THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

Altitude and Velocity Estimation of a Model Rocket

LILLIE MOHN
SPRING 2026

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Mechanical Engineering
with honors in Mechanical Engineering

Reviewed and approved* by the following:

Matthew Rhudy
Associate Professor of Engineering
Thesis Supervisor

Bryan Wang
Teaching Professor of Biology
Thesis Honors Adviser

Cesar Martínez-Garza
Associate Professor of Mathematics
Thesis Faculty Reader

* Electronic approvals are on file.

ABSTRACT

The goal of this project is to develop a low-cost rocket altitude sensor for students and beginner rocketeers. A sensor package consisting of a pressure sensor, a GPS, two inertial measurement units (IMUs), and a datalogger were tested on a high-power rocket. The sensors were coded using Arduino. Other measurements taken included altitude and speed using a JollyLogic AltimeterTwo and altitude using an Estes AltiTrak. Results from the custom sensor package were compared against the AltiTrak and OpenRocket simulation data. The AltimeterTwo did not produce a reading. Results were also run through an Unscented Kalman Filter (UKF) to estimate speed. From this flight, it was determined that the GPS was not a reliable altimeter, while the pressure sensor was more accurate and easier to use. Results from the two IMUs closely resembled one another. Speed estimation from the UKF somewhat matched the shape of the simulated speed curve, but more testing was required.

Then, a smaller sensor package was made with just the datalogger, the pressure sensor, and the smaller of the IMUs. This streamlined sensor was tested in a series of low-power flights to resemble launch conditions of the target audience. Results were run through a UKF, and were compared against the OpenRocket simulations, the AltimeterTwo, and the AltiTrak. Altitude from the pressure sensor closely matched the simulated altitude, while the AltimeterTwo consistently read higher altitudes, and the AltiTrak consistently read lower altitudes. Speed estimation from the UKF was not reliable, and varied significantly between flights.

Removing the IMU, which is not necessary for altimetry, the final sensor package costs $36.30. This is a significant price cut from the $79.95 cost of the JollyLogic. For simple altitude estimation, the sensor package developed in this work is a viable option for beginner rocketeers.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank several individuals for their contributions to this project. First, this project would not have been possible without the support of my Thesis Supervisor, Dr. Matthew Rhudy. His knowledge of sensor data combination, as well as his overall guidance and feedback have been invaluable to this work, and I am incredibly grateful for his mentorship.

I would also like to thank Dr. Bryan Wang and Dr. Cesar Martínez-Garza, my Honors Advisor and Faculty Reader. They have been very helpful in guiding me through the university's thesis-writing process.

This project would not have been the same if it were not for Dr. Garza, Rafael Uhle, and the Berks Rocketry Club. From teaching me how to build my first model rocket in freshman year, to now guiding my high-power endeavors, these individuals have consistently encouraged me to continue learning and share my love of rocketry with others.

I am thankful for the assistance of Mark DeChristopher, who aided me in both the construction and launch of my rockets for this project. In addition to lending his 3D-printing skills over the course of the year, he was always happy to help whenever I needed a second set of hands to epoxy, sand, and navigate muddy launch sites.

I gratefully acknowledge the support of the Penn State Berks Research Development Grant, who funded the electronic components used in this research. The Berks Rocketry Club was also generous enough to provide some rocket construction materials, for which I am also appreciative.

Finally, I would like to thank my friends and family for their love and encouragement. They continually push me to make my dreams a reality, and are the reason I am where I am today.

Chapter 1

INTRODUCTION

An Introduction to Rocketry

Young or old, novice or experienced, rocketry provides a fun and exciting way for anyone to explore aerospace concepts. When first being introduced to model rockets, many people start with predesigned kits, where they can learn the process of constructing a rocket. These kits have simple instructions and short build times, allowing them to be completed within a sitting or two. Then, advanced kits allow users to explore more complicated construction techniques and rocket builds.

However, construction technique is of low importance if the rocket is poorly designed to begin with. This is not a concern when building from kits, as manufacturers are experienced in rocket design and simulation. Still, simulating these rockets at home is a good way to become familiar with necessary software and see what parameters go into a rocket's design. When designing a rocket from scratch, though, it becomes important to be well-versed in the simulation software. This allows users to test how changes in the rocket will affect factors such as stability, air friction, and weight, all of which affect the performance of the rocket.

After gaining experience, many rocketeers continue into high-power rocketry. These rockets use motors with an impulse greater than 160 Ns, which require certification to purchase and launch. Here, it is necessary not just to simulate the rocket, but to measure and analyze its real-world performance.

The Need for Altimeters

In high-power and low-power rocketry alike, one of the most frequent measurements performed is altimetry. Aside from the sense of achievement that reaching a high altitude can bring, altitude is used as a method of comparison between a rocket's simulated flight performance and actual performance—in other words, it is used to see if the rocket flew as expected. It is even a criterion for judgement in student competitions, such as NASA Student Launch, the American Rocketry Challenge, the International Rocket Engineering Competition, and Base 11 Space Challenge, which all specify a target apogee.

Additionally, when launching high-power rockets, altitude quickly becomes a legal issue. Anyone wishing to fly a rocket with a motor above 160 Ns must file for a waiver with the Federal Aviation Administration (FAA) if their launch site is in controlled airspace or within 5 miles of an airport. The waiver establishes a flight ceiling, or a specified altitude that the rocket shall not exceed, for the launch site (Federal Aviation Administration, 2025). Clouds pose another issue, as the Tripoli Rocketry Association (2025) Safety Code states that rockets should not fly past altitudes with 50% or greater cloud cover. That being said, rocketeers should simulate the estimated apogee of their rocket before flying, and should not fly if they think there is a chance they will exceed any waiver or cloud ceiling. Altimeters used in this case are not preventative measures, but they can still be used for verification purposes.

However, altitude measurements are not solely used for post-flight analysis. Altimeters are also the primary sensors used when timing drogue and parachute deployment on mid to high-power rockets. These rockets, flying with motors at an impulse of 20 Ns or higher, fly to several thousand feet. To reduce the distance that the rocket drifts during its descent, a smaller parachute

called a “drogue” is used in place of the “main” large parachute. The drogue allows the rocket to descend in a controlled manner, but still at a high speed. Then, when the rocket is closer to the ground, the main parachute is deployed to slow the rocket and ensure safe landing. This process is called dual deployment. Altimeters are integral to this process, as failure to deploy the drogue and main parachutes at their desired altitudes could, at best, change the landing timing and location, and at worst, lead to an uncontrolled descent and landing that destroys the rocket.

Another mid-flight application of altimeter data, though less conventional, is shown in Trine University’s aerobraking system, developed by Chew et al. (2023). Their closed-loop feedback system uses altitude to determine whether the brake should expand or contract, thus altering the surface area and drag on the rocket. This braking system allows the rocket to respond to day-of launch conditions, such as wind and atmospheric pressure, without major redesign to the rocket body.

Previous Studies

Because altitude-controlled events are critical to the success of the rocket, it is important to know what altimetry options are available, as well as their benefits and drawbacks. Previous studies have compared commercially available altimeters, as well as custom programs with more cost-effective sensors. Kidwell and Ash-Poole (2004) have compared 11 different electronic barometric, accelerometer, and hybrid altimeters, as well as handheld optical scopes to find the altitude of their high-power rocket. They found that among the electronic altimeters, altimeters with 8-bit analog-to-digital converters (ADCs) had lower accuracy, while 12-bit or higher ADC

altimeters performed more accurately. However, the overall standard deviation on the electronic altimeters was lower than on the optical sensors.

Albéri et al. (2017) have also compared altimeters, but in the context of providing better altitude estimations for unmanned aerial vehicles (UAVs). They tested 3 GPS units, 1 IMU, 1 radar altimeter, and 2 barometers on a gyrocopter flying between 35 and 2194 meters over the Tyrrhenian Sea. They found that below 70 meters, barometric altimeters were the most accurate, with correction from the radar altimeter. GPS data could not be used at this altitude due to the multipath effect, in which GPS signals reflect on the sea and cause significant noise. GPS correction was able to be used over 80 meters. Additionally, radar correction was not feasible above 340 meters, as the readings were not accurate enough when compared to the other altimeters.

Brown et al. (2019) have compared a commercial TeleMega altimeter with the FASTsensor, their custom altimeter which utilizes several Adafruit components, including an accelerometer, GPS, and barometer. Their final design costs around $175, and accurately measured angular positions and velocities of their rocket. However, vertical acceleration was not able to be measured with the custom system due to calibration limitations on the accelerometer. Additionally, sensors were programmed using an Arduino Mega, which has a width of 2.09 inches. Therefore, the size of the board limits use to rockets over 2.09 inches in diameter. However, depending on sensor configuration and height and weight of the total altimeter package, a larger rocket is likely required. The rocket flown in the study had a diameter of 4 inches.

Mustata et al. (2024) have attempted to develop a low-cost custom altitude sensor package by utilizing the HX710B pressure sensor, a commercially available piezoresistive model. They tested their sensor up to 1600 meters and found that its average error was 18.19 meters, with a maximum error of 33 meters, as error increased linearly with increasing altitude. Additionally, their system was not set up to run on battery power, but instead relied on laptop connection while running.

With all of these options in mind, there are many considerations to make when deciding on an altimeter. If using a barometric altimeter, one must consider the environment that the sensor is placed in. Changes in weather will affect the ground-level pressure reading, and so it is necessary to check the ground-level pressure before taking other measurements. Bolanakis et al. (2015) have shown that a separate, stationary reference barometer can also be used to accomplish this task, and is actually preferred because ground pressure fluctuations can be measured throughout the entirety of the sampling window. However, if a separate reference barometer is being used, the two barometers must be calibrated against each other regularly, as their readings will deviate from one another over time.

GPS devices are also popular, though their accuracy can be questionable at times. Altitude measurement error through GPS is, on average, 3 times as large as error on latitude and longitude. Depending on the GPS, readings can deviate by 30-50 meters from actual altitude (International Civil Aviation Organization, 2012). However, some studies have shown much lower error on high-frequency GPS devices. In their development of a tracking device for birds, Bouten et al. (2013) have tested lightweight GPS altimeters over several sampling rates, and their lowest average error was 1.42 meters with readings being taken every 6 seconds. However, when

increasing the sampling time to 60 seconds and 600 seconds, average error increased to 3.99 meters and 26.27 meters respectively, despite the GPS remaining stationary. Schaub et al. (2023) have also looked at GPS altimeters for this application, and found that high frequency GPS altimeters were more accurate than barometric ones, but low frequency GPS altimeters were less accurate. Additionally, they found that barometric sensors generated less noise than any GPS configuration, but were more prone to bias. It is uncertain what quality of GPS was tested in these trials, as exact model names were not published.

For rocketeers who do use a GPS altimeter, care must be put into the configuration of a GPS within the rocket. The GPS must have access to satellite signals, and too many additional electronics at one spot can interfere with the GPS reading. Students at Morgan University have tested GPS placement by placing a Featherweight GPS in the nosecone of their rocket and launching it with a 303 Ns impulse motor. The GPS had an antenna attached, also located in the nosecone. After comparing the GPS altitude with simulated data and a barometric altimeter, the team concluded that the nosecone was a sufficient location for the GPS to gather reliable data, with the rest of the rocket's sensors and computers housed in a separate electronics bay (Caballes et al., 2021).

Aside from electrical sensors, handheld devices are a low-cost method to track altitude. Altimeters such as the Estes Altitrak have protractors that can measure the angle between the rocket, the user, and the launchpad. Then, some simple trigonometry can give the rocket's height. However, optical methods such as this are not very accurate, and are impossible to perform if a rocket ascends beyond a viewable altitude.

Proposed Design

Cost is another major factor in choosing an altimeter, especially for students and those who are new to rocketry. While past studies have compared altimeter types, many of these altimeters were quite costly, and most studies were not focused on applications within rocketry. Studies that are specifically designed with rocketry in mind tend to use high-power rockets, which do not have as many space restrictions as low-power or beginner rockets. This study attempts to develop an affordable yet accurate sensor system for novice rocketeers. The system should also be as small as possible, both in weight and geometry. This allows the system to fit inside smaller rockets and not significantly affect the rocket's stability.

Four sensors are implemented to do this: a GPS unit, two inertial measurement units (IMUs), and a pressure sensor. They are all connected to a microcontroller and datalogger, and data is saved to a microSD card. The sensor package is powered by a 3.7V lithium polymer rechargeable battery, making battery replacement convenient and inexpensive. Altitude measurements from the four devices are compared to one another, as well as to a JollyLogic AltimeterTwo, a popular commercial altimeter. The altimeters are also tested against an Estes AltiTrak. The sensor package is tested in multiple iterations over several rocket flights and across different rocket types. After comparing results, the best-performing sensor(s) will be selected and tested as part of the final device, or the "end product," thus completing the goal of developing an affordable, accurate system.

Chapter 2

CURRENT ALTIMETRY METHODS

Before further discussing the proposed design, it is beneficial to get an overview of common altimetry methods used in the rocketry community. Though not explicitly tested and documented in literature, many of these devices make use of the altimeter types presented in Chapter 1. Other, non-electronic methods also exist.

