAfter
```

```
import math
import pickle
import struct

stop_event = threading.Event()

#TIMES AND MODES
realtime, mode = [0],[0]
launchTime, descendingTime, landedTime, avgTransmissionTime = 27.5439, 30.1589, 36.2895, 0.23
modeText = ['N/A']

#ACCELERATIONS
accelX, accelY, accelZ, accelVertical = [0],[0],[0],[0]
aXMax, aXMin, aYMax, aYMin, aZMax, aZMin, maxAccelVert, minAccelVert = 0, 0, 0, 0, 0, 0, 0, 0

#ANGLES
Xtilt, Ztilt, absoluteTilt = [0],[0],[0]
XtiltMax, ZtiltMax, absoluteTiltMax = 0, 0, 0
XtiltMin, ZtiltMin, absoluteTiltMin = 10000000000, 10000000000, 10000000000

#ENVIRONMENTAL SENSORS
pressure, temperature = [0],[0]
maxPressure, minPressure, refPress, maxTemperature, minTemp, refTemp = 0, 0, 0, 0, 0, 0

# ALTITUDES
altitudeE, maxAltE, minAltE, altitudeA, maxAltA, minAltA = [0], 0, 0, [0], 0, 0

# VELOCITIES
velX, vXMax, vXMin, velY, vYMax, vYMin, velZ, vZMax, vZMin, maxVelVert, minVelVert, velVertical = [0], 0, 0, [0], 0, 0, [0], 0, 0, 0, 0, [0]

# FORCES
forceX, fXMax, fXMin, forceY, fYMax, fYMin, forceZ, fZMax, fZMin, maxForceVert, minForceVert, forceVertical = [0], 0, 0, [0], 0, 0, [0], 0, 0, 0, 0, [0]
```

```
# CALIBRATIONS
slopePosX, calib0X, slopeNegX, slopePosY, calib0Y, slopeNegY, slopePosZ, calib0Z,
slopeNegZ, calibTemp = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0


#MISCELANEOUS
serial_port = '/dev/cu.usbserial-0001'
baud_rate = 9600
timeout = 2
beginCalcs = 0
rocketMass = 0.1
capsuleMass = 0.05
g = 9.81
seaLevelPressure = 1013.25 #Sea Level Air Pressure in hPa


lists_to_process = [mode, modeText, accelX, accelY, accelZ, pressure, temperature,
    altitudeE, altitudeA, velX, velY, velZ, forceX, forceY, forceZ, Xtilt, Ztilt,
absoluteTilt]


def appendMaxMin(list, max, min):
    global aXMax, aXMin, aYMax, aYMin, aZMax, aZMin, maxAccelVert, minAccelVert,
XtiltMax,  ZtiltMax,  absoluteTiltMax,  XtiltMin,  ZtiltMin,  absoluteTiltMin,
maxPressure, minPressure, refPress, maxTemperature, minTemp, refTemp, altitudeE,
maxAltE, minAltE, altitudeA, maxAltA, minAltA, velX, vXMax, vXMin, velY, vYMax,
vYMin, velZ, vZMax, vZMin, maxVelVert, minVelVert, velVertical, forceX, fXMax,
fXMin, forceY, fYMax, fYMin, forceZ, fZMax, fZMin, maxForceVert, minForceVert,
forceVertical
    if list[-1]>max:
        max = list[-1]


    if list[-1]<min:
        min = list[-1]


def equalize_list_lengths():
    global lists_to_process, mode, modeText, accelX, accelY, accelZ, pressure,
temperature, altitudeE, altitudeA, velX, velY, velZ, forceX, forceY, forceZ,
accelVertical, velVertical, forceVertical, Xtilt, Ztilt, absoluteTilt
    # Check each list in list_of_lists
    for lst in lists_to_process:
        while len(lst) < len(realtime):
```

```
            # Append the last value of lst if it has fewer elements than realtime
            lst.append(lst[-1])

        while len(realtime) < len(lst):
            # Append the last value of realtime if it has fewer elements than lst
            realtime.append(realtime[-1])


    return lists_to_process, realtime

def checkTransmissionTime():
    global avgTransmissionTime
    avgTransmissionTime = realtime[-1]/(len(realtime))

def equalize_before_plot(y_list, listName):
    #print(listName, " before equalization:  ", y_list[-1])
    if len(y_list) < len(realtime):
        print(listName, ":  ",len(y_list), "Realtime:  ", len(realtime))
        y_list += [y_list[-1]] * (len(realtime) - len(y_list))   # Append last
value if shorter
        print(listName,":  ",len(y_list), "Realtime:  ", len(realtime))
    elif len(y_list) > len(realtime):
        print("Ah, length error (list longer), how quaint")
        y_list = y_list[:len(realtime)]  # Trim if longer
    #print(listName, " after equalization:  ", y_list[-1])
    return y_list

def calculateTilt():
    global Xtilt, Ztilt, absoluteTilt
    if accelX:  # Check if accelX is not empty
        # Ensure the value passed to math.asin is within [-1, 1]
        value = accelX[-1] / g
        clamped_value = max(min(value, 1), -1)
        tilt = math.asin(clamped_value)
        Xtilt.append(tilt)
        xComponent = math.sin(Xtilt[-1])
    else:
```

```
        Xtilt.append(0)
    if accelZ:  # Check if accelX is not empty
        # Ensure the value passed to math.asin is within [-1, 1]
        value = accelZ[-1] / g
        clamped_value = max(min(value, 1), -1)
        tilt = math.asin(clamped_value)
        Ztilt.append(tilt)
        zComponent = math.sin(Ztilt[-1])
    else:
        Ztilt.append(0)

    appendMaxMin(Xtilt,XtiltMax,XtiltMin)
    appendMaxMin(Ztilt,ZtiltMax,ZtiltMin)

    yComponent = math.cos(Xtilt[-1]) * math.cos(Ztilt[-1])
    vectorMagnitude = math.sqrt(xComponent**2+yComponent**2+zComponent**2)

    absoluteTilt.append(math.acos(yComponent/vectorMagnitude))
    appendMaxMin(absoluteTilt,absoluteTiltMax, absoluteTiltMin)

