

UK CanSat Competition

*'Canned Beans'*

*The King's School, Peterborough*

*Critical Design Review*

*Date: 31/01/25*

# 1 INTRODUCTION

## 1.1 Team Organisation and Roles

Our team consists of 4 members: Students E, D, S and G.

The project manager role is held by Student E who will ensure the team work together to make decisions, but he has the final say. He will try to keep everybody on track with their tasks aiding the team with a timeline so that all tasks can be completed with adequate time to check for errors. Student E will oversee outreach with companies and social media to promote the CanSat project. Student E will also keep on top of the budgets and part ordering to ensure that the team stay on track financially. Student E has knowledge of leading tasks on numerous occasions and his role as captain of both Rugby and Cricket teams allows him to be able to direct well and communicate effectively in time-pressed situations to achieve the best results. This will allow the team to thrive and meet the deadlines of the CanSat competition.

The software lead role is also held by Student E, and he will be using the C++ language to interact with the Arduino Nano in order to allow the CanSat to measure and calculate values during its launch and flight, thus allowing it to fulfil both its primary and secondary missions. He will be responsible for all elements of the software and so must communicate with the Hardware lead to ensure it's well compatible. As a Computer Science and Further Mathematics, A-level student, Student E will use his vast knowledge of programming as well as statistics and algorithmic thinking to enable a strong secondary mission which can be met realistically despite complex algorithms which are later discussed.

The communications role is undertaken by Student D, and his role is to devise a way for ground control to receive information from the CanSat such as weather info, wind measurements etc. He shall be using APC 220 – because it is a type of FSK [Frequency Shift Keying] - along with a Yagi antenna designed to special specifications and learn how the 'sweet spot' works so that the CanSat can broadcast waves to the antenna. Additionally, Student D must also develop the right code that will go well with the Arduino and APC 220, so that the Arduino and APC 220s can send and receive the right signals. He shall be testing for the right code using a computer and DFRobot website.

The Hardware lead role is taken by Student S. His role is to research and assemble all the components that make up the CanSat. This includes primarily electronics and coming up with circuits that make the missions possible. The hardware lead must also take into consideration weight and size of the components going into the CanSat to make sure everything is within the ESA regulations on the CanSat mission.

The design lead role is taken by Student G. His role is to design a shell and parachute for the CanSat. This includes iterating through various designs improving each or altering the way it performs to optimise it and account for challenges such as areas which may be subjected to high amounts of physical stresses and account for the requirements of hardware within the shell. The design lead must also consider dimensional constraints ensuring the CanSat is within parameters of what is allowed.

## 1.2 Mission Overview

### 1.2.1 Mission Objectives

#### Primary Mission:

As stated, measuring temperature and atmospheric pressure and transmitting this data to the ground station via communications whilst conforming to the CanSat guidelines {CDR Appendix Reference 1}.

The CanSat is using an 'Arduino Nano Every' as our micro-controller. It will be connected to the BMP280 sensors which allow us to measure both temperature and air pressure to deliver our primary mission. The Arduino Nano Every has dimensions of 18mm x 45 mm it easily fits inside a 'soda can' size. With a mere weight of 5 grams, we will easily be able to conform to the weight limits of the CanSat, we can always add extra weights to our final design to ensure we reach the required velocity and weight requirements. The BMP280 has dimensions of 15mm x 11mm once more allowing us to easily build our CanSat within the limits that are set by the requirements.

Our programming lead will use the C++ language through the Arduino IDE to communicate with the Arduino Every Nano and it's BMP280 sensor to retrieve the data recorded by them. This can then be directed through the communications equipment, set up by the communications lead, allowing the data to be sent back to the ground station and either be processed or saved for later use. We will use this with commands later discussed in the software section to get accurate readings for temperature and pressure using the sensors, every '**dt**', a small change in time allowing for correlations in the data to be monitored and analysed to mathematically model trends not only to provide the most accurate values, but also, to later help in our secondary mission.

These data points can then be received at our ground-base by using a suitable radio-transmission system. Utilising an Arduino and an APC 220, the CanSat will be able to transmit messages within the 300 MHz – 3GHz band, allowing us to access a UHF ranger within the ‘Sweet Spot’, so we can maximise the bandwidth of data. This will enable our Yagi Antenna to receive the data from our CanSat such as temperature and transmit them to our computer in the form of images and texts. We will also run multiple tests runs over different frequencies to find a unique frequency that isn’t being used by other teams or devices, allowing a clear and strong connection between the transmitter and receiver.

### Secondary Mission:

Our secondary mission will be to use recorded and obtained values with complex prediction algorithms to try and predict the weather.

Our idea is to create a device that will be launched up into the atmosphere of a local region with no weather forecast to try and predict the weather allowing them to gauge a possibility of what the weather may be like in the very near future. We’d use sensors to obtain values and then predict the following:

- Predicted Temperature & Heat Index
- Predicted wind Chill
- Predicted visibility
- Predicted precipitation & Condensation

Together we could combine these to predict the **movement of weather fronts**, predict the **precipitation forecasts** and predict the **likelihood of storms** and other unexpected events that could happen.

Whilst the flight is short (between 30 seconds and a minute roughly), with accurate values obtained and good data processing, our values could stay accurate and usable for up to a day after the flight.

Using the following formula for each value:

$$\text{rate of change of } n = \Delta n / \Delta \text{Time}$$

{where n is a value e.g. temperature}

Using the data obtained during the flight, we can find the gradient of the graph  **$\Delta n$  vs time elapsed** which will give us the average **rate of change of n**. This can then be used compared to the actual values of **rate of change of n** to see the accuracy of our values.

We can then **extrapolate** this data outside the range of values using **linear regression and interpolation** to retain accuracy and precision where possible. This can then give us a good predictor of when the values will be inaccurate afterwards.

Using pre-recorded test data from a reliable source as a predictor we can write a simple Python program to process the data obtained from a Large Data Set into plottable co-ordinates.

{Appendix Reference 5 – data Processing code}

Using an online graphing calculator such as [Desmos](#) we can plot these co-ordinates and find a correlation using a linear function in the form:

$$f(t) = mt + c$$

{where **m** is the gradient and **c** is the y-intercept}

By adjusting the sliders between a range of values, we can approximate our linear function as such:

The approximate equation:

{Appendix Reference 6 – Desmos Graph}

$$f(t) = 0.00905t + 13.4$$

gives us a good estimate of what the temperature over the time period should yield during our CanSat test runs.

This data and estimate give us the **rate of change of temperature** with respect to time or in other words:  $dT/dt$

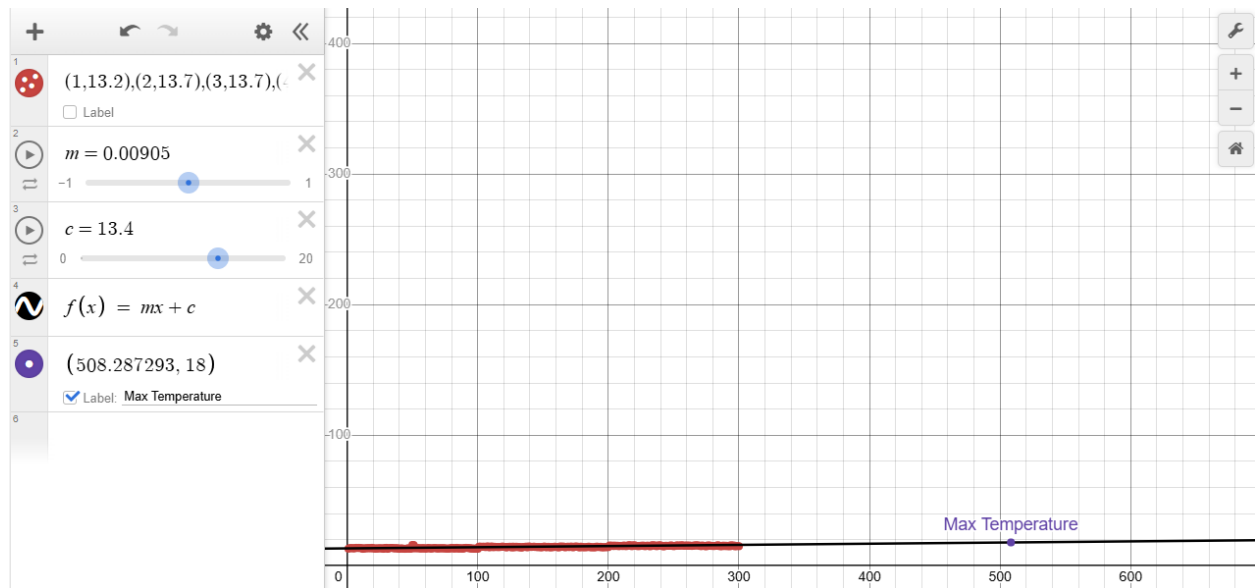
This shows the rate of change of temperature to be 0.00905°C per minute.

We know that the upper bound of the temperature for said day in the data is  $\sim 18^{\circ}\text{C}$

Setting  $f(t) = 18$  and rearranging the formula,  $t = (18 - 13.4) / 0.00905$  which yields a  $t$ -value of 508.287293

Knowing that a reading is taken every minute in this data, modelling a minute to a second, our values will stay accurate for  $\sim 508.3$  minutes which equates to  $\sim 8.47$  hours. These seems reasonable and good enough for a short 1-minute flight allowing us to attempt to predict weather for the next 8-hours providing our results and calculations are next-to perfect in accuracy.

This can be seen on the graph below where the purple represents the maximum temperature value of  $\sim 18^{\circ}\text{C}$



We can assume a similar trend will occur for ease of processing meaning that 8 hours is a good point. However, these values do assume perfect, theoretical accuracy which obviously is very unlikely in our flight time due to many factors, one being the specs and accuracy of the sensors themselves being a possible bottleneck in the system. Whilst we'll most certainly do what we can to retain accuracy of results, factors such as the internal accuracy of the hardware is out of our control and will need to be factored into our uncertainty and error calculations to ensure that we can correctly process our data.

It is important to note that the specific rate of change given for a 1-minute time period cannot be picked up by the equipment used because of the specific calibration, so it will need to be combined with the values for atmospheric pressure to build a more accurate simulation of future weather. Furthermore, it should be considered that the launch may not happen during a period of time where temperature change will not be positive, so the parameters for the equation should be reconsidered to build a more accurate picture.

This simple model allows us to predict the accuracy of our data values which we will record if we create a model for each value we are recording (stated below in 1.2.2).

### 1.2.2 What will you measure, why and how?

We will take measurements of the atmospheric pressure and temperature as the CanSat descends as our primary mission. For this we will use a GY-BMP 280 sensor, which can sense both temperature and pressure. This component will allow us to sense temperatures in the range of  $-40^{\circ}\text{C}$  and  $85^{\circ}\text{C}$  and can read pressure in the range of 300 to 1100 HPA, which will be perfect for our CanSat mission. We can then graph these values to show how the data changes as the CanSat descends.

For our microcontroller, we will be using the Arduino Nano Every. It is small and lightweight. It has the ATmega4809 processor with an AVR CPU up to 20MHz, 48kB flash, 6kB SRAM and 256B EEPROM, making it more powerful than many other Arduino units. It comes at the cost of needing a more powerful input compared to the original Arduino Nano, however the features it comes with are sure to help with the project.

For our secondary mission of weather prediction, we can use the temperature and pressure readings from the primary mission to improve efficiency and memory usage as there is no need to re-record them. The GY-BMP 280 sensor will also allow us to record the altitude as it's already built in. This is good as we will not only need the altitude as one of our measurements, but also, during descent we can use the other measurements to see how they change with altitude and therefore calculate the different weather conditions depending on the altitude you are at. At ground level, we can simply use linear regression once more to see how the data points regress to the last recorded values (closest to the ground) and therefore how the data stays and remains at ground level.

To determine the humidity, we will use the DHT22 (AM2302) due to its very high accuracy which is required in our complex calculations. It uses digital pins like our other components making it easily accessible and compatible with the Arduino microcontroller. It has a range of 0-100% humidity meaning we can observe even extremities in humidity readings, further increasing our accuracy. Finally, the DHT22's low power consumption is great for a small device for a CanSat as a low current reduces the risk of it heating up as well as reducing the need for a larger power supply and more cells which could end up causing more harm than good.

To determine windspeed, we are using the Adafruit Anemometer (ID: 1733) due to its lightweight and durability, perfect for a CanSat where weight is a restriction and damage to components is very possible {as later identified in the 2.3 risk matrix}. It has a high range of up to 32m/s (115km/h) which is more than enough for our readings, a higher windspeed than this would pose more of a risk to the actual CanSat than the readings being inaccurate. It handily has an analogue output via a pin which also makes compatible and accessible allowing us easily to obtain the readings and perform our calculations which will be further benefitted by the accuracy of the sensor.