Handheld devices such as the Estes AltiTrak are used from a known distance from the launchpad, and they measure the angle between the ground and the user's line of sight to the rocket. The tracker is shown in Figure 2.1 with callouts to key components.

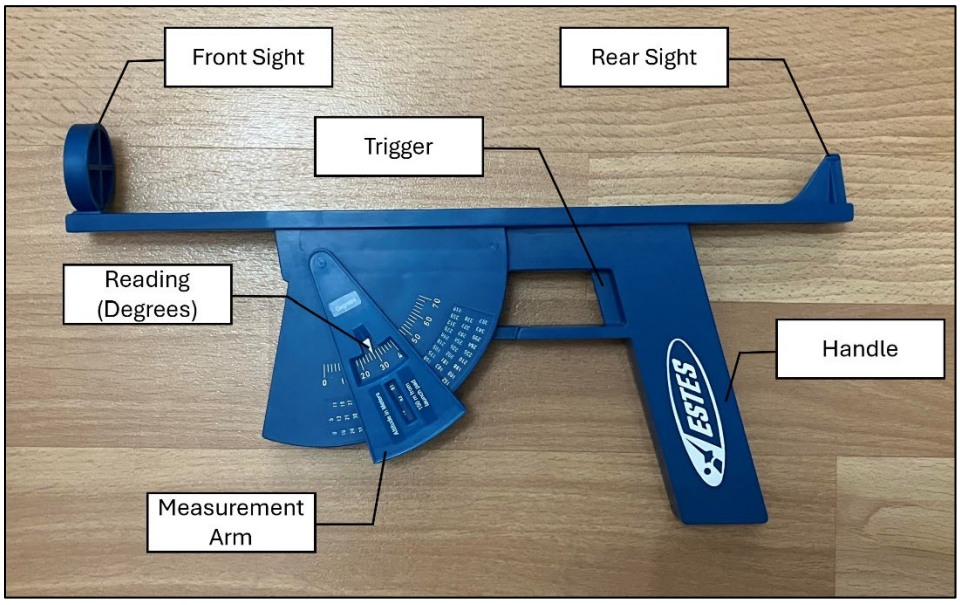

Figure 2.1. Components of the Estes AltiTrak

In this figure, the measurement arm reads 28 degrees.

To use the device, the tracker must be pointed at the rocket starting at takeoff. The user must hold the trigger, releasing the measurement arm on the tracker. Then, still holding the trigger, the

user continues to point the tracker at the rocket as it flies, following it through the sights. At apogee, the user releases the trigger, locking the arm in place, and then slowly lowers the tracker. The measurement is the angle between the ground and the user's line of sight to the rocket, shown as θ in Figure 2.2.

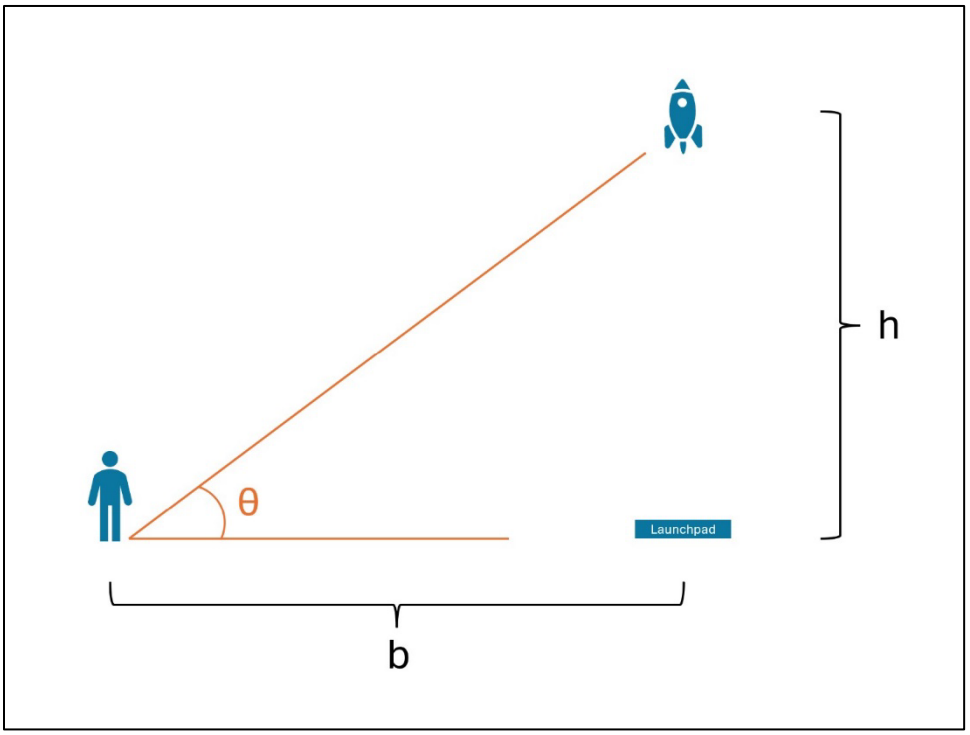

Figure 2.2. Altitude Calculation Parameters

Then, the user can apply some trigonometry to estimate the altitude. Equation (1) calculates the height of apogee, h, by using the AltiTrak angle reading, θ, and the user's distance from the launchpad, b.

$$h = b \tan \theta \tag{1}$$

The handheld device is used as an inexpensive alternative for rough altitude estimation. Homemade devices can even be constructed from common household items such as a straw,

string, a washer, tape, and a protractor (Structure&Civil, 2017). It is also simple to use, and can easily be operated without experience with electronic sensors. However, there is much room for error with this device. Readings will vary based on the height of the operator, vertical movement of the rocket (it never moves purely up), and general human error.

Another optical method of measuring altitude is by dropping a colorful object from the rocket upon parachute deployment, and timing how long it takes to land. The time can be used in Equation (2) to calculate the height where the object was dropped. This is an inexpensive method; however, it is not commonly used because parachute deployment happens slightly after apogee, so the resultant height would be lower than the true apogee of the rocket. Also, it can be difficult to keep sight of the object, especially because it should not have streamers or parachutes, which will decrease the speed of descent. Like the Estes AltiTrak method, this method depends on the user having sight of the rocket and the dropped object for its entire flight and landing. As the rocket flies to higher altitudes, the parachute or streamers are often the only visible part of the rocket during its descent.

$$h = \frac{1}{2} g t^2 \tag{2}$$

where h is height, g is gravitational acceleration (9.81 m/s^2 or 32.2 ft/s^2), and t is descent time

Because handheld methods generate large amounts of error, sensors in the electronics bay are generally considered to be more accurate, and work in several different ways. Most electronic altimeters use either a pressure sensor, an accelerometer, a GPS, or a combination of these to calculate altitude. In addition to the variety of available sensor types, altimeters range in price and accuracy, with those on the lower end costing between $40 and $50, and high-end ones

costing upwards of $600. When deciding on an altimeter, one needs to consider the tradeoffs between accuracy and price.

For less expensive models, EggTimer is a common brand that has several options for altitude tracking, most of which use pressure sensors as their main source of measurement. These models support dual deployment, but some models require separate systems to log data for use post-flight. To reduce costs, EggTimer models are sold as unassembled kits that must be soldered by the user. Assembly difficulty ratings on a scale of 1 to 5 are given by the seller for each kit (EggTimer, n.d.).

Like EggTimer, MissileWorks also has barometric altimeters. Their altimeters come fully assembled, though a separate LCD screen must be plugged into the sensor board if the user wishes to see programming settings or recorded data. The board comes with a beeper, and the beeps can be decoded in order to program without the LCD screen. These altimeters support dual deployment (MissileWorks, n.d.).

JollyLogic altimeters combine pressure readings with accelerometer data. There are several models, from simple altitude trackers (meant to be read post-flight) to flight computers which support chute release (in lieu of dual deployment). Anecdotally, the JollyLogic chute release flight computers are not as reliable as other dual deployment devices. However, simpler altimeters such as the AltimeterOne or AltimeterTwo are reliable and durable, making them a common choice when first exploring altimetry. They are also smaller in both size and weight, and do not require any programming, external hardware, or dedicated electronics bay (JollyLogic, n.d.). For this reason, the JollyLogic AltimeterTwo is chosen for this project as a point of comparison to a commercially-available altimeter.

Finally, the TeleMega altimeter blends barometer, accelerometer, and GPS data. This is one of the most high-end altimeters on the market, selling at or above $400, depending on supplier. It also transmits data real-time. In addition to tracking altitude, this board supports up to five ignition events (such as dual deployment or staged motor ignition). For this reason, the TeleMega is most commonly used for complex, high-altitude flights (Altus Metrum, n.d.).

There are many other altimeters on the market, but the aforementioned brands are some of the most popular. Table 2-1 gives an overview of these altimeters and some of their key characteristics.

Table 2.1. Common Rocket Altimeters

Note that prices do not include taxes or shipping charges

Brand	Name	Type	Price	Supports Dual Deployment?	Records Multiple Flights?
Estes	AltiTrak	Optical	$24.99	No	No
EggTimer	Quark	Barometer	$20.00	Yes	No
EggTimer	Quantum	Barometer	$40.00	Yes	Yes
MissileWorks	RRC3+	Barometer	$74.95	Yes	Yes
JollyLogic	AltimeterTwo	Barometer, Accelerometer	$79.95	No	Yes
JollyLogic	Chute Release	Barometer	$139.95	No	No
Altus Metrum	TeleMega	Barometer, GPS, Accelerometer, Magnetometer	$400.00	Yes	Yes

Chapter 3

ROCKET STATE ESTIMATION

Depending on the sensors used, some parameters can be easier to measure than others. Altitude and acceleration are relatively simple to measure, and sensors such as barometers and accelerometers or IMUs can give direct measurements. However, other parameters such as velocity require more sophisticated calculation methods. A GPS can perform one such calculation by analyzing the Doppler effect: how the signal frequency changes when both the GPS and the satellite are in motion (Van Sickle, n.d.). However, the GPS used in this study did not have a fast or accurate enough sampling rate to merit keeping it in the final design. More accurate GPS devices could be used, but they would increase cost by a considerable amount. To measure rocket velocity, another method must be used.

This work considers the estimation of rocket body-axis velocity components, u, v, and w and rocket altitude, h. This estimation is performed by combining information from IMU measurements of three-axis accelerations, a_x, a_y, and a_z, and angular rates, p, q, and r, as well as estimated values of Euler attitude angles ϕ (roll) and θ (pitch). From this information, a state space system can be formulated with the following state vector, $\vec{x}$, input vector, $\vec{u}$, and output vector, $\vec{y}$:

$$\vec{x} = [u \quad v \quad w \quad h]^T$$
$$\vec{u} = [a_x \quad a_y \quad a_z \quad p \quad q \quad r \quad \phi \quad \theta]^T \tag{3}$$
$$\vec{y} = h$$

The total rocket speed can be estimated from the estimated states as in:

$$V = \sqrt{u^2 + v^2 + w^2} \tag{4}$$

To implement this state space system within a state estimation filter, the state dynamic equations must be defined. The state dynamics for the body-axis velocity states are then given by (Rhudy et al., 2015):

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} 0 & -w & v \\ w & 0 & -u \\ -v & u & 0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} + \begin{bmatrix} a_x - g \sin\theta \\ a_y + g \sin\phi \cos\theta \\ a_z + g \cos\phi \cos\theta \end{bmatrix} \tag{5}$$

where g is the acceleration due to gravity, which is used here as a constant value of 9.80665 m/s^2. The altitude state equation can be defined by (Rhudy et al., 2024)

$$\dot{h} = u \sin\theta - v \sin\phi \cos\theta - w \cos\phi \cos\theta \tag{6}$$

The output equation is provided by a direct measurement of the altitude, for example from a pressure altitude measurement. Since these equations are all independent of the rocket model, i.e. the equations do not depend on inertial, aerodynamic, or geometric parameters, this formulation can be considered to be dynamic-model independent and therefore can be implemented on different rockets.