def calculateVerticalAcceleration():
    global accelVertical
    accelVertical.append(accelY[-1]*math.cos(absoluteTilt[-1]))

def calculateMaxAcceleration():
    global maxAccel, aXMax, aYMax, aZMax, maxAccelVert #Defining global variables
    appendMaxMin(accelX, aXMax, aXMin)
    appendMaxMin(accelY, aYMax, aYMin)
    appendMaxMin(accelZ, aZMax, aZMin)
    appendMaxMin(accelVertical, maxAccelVert, minAccelVert)

def calculateVelocity(aX, aY, aZ):
    global aXMax, aXMin, aYMax, aYMin, aZMax, aZMin, maxAccelVert, minAccelVert,
XtiltMax, ZtiltMax, absoluteTiltMax, XtiltMin, ZtiltMin, absoluteTiltMin,
maxPressure, minPressure, refPress, maxTemperature, minTemp, refTemp, altitudeE,
maxAltE, minAltE, altitudeA, maxAltA, minAltA, velX, vXMax, vXMin, velY, vYMax,
vYMin, velZ, vZMax, vZMin, maxVelVert, minVelVert, velVertical, forceX, fXMax,
fXMin, forceY, fYMax, fYMin, forceZ, fZMax, fZMin, maxForceVert, minForceVert,
```

```
forceVertical
    if abs(aX)>0.3: #Ensures that the acceleration taking place isn't noise
        velX.append(velX[-1]+(aX)*(realtime[-1]-realtime[-2]))   #Steps   forward
velocity
    else:
        velX.append(velX[-1])
    if abs(aY)>0.3:
        velY.append(velY[-1]+(aY)*(realtime[-1]-realtime[-2]))
    else:
        velY.append(velY[-1])
    if abs(aZ)>0.3:
        velZ.append(velZ[-1]+(aZ)*(realtime[-1]-realtime[-2]))
    else:
        velZ.append(velY[-1])
    if abs(accelVertical[-1])>0.3:
        velVertical.append(velVertical[-1]+(accelVertical[-1])*(realtime[-1]-
realtime[-2]))
    else:
        velVertical.append(velVertical[-1])


    appendMaxMin(velX, vXMax, vXMin)
    appendMaxMin(velY, vYMax, vYMin)
    appendMaxMin(velZ, vZMax, vZMin)
    appendMaxMin(velVertical, maxVelVert, minVelVert)


def calculateForce(aX, aY, aZ):
    global aXMax, aXMin, aYMax, aYMin, aZMax, aZMin, maxAccelVert, minAccelVert,
XtiltMax,  ZtiltMax,  absoluteTiltMax,  XtiltMin,  ZtiltMin,  absoluteTiltMin,
maxPressure, minPressure, refPress, maxTemperature, minTemp, refTemp, altitudeE,
maxAltE, minAltE, altitudeA, maxAltA, minAltA, velX, vXMax, vXMin, velY, vYMax,
vYMin, velZ, vZMax, vZMin, maxVelVert, minVelVert, velVertical, forceX, fXMax,
fXMin, forceY, fYMax, fYMin, forceZ, fZMax, fZMin, maxForceVert, minForceVert,
forceVertical


    if mode[-1] == 1 or mode[-1] == 2:
        forceX.append(aX*rocketMass)
        forceY.append((aY+g)*rocketMass)
        forceZ.append(aZ*rocketMass)
```

```
        forceVertical.append(accelVertical[-1]*rocketMass)

    elif mode[-1] == 3 or mode[-1] ==4:
        forceX.append(aX*capsuleMass)
        forceY.append(aY*capsuleMass)
        forceZ.append(aZ*capsuleMass)
        forceVertical.append(accelVertical[-1]*rocketMass)
    else:
        print("Force Calculation Mode Error")

    appendMaxMin(forceX, fXMax, fXMin)
    appendMaxMin(forceY, fYMax, fYMin)
    appendMaxMin(forceZ, fZMax, fZMin)
    appendMaxMin(forceVertical, maxForceVert, minForceVert)

def calculateAverage(value):
    return sum(value)/len(value) if value else 0

def checkEnvironmentalMaxs():
    global pressure, maxPressure, temperature, maxTemperature
    appendMaxMin(pressure, maxPressure, minPressure)
    appendMaxMin(temperature, maxTemperature, minTemp)

def calculateEnvironmentalAltitude(press, temp):
    global altitudeE, maxAltE
    altitudeE.append(((press / refPress) ** (1 / -5.255877) - 1) * ((temp + 273.15)
/ 0.0065))
    appendMaxMin(altitudeE, maxAltE, minAltE)

def  calculateAccelerometerAltitude():  #Attempts  to  calculate  altitude  from
accelerometer values, for experimentation purposes
    global altitudeA, maxAltA
    altitudeA.append(altitudeA[-1]+velVertical[-1]*(realtime[-1]-realtime[-2]))
    if altitudeA[-1]<0:
        altitudeA[-1]=0
    appendMaxMin(altitudeA, maxAltA, minAltA)
```

```
def checkModeName(): #Maps the values sent by the lander to English mode names
    global launchTime, descendingTime, landedTime
    if checkForModeChange() == True:
        if mode[-1]==0:
            modeText.append('Error')
        elif mode[-1]==1:
            modeText.append('Pre-Launch')
        elif mode[-1]==2:
            modeText.append('Ascending')
            launchTime=realtime[-1]

        elif mode[-1]==3:
            modeText.append('Descending')
            descendingTime=realtime[-1]

        else:
            modeText.append('Landed')
            landedTime=realtime[-1]
    elif launchTime != 0 and realtime[-1] - launchTime > 20:
        mode.append(4)
        modeText.append('Landed')

def checkForModeChange():
    if mode[-1]!=mode[-2]:
        return True
    else:
        return False



def checkMode():
    if mode[-1] == 2 or mode[-1] == 3:
        if velVertical[-1] > 0:
            mode[-1] = 2
        else:
```