Using our obtained values, we can make the following calculations:

### 1. Lapse Rate for Stable vs Unstable air

$$\Gamma = \frac{T2 - T1}{H2 - H1}$$

{where  $\Gamma$  is lapse rate, T is temperature and H is altitude}

- By using the temperature change over the altitude change, every reading we get, we can recalculate lapse rate and form it into a linear function of lapse rate over altitude. This means that we can then take the lapse rate at the end of the function as we extend the data for ground-level predictions.
- We can compare our calculated lapse rate to the standard tropospheric lapse rate of **6.5°C/km** to predict storms or clouds forming:
  - If  $\Gamma > 6.5^\circ\text{C/km} \rightarrow$  **Unstable Air**  $\rightarrow$  Possible storms or cloud formation
  - If  $\Gamma < 6.5^\circ\text{C/km} \rightarrow$  **Stable Air**  $\rightarrow$  Clear weather

### 2. Dew Point via Humidity for Fog, Clouds and Rain predictions

$$\text{Dew Point} = \frac{243.12 \cdot \left( \ln \left( \frac{RH}{100} + \left( \frac{17.62 \cdot T}{243.12 + T} \right) \right) \right)}{17.62 - \left( \ln \left( \frac{RH}{100} + \left( \frac{17.62 \cdot T}{243.12 + T} \right) \right) \right)}$$

{where RH is relative humidity, T is temperature}

- By using the empirically tested **Magnus-Tetens** formula above, for dew point we can obtain a highly accurate value for dew point
- The Dew point tells us how much **moisture** can be detected in the air; by further comparing it to known changes in dew point over large amounts of data processing, we can safely make conclusions from it
- If  $(T - T_d) > 2^\circ\text{C} \rightarrow$  **High Fog / Precipitation likelihood**
- If  $(T - T_d) < 5^\circ\text{C} \rightarrow$  **Lower chances of clouds/rain**

### 3. Barometric Pressure Trends

$$P_{sea-level} = P \cdot \left(1 - \frac{Lh}{T_{sea-level}}\right)^{\frac{gM}{RL}}$$

{where P is Pressure, h is altitude, L is standard lapse rate, g is gravitational field strength, M is molar mass of air, R is universal gas constant}

- By using this formula for pressure at sea-level, we can find the change in pressure with respect to altitude which hopefully should be minimal as we are taking altitude into account every calculation
- This is, however, useful for testing purposes as it allows us to see our error margin and re-calculate if necessary.
- **The constants used:**
  - **L = Standard lapse rate (0.0065 K/m)**
  - **g = Gravity (9.80665 m/s<sup>2</sup>)**
  - **M = Molar mass of air (0.028964 kg/mol)**
  - **R = Universal gas constant (8.314 J/(mol·K))**

If the values are off, we can use another estimation formulae, and we may perhaps use this to then find an average and improve accuracy.

$$P_{sea-level} = P \cdot e^{\left(\frac{h}{8400}\right)}$$

{where P is pressure, h is altitude}

### 4. Wind Trends

- By measuring wind every second, we can plot a graph of wind against time and find the gradient of this graph giving us a value of (wind / time)
- Next, we can plot a graph of wind vs altitude for every value recorded and instead extrapolate it outwards very slightly beyond the data set to find the wind at ground level.
- We can then strength of the change in wind over time at the point where wind is equal to the wind at ground level. This will give us the rate of change of wind where the strength of the wind in combination with pressure tells us the possibility of entering a storm or weather front.

Using well-tested and known conclusions and trends for a basic forecast, we can use our calculated values to make a conclusion about the weather to come, here are the possible scenarios summarised into a table:

Lapse Rate	$T - T_d$	Pressure Trend	Wind Speed	Prediction
> 6.5°C/km	< 2°C	Rapid Drop	Increasing	Thunderstorm likely ⚡
> 6.5°C/km	< 2°C	Gradual Drop	Steady	Cloudy, rain possible ☁
< 6.5°C/km	> 5°C	Steady/Rising	Calm	Clear weather ☀
> 6.5°C/km	> 5°C	Rapid Drop	Strong Winds	Cold front/storm incoming 🌪
< 6.5°C/km	< 2°C	Steady	Calm	Fog likely 🌫

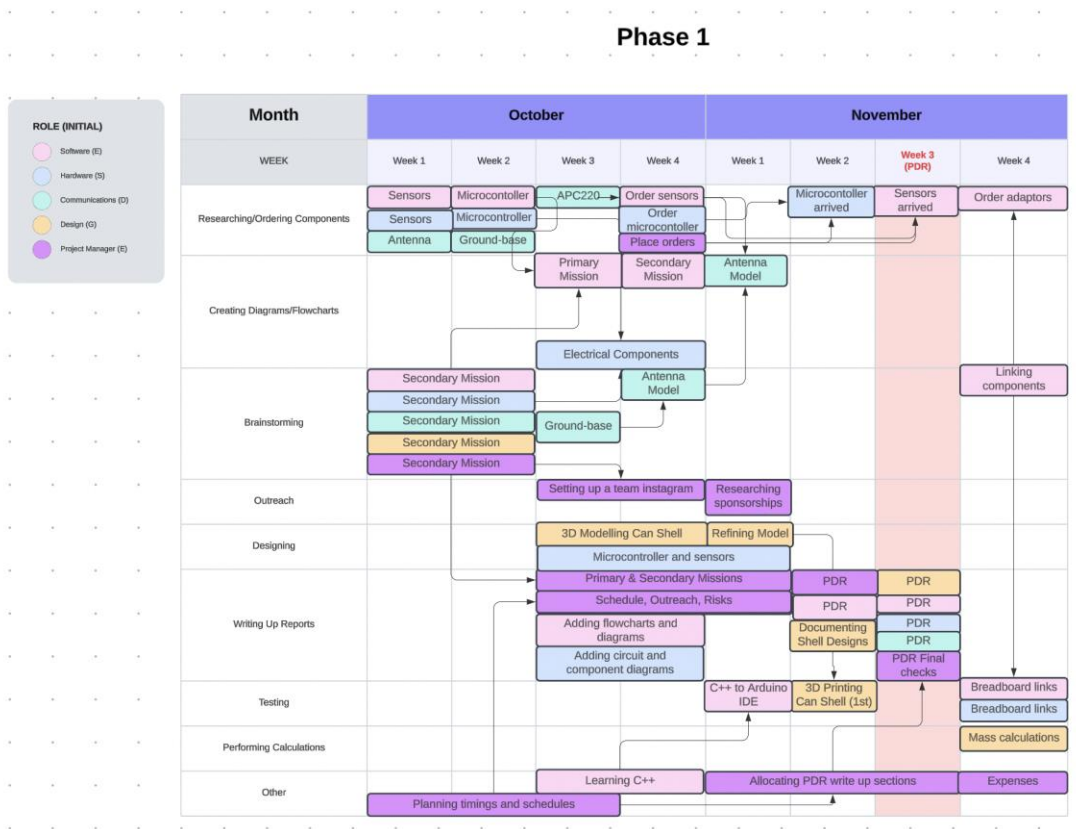
Not only would we be able to produce predictions like this, but we'd also be able to show the predicted temperature, humidity, air pressure and wind speeds over a short future period of time due to the measurements we have taken and basic data processing of analysing linear trends between them, much like the method used to determine the change in temperature with respect to time earlier.

## 2 PROJECT PLANNING

### 2.1 Time schedule

#### Phase 1: October to November

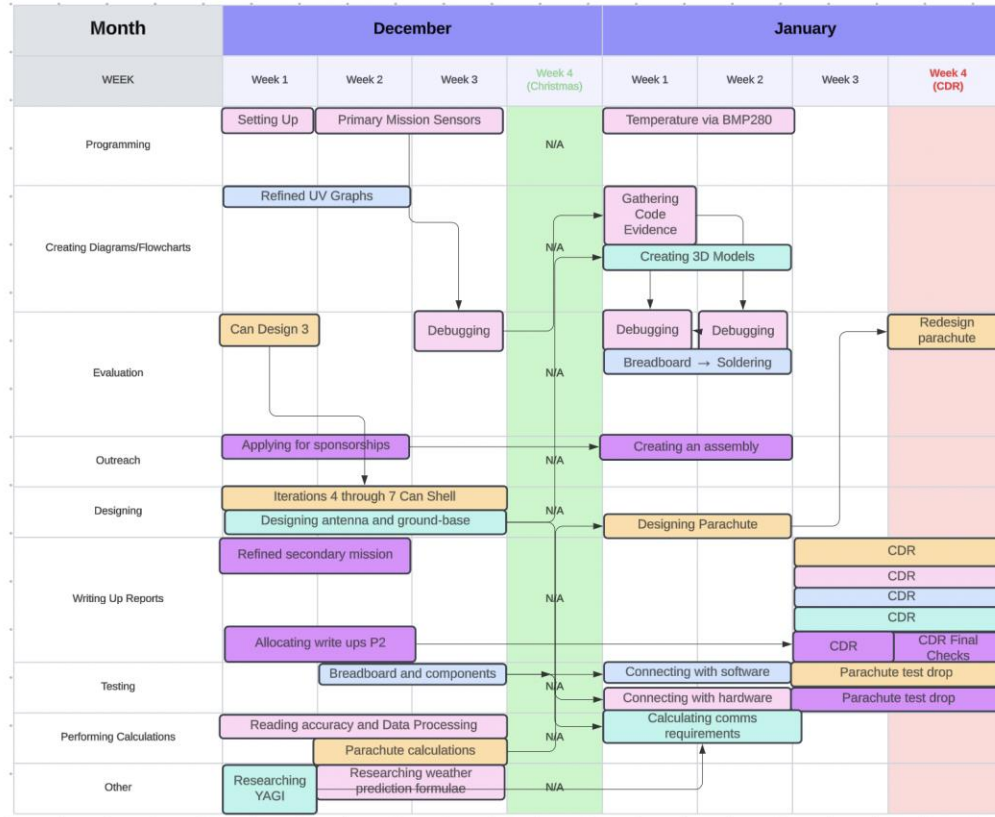
Do the following, each block represents a week's work which is roughly 2-4 hours per week spent on that task. Towards the PDR deadline at the end of November, this increased to 3-5 hours per week. The Key shows each person's colour and their initials e.g. Software is in pink, by Student E. The arrows represent the flow of ideas when creating this chart from the project manager and how the team can co-operate together to link tasks together for optimal efficiency.



#### Phase 2: December to January

Similarly to Phase 1, except ramped up a bit as more hands-on work needs to be done to create the actual CanSat. Each block of work is roughly 3-5 hours per week with it being accelerated up towards the CDR deadline to 4-6 hours per week. This change in hours allows us to avoid burn-out and continue motivation in the project. This is also accounted for in the weeks break for Christmas and New Years where the team members can prioritise family time and enjoyment to take a break and relax from CanSat, ready to work even harder in the new year.

## Phase 2



## 2.2 Team and External Support

Student E, the software lead is very confident in programming using Python, however we have opted for the Arduino due to its better functionality and therefore must use the C++ language. Student E has used this before; however, he is less proficient using it and so is taking an online course to refine his C++ programming skills in order to be better able to program the software functionality for the CanSat. Student E recognises that the basic programming constructs are the same in all languages and so his background of coding can help with this. However, data and memory handling are very different due to the lower-level nature of C++ in comparison to Python and so Student E has refined his knowledge of hardware and its interaction with software by revising the processes and functions of the kernel and specifically how to interact with primary and secondary storage in a computers operating system.

Student D might struggle with communication and might require even more assistance in understanding how to create a Yagi antenna that can receive signals from the APC 220 Duck Antenna, as he's never undertaken a task like this before. As student D takes Physics, it shouldn't be too hard to understand how waves work and possible interference the CanSat might experience as it sends signals and subsequently create ways to prevent or limit the interference to receive good signals and information. Student D shall be referring to previous CanSat teams for guidance to fully understand the importance of communication for CanSat, and remembering how the algorithm for the Arduino ought to be coded, and how it works.

Student G is well experienced in design having studied Product Design at GCSE, in addition he is well practiced with 3D computer aided design softwares such as "2D design" and "onshape", this will aid greatly with the design and development of the CanSat shell and parachute. Student G takes Physics at the A-level this will help him to understand the forces acting upon the parachute and therefore the expected result when taking all factors into account. Student G also takes A-level Mathematics which combined with Physics will enable him to undertake the necessary calculations for the parachute, by combining his knowledge of Product Design and Physics with his application of Mathematics, this hopefully should mean that the design and deployment of the parachute should go smoothly.

Student S has a passion for hardware and circuits, especially with sensors and electrical components. He has experience with electronics via some side projects he has completed and his own research, however, he is still unfamiliar with some aspects of it. Student S takes Physics A-level and the learning of electrical circuits will greatly help him to analyse possible risks as well as take measurements and process data in both testing and final iterations of the CanSat. Using various equations and calculations for Power and other values, Student S can determine the power needed to be supplied to the hardware components via the (power source in the form of a cell). Student S also takes Chemistry A-level, allowing him to understand the inner workings of cells and batteries with the



chemical reactions which take place transferring chemical energy into electrical energy, this will greatly benefit the ability to maintain power in the CanSat.

## 2.3 Risk Analysis

There are possibilities for software risks and so code and programs should be saved and tested with every change in an iterative testing style to ensure that all is working well. It should be frequently tested with the hardware so any issues can be isolated and fixed immediately. Multiple copies of files and scripts being used should be saved as backups with some cloud-based and some locally-based to prepare for if something were to happen to the main copy. This means that no matter what, there will always be a spare copy and so the code is always downloadable and available. Software will also be rigorously tested to remove logic errors and syntax errors. All members of the team will be white box tested on the software ensuring they can proficiently navigate the Arduino IDE and alter the software slightly if, for some reason, Student E isn't available on the launch day. This means that everyone will be able to fix any last minute logical or syntactical errors which may occur in development.

In regard to mechanical design, there are many risks which must be accounted for with the mechanical design such as lack of airflow leading to heat generated by the components affecting the performance of equipment or leading to inaccuracy of sensor readings which take measurements that can be affected by changes in temperature. To account for this the case has been designed with several holes to ensure that airflow is maximised, and temperature of components is kept reasonable. There is also a risk of the parachute not deploying or the shell of the CanSat and components inside breaking on impact due to an inability to slow it down sufficiently.

To prevent this, we will run parachute drop tests to ensure it is tested for. The strings will always be kept untangled with simple knots so they cannot be caught in each other. The parachute will take a simple shape with a strong yet flexible fabric like rip stock nylon being used to ensure it doesn't "tear" or "rip". By lining the inside of the CanSat shell with softer material, the components would collide with that rather than the CanSat shell, this will increase the time taken for the momentum to change and thus decrease the force on the components from the shell ensuring they are less likely to break.