Due to the nonlinearity in the state dynamics equations, a nonlinear state estimation technique is necessary. The Unscented Kalman Filter (UKF) is considered here due to its capability of handling nonlinearity and its relative ease of implementation when compared to the Extended Kalman Filter (EKF). Due to the use of IMU measurements in the input vector, a non-additive noise model was assumed for the system. Thus, the following nonlinear discrete-time state space system was utilized:

$$\begin{aligned} \boldsymbol{x}_k &= \boldsymbol{f}(\boldsymbol{x}_{k-1}, \boldsymbol{u}_{k-1}, \boldsymbol{w}_{k-1}) \\ \boldsymbol{y}_k &= \boldsymbol{h}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{v}_k) \end{aligned} \tag{7}$$

where k is the discrete time index, **f** is the discrete-time vector-valued state prediction function, **h** is the nonlinear vector-valued observation function, **w** is the non-additive process noise vector, and **v** is the non-additive measurement noise vector. For this particular system, the observation equation is a direct measurement of the state, resulting in a simple linear system update with additive measurement noise, as in:

$$\begin{aligned} \boldsymbol{y}_k &= \boldsymbol{H}_k \boldsymbol{x}_k + \boldsymbol{v}_k \\ \boldsymbol{H}_k &= [0 \quad 0 \quad 0 \quad 1] \end{aligned} \tag{8}$$

where the measurement noise vector is due to uncertainty in the pressure altitude measurement, as in:

$$\boldsymbol{v} = v_h \tag{9}$$

The state dynamics were defined in continuous time, therefore a first order discretization is used to put into a discrete time format, as in:

$$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \Delta t \boldsymbol{f}_c(\boldsymbol{x}_{k-1}, \boldsymbol{u}_{k-1}, \boldsymbol{w}_{k-1}) = \boldsymbol{f}(\boldsymbol{x}_{k-1}, \boldsymbol{u}_{k-1}, \boldsymbol{w}_{k-1}) \tag{10}$$

where Δt is the sampling time. Then, the process and measurement noise covariance matrices can be represented by:

$$\begin{aligned} \boldsymbol{Q} &= diag\left[\sigma_{w_{acc}}^2 \quad \sigma_{w_{acc}}^2 \quad \sigma_{w_{acc}}^2 \quad \sigma_{w_{gyro}}^2 \quad \sigma_{w_{gyro}}^2 \quad \sigma_{w_{gyro}}^2 \quad \sigma_{w_{att}}^2 \quad \sigma_{w_{att}}^2\right] \\ \boldsymbol{R} &= \sigma_{v_h}^2 \end{aligned} \tag{11}$$

where each of the individual variance terms are considered to be zero-mean uncorrelated Gaussian noise. $\sigma_{w_{acc}}^2$ represents the error in the accelerometer measurements, $\sigma_{w_{gyro}}^2$ represents the error in the rate gyroscope measurements and $\sigma_{w_{att}}^2$ represents the error in the attitude

estimates. The measurement noise covariance matrix, **R**, here consists of a single term that corresponds to the uncertainty in the pressure altitude measurement.

Using these definitions, the UKF for non-additive noise can be implemented. Due to the linear observation equations, the linear KF update can be used in this case. The UKF equations are presented with more details for a similar formulation in (Rhudy et al., 2024) and are summarized here. For non-additive process noise, the state vector of the UKF can be augmented with additional "states" to model the non-additive process noise terms, as in

$$\boldsymbol{x}_k^a = \begin{bmatrix} \boldsymbol{x}_k \\ \boldsymbol{w}_k \end{bmatrix} \tag{12}$$

where the superscript '*a*' denotes the augmentation. The error covariance matrix must also be augmented accordingly, taking the form of a block diagonal matrix

$$\boldsymbol{P}_k^a = \begin{bmatrix} \boldsymbol{P}_k & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_k \end{bmatrix} \tag{13}$$

Now, the augmented state vector and covariance matrix are used to generate the augmented sigma-points

$$\boldsymbol{\chi}_{k-1}^a = \begin{bmatrix} \boldsymbol{\chi}_{k-1}^x \\ \boldsymbol{\chi}_{k-1}^w \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{x}}_{k-1}^a & \hat{\boldsymbol{x}}_{k-1}^a + \sqrt{L + \lambda_{UKF}}\sqrt{\boldsymbol{P}_{k-1}^a} & \hat{\boldsymbol{x}}_{k-1}^a - \sqrt{L + \lambda_{UKF}}\sqrt{\boldsymbol{P}_{k-1}^a} \end{bmatrix} \tag{14}$$

where L is the dimension of the augmented state vector, and λ_{UKF} is an UKF scaling parameter, defined as

$$\begin{aligned}
\lambda_{UKF} &= \alpha_{UKF}^2(L + \kappa_{UKF}) - L \\
\boldsymbol{\eta}_0^m &= \lambda_{UKF}/(L + \lambda_{UKF}) \\
\boldsymbol{\eta}_0^c &= \lambda_{UKF}/(L + \lambda_{UKF}) + 1 - \alpha_{UKF}^2 + \beta_{UKF} \\
\boldsymbol{\eta}_i^m &= \boldsymbol{\eta}_i^c = 1/[2(L + \lambda_{UKF})], \quad i = 1, \ldots, .2L
\end{aligned} \tag{15}$$

where α_{UKF}, β_{UKF}, and κ_{UKF} are the primary, secondary, and tertiary UKF scaling parameters. The **η** terms are weight vectors used to recover the mean and covariance after the nonlinear transformation. The sigma-points can be separated into two sections, that is state, **x**, and process noise, **w**, as noted by the superscripts. Now, only the state sigma-points need to be predicted using prior information from both the state and process noise sigma-points

$$\boldsymbol{\chi}_{k|k-1}^{x,(i)} = \boldsymbol{f}\left(\boldsymbol{\chi}_{k-1}^{x,(i)}, \boldsymbol{u}_{k-1}, \boldsymbol{\chi}_{k-1}^{w,(i)}\right), \qquad i = 0,1, \ldots, 2L \tag{16}$$

The *a priori* statistics are then recovered using

$$\begin{aligned}
\widehat{\boldsymbol{x}}_{k|k-1} &= \sum_{i=0}^{2L} \boldsymbol{\eta}_i^m \boldsymbol{\chi}_{k|k-1}^{x,(i)} \\
\boldsymbol{P}_{k|k-1} &= \sum_{i=0}^{2L} \boldsymbol{\eta}_i^c \left(\boldsymbol{\chi}_{k|k-1}^{x,(i)} - \widehat{\boldsymbol{x}}_{k|k-1}\right)\left(\boldsymbol{\chi}_{k|k-1}^{x,(i)} - \widehat{\boldsymbol{x}}_{k|k-1}\right)^T
\end{aligned} \tag{17}$$

Since the measurement equations are linear, the linear Kalman filter update can be used

$$\begin{aligned}
\boldsymbol{K}_k &= \boldsymbol{P}_{k|k-1}\boldsymbol{H}_k^T\left(\boldsymbol{H}_k\boldsymbol{P}_{k|k-1}\boldsymbol{H}_k^T + \boldsymbol{R}_k\right)^{-1} \\
\widehat{\boldsymbol{x}}_k &= \widehat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k\left(\boldsymbol{y}_k - \boldsymbol{H}_k\widehat{\boldsymbol{x}}_{k|k-1}\right) \\
\boldsymbol{P}_k &= (\boldsymbol{I} - \boldsymbol{K}_k\boldsymbol{H}_k)\boldsymbol{P}_{k|k-1}
\end{aligned} \tag{18}$$

where **K** is the Kalman gain matrix and **I** is an identity matrix of appropriate dimensions.

Chapter 4

EXPERIMENTAL SETUP

The experiment is set up in two parts. The first part compares several sensors to test their performance in a high-power rocket. This allows for a larger electronics bay and testing over a greater altitude range. The second part fits the best-performing sensors in a smaller package for a low-power rocket, which is thinner and lighter in weight. This represents the rocket characteristics for which the end product sensors are being developed. Board models are listed in Table 4-1, along with other electronic components used in the testing of the boards. Each board is given a Diagram Number for reference throughout the chapter.

Table 4.1. List of Electronic Components

Diagram Number	Brand	Model	Adafruit Product ID	Description
1	Adafruit	STEMMA QT	4399	4-Pin Cable, 50mm
2	Adafruit	Feather RP2040 Adalogger	5980	Microcontroller/Data-logger
3	Adafruit	BMP390	4816	Pressure/Temperature Sensor
4	Adafruit	BNO055	4646	IMU
5	Adafruit	ISM330DHCX + LIS3MDL	4569	IMU
6	Adafruit	Ultimate GPS FeatherWing	3133	GPS
N/A	Adafruit	MicroSD Card	1294	MicroSD Card, 8 GB SDHC
N/A	Adafruit	LiPo Battery	1570	LiPo Battery, 3.7 V, 100mAh
N/A	PKCELL	CR1220	N/A	CR1220 Battery, 3 V
N/A	JollyLogic	AltimeterTwo	N/A	Pressure & Accelerometer Altimeter

In both parts of the experiment, the rocket flight is simulated on OpenRocket, a free software which allows users to model rocket designs and simulate their performance. Default settings were used for the simulations, except for wind speed, which was set to match the launch-day winds. The Extended Barrowman method is used to perform simulation calculations. This

method makes simplifications to a more complex model so that it can be solved algebraically rather than using calculus, and was developed starting in 1966 and continuing into the 1990s. More information about these equations can be found in the original paper by John and Judith Barrowman (1966), as well as a later work by Edward LaBudde (1999).

Part One: Initial Board Comparison

For the first part, four sensors are used: two IMUs, one pressure sensor, and one GPS. They are connected to a microcontroller, which records data to a microSD card. Because the boards are sold individually, the three biggest boards (Diagram Numbers 2, 5, and 6) must be soldered to headers so they can connect, shown in Figure 4-1. They are then stacked according to the diagram in Figure 4-2. The other two boards (Diagram Numbers 3 and 4) have ports for STEMMA QT connectors, Adafruit's custom plug-and-play connection method. STEMMA QT wires connect these boards to another port on the large IMU (Diagram Number 5), and information travels through the wires, and then through the stacking headers, to the microcontroller.

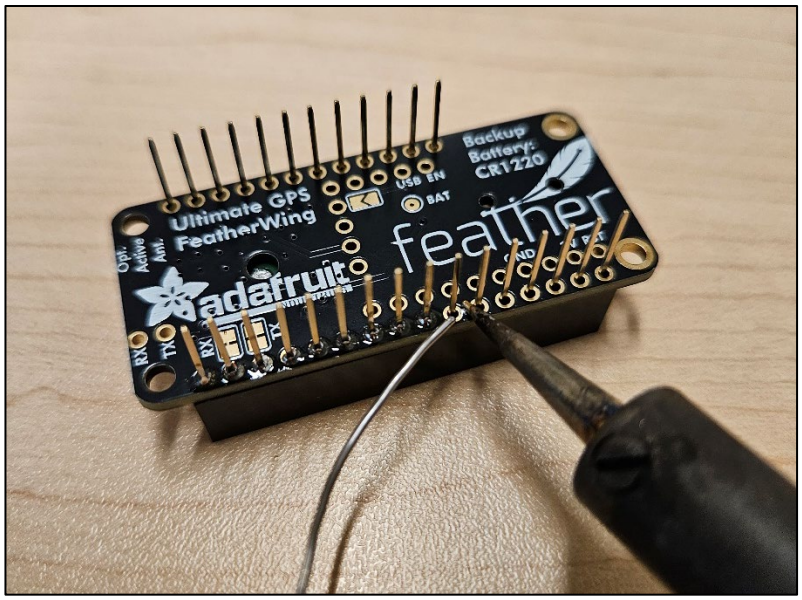

Figure 4.1. Soldering Headers Onto the GPS

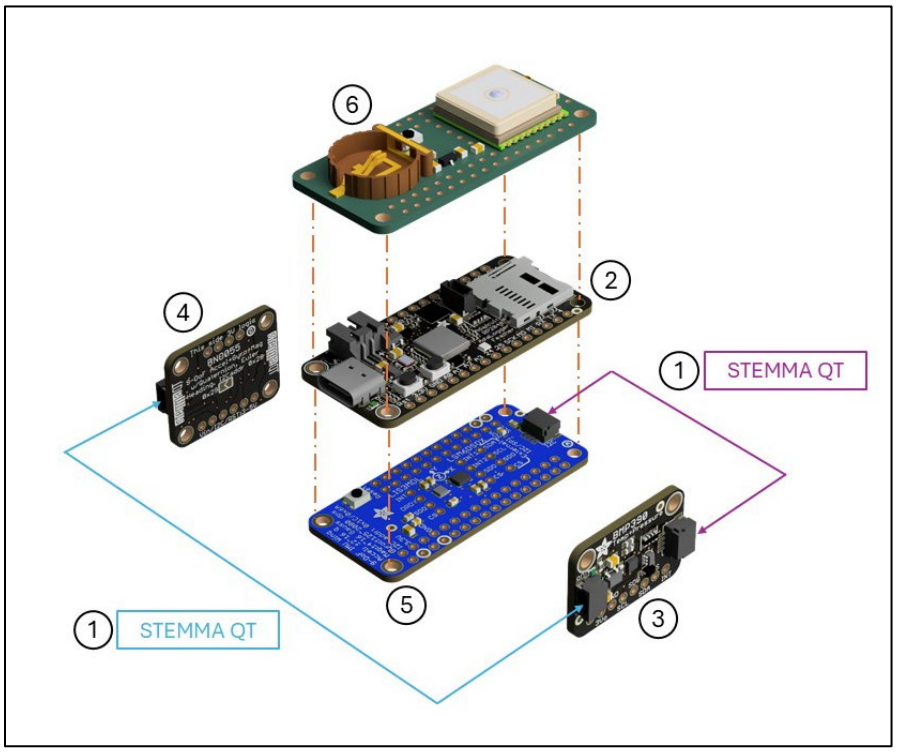

Figure 4.2. Part One Board Configuration

The system is powered by a Lithium Polymer (LiPo) battery that plugs into the microcontroller. The GPS also has its own button cell battery so that it can maintain constant satellite connection. This allows the GPS to remember satellite data between launches—otherwise, it would need to reconnect to the satellites every time the system was powered, which can take several minutes.