```
            mode[-1] = 3


def appendExcelFunc(): #Appends raw data to the excel sheet
    appendExcel(realtime, modeText, accelX, accelY, accelZ, accelVertical, velX, velY, velZ, velVertical, forceX, forceY, forceZ, forceVertical, altitudeE, altitudeA, pressure, temperature, maxAccelVert, aYMax, aXMax, aZMax, maxVelVert, vXMax, vYMax, vZMax, maxForceVert, fYMax, fXMax, fZMax, maxAltE, maxAltA, maxPressure, maxTemperature, launchTime, descendingTime, landedTime, avgTransmissionTime, Xtilt, Ztilt, absoluteTilt, minAccelVert, minForceVert, minVelVert, minAltA, minAltE, aXMin, aYMin, aZMin, vXMin, vYMin, vZMin, fXMin, fYMin, fZMin, XtiltMax, XtiltMin, ZtiltMax, ZtiltMin)


def map_accel_x(raw_x):
    if raw_x >= calib0X:
        # If the rawValue is between 0 and g, use the slope for that segment
        real_acceleration = slopePosX * (raw_x - calib0X)
    else:
        # If the rawValue is between -g and 0, use the slope for that segment
        real_acceleration = slopeNegX * (raw_x - calib0X)

    return real_acceleration

def map_accel_y(raw_y):
    if raw_y >= calib0Y:
        # If the rawValue is between 0 and g, use the slope for that segment
        real_acceleration = slopePosY * (raw_y - calib0Y)
    else:
        # If the rawValue is between -g and 0, use the slope for that segment
        real_acceleration = slopeNegY * (raw_y - calib0Y)

    return real_acceleration

def map_accel_z(raw_z):
    if raw_z >= calib0Z:
        # If the rawValue is between 0 and g, use the slope for that segment
        real_acceleration = slopePosZ * (raw_z - calib0Z)
    else:
        # If the rawValue is between -g and 0, use the slope for that segment
```

```
        real_acceleration = slopeNegZ * (raw_z - calib0Z)


    return real_acceleration


def saveLists():
    dataToPickle = {
        "realtime": realtime,
        "modeText": modeText,
        "accelX": accelX,
        "accelY": accelY,
        "accelZ": accelZ,
        "accelVertical": accelVertical,
        "velX": velX,
        "velY": velY,
        "velZ": velZ,
        "velVertical": velVertical,
        "forceX": forceX,
        "forceY": forceY,
        "forceZ": forceZ,
        "forceVertical": forceVertical,
        "altitudeE": altitudeE,
        "altitudeA": altitudeA,
        "pressure": pressure,
        "temperature": temperature,
        "maxAccelVert": maxAccelVert,
        "aYMax": aYMax,
        "aXMax": aXMax,
        "aZMax": aZMax,
        "maxVelVert": maxVelVert,
        "vXMax": vXMax,
        "vYMax": vYMax,
        "vZMax": vZMax,
        "maxForceVert": maxForceVert,
        "fYMax": fYMax,
        "fXMax": fXMax,
        "fZMax": fZMax,
```

```
        "maxAltE": maxAltE,
        "maxAltA": maxAltA,
        "maxPressure": maxPressure,
        "maxTemperature": maxTemperature,
        "launchTime": launchTime,
        "descendingTime": descendingTime,
        "landedTime": landedTime,
        "avgTransmissionTime": avgTransmissionTime,
        "Xtilt": Xtilt,
        "Ztilt": Ztilt,
        "absoluteTilt": absoluteTilt,
        "minAccelVert": minAccelVert,
        "minForceVert": minForceVert,
        "minVelVert": minVelVert,
        "minAltA": minAltA,
        "minAltE": minAltE,
        "aXMin": aXMin,
        "aYMin": aYMin,
        "aZMin": aZMin,
        "vXMin": vXMin,
        "vYMin": vYMin,
        "vZMin": vZMin,
        "fXMin": fXMin,
        "fYMin": fYMin,
        "fZMin": fZMin,
        "XtiltMax": XtiltMax,
        "XtiltMin": XtiltMin,
        "ZtiltMax": ZtiltMax,
        "ZtiltMin": ZtiltMin
    }
    with open('graphingLists.pkl', 'wb') as file:
        pickle.dump(dataToPickle, file)

broadcastedFinalResults = False


def averageDifference(lst):
```

```
    if len(lst) < 2:  # If the list has fewer than two elements, return 0 as there
are no differences to calculate.
        return 0
    differences = [abs(lst[i] - lst[i-1]) for i in range(1, len(lst))]
    average_diff = sum(differences) / len(differences)
    return average_diff


def printData():
    print("Max Acceleration X:  ", max(accelX))
    print("Max Acceleration Y:  ", max(accelY))
    print("Max Acceleration Z:  ", max(accelZ))

    print("Min Acceleration X:  ", min(accelX))
    print("Min Acceleration Y:  ", min(accelY))
    print("Min Acceleration Z:  ", min(accelZ))

    print("Average Acceleration X:  ", sum(accelX)/len(accelX))
    print("Average Acceleration Y: ", sum(accelY)/len(accelY))
    print("Average Acceleration Z: ", sum(accelZ)/len(accelZ))

    print("Maximum Vertical Acceleration:  ", max(accelVertical))
    print("Minimum Vertical Acceleration:  ", max(accelVertical))
    print("Average          Vertical          Acceleration:                    ",
sum(accelVertical)/len(accelVertical))

    print("Max Velocity X: ", max(velX))
    print("Max Velocity Y: ", max(velY))
    print("Max Velocity Z: ", max(velZ))

    print("Min Velocity X:  ", min(velX))
    print("Min Velocity Y: ", min(velY))
    print("Min Velocity Z: ", min(velZ))

    print("Average Velocity X:", sum(velX)/len(velX))
    print("Average Velocity Y:", sum(velY)/len(velY))
    print("Average Velocity Z:", sum(velZ)/len(velZ))
```