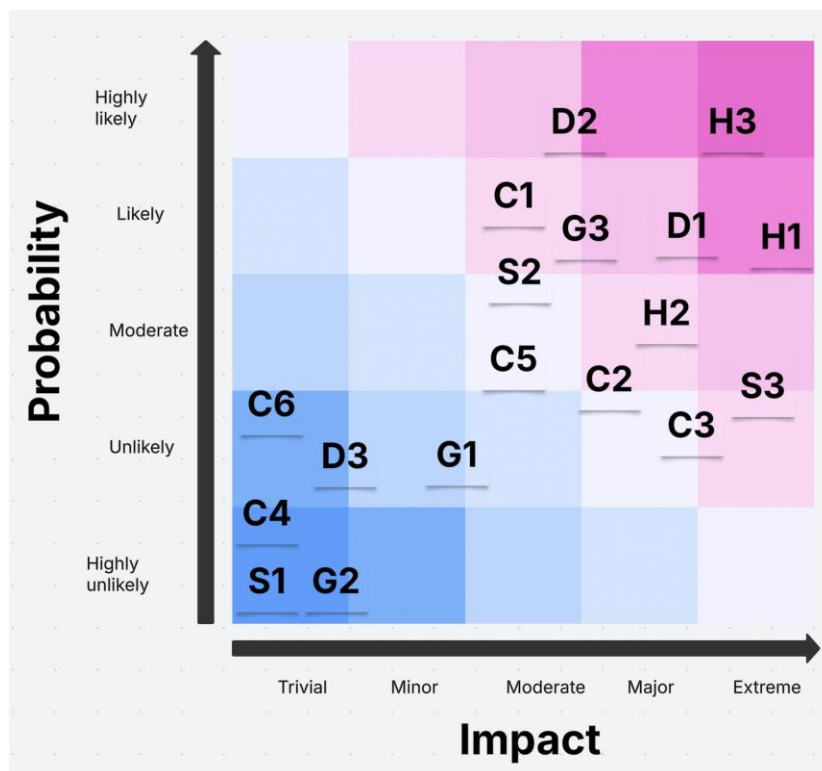
Linked in with this, electrical design components have the risk of becoming overheated and not functioning properly or at all. This issue is reduced by the mechanical design of airflow to properly cool the components down, however, in a situation where this ventilation does not prove enough cooling compared to the heat being given off, extra cooling systems such as small fans could be used to ensure adequate and optimal performance of said components. To prepare for launch, all components will be kept at a safe, yet cool, temperature before with limited use to ensure that during the launch they are initially cool and so are less likely to overheat. Whilst extensive cooling systems inside a CanSat are unrealistic and not easily available or accessible, ventilation and thorough testing of individual components as well as the whole system together should remove the need for this, yet emergency systems will be put in place just in case.

Regarding the communications systems, Student D's inexperience may cause for unexpected issues initially, however, by familiarising himself completing courses and thoroughly researching the topic, he will be able to ensure that all communication systems are available for the team's use. However, other issues may occur, one being the desired frequency being already in use. To prevent the possibility of this disrupting our communication we will ensure to test many different frequencies so that we know our hardware and software are compatible with different frequencies in case a sudden change is needed. We will also use a form of signature-based filtering by using the **preamble and header** method. By adding a unique sequence of bits, characters and strings before the data transmission, we can then check that every piece of data received has this sequence of bits at the beginning using very simple data filtering and processing techniques. Any data received that doesn't contain this sequence will not meet the condition and will thus be discarded from the data received. This allows us to only collect our own data and to keep it clean and accurate as well as, if it were a more private circumstance, allowing for easy encryption to prevent interception of data. However, this is not an issue in this circumstance.

The risks can be summarised and added to in a two-way categorical risk chart below:

	Risk Title & Details			How we will mitigate the risk			Risk Impact (Initial : Post Mitigation)			Probability of Occurrence		
General	Going over budget	Parts don't arrive	Unfinished CanSat	Student E uses a budget tracker	Student E pre-orders in time	Stick to project planning Gantt Chart	High : Low	Very High : Very Low	High : Medium	Unlikely	Very Unlikely	Possible
Software	Syntax Error	Logic Error	File Corruption	Fix using error message information	Vigorous White-box testing	Keep cloud-based and local backups	Medium : Low	High : Low	Very High : Very Low	Very Unlikely	Possible	Possible / Likely
Hardware	Damage to components	Overheating components	Power loss	Keep backups where possible, test safely	Ventilation and cooling	Spare batteries	Very High : High	High : Low	Very High : Very Low	Possible / Likely	Possible / Unlikely	Very Possible
Design	Parachute doesn't deploy	CanSat damaged on impact	CanSat doesn't meet guidelines	Parachute drops, connections tested and untangled	Vigorous Testing	Test dimensions and be familiar with guidelines	Very High : High	Very High : Medium	Medium : Very Low	Possible	Very Possible	Very Unlikely
Communications	Damage to antenna	Signals not received	Power Lost	Reinforcements	Vigorous testing by comms. lead	Pre-test power supply	High : Low	Very High : Medium	Very High : Medium	Possible	Possible	Possible
	Incorrect frequency	Used frequency	Injuries via moving components	Pre-determine a frequency	Choose a unique frequency	All members stand back	Low : Very Low	High : Very Low	Medium : Very Low	Very Unlikely	Possible	Very Unlikely

These risks can then be helpfully categorised further into a risk assessment matrix allowing a clear directory as to what needs to be mitigated first and what order is optimal to reduce as many risk as possible.



By analysing this Risk assessment matrix, we can see that many risks are either very extreme and likely which need immediate attention and mitigation, these are mainly concentrated in the hardware section with some from each section. The low-impact, highly unlikely risks will still be assessed, attested to and mitigated where possible however are much less urgent than the rest. These are more in the general areas as well as some in the communications areas. By reviewing this, we need to ensure our hardware is top-notch and we cannot have room for error for the CanSat to work and function effectively

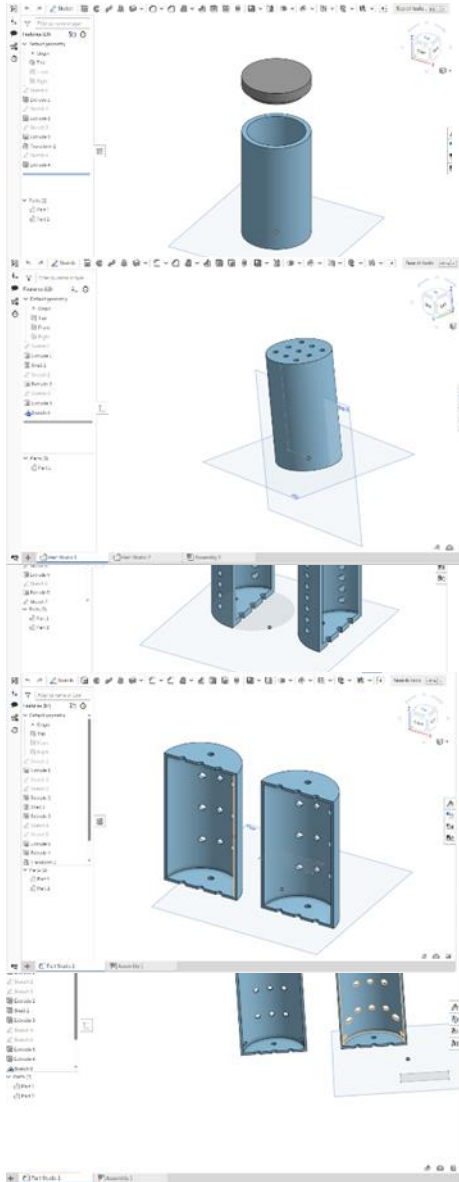
Risk G1's mitigation via a budget tracker can be found:

{Appendix 22, a screenshot of our budget tracker} {Appendix 23, a screenshot of item order and purchase log}

## 3 CANSAT DESIGN

### 3.1 Mechanical design

Design one



I made an initial design of a case for the can but realised this initial design had a few issues such as being difficult to access and having very little airflow. The lack of airflow and awkward to work with design, in the sense that it may be difficult to access hardware within it leading to this design needing to be discarded.

Design two

On design two I experimented with how I could space out the vents on the top.

Design three

My improvement on my first design featured Several holes in its sides for ventilation and a design that opened across the long cross section of the case.

Design four

To improve on the previous design, I spaced the holes out more evenly to prevent creating weaknesses.

Design five

On Design five I added holes for bolts to be fed through to hold the case together and holes to feed the parachute ropes through, opting for the holes to be on the side of the case, I did this with the aim of minimising the risk of any entanglement with the ropes by keeping them as separated as reasonably possible.

Prints of design five:

Design six

{Appendix 21 - CanSat Shell Design Iteration 6}

For design 6 I increased the size of the bolt holes to allow for thicker bolts to fit through the holes on design 5 were too small to fit the bolts I wanted to use. I moved the holes for the parachute to prevent the CanSat from getting stuck in the rocket tube before needing to drop and added to more holes to accommodate for the number of ropes the parachute would have and spaced them equidistantly.

### Parachute design

After being deployed the CanSat is in freefall, the CanSat experiences weight due to gravity, this causes an acceleration if allowed to accelerate eventually the CanSat will reach a velocity at which the force of drag due to air resistance is equal to the force of weight due to gravity causing acceleration to cease, this is terminal velocity, the terminal velocity of the CanSat would be so great that upon colliding with the ground the impulse involved and forces involved would cause the CanSat to be destroyed. To limit the terminal velocity of the CanSat a parachute is used, the parachute increases the area of the CanSat increasing the force of drag

which acts on it in proportion to the velocity at which it is traveling at this means that the terminal velocity of the CanSat can be decided via a particular area of parachute being used.

To calculate the necessary area for the parachute I rearranged the equation

$$\text{"Drag force} = -1/2(\text{Drag coefficient})(\text{Local air density})(\text{Area})(\text{velocity})^2\text{"}$$

to

$$\text{"Area} = (-2(\text{Drag force})/((\text{drag coefficient})(\text{Local air density})(\text{target velocity})^2))\text{"}$$

Since the force being used in the calculation is counteracting weight using newtons second law that the resultant force is equal to the sum of all forces acting on an object and the equation Force due to gravity = (mass) (gravitational field strength) we can replace the force in the equation with (mass) (gravitational field strength) and since it acts in the opposite direction to the drag force we make it negative so we get:

$$\text{"Area} = (2(\text{mass})(\text{gravitational field strength})/((\text{drag coefficient})(\text{Local air density})(\text{target velocity})^2))\text{"}$$

And entering the values:

$$\text{Mass} = 0.4\text{kg}$$

$$\text{Target velocity} = 10\text{ms}^{-1}$$

$$\text{Gravitational field strength} = 9.81\text{Nkg}^{-1}$$

(obtained from the European space agency "Design your parachute" pdf document):

$$\text{Drag coefficient of a ripstop nylon hexagon} = 0.8$$

$$\text{Local air density: } 1.225\text{kgm}^{-3}$$

Plugging these values into the equation

$$(2(0.4)(9.81))/((1.225)(0.8)(10)^2)$$

$$\frac{(2(0.4)(9.81))}{((1.225)(0.8)(10)^2)}$$

$$\approx 0.08082\text{m}^2$$

And rearranging the equation that links the side length of a polygonal hexagon  $A = ((3\sqrt{3})/2)s^2$

To solve for side length from area we get the equation

$$S = \sqrt{(2A)/3\sqrt{3}}$$

$$S = \sqrt{(2(0.08082/3\sqrt{3}))} \approx 0.18\text{m side length}$$

Knowing this I used the 2D computer aided design software "2D Design" to create a stencil for a parachute, I then laser cut that stencil and traced it onto a sheet of ripstop nylon along with 2 cm of space to hem the edges of the parachute. After cutting out the parachute and hemming its edges I punched holes 1.5cm away from the edges of the hem in preparation to install rivets where the parachute rope was to run through. Then I hammered the rivets into the parachute and threaded the rope into the riveted holes tying knots on either side of the riveting to secure the rope.

### 3.2 Electrical design

#### Components:

##### Microcontroller: Arduino Nano Every

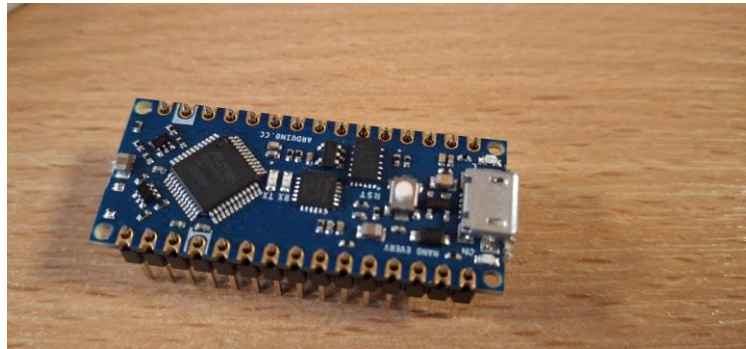
Lightweight and small microcontroller that can fit anywhere. It has been tested to support the necessary sensors that we need and relay information efficiently. It was chosen as opposed to our second option, the Arduino UNO, as it was smaller, lighter and a higher processing power.

The processing power is higher on the Arduino nano every as it uses the ATmega4809P processor as opposed to the ATmega328P processor on the Arduino UNO.

The Arduino Nano Every also has a higher flash memory of 48KB, which is higher and more efficient to work with than the UNO's 32KB. In terms of SRAM, the Arduino Nano Every can hold up to 6KB rather than UNO's 2KB, which makes it a better fit for running larger and more complex programs.

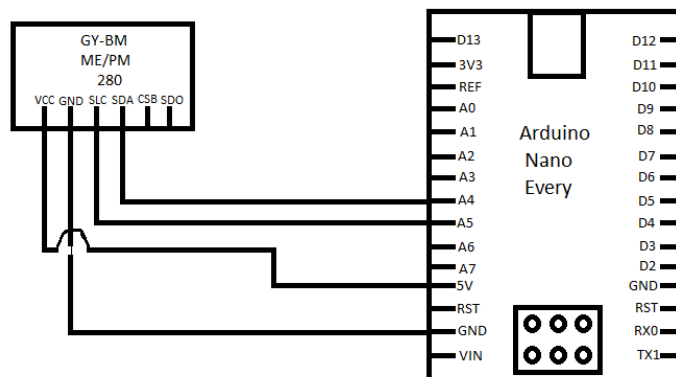
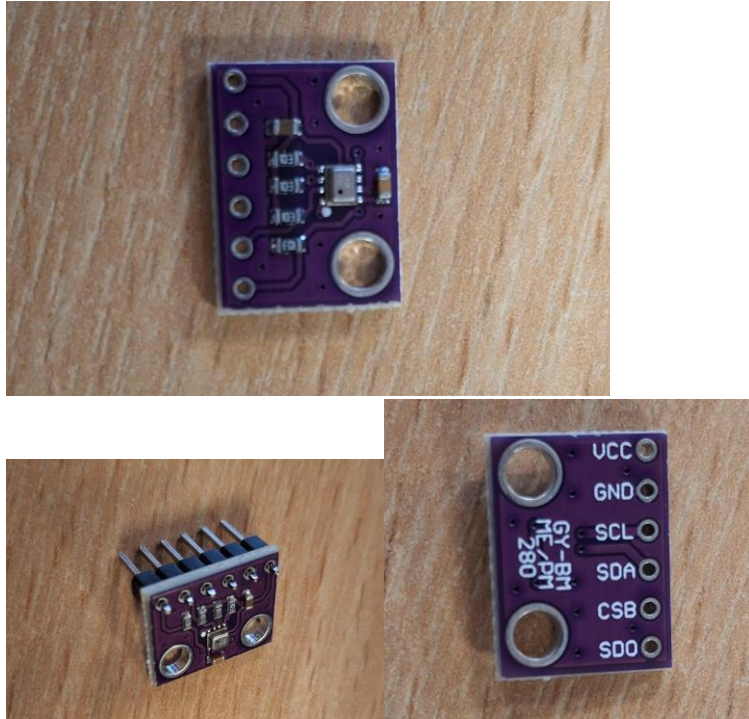
The Arduino Nano Every has a greater number of I/O Pins (20) as compared to UNO's 14. This provides us with a greater flexibility to connect sensors.