Sensors were programmed using Arduino. Example code from Adafruit is freely available for individual boards, so their example code was integrated into the combined system. The full Arduino sketch used in this part of the experiment is available in Appendix A.

Several considerations were made when arranging the boards. The GPS is the topmost board so that the others will not interfere with the satellite signal. Meanwhile, the two smallest boards are affixed to the sides of the stacking headers with hot glue (see Figure 4-3). This is done to keep a compact sensor package while still fixing the boards in place. The boards are also rotated by 90 degrees in relation to the other boards—this ensures that once the package is mounted inside the rocket, the *x*-axis of the IMU will point towards the nose of the rocket, or in other words, in the direction of travel. Finally, the small IMU is glued at the left edge of the stacking header as far from the battery port as possible. This is done so that the battery can be more easily accessed for replacement.

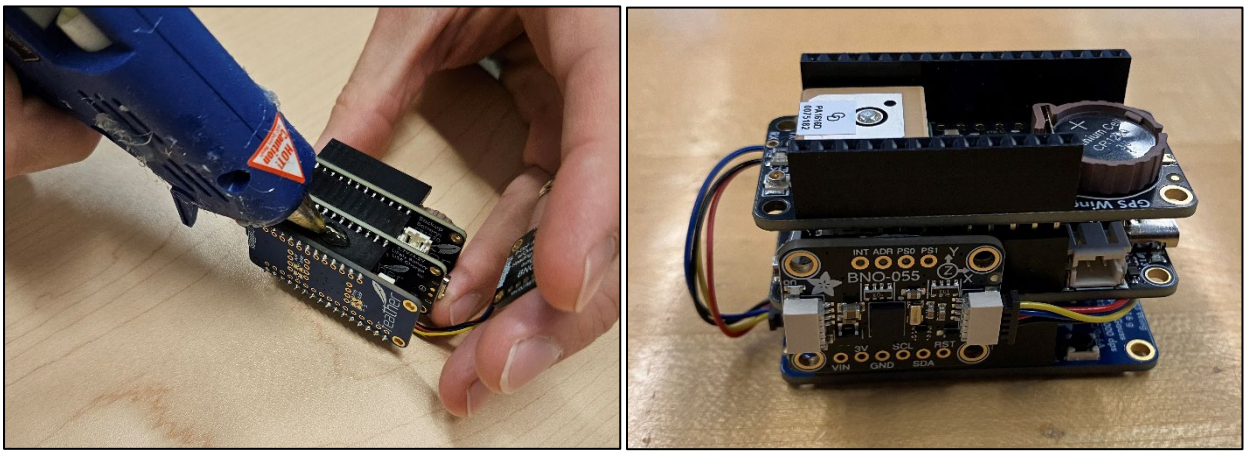

Figure 4.3. Gluing the Smaller Boards

Left: Hot glue is applied to the plastic walls of the stacking headers. Right: The IMU is affixed at a 90 degree angle to the other boards. It is as far as left as possible to maintain access to the datalogger battery port.

The sensor package is tested in a 54 inch scratch-built rocket. The rocket is designed to take a 38mm diameter motor ranging from H to J impulse class, corresponding to 160 to 1280 Ns of impulse. For this test, an H283 motor was used, bringing the rocket to 1048 feet in altitude in the simulated test. A labeled diagram of the rocket is shown in Figure 4-4, and Figure 4-5 shows the rocket on the launchpad. Note the gold ring towards the front (nose end) of the rocket—this is part of the electronics bay where the sensors are mounted.

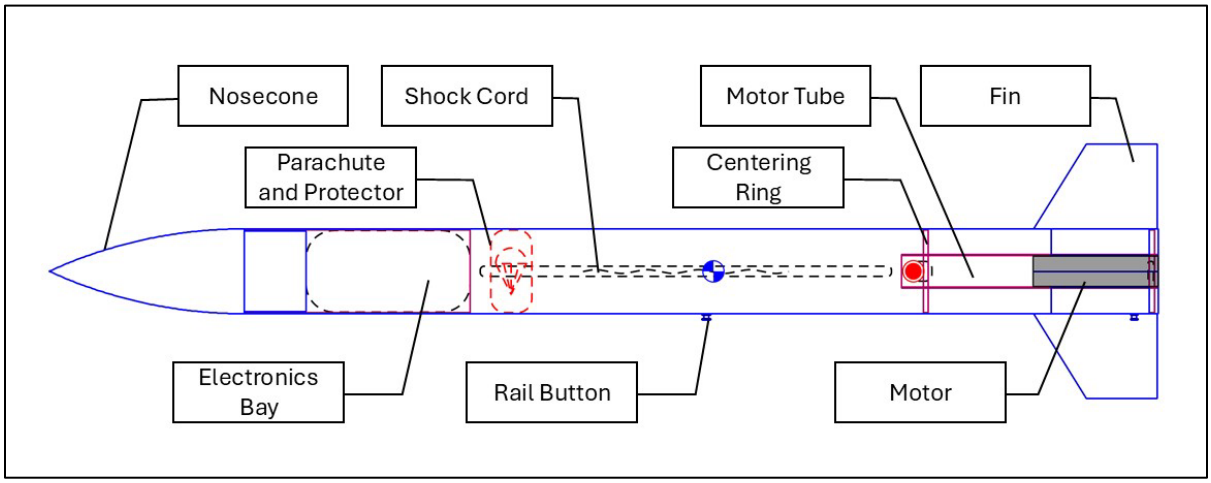

Figure 4.4. Labeled Diagram of the High-Power Rocket

Figure 4.5. High-Power Rocket on the Launchpad

The electronics bay, shown in Figure 4-6, has a baseplate with several holes for mounting the sensor package. There are also slots for ties to secure the battery and JollyLogic AltimeterTwo.

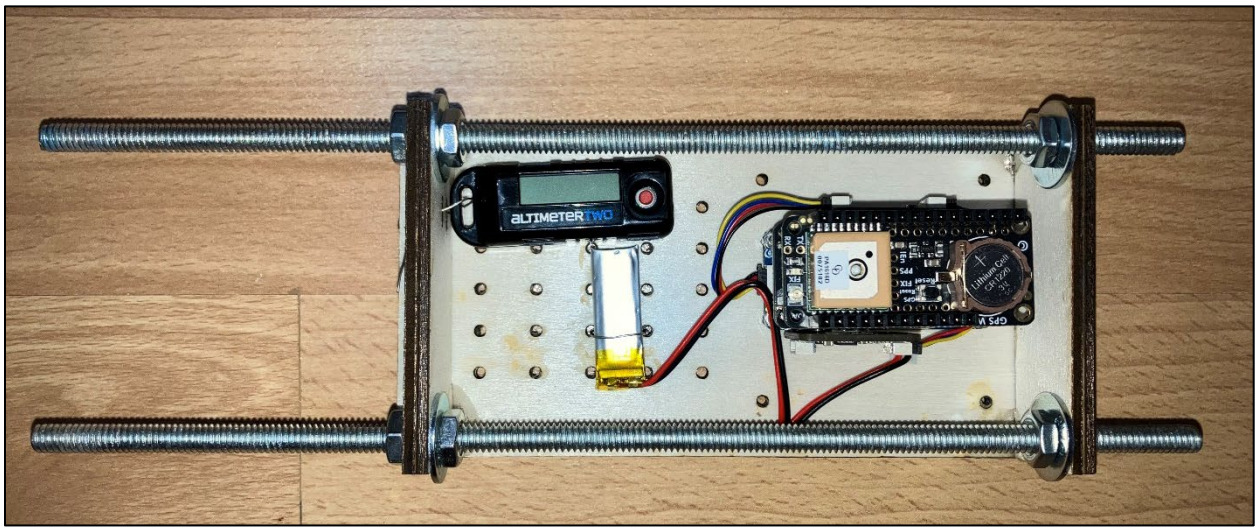

Figure 4.6. Electronics Bay on Sled

Because the sensor package and battery are mounted separately, the battery can easily be unplugged or replaced between flights, while the sensors can remain in place. The microSD card can also be switched between flights to keep data well-organized—in the event that the card is not swapped, data will still be recorded for both flights, but it will save to the same file and will need to be separated. Similarly, the JollyLogic AltimeterTwo can be turned on and off without removing it from the electronics bay. The plate is secured on rails, and a coupler tube and endcaps enclose the plate and rails. Then, the entire electronics bay serves as a coupler between the two body tubes of the rocket. Air is allowed to enter the electronics bay via a hole in the gold-colored switch band Figure 4-7. This allows the pressure sensors to get an accurate reading.

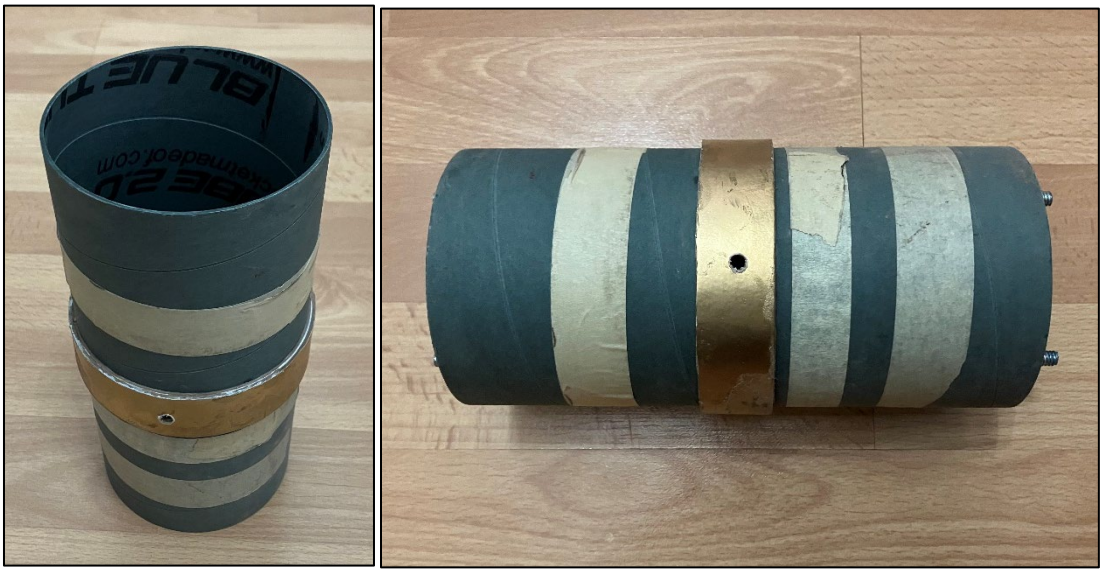

Figure 4.7. Outside of the Payload Bay

This tube fits inside the outer walls of the rocket, with the gold band being exposed to the outside surroundings. Note the hole in the band, which allows air to enter the payload bay so that the pressure sensor can get an accurate reading. Tape is applied to the tube to slightly increase its width and create friction against the outer rocket tube.

The Estes AltiTrak handheld altimeter was also used, and coordinates were recorded at the launch pad and the location of observation.

Part Two: A Slimmed-Down Model

In testing the first sensor package, it was determined that the GPS was highly inaccurate in altitude reading, and both IMUs gave similar results as one another. For the second part of testing, the GPS and the larger IMU (Diagram Numbers 5 and 6) were removed, leaving the microcontroller, the pressure sensor, and the smaller IMU (Diagram Numbers 2, 3, and 4),

powered by the same battery as in Part One. All of these boards can be connected with STEMMA QT cables, so no soldering or stacking headers are required.

In a similar manner to Part One, example code from Adafruit is integrated into the combined system for this project. The full Arduino sketch used for this part of the project is available in Appendix B.

To maintain a compact system, the boards are hot glued together. The pressure sensor is glued to the back of the datalogger, sandwiched between the datalogger and the small IMU as demonstrated in Figure 4-8. (In the figure, the datalogger also has soldered stacking headers. These headers are remaining from Part One of the experimental setup and play no role in data transfer for Part Two.)

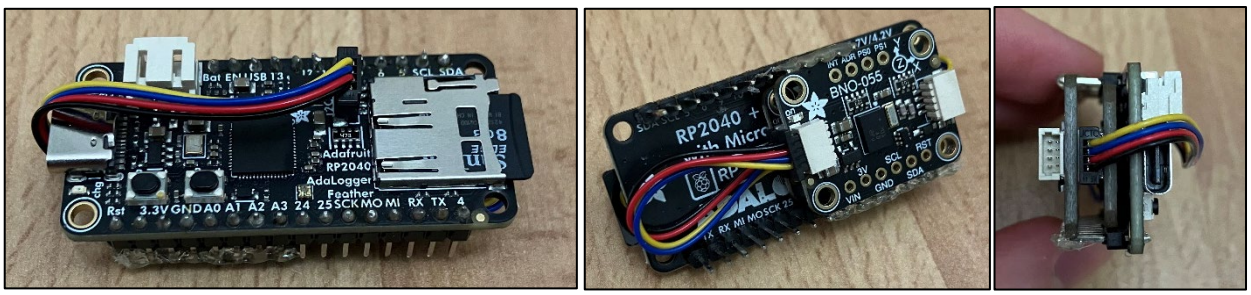

Figure 4.8. Part Two Board Configuration

A foam casing is used to hold the boards and JollyLogic altimeter inside the rocket, and the purpose of this is twofold: first, to protect the boards and prevent them from moving, and second, to keep the IMU x-axis aligned towards the nose of the rocket, in the direction of flight. The battery and pressure sensor do not need specific orientations, but the pressure sensor is given access to the air hole in the payload bay wall. The sensor and foam casing are shown inside the rocket in Figure 4-9.