```
    print("Maximum Vertical Velocity:  ", max(velVertical))
    print("Minimum Vertical Velocity:  ", min(velVertical))
    print("Average Vertical Velocity:  ", sum(velVertical)/len(velVertical))

    print("Max Pressure:  ", max(pressure))
    print("Min Pressure:  ", min(pressure))
    print("Average Pressure:  ", sum(pressure)/len(pressure))

    print("Max Temperature:  ", max(temperature))
    print("Min Temperature:  ", min(temperature))
    print("Average Temperature:  ", sum(pressure)/len(pressure))

    print("Max Altitude (Environmental):  ", max(altitudeE))
    print("Max Altitude (Accelerometer):  ", max(altitudeA))

    print("Min Altitude (Environmental):  ", min(altitudeE))
    print("Min Altitude (Accelerometer):  ", min(altitudeA))

    print("Average Altitude (Environmental):  ", sum(altitudeE)/len(altitudeE))
    print("Average Altitude (Accelerometer):  ", sum(altitudeA)/len(altitudeA))

    print("Max Force X:  ", max(forceX))
    print("Min Force X:  ", min(forceX))
    print("Average Force X:  ", sum(forceX)/len(forceX))

    print("Max Force Y:  ", max(forceY))
    print("Min Force Y:  ", min(forceY))
    print("Average Force Y: ", sum(forceY)/len(forceY))

    print("Max Force Z:  ", max(forceZ))
    print("Min Force Z:  ", min(forceZ))
    print("Average Force Z: ", sum(forceZ)/len(forceZ))

    print("Maximum Vertical Force:  ", max(forceVertical))
    print("Minimum Vertical Force:  ", min(forceVertical))
    print("Average Vertical Force:  ", sum(forceVertical)/len(forceVertical))
```

```
    print("Max Tilt X:  ", max(Xtilt))
    print("Min Tilt X:  ", min(Xtilt))
    print("Average Tilt X:  ", sum(Xtilt)/len(Xtilt))

    print("Max Tilt Z:  ", max(Ztilt))
    print("Min Tilt Z:  ", min(Ztilt))
    print("Avereage Tilt Z:  ", sum(Ztilt)/len(Ztilt))

    print("Max Absolute Tilt:  ", max(absoluteTilt))
    print("Min Absolute Tilt:  ", min(absoluteTilt))
    print("Average Absolute Tilt:  ", sum(absoluteTilt)/len(absoluteTilt))

    print("Launch Time:", launchTime)
    print("Descent Began:  ", descendingTime)
    print("Landing Time:  ", landedTime)
    print("Final Time:  ", realtime[-1])
    print("Average Transmission Time:  ", averageDifference(realtime))



def read_and_process_serial_data():
    print("reading")
    global  beginCalcs,  realtime,  mode,  accelX,  acceY,  accelZ,  pressure,
temperature, beginCalcs, refTemp, refPress, lists_to_process
    try:
        # Set up serial connection
        ser = serial.Serial(serial_port, baud_rate, timeout=timeout)
        print(f"Connected to {serial_port} at {baud_rate} baud.")
        # Define the terminator and the size of the float set
        terminator = b'\xff\xff\xff\xff'
        num_floats = 7
        float_size = 4
        packet_size = num_floats * float_size  # Size of data in bytes (7 floats)

        buffer = bytearray()
```

```
        # if realtime[-1] - launchTime > 15:
        #     mode = 4

        while True:
            #print("true")
            if mode == 4:
                    if broadcastedFinalResults == False:
                        printData()

            if ser.in_waiting > 0:
                buffer += ser.read(ser.in_waiting)

                terminator_pos = buffer.find(terminator)
                while len(buffer>=packet_size):
                    while terminator_pos != -1:
                        # Ensure there is a complete packet before the terminator
                        if terminator_pos >= packet_size:
                            packet_start_pos = terminator_pos - packet_size
                            packet = buffer[packet_start_pos:terminator_pos]

                            if refTemp == 0: #Sets up reference values for the
altitude formula
                                refTemp=temperature[-1]
                            if refPress == 0:
                                refPress=pressure[-1]



                            try:
                                dataList  =  list(struct.unpack('<9B',  packet))
#This unpacks the data, reading it as a series of unsigned integers


                                try:
                                    realtime.append((float((dataList[0] << 4) |
(dataList[1] >> 4)))/1000)
                                except ValueError:
                                    if realtime:  # Check if not empty to avoid
IndexError
```

```
                                        realtime.append(realtime[-1])


                                    try:
                                        if ((dataList[1] >> 3) & 0x08) == 1:
                                            mode.append(mode[-1]+1)
                                        else:
                                            mode.append(mode[-1])
                                    except ValueError:
                                        if mode:
                                            mode.append(mode[-1])



                                    try:

accelY.append(float(map_accel_y(((dataList[1] & 0x70) >> 4) << 7 | ((dataList[2]
& 0xFE) >> 1))))

                                        if accelY[-1]<0:
                                            accelY[-1]+=g
                                        else:
                                            accelY[-1]-=g
                                    except ValueError:
                                        if accelY:
                                            accelY.append(accelY[-1])


                                    try:
                                        tempX = (((dataList[2] & 0x01) << 7) |
(dataList[3] >> 1))
                                        if tempX > 128:
                                            tempX -= 256
                                        tempX += calib0X
                                        accelX.append(float(map_accel_x(tempX)))

                                    except ValueError:
                                        if accelX:
                                            accelX.append(accelX[-1])
```

```
                                try:
                                    tempZ  =  (((dataList[3]  &  0x01)  <<  7)  |
(dataList[4] >> 1))
                                    if tempZ > 128:
                                        tempZ -= 256
                                    tempZ += calib0Z
                                    accelZ.append((float(map_accel_z(tempZ))))

                                except ValueError:
                                    if accelZ:
                                        accelZ.append(accelZ[-1])

                                try:
                                    tempTemp  =  (((dataList[4]  &  0x01)  <<  8)  |
dataList[5])
                                    if tempTemp > 512:
                                        tempTemp -= 1024
                                    tempTemp = tempTemp/1000 + calibTemp
                                    temperature.append(float(tempTemp))


                                except ValueError:
                                    if temperature:
                                        temperature.append(temperature[-1])


                                try:
                                    tempPress = (((dataList[5] & 0x03) << 12) |
(dataList[6] << 4) | ((dataList[7] & 0xF0) >> 4))
                                    if tempPress > 8192:
                                        tempPress -= 16384
                                    tempPress = tempPress/1000 + seaLevelPressure
                                    pressure.append(tempPress)
                                except ValueError:
                                    if pressure:
                                        pressure.append(pressure[-1])