Clock speed was also important to us, so we found that the Nano Every provides us with 20MHz, which is faster than UNO's 16MHz.



##### Pressure and Temperature Sensor: GY-BMP280

- This sensor was chosen according to its great capabilities:
  - It has a good sensing range for temperature between the regions of  $-40^{\circ}\text{C}$ ...  $+85^{\circ}\text{C}$ . Its accuracy is within  $\pm 1.0^{\circ}\text{C}$ .
  - The pressure sensing range lies between 300-1100hPa. At sea level, the atmospheric pressure lies at around 1013 hPa, according to MetroSwiss and reaching a kilometre in altitude, as you gain altitude, the pressure is not linear in trend, however we can approximate this to be around 890hPa, which lies within the sensing range of the GY-BMP280 sensor.
  - This model can also calculate altitude with the change of pressure and is listed to possess an accuracy of  $\pm 1$  meter so it can also be used as an altimeter.
- The Photo and Connections of the Sensor to the Arduino Nano Every:

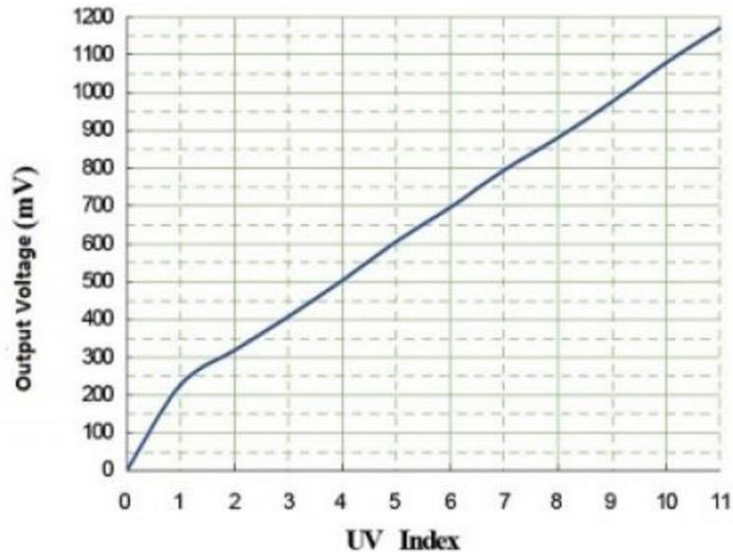


Together assembled into a breadboard for prototyping and testing the software, the hardware looks like this:

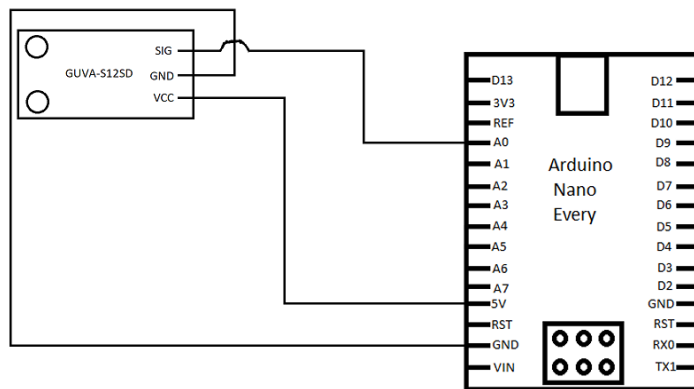
{Appendix Reference 4 – Breadboard & Sensors}

### **The Ultraviolet Light sensor:**

- The GUVVA-S12SD sensor is a sensor designed to detect the intensity of Ultraviolet light in the range of 200-370 nm at an operating voltage of 3-5V. This sensor outputs an analogue voltage but the Arduino nano every comes with an analogue-to-digital converter with a 10-Bit resolution.
- This model uses an analogue output that changes in accordance the intensity of UV light. This output varies between 0-1 volts so we can use a graph to calculate the UV intensity.



- Now, we can only determine the intensity of UV light during the launch in March. It is known that the intensity changes through the year so to determine an accurate prediction for the weather through using the UV intensity sensor, we would need to measure the UV intensity through the year, however we only have a chance to do this in March and the maximum UV intensity is a 2 in March in the UK, according to Weather2Travel, which is quite low. However, this can still be beneficial to determine the short-term weather around this time of year. This can give us information on cloud coverage reflection from snow or raindrops or any other reflective surfaces and the amount of ozone in the atmosphere. These would all be important variables needed to predict the weather in the short-term.
- This model is well compatible with the Arduino Nano Every and the diagram showing connections is listed below:



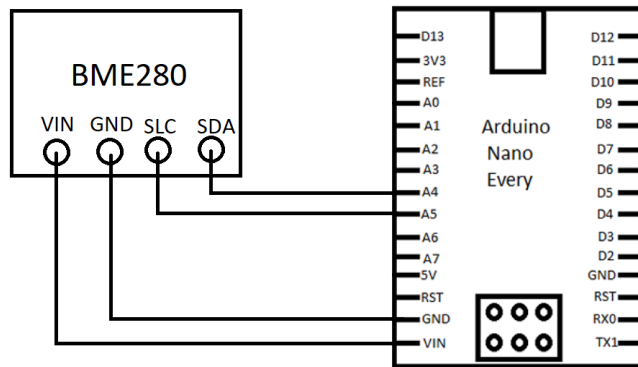
A few factors need to be considered, however. For example:

- We must calibrate the sensor to match the results around the area of launch.
- The temperature may affect the results of the guva-s12sd sensor. To see how the temperature will affect this sensor, we could use the temperature sensor that will also be in the can sat to account for this change.
- At higher altitudes, there is less atmospheric filtering so the UV intensity will increase.
  - In our data, we will need to account for this as to provide valid results in the end.

#### **Humidity Sensor:**

The BME280 is a pressure and humidity sensor combined. Its operating range is between  $-40^{\circ}\text{C}$ ...  $+85^{\circ}\text{C}$ , 300-1100 hPa. Response time is 1 second and its accuracy for humidity sensing is  $\pm 3\%$ , which makes it pretty accurate and accurate enough for our selected mission. The BME280 is compatible with the Arduino nano every and the circuit diagram for this is shown below:

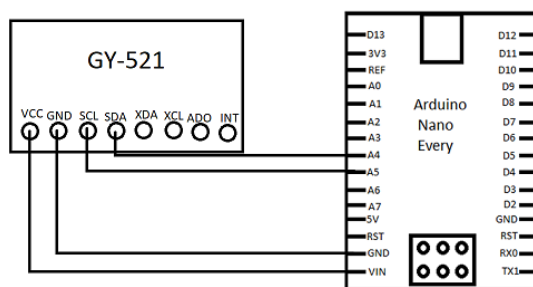




## Accelerometer:

The accelerometer will allow us to measure the acceleration in not just the y-axis (gravity), but it can measure the acceleration in one or more directions. These can include forces such as vibration.

The sensor I have decided to go with is the GY-521 model that can measure in 3-axis. The gyroscope range is  $\pm 250$ , 500, 1000, 2000°/s. The accelerometer range is  $\pm 2$ ,  $\pm 4$ ,  $\pm 8$ ,  $\pm 16g$ , according to the datasheet. The sensor is compatible with the Arduino nano every and is connected as shown below:



These sensors can be used in various ways to predict the weather and here is my plan:

- The GUVa-s12SD measures the Ultraviolet radiation intensity and can be used to analyse atmospheric weather conditions. It is important to note that:
  - Higher altitudes have a stronger UV intensity.
  - Lower than what was predicted readings at the high altitudes can mean an increase in ozone or pollution.
  - This helps analysing solar exposure, which affects weather stability.
  - We can map the clouds by considering these points:
    - Clear skies- Higher UV levels
    - Partially cloudy- Lower UV levels
    - Overcast- Large UV reduction
    - Thick clouds- Almost no UV
- GY-521 senses the vibrations with its gyroscope. If the CanSat shakes violently, there are high winds and signs of a storm incoming. As the CanSat falls, we can identify layers of wind turbulence, which can help us determine weather patterns.
  - Updrafts- rising air will be recorded on the accelerometer. These are called Thermals and indicate a convection current, which can lead to good weather.
  - It is probably a good idea to keep this sensor from vibrations that may be caused by the electronics.
  - Smooth motion- Calm weather
  - Turbulence- Strong winds (storm)
  - Rotations- Unstable air currents Updrafts- Thermals (Convection current)
  - Rapid temperature change- Weather shift.
- The BME280 can record the temperature and pressure, however the important bit is the humidity sensor. It can determine the moisture levels, which can indicate possible rain or fog.
  - If humidity is over 80% at a high altitude, there is possibly a cloud.
  - If humidity is increasing with increasing altitude, this can indicate cloud layers.
  - High humidity and a low pressure is a sign of a storm.
  - If there is a sudden temperature increase at a high altitude, this could be an inversion layer, which means fog.



- The GY-BMP280 has real time atmospheric pressure readings measured in hectopascals. The atmospheric pressure can vary with weather conditions with the statistics below:
  - High pressure (1020+), clear weather and good, stable conditions.
  - Low pressure (1000 and below), weather conditions are cloudy and wet, stormy and so on.
  - If there is a rapid pressure drop, it means a storm will be approaching, however we are unlikely to record this as this will require measuring for a longer period of time. It is worth to note that the bigger the drop in pressure, the more possibility of a storm approaching,

### 3.3 Software design

As a team, together, we have decided to use an 'Arduino' due to its better compatibility with our hardware designs and its higher efficiency which should ultimately lead to more accurate results.

We have decided to opt for the BMP280 sensors (as displayed above in the electrical design) to complete our primary mission of measuring temperature and pressure.

Other sensors such as the MH-Z19 NDIR, DHT22 and Figaro TGS 5342 sensors as well as a Geiger-Muller Tube may all be used deliver the secondary mission as well, however, these haven't been fully decided yet.

A flowchart below (on the left) shows how the CanSat may operate to deliver the primary mission and secondary missions (with **RunPrimaryMission()** and **RunSecondaryMission()** being predefined functions. The predefined **RunPrimaryMission()** function is shown on the right.

{Appendix 18 – Flowchart of full algorithm of CanSat Software}

{Appendix 19 – Flowchart of RunPrimaryMission() }

The **RunPrimaryMission()** again contains predefined functions of **CalculateTemperature()** and **CalculatePressure()**. A basic sequence of how they may run is shown below.

{Appendix 20 – CalculateTemperature() in-detailed flowchart}

{Appendix 21 – CalculatePressure() in-detailed flowchart}

Using the Arduino IDE and installing all necessary drivers, below is the starting code for our primary mission using the GY-BMP280 sensor to detect the temperature.

Firstly, we need to use the **#include** command to ensure we have all modules desired in the program:

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BMP280.h>

Adafruit_BMP280 bmp;
```

After declaring the Adafruit\_BMP280 we can next use the **setup()** function to initialize the Serial and anything else needed in the program.

The **Serial.begin(9600)** will allow us to start the serial and we can test this works by printing a simple string such as **"HELLO"** to the serial. In the Serial Monitor, we can see this is printed out and therefore we know that the code is working so far, and we can continue building the program.

```
void setup() {  
  
    Serial.begin(9600);  
    Serial.println("HELLO");  
}
```

We can then use the selection statement `if (!bmp.begin())` to return a Boolean value, determining whether or not the BMP has indeed been accessed. Using the `!` operator acting as 'NOT' in Boolean logic, the statement is only `True` if the BMP has not been accessed. By adding a few more print statements we can look at the Serial monitor again to see the result and using this information we can decide what to do next.

```
void setup() {  
  
    Serial.begin(9600);  
    Serial.println("HELLO");  
  
    if (!bmp.begin()) {  
        Serial.println("BMP cannot be accessed.");  
        while (1);  
    }  
    else {  
        Serial.println("BMP Accessed.");  
    }  
}
```

Providing this works, we can now move on to establish the sampling of the BMP. Using the `bmp.SetSampling()` command helps us to set and control how the data is sampled. This allows us to try and obtain the most accurate readings possible from its sensors so we can return these values and later then retain this accuracy in further calculations to make precise predictions.

{Appendix Reference 15, Code Snippet, setup() function for sensor software}

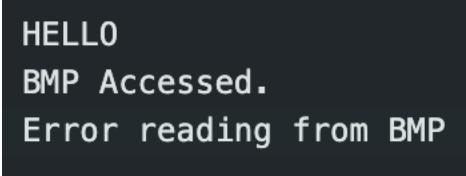
We can then start to set up our main `loop()` function declaring a value of our temperature. Using the inbuilt `bmp.readTemperature()` command, after sampling, we can easily obtain the temperature value read from the sensor and store it in a float variable respectively named.