Figure 4.9. Low-Power Rocket with Sensors in Electronics Bay

The payload is part of an Estes Loadstar II kit rocket. This rocket is 23.25 inches in length and takes two motors, one in each stage. The motors can be A, B, or C class, corresponding to an impulse between 1.26 and 10 Ns. For this test, B6 motors were used for the sustainer (the motor that stays on the rocket for the entire duration of the flight), and the booster (the section that burns first and then falls off the rocket) was not used. In simulation, the rocket reaches 157 feet in altitude. A labeled diagram of the rocket is shown in Figure 4-10.

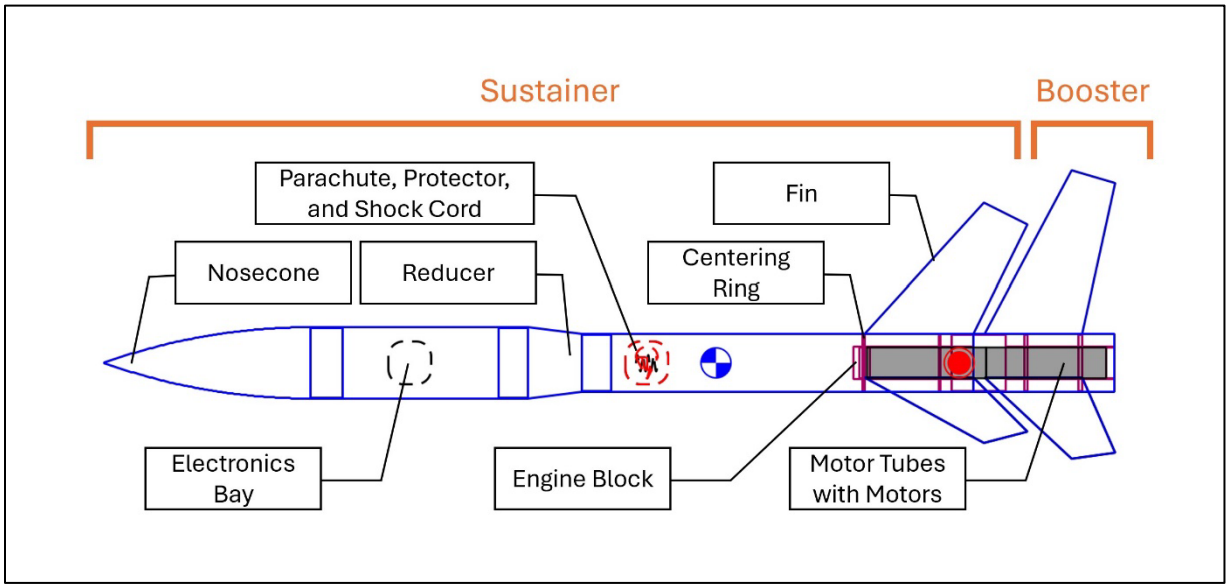

Figure 4.10. Labeled Diagram of the Low-Power Rocket

The Estes AltiTrak handheld altimeter was also used, and the distance between the launchpad and the observation location was measured.

Chapter 5

RESULTS

Results were collected for both Parts One and Two of the experiment. MATLAB was used for plotting and analysis.

Analysis of Part One

The first launch was performed at 50 degrees Fahrenheit and 29.4 inHg sea level pressure, at an elevation of 399 feet above sea level. Wind was strong, blowing between 15 and 20 mph. Because of this, only one launch was possible before winds became too heavy.

From the simulation, the rocket was estimated to reach 1048 feet in altitude. Maximum speed was 284 ft/s, and speed at landing was 25.7 ft/s. Speed at parachute deployment was 13.4 ft/s. Simulated altitude, vertical velocity, and vertical acceleration graphs can be seen Figure 5-1.

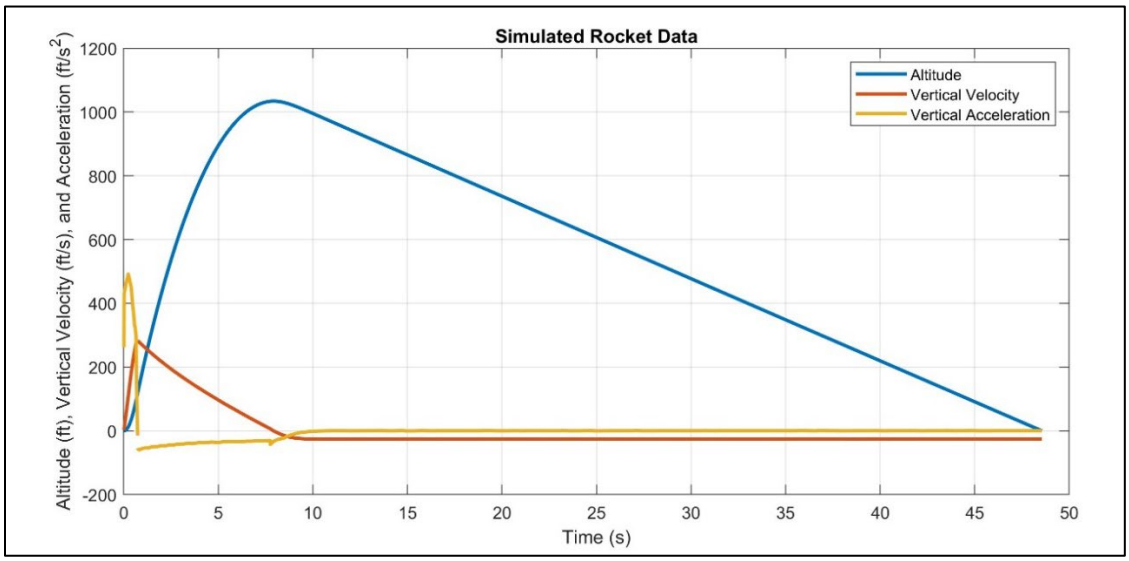

Figure 5.1. Simulated Data, Part One

Altitude was recorded on the custom sensor package from the GPS and pressure sensor. Figure 5-2 shows a comparison between pressure sensor (BMP390) and GPS readings.

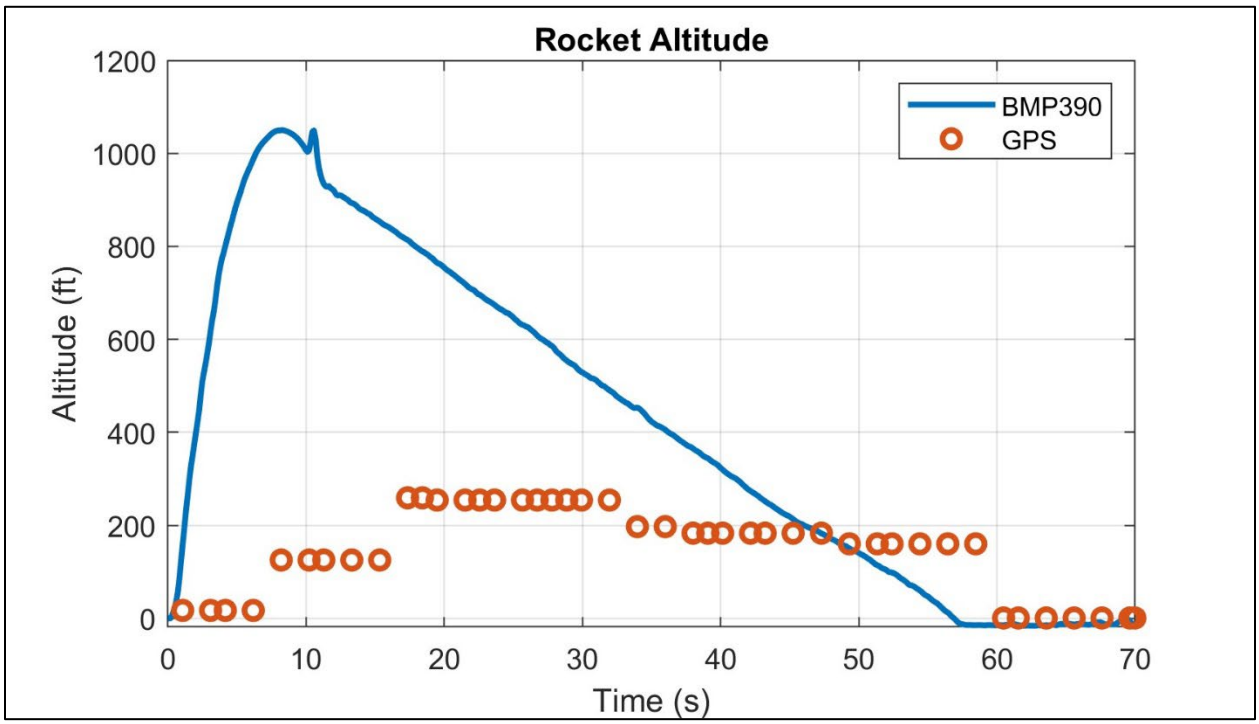

Figure 5.2. Altitude Readings from the Custom Sensor

The GPS did not record data as frequently as the other sensors. While the other sensors operated at around 8.4 Hz, the GPS recorded data at around 0.63 Hz. Additionally, the altitude as recorded by the GPS was found to be exceedingly low in comparison with the pressure sensor and simulated results.

The pressure sensor altitude reading was very close to the simulated altitude during the rocket's ascent, as shown in Figure 5-3. Then, there is a spike in altitude during around time t = 10s, which is the time of parachute deployment. At first it was suspected that this was due to high pressure caused by the motor ejection charge, but upon reviewing footage of the flight, it

appears that the rocket was pulled upward by the parachute, and the spike does actually correspond to a momentary increase in altitude.

Velocity was calculated from simulated data using an Unscented Kalman Filter (UKF), also plotted in Figure 5-3. Note that the domain of this filter only ranges from around time 0 s < t < 10 s, after which the parachute deploys. Velocity rises during the propellant burn time, lasting 0.7 seconds. During that time, the velocity from the filter and the simulation are similar, but from there, the filter becomes unsteady.

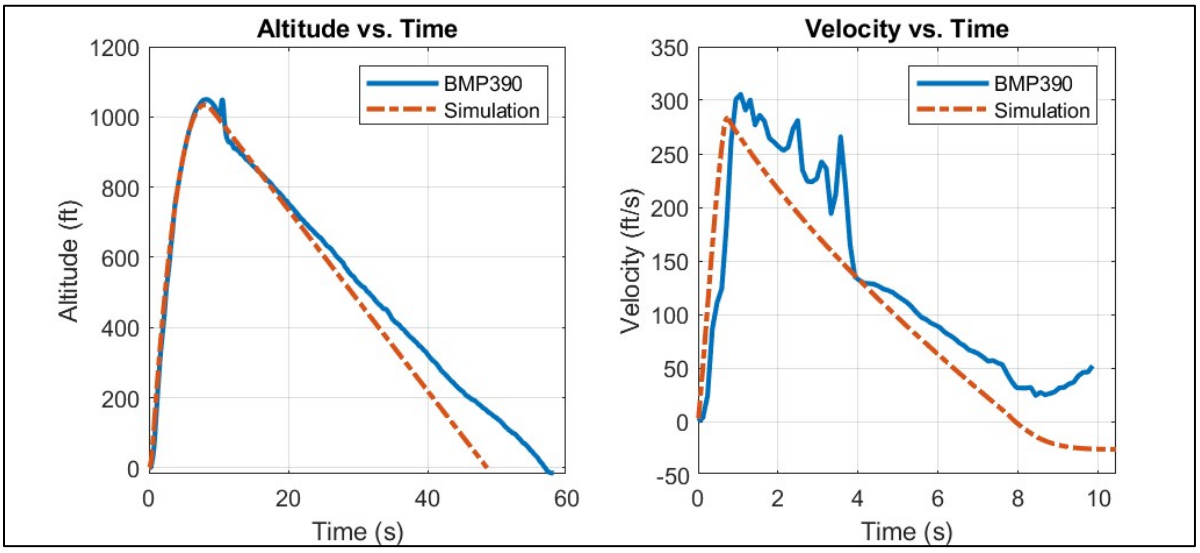

Figure 5.3. Simulation Compared to Measured and Filtered Data

Acceleration was recorded from two IMUs. Acceleration readings are shown in Figure 5-4, and gyroscope readings are shown in Figure 5-5. Note that accelerations had to be rotated so that they start at 0 degrees.