                                #Uses all the functions declared before
```

```
                                calculateTilt()
                                calculateVerticalAcceleration()
                                calculateMaxAcceleration()
                                calculateVelocity(accelX[-1],        accelY[-1],
accelZ[-1])
                                checkMode()
                                calculateForce(accelX[-1], accelY[-1], accelZ[-
1])
                                checkEnvironmentalMaxs()
                                calculateEnvironmentalAltitude(pressure[-1],
temperature[-1])
                                calculateAccelerometerAltitude()
                                checkModeName()
                                checkForModeChange
                                appendExcelFunc()
                                saveLists()

                            except struct.error as e:
                                print(f"Error unpacking data: {e}")

                            # Remove processed packet and its terminator from the
buffer
                            buffer = buffer[terminator_pos + len(terminator):]
                        else:
                            # Not enough data before this terminator, remove up
to and including this terminator
                            buffer = buffer[terminator_pos + len(terminator):]

                        # Look for the next terminator in the remaining buffer
                        terminator_pos = buffer.find(terminator)


                    time.sleep(0.5)  # For Debugging

    except serial.SerialException as e: #Error handling
        print(f"Error opening serial port: {e}")

    except KeyboardInterrupt:
```

```
        print("\nProgram terminated by user")

    finally:
        if 'ser' in locals() or 'ser' in globals():
            #ser.close()
            print("Serial port error")


windowSize = 30 #How many seconds are displayed by the graphs

def update_limits(ax, time_data, timecheck, window_size=60):  # Adjust window_size
as needed
    if timecheck > 0:
        right_limit = timecheck
        left_limit = max(0, right_limit - window_size)
        # Ensure there's a minimal difference between left_limit and right_limit
        if left_limit == right_limit:
            left_limit -= 1  # Decrement left_limit
            right_limit += 1   # Increment right_limit to ensure they are not
identical
        ax.set_xlim(left_limit, right_limit)
        ax.figure.canvas.draw()

scroll_offset = 0 #Artefact of proposed manual scrolling mechanic



def start_graphing():
    global scroll_offset
    fig1, axs1 = plt.subplots(3, num='Acceleration Data')
    fig2, axs2 = plt.subplots(3, num='Environmental Data')
    fig3, axs3 = plt.subplots(3, num='Velocity Data')
    fig4, axs4 = plt.subplots(3, num='Force Data')
    fig5, axs5 = plt.subplots(3, num='Vertical Data')


    def animate_accel(i):
        try:
            for j, ax in enumerate(axs1):
```

```
                ax.clear()
                if j == 0:
                    equalized_accelX = equalize_before_plot(accelX, "accelX")
#Ensures list lengths are equal to "realtime", which they are plotted against, to
avoid Matplotlib errors
                    ax.plot(range(len(realtime)), equalized_accelX, label='Accel
X') #Calls the Matplotlib plot() function
                    ax.set_title('Real-time Acceleration X')
                elif j == 1:
                    equalized_accelY = equalize_before_plot(accelY, "accelY")
                    ax.plot(range(len(realtime)), equalized_accelY, label='Accel
Y')
                    ax.set_title('Real-time Acceleration Y')
                elif j == 2:
                    equalized_accelZ = equalize_before_plot(accelZ, "accelZ")
                    ax.plot(range(len(realtime)), equalized_accelZ, label='Accel
Z')
                    ax.set_title('Real-time Acceleration Z')
                update_limits(ax,       range(len(realtime)),       realtime[-1],
scroll_offset) #Updates the axes to give an animated effect
        except Exception as e:
            print(f"Error in animate_accel: {e}") #Prevents the graphing from
ceasing should an error occur


    def animate_env(i):
        try:
            for j, ax in enumerate(axs2):
                ax.clear()
                if j == 0:
                    equalized_temp      =      equalize_before_plot(temperature,
"temperature")
                    ax.plot(range(len(realtime)),                equalized_temp,
label='Temperature')
                    ax.set_title('Real-time Temperature')
                elif j == 1:
                    equalized_press = equalize_before_plot(pressure,'Pressure')
                    ax.plot(range(len(realtime)),                equalized_press,
label='Pressure')
                    ax.set_title('Real-time Pressure')
```

```
                elif j == 2:
                    equalized_alt = equalize_before_plot(altitudeE,'Altitude')
                    ax.plot(range(len(realtime)),                 equalized_alt,
label='Altitude')
                    ax.set_title('Real-time Altitude')
                update_limits(ax,       range(len(realtime)),       realtime[-1],
scroll_offset)
        except Exception as e:
            print(f"Error in animate_env: {e}")

    def animate_vel(i):
        try:
            for j, ax in enumerate(axs3):
                ax.clear()
                if j == 0:
                    equalized_velX = equalize_before_plot(velX,'Vel X')
                    ax.plot(range(len(realtime)), equalized_velX, label='Vel X')
                    ax.set_title('Real-time Velocity X')
                elif j == 1:
                    equalized_velY = equalize_before_plot(velY,'Vel Y')
                    ax.plot(range(len(realtime)), equalized_velY, label='Vel Y')
                    ax.set_title('Real-time Velocity Y')
                elif j == 2:
                    equalized_velZ = equalize_before_plot(velZ,'Vel Z')
                    ax.plot(range(len(realtime)), equalized_velZ, label='Vel Z')
                    ax.set_title('Real-time Velocity Z')
                update_limits(ax,       range(len(realtime)),       realtime[-1],
scroll_offset)
        except Exception as e:
            print(f"Error in animate_vel: {e}")

    def animate_force(i):
        try:
            for j, ax in enumerate(axs4):
                ax.clear()
                if j == 0:
                    equalized_forceX = equalize_before_plot(forceX,'force X')
```