```
void loop() {  
  
    float temperature = bmp.readTemperature();  
}
```

We can then use some more if statements to check the BMP is correctly reading the temperature. The `isnan(temperature)` function will take a parameter, in this case temperature and return a Boolean value as to whether the temperature value is 'not-a-number'. As we are looking for a numerical value, if this is returned as `False`, we know some error has occurred in the reading stage and therefore we can fix that. After this, if a number is indeed returned, we will print the temperature obtained from the BMP. By delaying for 1000ms afterwards we can ensure the readings are taken every 1 second for accuracy and precision.

{Appendix 14 – Software Code Snippet, isnan() function}

Here's what the serial monitor could look like on a test:



```
HELLO
BMP Accessed.
Error reading from BMP
```

This error occurred due to an error in the hardware, specifically the connections in the breadboard used for testing. To amend this, we will need to solder the components together to ensure proper connections and hopefully get an accurate reading from the sensor.

### 3.4 Landing and recovery system

CanSat may include a tracker to track its path, which would give us approximate GPS locations allowing us to retrieve the CanSat from the ground once launched as quickly as possible.

Systems to reduce the deceleration of the landing with the aim of preventing damage to internal components have been considered. These include spring systems and internal systems to isolate the components from the outer case with the aim of preventing vibrations from being transferred to internal components which may damage them.

By lining the inside of the CanSat shell, with softer material, we can reduce the force due to general physics principles such as the change of momentum given by the following equation:  $F = \frac{\Delta mv}{t}$

A softer, more absorbent material, rather than that of the Can Shell which needs to remain firm would increase the time taken for the change in momentum, thus increasing the denominator on the right-hand side and therefore the total Force would also decrease. This means the components inside the CanSat will feel less 'impact' from the CanSat landing and so will be less likely to break or be damaged upon the landing of the device.

The parachute may be used to increase surface area of the CanSat which would increase the air resistance and thus the drag force opposing that of gravity. This would allow us to reduce the terminal velocity which is reached and prevent further damage to components and slow down the rate of descent. The hardware lead will design a parachute that is heavy enough to slow down the CanSat's descent, but not light enough to suddenly increase the total Surface Area and increase air resistance thereby floating the CanSat away from ground control.

### 3.5 Ground support equipment

Ground control will use a Yagi antenna to receive the information from the duck antenna. We shall be using a UHF (Ultra High Frequency) range of 300 MHz to 3GHz because that is where the 'Sweet spot' falls under. This means that our CanSat will be able to transmit data with sufficient bandwidth without sacrificing range. Therefore, ground control should be able to receive information with little to no interference, so long as the CanSat does not surpass 1km.

Additionally, to power the APC 220 – which shall be connected to the Yagi antenna – we shall be using a battery, allowing us to transport the APC 220 anywhere within range of our CanSat, so that we don't lose power and aren't cut off from the system.

Instead of using 'FM [Frequency modulation]' or 'AM [Amplitude Modulation]', we shall be using 'FSK [Frequency Shift Keying] modulation' by purchasing an APC 220 transceiver for our CanSat, the reason being that it transmits more data to ground control from our CanSat.

{Appendix 16 – Arduino Uno's, power source (cell) and radio transmitter/receiver setup}

*This image shows the Arduino Uno (Blue board) being powered by a battery and plugged into an APC 220 so that it sends and receives signals to the computer.*

**Yagi antennas:**

{Appendix 17 – Yagi Antenna diagrams and pictures, labelled}

Boom Length: 345.55 mm

Width: 116.84 mm

Height: 66.04 mm

{Diagrams Sources Referenced in Appendix Two}

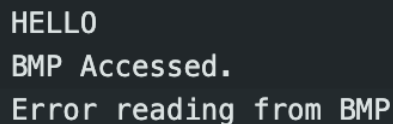
### 3.6 Testing

To meet the requirements, we'll make sure to test and measure our CanSat so it can be in line with the rules.

We would also rigorously test the different elements of our CanSat separately using iterative testing to find any issues and tend to them immediately.

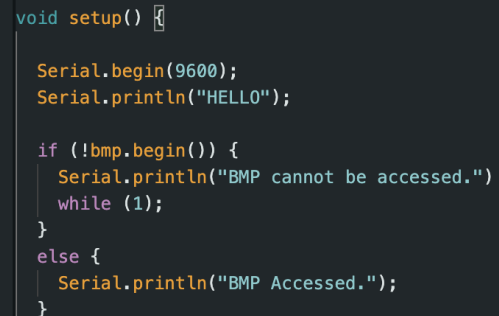
Software testing would consist of running different functions of the code and ensuring they work with the hardware to return the correct results. It'd consist of both debugging and editing to ensure all syntax and logical errors are removed where possible. Test data could be used to ensure that all mathematical, mechanical and statistical calculations are being executed accurately and effectively.

Print statements after each new operation can be used to output things into the **Serial Monitor** based on whether conditions have been met regarding the hardware and software working together correctly. For example, the code below on the right contains various print statements, 'HELLO' after beginning the Serial to see if the program starts running then some more after the `bmp.begin()` command to see whether or not the BMP Sensor has correctly been accessed. We originally came over this issue with the following Serial Monitor log below on the left:



```
HELLO
BMP Accessed.
Error reading from BMP
```

This shows the BMP was accessed however we had an error reading the temperature from the BMP. After testing the hardware components, we discovered the issue was with the breadboard we used to test it with. The connections were faulty meaning the sensor could not be properly accessed and read from. This means that we needed to use proper soldering with the components to ensure they were connected properly and that more testing could be undertaken with proper readings.



```
void setup() {
  Serial.begin(9600);
  Serial.println("HELLO");

  if (!bmp.begin()) {
    Serial.println("BMP cannot be accessed.");
    while (1);
  }
  else {
    Serial.println("BMP Accessed.");
  }
}
```

Hardware testing would work alongside the software testing in ensuring that the correct values are returned. By testing wires and connectors one-by-one we can isolate if there are any bottlenecks or weakness in the physical system. As we're using 'male' connectors, we could pair them with known 'female' connectors that we are sure work to individually test them.

Whilst in the final iteration we'd solder hardware components together using permanent soldering techniques, for testing and prototyping purposes this is unnecessary and would cause more issues than benefits. We can use a device such as a breadboard to connect components via a plastic box which holds a matrix of electrical sockets of a size suitable for gripping thin connecting wire allowing components and integrated circuits. We have set this breadboard configuration up with the GY-BMP280 (on the left) and the Arduino Nano Every (on the right), this can be seen below:

{Appendix Reference 4}

We can test parachutes and landing systems from a high point like a balcony or building allowing us to simulate a somewhat similar environment to the launch day when the CanSat is falling. Adding extra weights and masses will allow us to simulate the  $10\text{ms}^{-2}$  freefall rate.

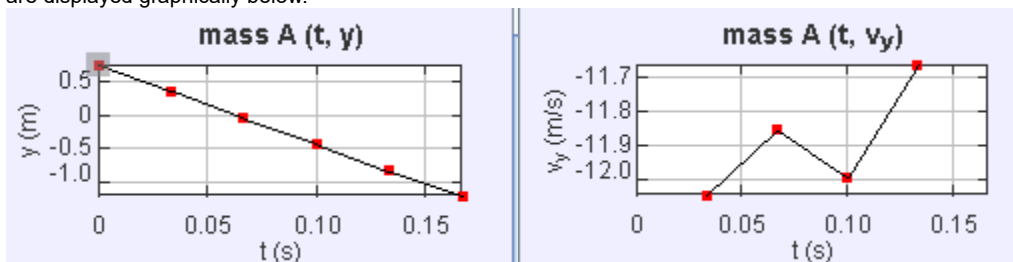
Below you can see the two screenshots of the video editor from the video recorded of the parachute drop test, earlier proposed, from the top of a low balcony. Using 2 one-meter rules, allows us to calculate the velocity of the parachute to ensure it fits with the CanSat guidelines {CDR Appendix Reference 1}.

{Appendix 13 – Parachute Initial Drop Test, Start Frame}  
{Appendix 14 – Parachute Initial Drop Test, End Frame}

From the appendix references we can see images of the parachute drop test from the first and last frame of the 2m scale.

Counting frame-by-frame, the number of frames between the first and second screenshot is 18 frames. Use the video details and knowing the video was recorded in 120fps we can use re-order to get:  $time = \frac{frames}{fps}$  and so, the time taken to fall 2m is (18 frames / 120fps) = 0.15 seconds. Using the formula  $S = vt$  rearranged into  $v = \frac{S}{t}$ , the velocity of the parachute is (2 / 0.15) = 13.33 m/s (to 2dp).

A rough estimate using the video analysis software “Tracker” also gives the velocity to be roughly 12m/s, seen through the corresponding displacement-time and velocity-time graphs for the y components of the dummy CanSat during the drop test, which are displayed graphically below.



This fits perfectly within the CanSat guidelines {CDR Appendix Reference 1} between 10m/s and 15m/s. The bag the parachute is connected to was also measured using a mass balance to 342 grams which is also within the CanSat guidelines {CDR Appendix Reference 1}. By using a temporary mass whilst the full CanSat is not developed, this allows us to simulate having a CanSat on it, using a mass to simulate the acceleration which the actual device would go through.

Communications test would first look like basic radio transmission tests as we attempt to send very basic signals from the transmitter to the receiver. The APC 220 comes with both a transmitter and a receiver. For initial testing purposes, the transmitter would be connected to a desktop computer and the receiver would be connected to an Arduino (Nano Every). We'd put them right next to each other and using basic code and the [<SoftwareSerial.h>](#) module to communicate via Universal Asynchronous Receiver-Transmitter (UART) serial systems first transmit some basic messages.

For the transmitter, our very simple testing code looks like this:

{Appendix Reference 12: Communications Transmitter Code via APC220}

Using the pins to connect to the APC220 and the [apcSerial](#) as well as the normal [Serial Monitor](#) (both at a convenient 9600 baud rate) we can print strings using the two different systems to check for logic errors. For example, if we look at the Serial Monitor and see the following, we know that the transmitter code has worked and been executed as desired:

```
APC220 Transmitter Ready
Message Sent: Hello this is a test run
```

Therefore, if we look at the apcSerial via the receiver utilising an Arduino Microcontroller, and don't see the output **'Hello this is a test run'**, we can safely isolate the receiver system as the 'bottleneck' not correctly performing its task. By also sending a message every 2000ms (2 seconds) we can ensure that perhaps a one-off internal hardware or software 'spurious' error, perhaps in compiling, that may occur is not happening as we should receive the message every 2 seconds with correct systems in place. The receiver, using an Arduino rather than a desktop computer for more realistic simulation of the CanSat launch, would run the code below for simple tests, using the built-in [apcSerial](#) commands for ease. The '\n' string represents that of a new line, by using this we can ensure that all data transmitted is separate and that no unwanted data is received. Each section of data sent, in the test runs, is separated by a new line anyway so this won't stop using receiving any of the data which we need to receive for testing purposes. Once again use many Serial commands and printing lines, we can easily undergo debugging via the in-built Arduino Serial Monitor. The monitor

should read the **'APC220 Receiver Ready'** and **'Received: {receivedMessage}'** strings and if this doesn't occur, we can easily isolate any issues which we may find in running our tests:

{Appendix Reference 11: Communications Receiver Code via APC220}

### 3.7 Overall testing for launch

Student E, as the software lead, decided it would be useful to create, in Python from scratch only using a library for graphics, a fully functional CanSat simulator. This allows us to simulate the flight of a CanSat both with and without a parachute so we can see how it may roughly behave under various conditions providing flight statistics and data for processing and understanding:

The (very untidy) **source code** for this can be found here:

<https://bit.ly/CanSatSim>

By providing a basic GUI, it allows us to change values which are critical in the flight of the CanSat such as the Parachute Upthrust, Horizontal Wind and Mass.

This allows us to simulate many different scenarios and find out whether our current values will work well and which values may work best for the CanSat.

{Appendix Reference 7 – CanSat Simulator Software, GUI Main Menu}

The program then allows us to see, in real-time, the falling of the two CanSat's side by side. One has a parachute, and one doesn't, this allows us to see the difference in flight time depending on the Parachute allowing us to see how resistant the parachute needs to be to the motion of the CanSat. This is crucial as it gives us a visual representation as well as a scale to put it well into perspective.

{Appendix Reference 8 – CanSat Simulator Software, Scaled Flight}

The simulation then very helpfully provides statistics calculated from the flight such as time, velocity and acceleration. This is extremely helpful as it allows us to tweak things and it allows us to compare our simulated flight with the CanSat guidelines {CDR Appendix Reference 1} so that we know if we are sticking to the correct simulation statistics. A force diagram is also shown at the bottom so we can understand the forces acting on the CanSat and how either maximize or minimize these to meet our interests.

{Appendix Reference 9 – CanSat Simulator, Flight Statistics Display}

Finally, we can use the 'Get Graphs' button at the bottom to obtain the velocity, acceleration and displacement time graphs (as shown below in red, green and blue respectively). Whilst there are no values, it provides another good visual representation as a kind of benchmark to see if we match with how these values should change over time.

{Appendix Reference 10 – CanSat Simulator, Graphical Flight Statistics Display}

This overall application is handy for running quick simulations, obtaining rough yet accurate values and seeing what to tweak to make the CanSat meet the guidelines {CDR Appendix Reference 1} necessary and to ensure a safe and well thought out flight. However, this does not simulate real-life, and we still need to use physical tests to fully understand how the CanSat will operate, and the real physical changes needed. The real-life overall testing for launch can be found below.

To simulate the overall testing for launch, we can first test all components individually, hardware, software, communications and design. Once we are happy with all individual attributes of the CanSat, we can then start to piece it together.

Firstly, we can integrate hardware with software combining code with sensors and a microcontroller. We can perform basic tests initially and then run the actual primary and secondary missions on ground to hopefully receive the correct data values both pre and post processing. This will allow us to randomly select some data points and process them ourselves hopefully matching with the post-processed data retrieved from the CanSat.

Providing all runs well, we can then add the communications systems into it sending the received data via radio signals and transmission to our ground-base. Checking these values match up and are properly transmitted with limited latency and delay, we can continue to run test over the possible ranges of transmission and find the furthest point from the CanSat that the receiver can accurately detect signals. We can also use multiple frequencies in testing the transmission to ensure that any unexpected problems, e.g another team using the same frequency as us can be dealt with without panic on the actual launch day.

After all testing of other individual components are done, we can finally integrate the design elements. Firstly, by placing all the components inside the Can shell, we can check the sizes meet our expectations and calculated values. We can then use a mass balance to see if the total mass is firstly, to match our calculations and secondly within the CanSat guidelines {CDR Appendix Reference 1}. If it is out, it will most likely be underweight meaning we can alter this by adding additional masses to either fit our calculations or fit the guidelines {CDR Appendix Reference 1} set. This will allow us to carry on to the next stage of our overall test by adding the parachute. By using masses to simulate the tension in the strings we can firstly test the tensile strength of the connections between the CanSat and the parachute to ensure that they will be able to withstand the forces predicted during the flight time. Once done, we will connect everything together for some drop tests of the whole CanSat, fully assembled.

By using a soft/grass ground much like the field the CanSat may land in we can further simulate the conditions of an actual launch. We can test our hardware and durability here with all our components to check they work, firstly we can just test the shell to not damage the components. After that is completed and we can be more certain that the internal components won't get overly damaged, we can start to add them back in and test the CanSat model. We can use the feedback from these tests to try and add to the model and fix any issues which we find.

We can keep re-running this process tweaking and adjusting elements till we reach our desired results where we can then make any final polishes until we have a full working product that we are ready with. Whilst it's unfeasible to be able to drop the CanSat from 400m due to the inaccessibility of such a height locally without causing risks to people below, we can certainly simulate these conditions on a much smaller scale and see how it acts, allowing us to come to rough conclusions which would be enough to provide any information about inaccuracies and things to change. To look for more minor errors, whilst not necessarily 100% accurate, we can extrapolate the data past the range of data we have and use mathematics and modelling via functions to simulate how it would act on a larger scale.

### **3.8 Evidence of CanSat build**

{Appendix Reference 3 – CanSat shell iteration 5}

{Appendix Reference 4 – Breadboard and Electronics}

## **4 OUTREACH PROGRAMME**

We have started our outreach program using social media to increase our exposure and allow people to recognise us as a team as well as the benefits of the CanSat itself.

We have created an Instagram account where we create short 'reels' of tests and builds showing our successes and achievements as we progress through the competition. We also post pictures and videos, documenting the process of our builds, not only highlighting our successes but also the actual journey along the way creating a realistic image in an age of many faked social media posts.

Whilst using social media for exposure is great, it's hard to show and explain our thought process properly. To do this, we can make an assembly or presentation which we could direct towards younger students, specifically locally or in our school. This will allow us to spread a proper message of what CanSat involves and to show people the true experience which many miss out on due to prejudice of the competition.

I, Student E, personally remember being delivered an assembly about CanSat from past candidates. This was interesting however much wasn't explained, and I was left with many questions which resulted in me doing my own research about the competition. Whilst this was okay, many would simply dismiss it as they didn't receive the full picture of what CanSat was all about. So, to combat others feeling the same way, our assembly will be more in detail whilst remaining fun and interactive.

**Our assembly covers the following topics:**

- Introduction (What is CanSat?)
- Evidence of the build (Pictures and videos)
- The actual launch (testing and launch day footage)
- A simple problem (regarding physics how they'd solve it)
- Any questions (leaving contact details for those interested to find out more)
- 

We would also show the appeal in terms of the Silver Industrial Cadets award and how employers recognise the CanSat. Careers in the ESA and similar fields could also be presented allowing us to show how even after the competition people can continue this workflow and pursue their goals in similar aspects.

## **Appendices:**

### **Appendix 1 – Requirements:**

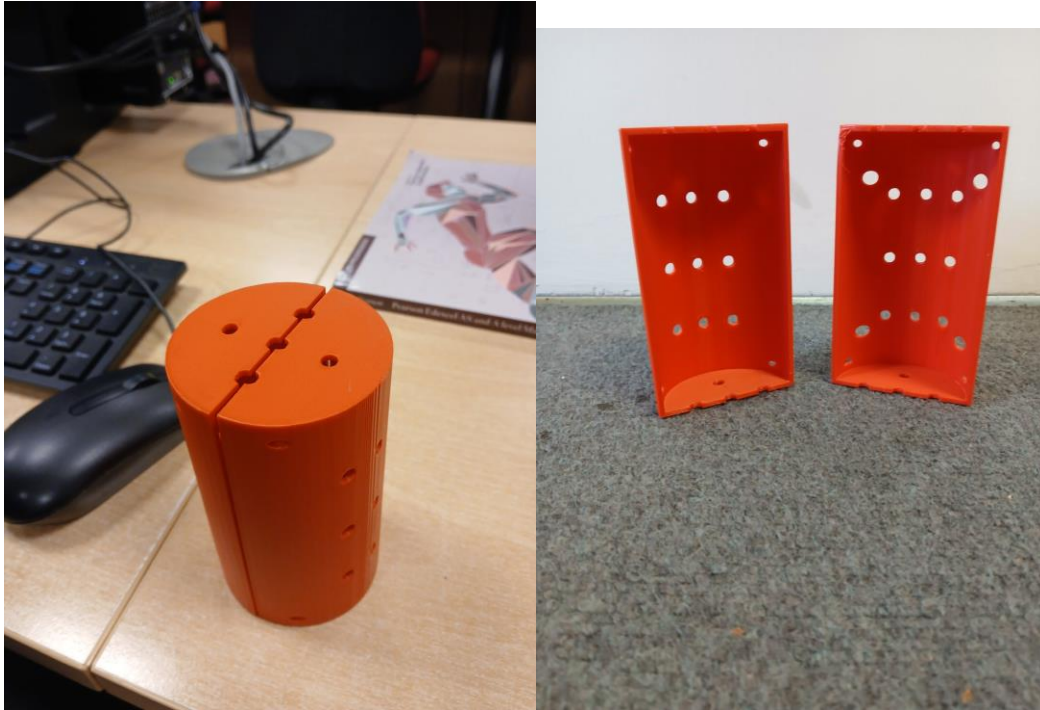
[CanSat Requirements {ESA Official Website}](#)



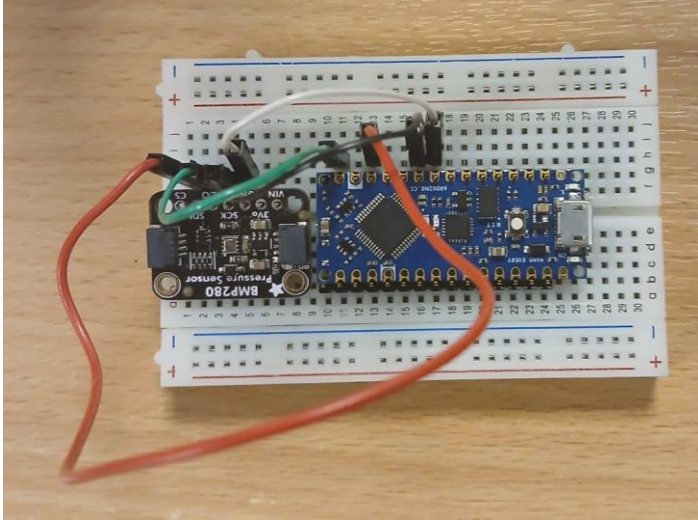
## Appendix 2 – Antenna Diagrams:

### [Yagi Antenna Diagrams](#)

## Appendix 3 – Can Shell Iteration 5



## Appendix 4 – Breadboard



#### Appendix 5 – Processing Data Python Code

```
'''PROCESSING.py

- Turns a .txt file of data into plottable co-ordinates of said data against time

'''

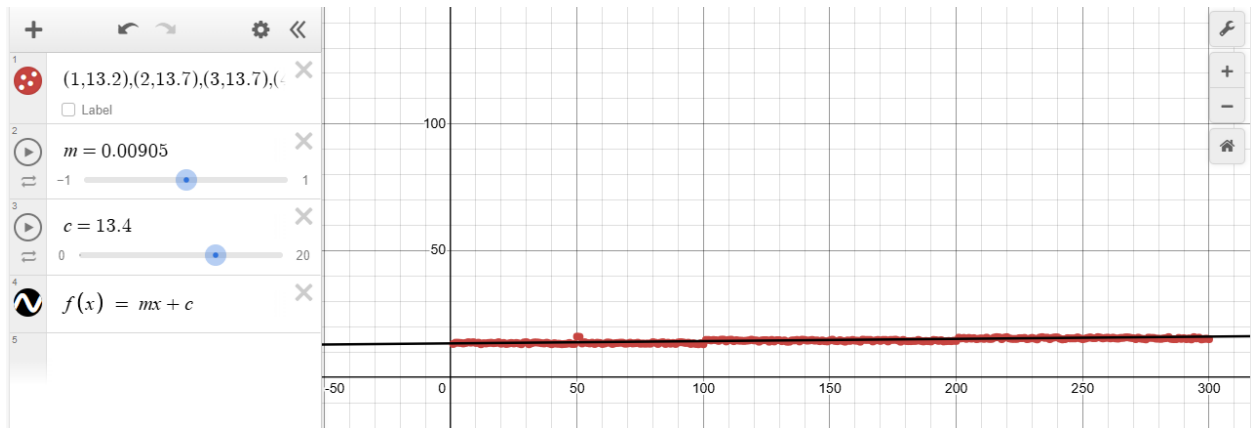
PROCESSED_VALUES = []

with open('TEMPERATURES.txt', 'r') as DATA:
    VALUES = DATA.read().split('\n')

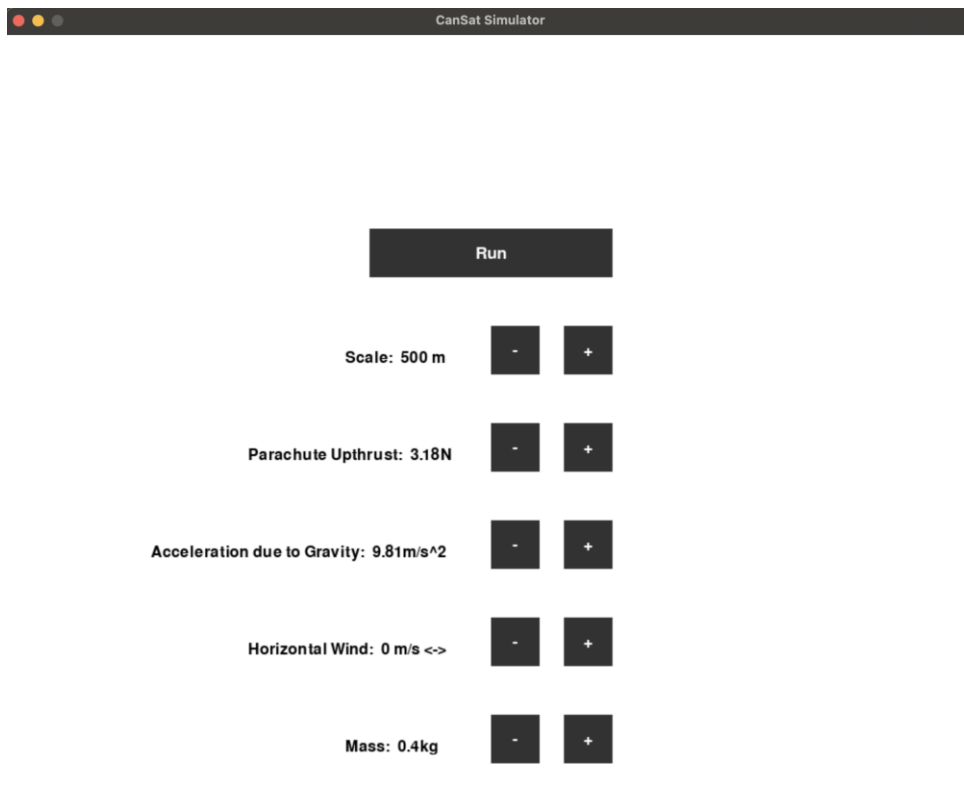
for INDEX in range(1, len(VALUES) - 1):
    PROCESSED_VALUES.append((INDEX, float(VALUES[INDEX])))