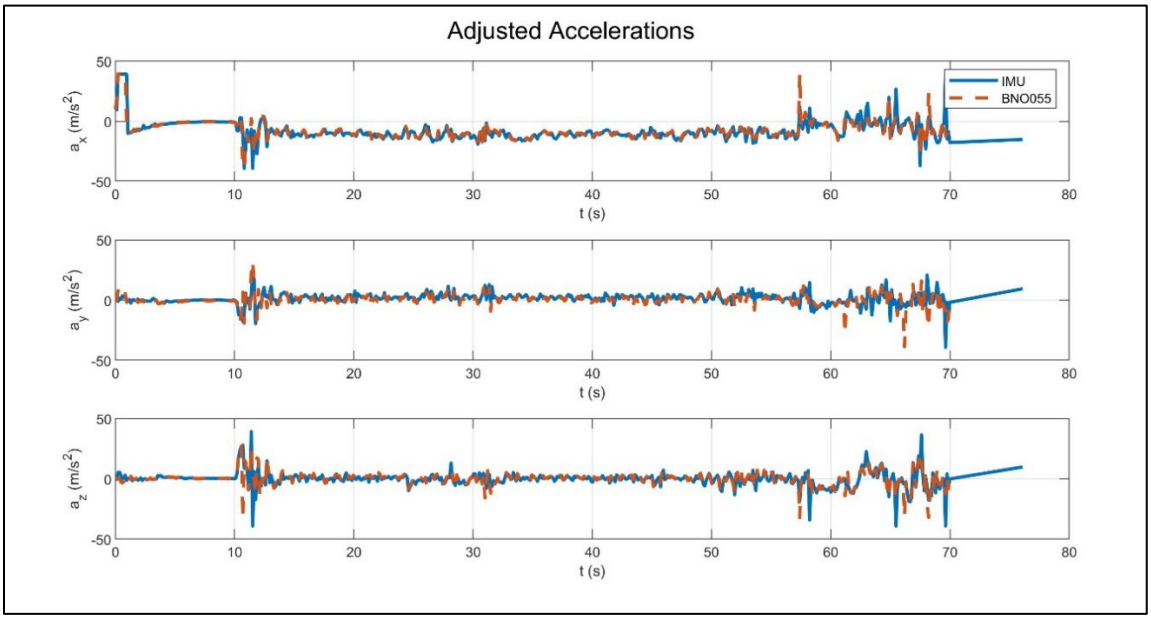

Figure 5.4. Acceleration Readings from the Inertial Measurement Units

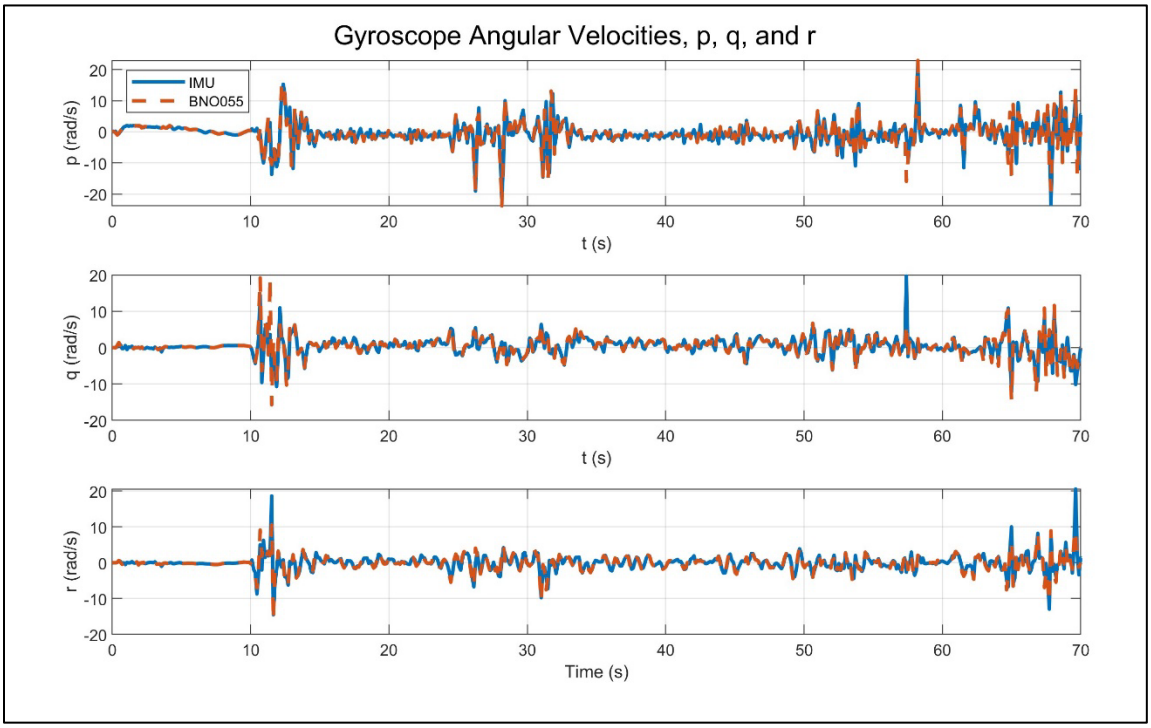

Figure 5.5. Gyroscope Readings from the Inertial Measurement Units

From these graphs, it can be seen that acceleration and angular velocity readings were similar between the two sensors. Seeing that the difference in data is not significant, for the purposes of this project, the smaller of the two IMUs will be chosen for the final design in Part Two. This board will be lighter in weight and contain a port for a STEMMA QT connector.

Analysis of Part Two

The second part was performed at 31 degrees Fahrenheit and 30.2 inHg ground level pressure. Elevation was 434 feet above sea level, and wind blew between 5 and 10 mph. Three launches were performed with the streamlined custom sensor package, and three others were performed with the JollyLogic altimeter to allow for comparison. The Estes AltiTrak was used for all launches.

In simulation, the rocket was estimated to reach an apogee of 157 feet. Maximum speed was 93.2 ft/s, and speed at landing was 12.2 ft/s. Speed at parachute deployment was 43.4 ft/s. Simulated altitude, vertical velocity, and vertical acceleration graphs can be seen in Figure 5-6.

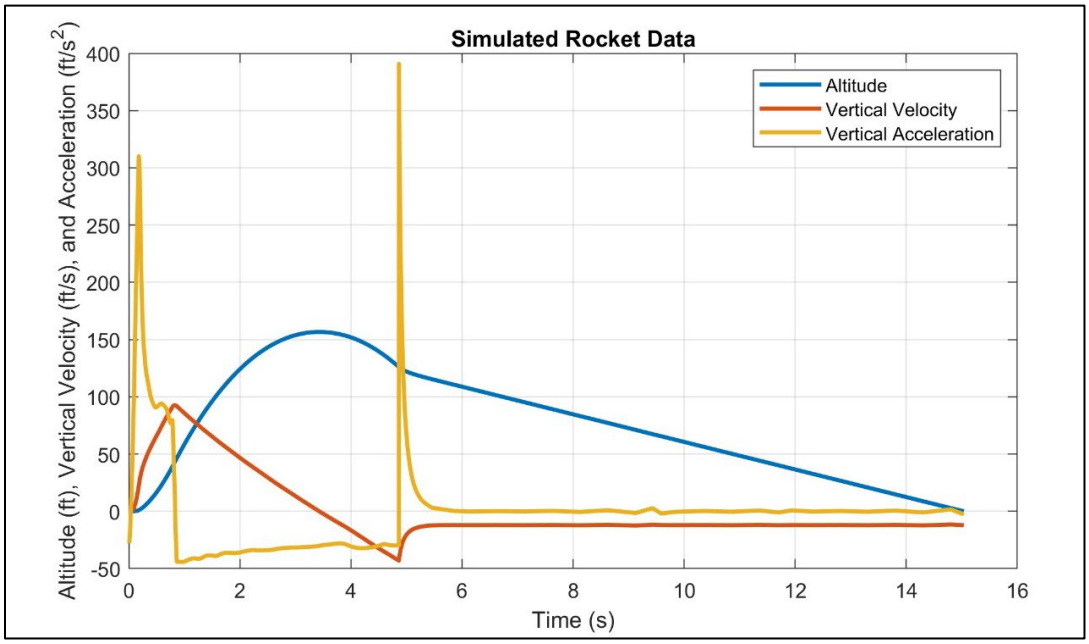

Figure 5.6. Simulated Data, Part Two

A summary of flights with low-power motors is given in Table 5-1.

Table 5.1. Low-Power Flight Summary

Flight Number	Altitude (ft) from AltiTrak	Altitude (ft) from AltimeterTwo	Altitude (ft) from BMP390	Max Speed (ft/s) from AltimeterTwo	Max Speed (ft/s) from UKF
1	154	196	-	122	-
2	160	182	-	123	-
3	154	193	-	122	-
4	148	-	No reading	-	No reading
5	160	-	186	-	111
6	No reading	-	173	-	102

Additionally, one other flight was performed on the same day and contained both the JollyLogic AltimeterTwo and the streamlined sensor package. This launch serves as another point of comparison between the sensors, especially given that no AltimeterTwo data was able to be collected in Part One of this work. This flight used a C6 motor. The flight summary is shown in Table 5-2.

Table 5.2. Comparison between JollyLogic AltimeterTwo and Streamlined Custom Sensor Package

Altitude (ft) from Simulation	Altitude (ft) from AltiTrak	Altitude (ft) from AltimeterTwo	Altitude (ft) from BMP390	Max Speed (ft/s) from Simulation	Max Speed (ft/s) from AltimeterTwo	Max Speed (ft/s) from UKF
429	265	498	435	161	173	204

And the full flight profile is graphed in Figure 5-7 .

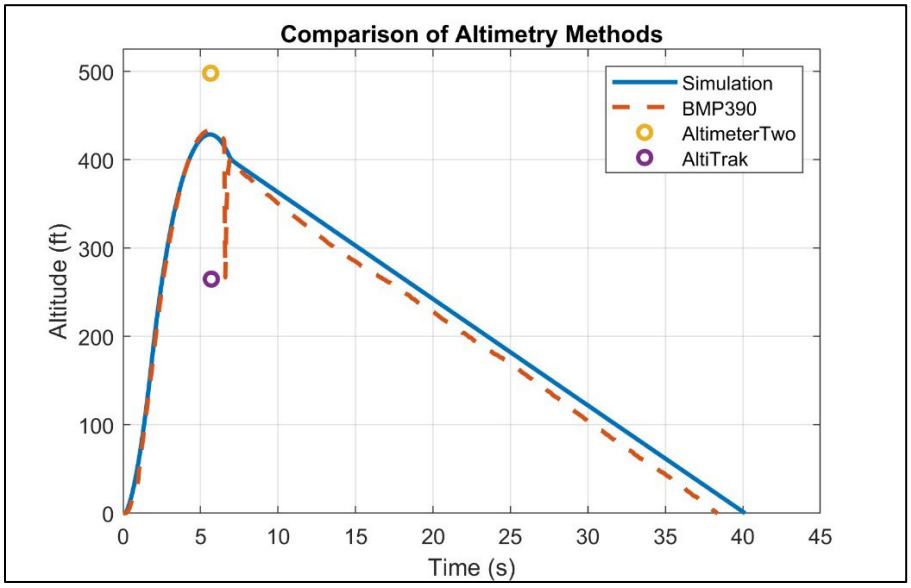

Figure 5.7. Comparison of Altimetry Methods

From these launches, it can be seen that the BMP390 tends to read lower altitudes than the AltimeterTwo. However, the custom sensor readings are still similar. There was also one flight (Flight 4) that did not produce a reading on the custom sensor. While two minutes of data were logged on the sensor during rocket preparation, data collection stopped before the launch occurred. After discovering this, the battery was examined and was found to be charged, so battery failure was not the cause. Additionally, the data that was collected appears normal, apart from ending collection early. One possible explanation is simply that the battery was not securely plugged into the sensor, and it came loose during rocket preparation.

Speed calculations were higher than simulated maximum speed, but lower than the AltimeterTwo readings for Flights 1 through 6. The flights with the AltimeterTwo measurements had the same maximum velocity (between 122 and 123 ft/s), while speed calculations from the UKF varied by 9 ft/s. This indicates that the speed calculations from the UKF are not as reliable as those from the AltimeterTwo.

Readings from the Estes AltiTrak were the least accurate, but this is to be expected based on the many opportunities for human error and imprecision of the measurement technique. Perhaps readings could be improved by increasing distance from the launchpad or checking the wind direction more frequently.

Chapter 6

CONCLUSION

The final model for this project has some variations depending on what applications are required. Overall, the streamlined sensor includes the setup as outlined in Part Two of the Experimental Setup: the datalogger and microSD card, BMP390 pressure sensor, BNO055 IMU, a LiPo battery, and 2 STEMMA QT cables. Data collected with this setup was at most 4 MB per flight, so a low-capacity microSD card could be substituted for the 8GB card used in this setup. Replacing the microSD card with an Adafruit 64MB card, this setup costs $67.20. This is not much less than the JollyLogic AltimeterTwo, which costs $79.95. For this reason, it is only recommended using the full setup if an IMU application is required.