```
                    ax.plot(range(len(realtime)), equalized_forceX, label='force
X')
                    ax.set_title('Real-time Force X')
                elif j == 1:
                    equalized_forceY = equalize_before_plot(forceY,'force Y')
                    ax.plot(range(len(realtime)), equalized_forceY, label='force
Y')
                    ax.set_title('Real-time Force Y')
                elif j == 2:
                    equalized_forceZ = equalize_before_plot(forceZ,'force Z')
                    ax.plot(range(len(realtime)), equalized_forceZ, label='force
Z')
                    ax.set_title('Real-time Force Z')
                update_limits(ax,       range(len(realtime)),       realtime[-1],
scroll_offset)
        except Exception as e:
            print(f"Error in animate_force: {e}")

    def animate_vert(i):
        try:
            for j, ax in enumerate(axs5):
                ax.clear()
                if j == 0:
                    equalized_vel     =     equalize_before_plot(velVertical,'Vel
Vertical')
                    ax.plot(range(len(realtime)),    equalized_vel,    label='Vel
Vertical')
                    ax.set_title('Real-time Vertical Velocity')
                elif j == 1:
                    equalized_accel  =  equalize_before_plot(accelVertical,'Accel
Vertical')
                    ax.plot(range(len(realtime)),  equalized_accel,  label='Accel
Vertical')
                    ax.set_title('Real-time Vertical Acceleration')
                elif j == 2:
                    equalized_force  =  equalize_before_plot(forceVertical,'Force
Vertical')
                    ax.plot(range(len(realtime)),  equalized_force,  label='Force
Vertical')
```

```
                    ax.set_title('Real-time Vertical Force')
                update_limits(ax,       range(len(realtime)),       realtime[-1],
scroll_offset)
        except Exception as e:
            print(f"Error in animate_vert: {e}")


    # Setup animations
    ani1       =       FuncAnimation(fig1,       animate_accel,       interval=50,
cache_frame_data=False)
    ani2 = FuncAnimation(fig2, animate_env, interval=50, cache_frame_data=False)
    ani3 = FuncAnimation(fig3, animate_vel, interval=50, cache_frame_data=False)
    ani4       =       FuncAnimation(fig4,       animate_force,       interval=50,
cache_frame_data=False)
    ani5 = FuncAnimation(fig5, animate_vert, interval=50, cache_frame_data=False)


    plt.show()



def setupFunc():
    seri = serial.Serial(serial_port, baud_rate, timeout=timeout)
    print(f"Connected to {serial_port} at {baud_rate} baud.")
    # message = "BEGIN"


    def    waitForMessage(message,    expected_message,    firsttimeout    =    1,
timeout_duration=5, reset_message="R", reset_ack="R"):
        start_time = time.time()
        secondtime = time.time()
        while True:
            if seri.in_waiting:
                line = seri.readline().decode('utf-8', errors='ignore').rstrip()
                if expected_message in line:
                    return "SUCCESS"
                elif reset_ack in line:  # Handle receiving reset acknowledgment
                    print("Reset acknowledged. Starting calibration over.")
                    return "RESET"
            else:
                current_time = time.time()
```

```
                if (current_time - start_time) > timeout_duration:
                    seri.write(reset_message.encode())  # Send RESET command
                    print("No response, sending RESET.")
                    start_time = current_time  # Reset the start time for the next
interval
                    timeout_duration = 2  # Set the new timeout duration for RESET
handling
                elif    ((current_time-secondtime)    >    firsttimeout)    and
((current_time-start_time) < timeout_duration):
                    print("No response, retrying")
                    seri.write(message.encode())
                    secondtime = current_time


    def waitForReset(timeoutDuration = 2,reset_message="R", reset_ack="R"):
        start_time = time.time()
        while True:
            if seri.in_waiting:
                line = seri.readline().decode('utf-8', errors='ignore').rstrip()
                if reset_ack in line:  # Handle receiving reset acknowledgment
                    print("Reset acknowledged. Starting calibration over.")
                    return "RESET"
            else:
                current_time = time.time()

                if (current_time - start_time) > timeoutDuration:
                    seri.write(reset_message.encode())  # Send RESET command
                    print("Sending RESET.")
                    start_time = current_time  # Reset the start time for the next
interval


    def findCalibValues():
        global  identifier,  calibPosX,  calib0X,  calibNegX,  calibPosY,  calib0Y,
calibNegY, calibPosZ, calib0Z, calibNegZ, calibTemp
        line = seri.readline().decode('utf-8').strip()


        # Assuming the line format is as you described, with commas separating
values
```

```
        # Check if the line is not empty
        if line:
            # Split the line into values
            values = line.split(",")

            # Ensure that there are enough values in 'values' to unpack successfully

            if len(values) == 10:
                identifier, calibPosX, calib0X, calibNegX, calibPosY, calib0Y, calibNegY, calibPosZ, calib0Z, calibNegZ, calibTemp = values

            else:
                waitForReset()

        else:
                waitForReset()

    def calibrationStep(instruction, message, expectedMessage, success_message):
        while True:
            numberIn = int(input(instruction))
            if numberIn == 1:
                if message == "C7":
                    findCalibValues
                seri.write(message.encode())
                response = waitForMessage(message, expectedMessage)
                if response == "SUCCESS":
                    print(success_message)
                    return True
                elif response == "RESET":
                    return False
            elif numberIn == 404:
                response = waitForReset()
                if response == "RESET":
                    return False
            else:
                print("Input not recognised. Try again.")
```

```
    while True:

        ChoiceNum = input("Use old calibration values (1), recalibrate (2), or
read pre-initialised data-stream (3):   ")

        if ChoiceNum == "1":
            if calibrationStep("Enter 1 to confirm choice:  ", "OLD", "X",
                               "Calibration Complete, Data Transmission Has
Begun"):
                break
            else:
                continue

        elif ChoiceNum == "2":

            if not calibrationStep("Please orient Arduino on the Z-axis. Enter
'1' to confirm Z-axis calibration (Note: at any point you can input '404' to
restart calibration):   ", "C1","C",
                                   "Z  Calibration  1  Complete.  Please  Invert
Arduino."):
                continue  # Restart calibration if RESET is acknowledged
            if  not  calibrationStep("Enter  '1'  to  confirm  inverted  Z-axis
calibration:   ","C2","C",
                                   "Z Calibration 2 Complete. Please orient Arduino
on X-axis."):
                continue
            if  not  calibrationStep("Enter  '1'  to  confirm  X-axis  calibration:
","C3","C",
                                   "X  Calibration  1  Complete.  Please  Invert
Arduino."):
                continue
            if  not  calibrationStep("Enter  '1'  to  confirm  inverted  X-axis
calibration:   ","C4","C",
                                   "X Calibration 2 Complete. Please orient Arduino
on Y-axis."):
                continue
            if  not  calibrationStep("Enter  '1'  to  begin  Y-axis  (vertical)
calibration:   ","C5","C",
```