print(PROCESSED_VALUES)
```

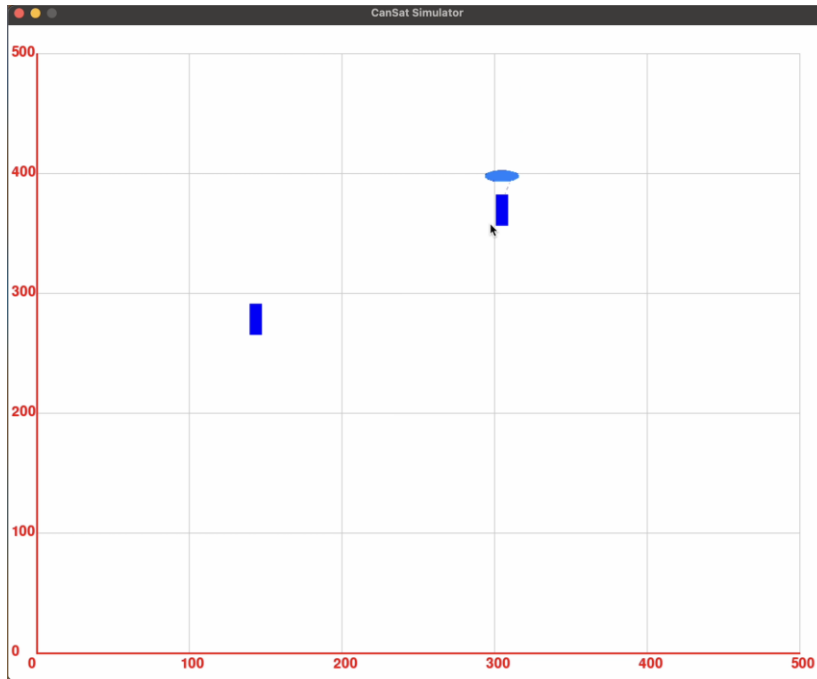
#### Appendix 6 – Desmos Graph of Processed Data



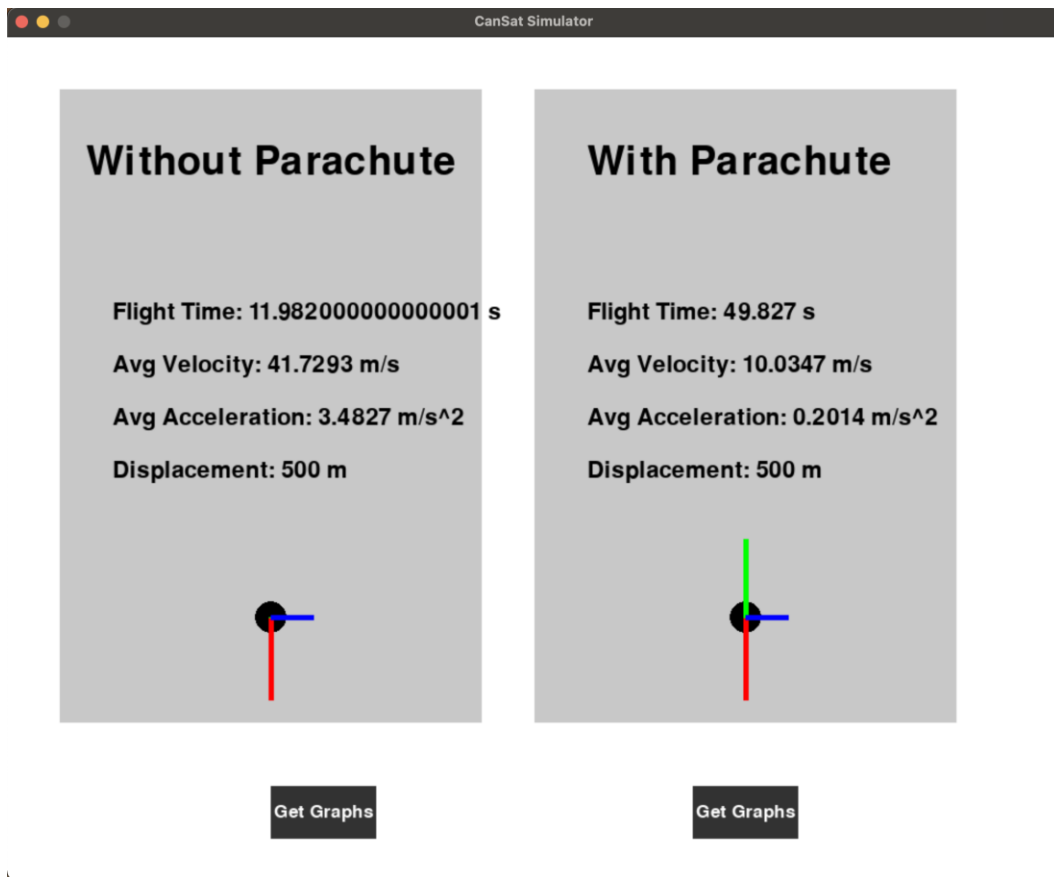
## Appendix 7 – CanSat Simulator Software, GUI Main Menu



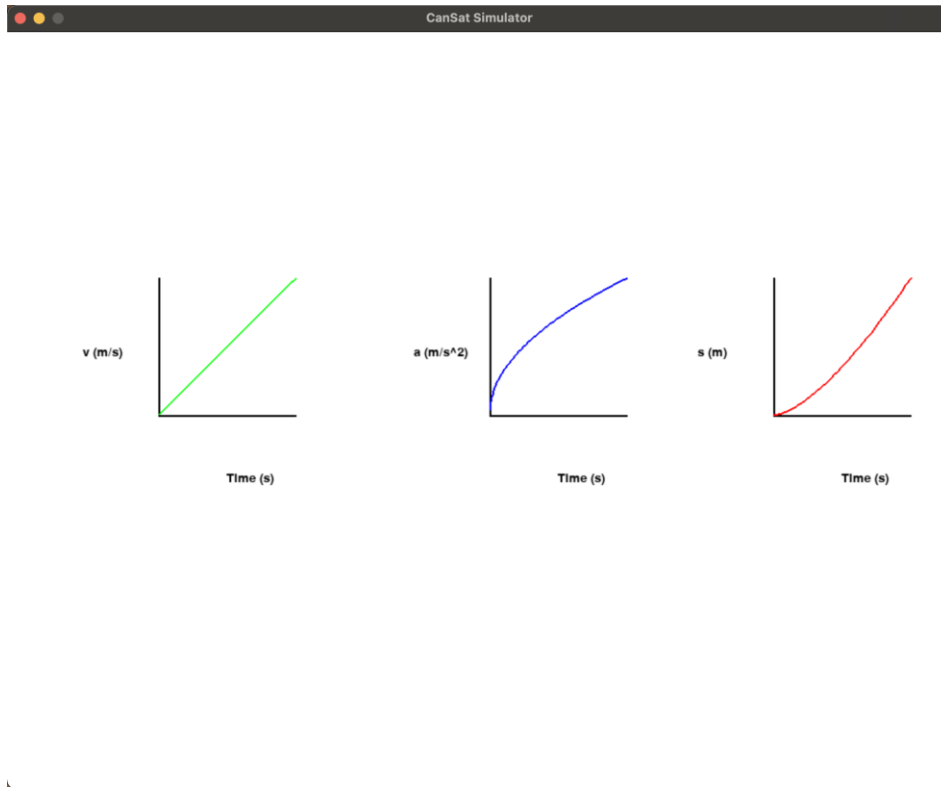
## Appendix 8 – CanSat Simulator Software, Scaled Flight



Appendix 9 – CanSat Simulator Software, Flight Statistics Display



Appendix 10 – CanSat Simulator Software, Graphical Flight Statistics Display



#### Appendix 11 – Communications Receiver Code via APC220

```
#include <SoftwareSerial.h>

#define TX_PIN 10 // APC220 RX (Data In)
#define RX_PIN 11 // APC220 TX (Data Out)

SoftwareSerial apcSerial(RX_PIN, TX_PIN);

void setup() {
  Serial.begin(9600); // Serial Monitor
  apcSerial.begin(9600); // APC220 baud rate

  Serial.println("APC220 Receiver Ready");
}

void loop() {
  if (apcSerial.available()) {
    String receivedMessage = apcSerial.readStringUntil('\n');
    Serial.println("Received: " + receivedMessage);
  }
}
```

## Appendix 12 – Communications Transmitter Code via APC220

```
#include <SoftwareSerial.h>

#define TX_PIN 10 // Connect to APC220 RX
#define RX_PIN 11 // Connect to APC220 TX

SoftwareSerial apcSerial(RX_PIN, TX_PIN); // Create SoftwareSerial object

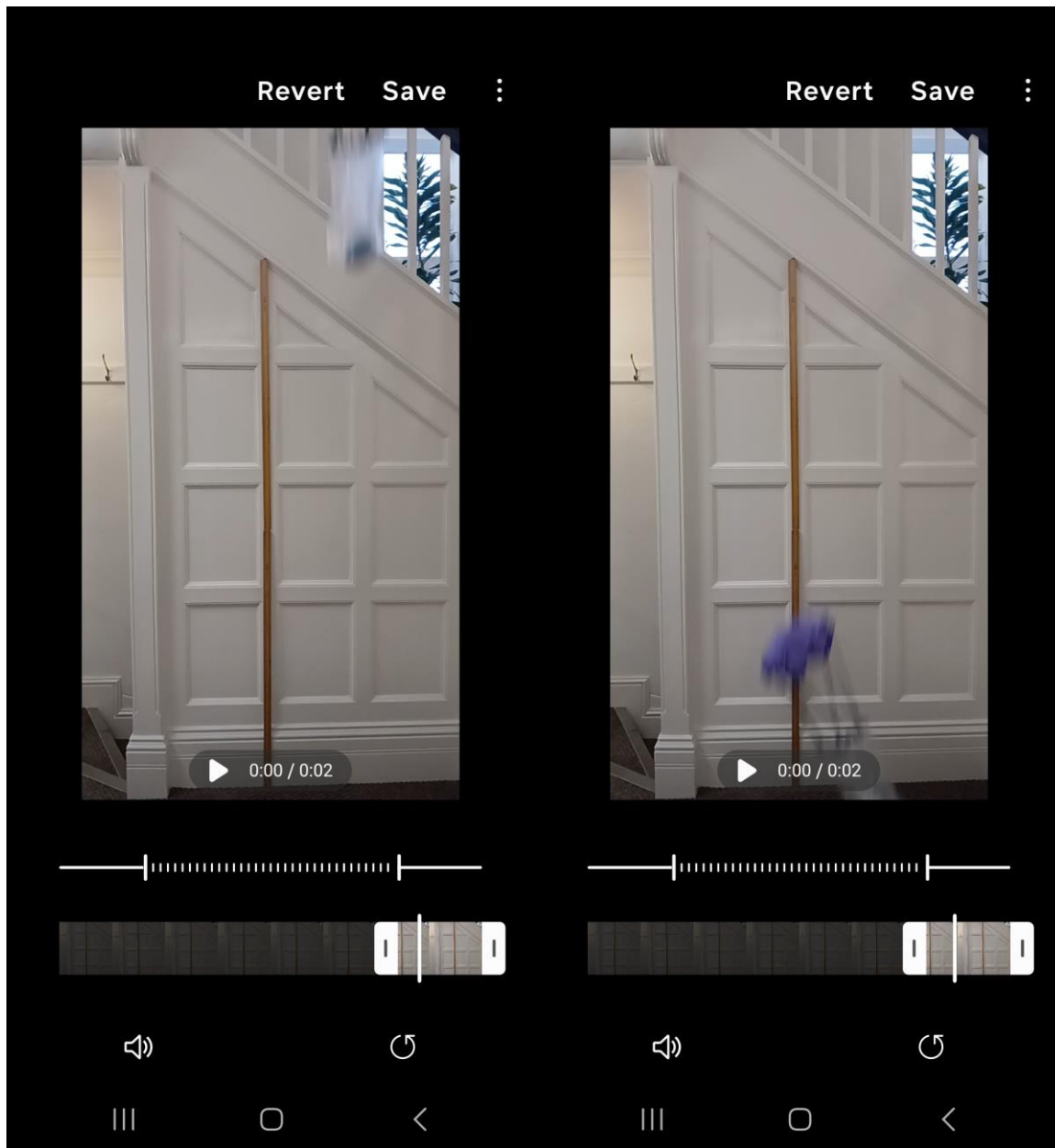
void setup() {
    Serial.begin(9600); // Serial Monitor
    apcSerial.begin(9600); // APC220 Module baud rate

    Serial.println("APC220 Transmitter Ready");
}

void loop() {
    String message = "Hello this is a test run";
    apcSerial.println(message); // Send the message via APC220
    Serial.println("Message Sent: " + message);
    delay(2000); // Send every 2 seconds
}
```

Appendix 13 – Parachute Initial Drop Test, First Frame

Appendix 14 – Parachute Initial Drop Test, Last Frame



Appendix 14 – Software Code Snippet, isnan() function

```
if (isnan(temperature)) {  
    Serial.println("Error reading from BMP");  
} else {  
  
    Serial.print("Temperature");  
    Serial.print(temperature);  
}
```

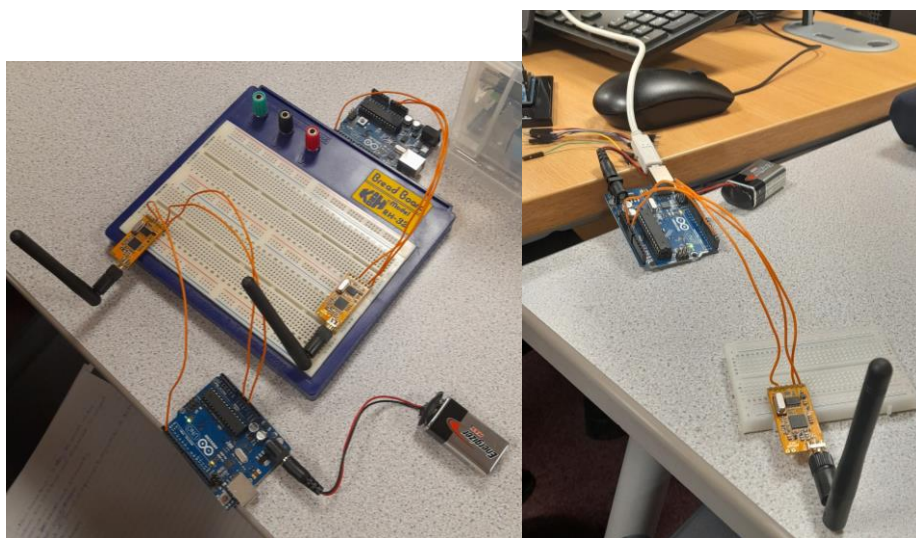
```
void loop() {  
  
    float temperature = bmp.readTemperature();  
  
    if (isnan(temperature)) {  
        Serial.println("Error reading from BMP");  
    } else {  
  
        Serial.print("Temperature");  
        Serial.print(temperature);  
    }  
  
    delay(1000);  
}
```



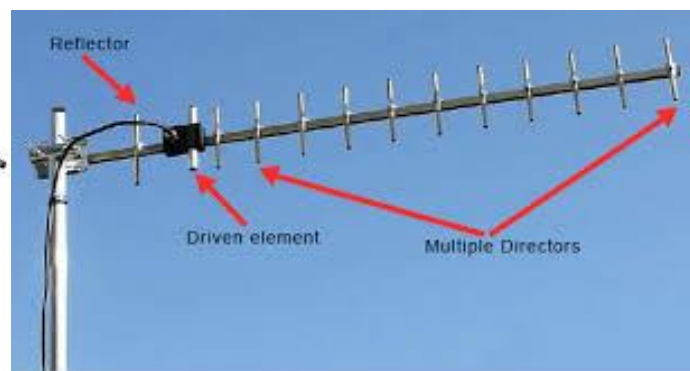
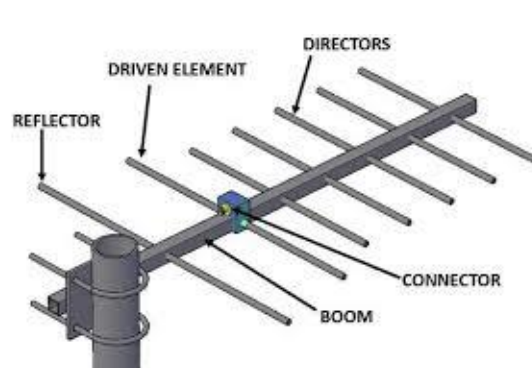
#### Appendix 15 - Code Snippet, setup() function for sensor software