However, if the IMU is not needed, the BNO055 IMU and one STEMMA QT connector can be removed from the design, bringing the cost down to $36.30. This makes the cost of the custom sensor comparable to the EggTimer altimeters, but with the benefit of not requiring any soldering. Costing less than half of the price of the JollyLogic AltimeterTwo, this could be incredibly beneficial for novices and student groups looking to estimate the altitude of their rockets. Additionally, the custom sensor allows the user to visualize the entire flight path, from takeoff to landing. Most other devices only have key events such as takeoff, apogee, parachute deployment, and landing. Having a full flight path could be useful for troubleshooting in the event of a flight issue, and overall gives a better picture of the rocket's behavior.

Removing the IMU would also make requirements for the electronics bay less restrictive. With the IMU, the sensor must be placed such that its orientation relative to the rocket is known, and the sensor must be fixed in place. Barometers do not have these limitations, and while they

should not be loose inside the rocket, less precise attachment methods are acceptable to use. Still, in the event that a cable connection comes undone, the sensor should be placed in such a way that components will remain affixed to the rocket.

Overall, users of this custom sensor package should be aware that it is not perfect, but it gets the job done. Like with any other altimeter, power and cable connections should be verified before each launch, and readings may not be as accurate as higher-end altimeters. However, for the purposes of this study, the custom sensor is useful for low-cost altitude estimation and solder-free installation, thus achieving the goal of lowering barriers to entry for students and novice rocketeers.

Appendix A

FIRST ITERATION OF ARDUINO CODE

combineSensors.ino

```
//Read to SD

/*
Combines the following microcontroller and sensors so that they read to an SD
card:
Adafruit Feather RP2040
Adafruit Ultimate GPS Featherwing
Adafruit Adafruit ISM330DHCX + LIS3MDL FeatherWing
Adafruit BMP390
Adafruit BNO055

Code is adapted from Adafruit's open-source examples
*/

// time variable
unsigned long currentTime;

// boolean variable
bool goWrite = false;

//SD
#include "SdFat.h" //include SD library
#include <SPI.h>
#define SD_CS_PIN 23
SdFat SD;
FsFile myFile;
SdSpiConfig config(SD_CS_PIN, DEDICATED_SPI, SD_SCK_MHZ(16), &SPI1);

//GPS
#include <Adafruit_GPS.h>
// what's the name of the hardware serial port?
#define GPSSerial Serial1
```

```
// Connect to the GPS on the hardware port
Adafruit_GPS GPS(&GPSSerial);
// Set GPSECHO to 'false' to turn off echoing the GPS data to the Serial console
// Set to 'true' if you want to debug and listen to the raw GPS sentences
#define GPSECHO false
uint32_t timer = millis();

//IMU
#include <Adafruit_ISM330DHCX.h>
Adafruit_ISM330DHCX lsm6ds;
#include <Adafruit_LIS3MDL.h>
Adafruit_LIS3MDL lis3mdl;

//BMP390
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include "Adafruit_BMP3XX.h"
#define BMP_SCK 13
#define BMP_MISO 12
#define BMP_MOSI 11
#define BMP_CS 10
#define SEALEVELPRESSURE_HPA (1013.25)
Adafruit_BMP3XX bmp;

//BNO055
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
Adafruit_BNO055 bno = Adafruit_BNO055(55);

uint32_t loopTimer = millis();
uint32_t delayTime = 50;
uint32_t gpsTimer = millis();
uint32_t gpsDelay = 1000;

void setup() {

  // Set up sd card and sensors
  sdSetup();
  myFile = SD.open("test.txt", FILE_WRITE);
  gpsSetup();
  imuSetup();
```

```
  bmpSetup();
  bnoSetup();
}

void loop() {

  currentTime = millis();

  if((millis() - loopTimer) > delayTime){
    myFile.print(currentTime); myFile.print(",");

    imuLoop();
    bmpLoop();
    bnoLoop();

    goWrite = true;
  }

  gpsLoop();

  if(goWrite){

    loopTimer = millis();
    myFile.println("\n");
    myFile.flush();

    goWrite = false;
  }

}
```

bmpLoop.ino

```
void bmpLoop(){

  if (!bmp.performReading()) {
    Serial.println("Failed to perform reading :(");
    myFile.println(",");
    myFile.println(",");
    myFile.println(",");
    return;
  } else {

  //print to serial monitor
  Serial.print("Temperature = ");
  Serial.print(bmp.temperature);
  Serial.println(" *C");
  Serial.print("Pressure = ");
  Serial.print(bmp.pressure / 100.0);
  Serial.println(" hPa");
  Serial.print("Approx. Altitude = ");
  Serial.print(bmp.readAltitude(SEALEVELPRESSURE_HPA));
  Serial.println(" m");
  Serial.println();

  //print to myFile
  //temp in deg C
  myFile.print(bmp.temperature); myFile.print(",");
  //pressure in hPa
  myFile.print(bmp.pressure / 100.0); myFile.print(",");
  //altitude in m
  myFile.print(bmp.readAltitude(SEALEVELPRESSURE_HPA)); myFile.print(",");
  }
}
```

bmpSetup.ino

```
void bmpSetup(){

  if (!bmp.begin_I2C()) { // hardware I2C mode, can pass in address & alt Wire
    Serial.println("Could not find a valid BMP3 sensor, check wiring!");
    while (1);
  }
  // Set up oversampling and filter initialization
  bmp.setTemperatureOversampling(BMP3_OVERSAMPLING_8X);
  bmp.setPressureOversampling(BMP3_OVERSAMPLING_4X);
  bmp.setIIRFilterCoeff(BMP3_IIR_FILTER_COEFF_3);
  bmp.setOutputDataRate(BMP3_ODR_50_HZ);

}
```

bnoLoop.ino

```
void bnoLoop(){

  // Get a new sensor event
  sensors_event_t event;
  bno.getEvent(&event);

  //print to serial monitor
  Serial.println("bno connected");

  //set up vectors
  imu::Quaternion quat = bno.getQuat();
  imu::Vector<3> magnet = bno.getVector(Adafruit_BNO055::VECTOR_MAGNETOMETER);
  imu::Vector<3> gyro = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
  imu::Vector<3> euler = bno.getVector(Adafruit_BNO055::VECTOR_EULER);
  imu::Vector<3> accel = bno.getVector(Adafruit_BNO055::VECTOR_ACCELEROMETER);
  imu::Vector<3> linaccel = bno.getVector(Adafruit_BNO055::VECTOR_LINEARACCEL);
  imu::Vector<3> acc = bno.getVector(Adafruit_BNO055::VECTOR_ACCELEROMETER);
  imu::Vector<3> gravity = bno.getVector(Adafruit_BNO055::VECTOR_GRAVITY);

  //print to myFile
  myFile.print(quat.w());     myFile.print(",");
  myFile.print(quat.x());     myFile.print(",");
  myFile.print(quat.y());     myFile.print(",");
  myFile.print(quat.z());     myFile.print(",");
  myFile.print(bno.getTemp());myFile.print(",");
  myFile.print(magnet.x());   myFile.print(",");
  myFile.print(magnet.y());   myFile.print(",");
  myFile.print(magnet.z());   myFile.print(",");
  myFile.print(gyro.x());     myFile.print(",");
  myFile.print(gyro.y());     myFile.print(",");
  myFile.print(gyro.z());     myFile.print(",");
  myFile.print(euler.x());    myFile.print(",");
  myFile.print(euler.y());    myFile.print(",");
  myFile.print(euler.z());    myFile.print(",");
  myFile.print(acc.x());      myFile.print(",");
  myFile.print(acc.y());      myFile.print(",");
  myFile.print(acc.z());      myFile.print(",");
  myFile.print(linaccel.x()); myFile.print(",");
  myFile.print(linaccel.y()); myFile.print(",");
  myFile.print(linaccel.z()); myFile.print(",");
```

```
  myFile.print(gravity.x());  myFile.print(",");
  myFile.print(gravity.y());  myFile.print(",");
  myFile.print(gravity.z());  myFile.print(",");
  myFile.print(event.orientation.x, 4); myFile.print(",");
  myFile.print(event.orientation.y, 4); myFile.print(",");
  myFile.print(event.orientation.z, 4); myFile.print(",");
}
```

bnoSetup.ino

```
void bnoSetup(){
  // Initialise the sensor
  if(!bno.begin())
  {
    // There was a problem detecting the BNO055 ... check your connections
    Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
    while(1);
  }

  bno.setExtCrystalUse(true);
}
```

gpsLoop.ino

```
void gpsLoop(){

  // read data from the GPS in the 'main loop'

  char c = GPS.read();

  if (GPSECHO)
    if (c) Serial.print(c);
  // if a sentence is received, we can check the checksum, parse it...
  if (GPS.newNMEAreceived()) {
    // a tricky thing here is if we print the NMEA sentence, or data
    // we end up not listening and catching other sentences!
    // so be very wary if using OUTPUT_ALLDATA and trying to print out data
    Serial.print(GPS.lastNMEA()); // this also sets the newNMEAreceived() flag to
false
    if (!GPS.parse(GPS.lastNMEA())) // this also sets the newNMEAreceived() flag
to false
      return; // we can fail to parse a sentence in which case we should just
wait for another
  }


  if((millis() - gpsTimer) > gpsDelay){
    //Serial monitor
    gpsTimer = millis();
    Serial.print("\nTime: ");
    if (GPS.hour < 10) { Serial.print('0'); }
    Serial.print(GPS.hour, DEC); Serial.print(':');
    if (GPS.minute < 10) { Serial.print('0'); }
    Serial.print(GPS.minute, DEC); Serial.print(':');
    if (GPS.seconds < 10) { Serial.print('0'); }
    Serial.print(GPS.seconds, DEC); Serial.print('.');
    if (GPS.milliseconds < 10) {
      Serial.print("00");
    } else if (GPS.milliseconds > 9 && GPS.milliseconds < 100) {
      Serial.print("0");
    }
    Serial.println(GPS.milliseconds);
    Serial.print("Date: ");
    Serial.print(GPS.day, DEC); Serial.print('/');
```

```
Serial.print(GPS.month, DEC); Serial.print("/20");
Serial.println(GPS.year, DEC);
Serial.print("Fix: "); Serial.print((int)GPS.fix);
Serial.print(" quality: "); Serial.println((int)GPS.fixquality);
if (GPS.fix) {
  Serial.print("Location: ");
  Serial.print(GPS.latitude, 4); Serial.print(GPS.lat);
  Serial.print(", ");
  Serial.print(GPS.longitude, 4); Serial.println(GPS.lon);
  Serial.print("Speed (knots): "); Serial.println(GPS.speed);
  Serial.print("Angle: "); Serial.println(GPS.angle);
  Serial.print("Altitude: "); Serial.println(GPS.altitude);
  Serial.print("Satellites: "); Serial.println((int)GPS.satellites);
  Serial.print("Antenna status: "); Serial.println((int)GPS.antenna);
}

//print to SD card
//print time to myFile
if (goWrite){
  if (GPS.hour < 10) { myFile.print('0'); }
  myFile.print(GPS.hour, DEC); myFile.print(':');
  if (GPS.minute < 10) { myFile.print('0'); }
  myFile.print(GPS.minute, DEC); myFile.print(':');
  if (GPS.seconds < 10) { myFile.print('0'); }
  myFile.print(GPS.seconds, DEC); myFile.print('.');
  if (GPS.milliseconds < 10) {
    myFile.print("00");
  } else if (GPS.milliseconds > 9 && GPS.milliseconds < 100) {
    myFile.print("0");
  }
  myFile.print(",");
  myFile.print(GPS.milliseconds);
  myFile.print(",");

  //print date to file
  myFile.print(GPS.day, DEC); myFile.print('/');
  myFile.print(GPS.month, DEC); myFile.print("/20");
  myFile.print(GPS.year, DEC);
  myFile.print(",");
  myFile.print((int)GPS.fix);
  myFile.print(",");
  myFile.print((int)GPS.fixquality); //quality
  myFile.print(",");
```

```
      if (GPS.fix) {
        //print location to file
        myFile.print(GPS.latitude, 4); myFile.print(GPS.lat);
        myFile.print(", ");
        myFile.print(GPS.longitude, 4); myFile.print(GPS.lon);
        myFile.print(", ");
        myFile.print(GPS.speed); //speed in knots
        myFile.print(", ");
        myFile.print(GPS.angle); //angle
        myFile.print(", ");
        myFile.print(GPS.altitude); //altitude
        myFile.print(", ");
        myFile.print((int)GPS.satellites); //satellites
        myFile.print(", ");
        myFile.print((int)GPS.antenna); //antenna status
        myFile.print(", ");
      } else {
        myFile.print(", ");
        myFile.print(", ");
        myFile.print(", ");
        myFile.print(", ");
        myFile.print(", ");
        myFile.print(", ");
        myFile.print(", ");
      }
    }
  }
}
```