```
                                "Y Calibration 1 Complete. Please Invert
Arduino."):
                continue
            if not calibrationStep("Enter '1' to begin inverted Y-axis
calibration:  ","C6","C",
                                "Y Calibration 2 Complete. Good job!"):
                continue
            if calibrationStep("Enter '1' to begin data transmission:
","C7","C",
                               "Data Transmission Has Begun"): #This transmission is
important, as it triggers the listening for calibration values
                break #exit the loop if all previous steps have been executed
sequentially without a "false" being returned
            else:
                continue

        elif ChoiceNum == "3":
            confirmNum = input("Enter 1 to confirm choice:  ")
            if confirmNum == "1":
                break
            else:
                continue




def command_endGraph():
    print("End Graph command called")
    plt.close('all')  # Close all Matplotlib plots
    # Additional commands to end graphing processes if necessary

def command_endTransmission():
    # Signal the serial plotting thread to stop
    stop_event.set()
    print("Serial transmission ending requested.")

def command_printData():
    printData()
```

```
def handle_user_input():
    while True:
        user_input = input()
        if user_input == "endGraph":
            command_endGraph()
        elif user_input == "print":
            print("Printing")
            command_printData()
        elif user_input == "endTransmission":
            command_endTransmission()
            break  # Exit the loop to end the program, or remove break if you want
to keep the program running
        elif user_input == "END ALL":
            command_endGraph()
            command_endTransmission()
            break  # Exit the loop to end the program
    print("User input handler stopped.")


if __name__ == "__main__":
    setupFunc()
    #Start the serial data handling in a separate thread
    serial_thread     =     threading.Thread(target=read_and_process_serial_data,
daemon=True)
    serial_thread.start()

    # Start the graphing on the main thread
    start_graphing()
    handle_user_input()


    time.sleep(5)
    graphAfter()
```

A1.2.2 Excel Manipulation (“excelFunctions.py”)

```
import openpyxl

workbookPath = '../TransmissionStorage.xlsx'
```

```
workbook = openpyxl.load_workbook(workbookPath)
sheet = workbook['MainSheet']

rawValueRow = 11
rawMaxColumn = 2

maxERow = 1
maxARow = 2
maxAccRow = 3
maxVelRow = 4
maxForceRow = 5
maxPressureRow = 6
maxTempRow = 7

maxAyRow = 14
maxAxRow = 15
maxAzRow = 16

maxVyRow = 19
maxVxRow = 20
maxVzRow = 21

maxFyRow = 24
maxFxRow = 25
maxFzRow = 26

rawTimeColumn = 8
rawModeColumn = 10

rawAxColumn = 12
rawAyColumn = 13
rawAzColumn = 14
rawA_AbsoulteColumn = 15

rawVxColumn = 17
rawVyColumn = 18
```

```
rawVzColumn = 19
rawV_AbsoluteColumn = 20

rawHeightEnvColumn = 22
rawHeightAccColumn = 23

rawFxColumn = 25
rawFyColumn = 26
rawFzColumn = 27
rawF_AbsoluteColumn = 28

rawPressureColumn = 30
rawTemperatureColumn =31

rawXTiltColumn = 33
rawZTiltColumn = 34
rawAbsoluteTiltColumn = 35


def appendExcel(time, modeText, accelX, accelY, accelZ, accelVertical, velX, velY,
velZ, velVertical, forceX, forceY, forceZ, forceVertical, altitudeE, altitudeA,
pressure, temperature, maxAccelVert, aYMax, aXMax, aZMax, maxVelVert, vXMax,
vYMax, vZMax, maxForceVert, fYMax, fXMax, fZMax, maxAltE, maxAltA, maxPressure,
maxTemperature, launchTime, descendingTime, landedTime, avgTransmissionTime,
Xtilt, Ztilt, absoluteTilt, minAccelVert, minForceVert, minVelVert, minAltA,
minAltE, aXMin, aYMin, aZMin, vXMin, vYMin, vZMin, fXMin, fYMin, fZMin, XtiltMax,
XtiltMin, ZtiltMax, ZtiltMin):
    sheet.cell(row=rawValueRow+len(time), column=rawModeColumn).value=modeText[-
1]
    sheet.cell(row=rawValueRow+len(time), column=rawAxColumn).value=accelX[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawAyColumn).value=accelY[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawAzColumn).value=accelZ[-1]
    sheet.cell(row=rawValueRow+len(time),
column=rawA_AbsoulteColumn).value=accelVertical[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawVxColumn).value=velX[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawVyColumn).value=velY[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawVzColumn).value=velZ[-1]
    sheet.cell(row=rawValueRow+len(time),
column=rawHeightEnvColumn).value=altitudeE[-1]
    sheet.cell(row=rawValueRow+len(time),
```

```
column=rawHeightAccColumn).value=altitudeA[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawFxColumn).value=forceX[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawFyColumn).value=forceY[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawFzColumn).value=forceZ[-1]
    sheet.cell(row=rawValueRow+len(time),
column=rawF_AbsoluteColumn).value=forceVertical[-1]
    sheet.cell(row=rawValueRow+len(time),
column=rawPressureColumn).value=pressure[-1]
    sheet.cell(row=rawValueRow+len(time),
column=rawTemperatureColumn).value=temperature[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawXTiltColumn).value=Xtilt[-1]
    sheet.cell(row=rawValueRow+len(time), column=rawZTiltColumn).value=Ztilt[-1]
    sheet.cell(row=rawValueRow+len(time),
column=rawAbsoluteTiltColumn).value=absoluteTilt[-1]


    sheet.cell(row=maxERow, column = rawMaxColumn).value = maxAltE
    sheet.cell(row=maxARow, column = rawMaxColumn).value = maxAltA
    sheet.cell(row=maxAccRow, column = rawMaxColumn).value = maxAccelVert
    sheet.cell(row=maxVelRow, column = rawMaxColumn).value = maxVelVert
    sheet.cell(row=maxForceRow, column = rawMaxColumn).value = maxForceVert
    sheet.cell(row=maxPressureRow, column = rawMaxColumn).value = maxPressure
    sheet.cell(row=maxTempRow, column = rawMaxColumn).value = maxTemperature