```
void setup() {  
  
  Serial.begin(9600);  
  Serial.println("HELLO");  
  
  if (!bmp.begin()) {  
    Serial.println("BMP cannot be accessed.");  
    while (1);  
  }  
  else {  
    Serial.println("BMP Accessed.");  
  }  
  
  bmp.setSampling(Adafruit_BMP280::MODE_NORMAL, //Operating  
                  Adafruit_BMP280::SAMPLING_X2, //Temperature oversampling  
                  Adafruit_BMP280::SAMPLING_X16, //Pressure oversampling  
                  Adafruit_BMP280::FILTER_X16, //Filtering  
                  Adafruit_BMP280::STANDBY_MS_500); //Standby Time  
}
```

#### Appendix 16 - Two Arduino Uno's, power source (cell) and radio transmitter/receiver setup

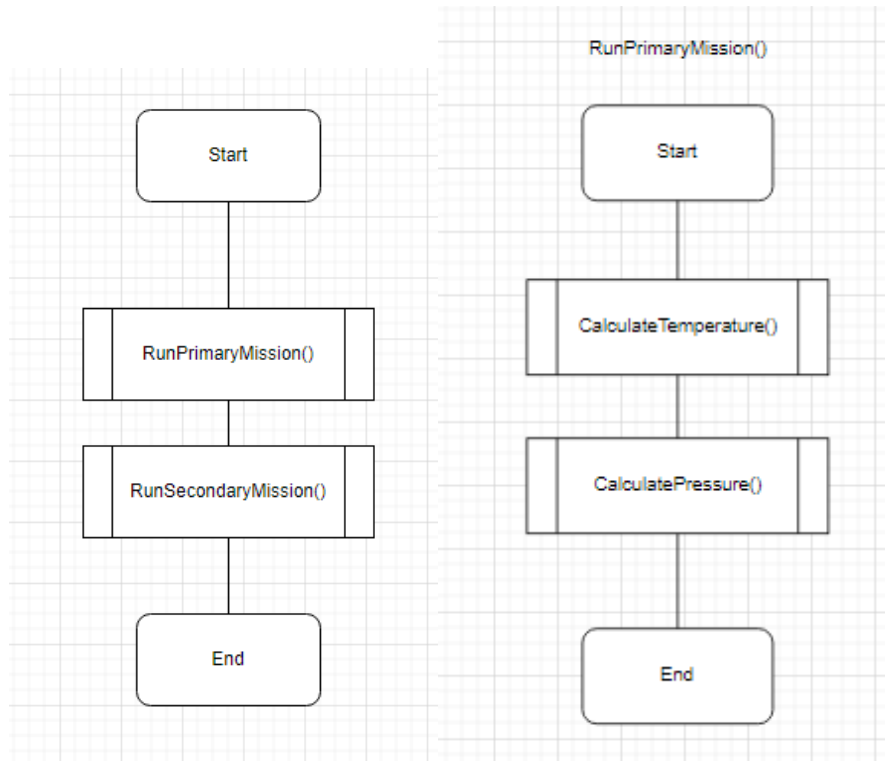


#### Appendix 17 – Yagi Antenna diagrams and pictures, labelled

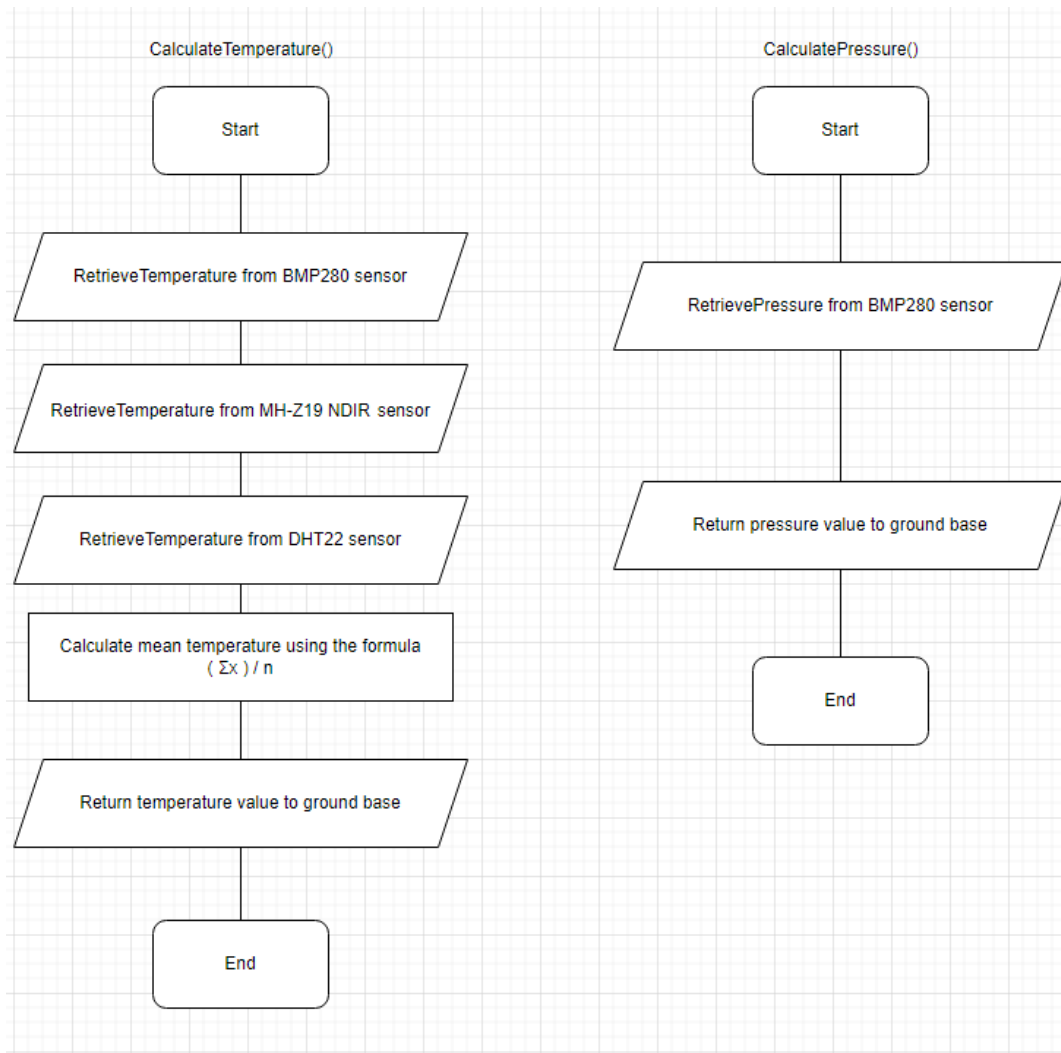


Appendix 18 – Flowchart of full algorithm of CanSat Software (left)

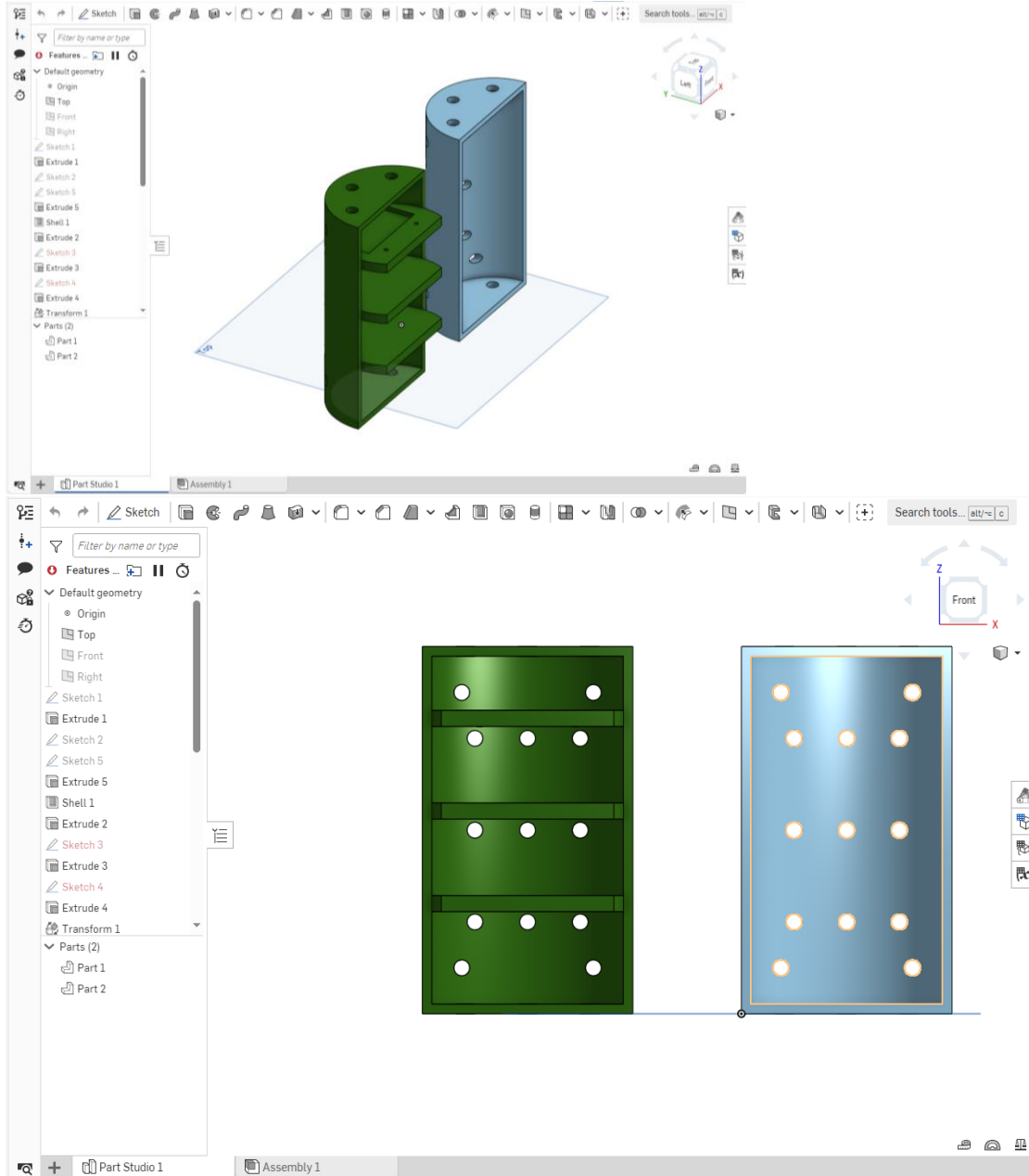
Appendix 19 – Flowchart of RunPrimaryMission() (right)

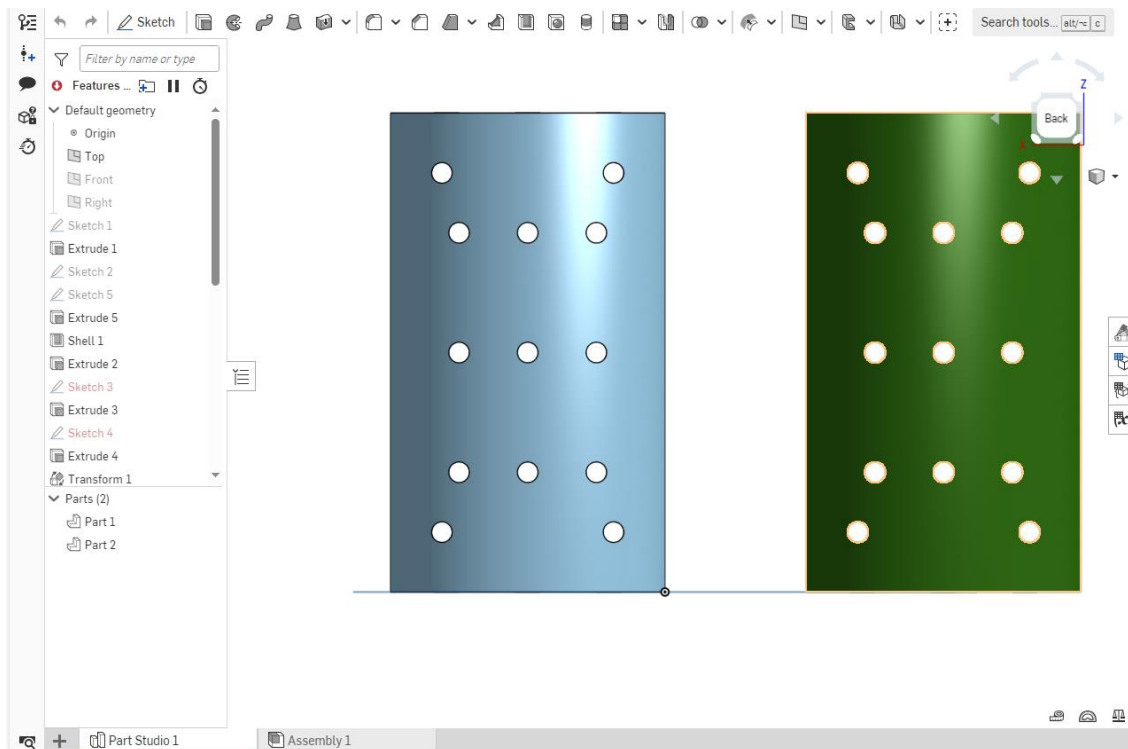


Appendix 20 – CalculateTemperature() in-detailed flowchart & Appendix 21 – CalculatePressure() in-detailed flowchart



## Appendix 21 - CanSat Shell Design Iteration 6





Appendix 22 – Budget Tracker Screenshot

URL Link to Item	Quantity Needed	Total Expected Cost	Required Delivery Date
<a href="#">BMP280</a>	1	<del>£7.57</del> £13.56	Received 9th October.
<a href="#">AZDelivery 3 x GY-BMP280 I2C IIC SPI BMP280</a>	1	£4.49	ASAP
<a href="#">Arduino Nano Every (Single Board) [ABX000]</a>	1	£13.95	ASAP
<a href="#">Adaptor</a>	1	£5.09	ASAP
<a href="https://www.amazon.co.uk/Prym-Canvas-rin">https://www.amazon.co.uk/Prym-Canvas-rin</a>	1	7.42	ASAP
<a href="#">6pcs Reflective Tent Guy Ropes - 5mm Tent</a>	1	13.99	Monday 16th December 2024
<a href="#">Connecting Wires</a>	n/a	£1	N/a
		58.50	

Appendix 23 – Item Order and Purchase Screenshot

Date Ordered	Status
Ordered 10th Sept	Received
Ordered 31st Oct	Received
Ordered 31st Oct	Received
Ordered 4th Jan	Received
Ordered 4th Jan	Received
Ordered 4th Jan	Received
Ordered 6th Jan	Received