    sheet.cell(row=maxAyRow, column = rawMaxColumn).value = aYMax
    sheet.cell(row=maxAxRow, column = rawMaxColumn).value = aXMax
    sheet.cell(row=maxAzRow, column = rawMaxColumn).value = aZMax

    sheet.cell(row=maxVyRow, column = rawMaxColumn).value = vYMax
    sheet.cell(row=maxVxRow, column = rawMaxColumn).value = vXMax
    sheet.cell(row=maxVzRow, column = rawMaxColumn).value = vZMax

    sheet.cell(row=maxFyRow, column = rawMaxColumn).value = fYMax
    sheet.cell(row=maxFxRow, column = rawMaxColumn).value = fXMax
    sheet.cell(row=maxFzRow, column = rawMaxColumn).value = fZMax
    sheet['D3']=launchTime #Stores the time of launch
    sheet['E3']=descendingTime #Stores when descent begins
    sheet['F3']=landedTime #Stores the time of landing
```

```
    sheet['I3']=avgTransmissionTime #Stores the time of landing

workbook.save(filename="/Users/eoghancollins/ArduinoLocal/TransmissionStorageReal.xlsx")
```

A1.2.3 Post-Landing Graphing ("readValues.py")

```
import pickle
import matplotlib.pyplot as plt
# from matplotlib.animation import FuncAnimation
import numpy as np


realtime = []
modeText = []
accelX = []
accelY = []
accelZ = []
accelVertical = []
velX = []
velY = []
velZ = []
velVertical = []
forceX = []
forceY = []
forceZ = []
forceVertical = []
altitudeE = []
altitudeA = []
pressure = []
temperature = []


def loadLists():
    global realtime, modeText, accelX, accelY, accelZ, accelVertical, velX, velY, velZ, velVertical, forceX, forceY, forceZ, forceVertical, altitudeE, altitudeA, pressure, temperature
    # Load the pickle file
    with open('graphingLists.pkl', 'rb') as file:  # 'rb' for reading in binary mode
        data_loaded = pickle.load(file)
```

```
    # Accessing lists from the dictionary
    realtime = data_loaded["realtime"]
    modeText = data_loaded["modeText"]
    accelX = data_loaded["accelX"]
    accelY = data_loaded["accelY"]
    accelZ = data_loaded["accelZ"]
    accelVertical = data_loaded["accelVertical"]
    velX = data_loaded["velX"]
    velY = data_loaded["velY"]
    velZ = data_loaded["velZ"]
    velVertical = data_loaded["velVertical"]
    forceX = data_loaded["forceX"]
    forceY = data_loaded["forceY"]
    forceZ = data_loaded["forceZ"]
    forceVertical = data_loaded["forceVertical"]
    altitudeE = data_loaded["altitudeE"]
    altitudeA = data_loaded["altitudeA"]
    pressure = data_loaded["pressure"]
    temperature = data_loaded["temperature"]

def equalize_before_plot(y_list):
    if len(y_list) < len(realtime):
        print("Ah, length error (list shorter), how quaint")
        y_list += [y_list[-1]] * (len(realtime) - len(y_list))   # Append last
value if shorter
    elif len(y_list) > len(realtime):
        print("Ah, length error (list longer), how quaint")
        y_list = y_list[:len(realtime)]  # Trim if longer
    return y_list


def plot_data():
    # Data sets to be plotted in each window
    datasets = [
        (accelX, accelY, accelZ, 'Acceleration X, Y, Z'),
        (velX, velY, velZ, 'Velocity X, Y, Z'),
```

```
        (pressure, temperature, altitudeE, 'Pressure, Temperature, Altitude'),
        (accelVertical, velVertical, forceVertical, 'Vertical Acceleration,
Velocity, Force')
    ]

    # Titles for each subplot
    titles = [
        ['Acceleration X', 'Acceleration Y', 'Acceleration Z'],
        ['Velocity X', 'Velocity Y', 'Velocity Z'],
        ['Pressure', 'Temperature', 'Altitude'],
        ['Vertical Acceleration', 'Vertical Velocity', 'Vertical Force']
    ]

    # Loop through each data set and create a window for each
    for i, (data1, data2, data3, window_title) in enumerate(datasets):
        fig, axs = plt.subplots(3, 1, figsize=(10, 8))
        fig.suptitle(window_title)

        axs[0].plot(realtime, equalize_before_plot(data1))
        axs[0].set_title(titles[i][0])
        axs[0].set_xlabel('Time')
        axs[0].set_ylabel(titles[i][0])

        axs[1].plot(realtime, equalize_before_plot(data2))
        axs[1].set_title(titles[i][1])
        axs[1].set_xlabel('Time')
        axs[1].set_ylabel(titles[i][1])

        axs[2].plot(realtime, equalize_before_plot(data3))
        axs[2].set_title(titles[i][2])
        axs[2].set_xlabel('Time')
        axs[2].set_ylabel(titles[i][2])

        plt.tight_layout()
        # plt.show(block=False)

def graphAfter():
```

```
    loadLists()
    plot_data()
    plt.show()
def average_difference(lst):
    if len(lst) < 2:
        # If there are fewer than 2 elements, the average difference is undefined
        return None

    total_difference = 0
    for i in range(len(lst) - 1):
        # Calculate the difference between consecutive elements
        difference = lst[i+1] - lst[i]
        total_difference += difference

    # Calculate the average difference
    avg_diff = total_difference / (len(lst) - 1)
    return avg_diff

# Example usage



if __name__ == "__main__":
    loadLists()
    plot_data()
    plt.show()
    # print(average_difference(realtime))
```

A2 Data

A2.1 Device Time vs Arduino Time (10s demo, printed once every 1s device time, truncated to 2 decimal places)

Device Time	Arduino Time
1.00	0.98
2.00	1.92
3.00	2.76
4.00	3.84
5.00	4.75

6.00	5.74
7.00	6.59
8.00	7.51
9.00	8.54
10.00	9.48