gpsSetup.ino

```
void gpsSetup(){
  // 9600 NMEA is the default baud rate for Adafruit MTK GPS's- some use 4800
  // 115200 is used here to match with the other sensors
  Serial.begin(115200);
  Serial.println("Adafruit GPS library basic parsing test!");
  GPS.begin(9600);

  // uncomment this line to turn on RMC (recommended minimum) and GGA (fix data)
including altitude
  GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);
  // uncomment this line to turn on only the "minimum recommended" data
  //GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCONLY);

  // Set the update rate, use 1Hz MAX
  GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ); // 1 Hz update rate

  // Request updates on antenna status, comment out to keep quiet
  GPS.sendCommand(PGCMD_ANTENNA);

  // Ask for firmware version
  GPSSerial.println(PMTK_Q_RELEASE);
}
```

imuLoop.ino

```
void imuLoop(){

  sensors_event_t accel, gyro, mag, temp;

  // Get new normalized sensor events
  lsm6ds.getEvent(&accel, &gyro, &temp);
  lis3mdl.getEvent(&mag);


  //print to serial monitor
  // Display the results (acceleration is measured in m/s^2)
  Serial.print("\t\tAccel X: ");
  Serial.print(accel.acceleration.x, 4);
  Serial.print(" \tY: ");
  Serial.print(accel.acceleration.y, 4);
  Serial.print(" \tZ: ");
  Serial.print(accel.acceleration.z, 4);
  Serial.println(" \tm/s^2 ");

  // Display the results (rotation is measured in rad/s)
  Serial.print("\t\tGyro X: ");
  Serial.print(gyro.gyro.x, 4);
  Serial.print(" \tY: ");
  Serial.print(gyro.gyro.y, 4);
  Serial.print(" \tZ: ");
  Serial.print(gyro.gyro.z, 4);
  Serial.println(" \tradians/s ");

  // Display the results (magnetic field is measured in uTesla)
  Serial.print(" \t\tMag X: ");
  Serial.print(mag.magnetic.x, 4);
  Serial.print(" \tY: ");
  Serial.print(mag.magnetic.y, 4);
  Serial.print(" \tZ: ");
  Serial.print(mag.magnetic.z, 4);
  Serial.println(" \tuTesla ");
  Serial.print("\t\tTemp :\t\t\t\t\t");
  Serial.print(temp.temperature);
  Serial.println(" \tdeg C");
  Serial.println();
```

```
  //print to file
  //acceleration in m/s^2
  myFile.print(accel.acceleration.x, 4); myFile.print(",");
  myFile.print(accel.acceleration.y, 4); myFile.print(",");
  myFile.print(accel.acceleration.z, 4); myFile.print(",");

  // rotation in rad/s
  myFile.print(gyro.gyro.x, 4); myFile.print(",");
  myFile.print(gyro.gyro.y, 4); myFile.print(",");
  myFile.print(gyro.gyro.z, 4); myFile.print(",");

  // magnetic field measured uTesla
  myFile.print(mag.magnetic.x, 4); myFile.print(",");
  myFile.print(mag.magnetic.y, 4); myFile.print(",");
  myFile.print(mag.magnetic.z, 4); myFile.print(",");

  // temperature in deg C
  myFile.print(temp.temperature); myFile.print(",");
}
```

imuSetup.ino

```
void imuSetup(){

  bool lsm6ds_success, lis3mdl_success;

  // hardware I2C mode, can pass in address & alt Wire
  lsm6ds_success = lsm6ds.begin_I2C();
  lis3mdl_success = lis3mdl.begin_I2C();

  if (!lsm6ds_success){
    Serial.println("Failed to find LSM6DS chip");
  }
  if (!lis3mdl_success){
    Serial.println("Failed to find LIS3MDL chip");
  }
  if (!(lsm6ds_success && lis3mdl_success)) {
    while (1) {
      delay(10);
    }
  }
  Serial.println("LSM6DS and LIS3MDL Found!");

  // lsm6ds.setAccelRange(LSM6DS_ACCEL_RANGE_2_G);
  Serial.print("Accelerometer range set to: ");
  switch (lsm6ds.getAccelRange()) {
    case LSM6DS_ACCEL_RANGE_2_G:
      Serial.println("+-2G");
      break;
    case LSM6DS_ACCEL_RANGE_4_G:
      Serial.println("+-4G");
      break;
    case LSM6DS_ACCEL_RANGE_8_G:
      Serial.println("+-8G");
      break;
    case LSM6DS_ACCEL_RANGE_16_G:
      Serial.println("+-16G");
      break;
  }

  // lsm6ds.setAccelDataRate(LSM6DS_RATE_12_5_HZ);
  Serial.print("Accelerometer data rate set to: ");
```

```
switch (lsm6ds.getAccelDataRate()) {
case LSM6DS_RATE_SHUTDOWN:
  Serial.println("0 Hz");
  break;
case LSM6DS_RATE_12_5_HZ:
  Serial.println("12.5 Hz");
  break;
case LSM6DS_RATE_26_HZ:
  Serial.println("26 Hz");
  break;
case LSM6DS_RATE_52_HZ:
  Serial.println("52 Hz");
  break;
case LSM6DS_RATE_104_HZ:
  Serial.println("104 Hz");
  break;
case LSM6DS_RATE_208_HZ:
  Serial.println("208 Hz");
  break;
case LSM6DS_RATE_416_HZ:
  Serial.println("416 Hz");
  break;
case LSM6DS_RATE_833_HZ:
  Serial.println("833 Hz");
  break;
case LSM6DS_RATE_1_66K_HZ:
  Serial.println("1.66 KHz");
  break;
case LSM6DS_RATE_3_33K_HZ:
  Serial.println("3.33 KHz");
  break;
case LSM6DS_RATE_6_66K_HZ:
  Serial.println("6.66 KHz");
  break;
}

// lsm6ds.setGyroRange(LSM6DS_GYRO_RANGE_250_DPS );
Serial.print("Gyro range set to: ");
switch (lsm6ds.getGyroRange()) {
case LSM6DS_GYRO_RANGE_125_DPS:
  Serial.println("125 degrees/s");
  break;
case LSM6DS_GYRO_RANGE_250_DPS:
```

```
  Serial.println("250 degrees/s");
  break;
case LSM6DS_GYRO_RANGE_500_DPS:
  Serial.println("500 degrees/s");
  break;
case LSM6DS_GYRO_RANGE_1000_DPS:
  Serial.println("1000 degrees/s");
  break;
case LSM6DS_GYRO_RANGE_2000_DPS:
  Serial.println("2000 degrees/s");
  break;
case ISM330DHCX_GYRO_RANGE_4000_DPS:
  Serial.println("4000 degrees/s");
  break;
}

// lsm6ds.setGyroDataRate(LSM6DS_RATE_12_5_HZ);
Serial.print("Gyro data rate set to: ");
switch (lsm6ds.getGyroDataRate()) {
case LSM6DS_RATE_SHUTDOWN:
  Serial.println("0 Hz");
  break;
case LSM6DS_RATE_12_5_HZ:
  Serial.println("12.5 Hz");
  break;
case LSM6DS_RATE_26_HZ:
  Serial.println("26 Hz");
  break;
case LSM6DS_RATE_52_HZ:
  Serial.println("52 Hz");
  break;
case LSM6DS_RATE_104_HZ:
  Serial.println("104 Hz");
  break;
case LSM6DS_RATE_208_HZ:
  Serial.println("208 Hz");
  break;
case LSM6DS_RATE_416_HZ:
  Serial.println("416 Hz");
  break;
case LSM6DS_RATE_833_HZ:
  Serial.println("833 Hz");
  break;
```

```
case LSM6DS_RATE_1_66K_HZ:
  Serial.println("1.66 KHz");
  break;
case LSM6DS_RATE_3_33K_HZ:
  Serial.println("3.33 KHz");
  break;
case LSM6DS_RATE_6_66K_HZ:
  Serial.println("6.66 KHz");
  break;
}

lis3mdl.setDataRate(LIS3MDL_DATARATE_155_HZ);
// You can check the datarate by looking at the frequency of the DRDY pin
Serial.print("Magnetometer data rate set to: ");
switch (lis3mdl.getDataRate()) {
  case LIS3MDL_DATARATE_0_625_HZ: Serial.println("0.625 Hz"); break;
  case LIS3MDL_DATARATE_1_25_HZ: Serial.println("1.25 Hz"); break;
  case LIS3MDL_DATARATE_2_5_HZ: Serial.println("2.5 Hz"); break;
  case LIS3MDL_DATARATE_5_HZ: Serial.println("5 Hz"); break;
  case LIS3MDL_DATARATE_10_HZ: Serial.println("10 Hz"); break;
  case LIS3MDL_DATARATE_20_HZ: Serial.println("20 Hz"); break;
  case LIS3MDL_DATARATE_40_HZ: Serial.println("40 Hz"); break;
  case LIS3MDL_DATARATE_80_HZ: Serial.println("80 Hz"); break;
  case LIS3MDL_DATARATE_155_HZ: Serial.println("155 Hz"); break;
  case LIS3MDL_DATARATE_300_HZ: Serial.println("300 Hz"); break;
  case LIS3MDL_DATARATE_560_HZ: Serial.println("560 Hz"); break;
  case LIS3MDL_DATARATE_1000_HZ: Serial.println("1000 Hz"); break;
}

lis3mdl.setRange(LIS3MDL_RANGE_4_GAUSS);
Serial.print("Range set to: ");
switch (lis3mdl.getRange()) {
  case LIS3MDL_RANGE_4_GAUSS: Serial.println("+-4 gauss"); break;
  case LIS3MDL_RANGE_8_GAUSS: Serial.println("+-8 gauss"); break;
  case LIS3MDL_RANGE_12_GAUSS: Serial.println("+-12 gauss"); break;
  case LIS3MDL_RANGE_16_GAUSS: Serial.println("+-16 gauss"); break;
}

lis3mdl.setPerformanceMode(LIS3MDL_MEDIUMMODE);
Serial.print("Magnetometer performance mode set to: ");
switch (lis3mdl.getPerformanceMode()) {
  case LIS3MDL_LOWPOWERMODE: Serial.println("Low"); break;
  case LIS3MDL_MEDIUMMODE: Serial.println("Medium"); break;
```

```
    case LIS3MDL_HIGHMODE: Serial.println("High"); break;
    case LIS3MDL_ULTRAHIGHMODE: Serial.println("Ultra-High"); break;
  }

  lis3mdl.setOperationMode(LIS3MDL_CONTINUOUSMODE);
  Serial.print("Magnetometer operation mode set to: ");
  // Single shot mode will complete conversion and go into power down
  switch (lis3mdl.getOperationMode()) {
    case LIS3MDL_CONTINUOUSMODE: Serial.println("Continuous"); break;
    case LIS3MDL_SINGLEMODE: Serial.println("Single mode"); break;
    case LIS3MDL_POWERDOWNMODE: Serial.println("Power-down"); break;
  }

  lis3mdl.setIntThreshold(500);
  lis3mdl.configInterrupt(false, false, true, // enable z axis
                          true, // polarity
                          false, // don't latch
                          true); // enabled!

}
```

sdSetup.ino

```
void sdSetup(){

  // Open serial communications and wait for port to open:
  Serial.begin(115200);
  Serial.print("Initializing SD card...");
  // Retry mechanism for SD card initialization
  while (!SD.begin(config)) {
    Serial.println("initialization failed! Retrying...");
  }
  Serial.println("initialization done.");
  // make sure file opens
  myFile = SD.open("test.txt", FILE_WRITE);
  if (myFile) {
    Serial.print("File opened successfully");
    myFile.close();
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }

}
```

Appendix B

SECOND ITERATION OF ARDUINO CODE

streamlinedSensors.ino

```
/*
Combines the following microcontroller and sensors so that they read to an SD
card:
Adafruit Feather RP2040
Adafruit BMP390
Adafruit BNO055

Code is adapted from Adafruit's open-source examples
*/

// time variable
unsigned long currentTime;

//SD
#include "SdFat.h" //include SD library
#include <SPI.h>
#define SD_CS_PIN 23
SdFat SD;
FsFile myFile;
SdSpiConfig config(SD_CS_PIN, DEDICATED_SPI, SD_SCK_MHZ(16), &SPI1);

//BMP390
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include "Adafruit_BMP3XX.h"
#define BMP_SCK 13
#define BMP_MISO 12
#define BMP_MOSI 11
#define BMP_CS 10
#define SEALEVELPRESSURE_HPA (1013.25)
Adafruit_BMP3XX bmp;

//BNO055
#include <Wire.h>
#include <Adafruit_Sensor.h>
```

```
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
Adafruit_BNO055 bno = Adafruit_BNO055(55);

void setup() {

  // Set up sd card and sensors
  sdSetup();
  myFile = SD.open("test.txt", FILE_WRITE);
  bmpSetup();
  bnoSetup();
}

void loop() {
  currentTime = millis();

  myFile.print(currentTime); myFile.print(",");
  bmpLoop();
  bnoLoop();
  myFile.println("\n");
  myFile.flush();
}
```

bmpLoop.ino

```
void bmpLoop(){

  if (!bmp.performReading()) {
    Serial.println("Failed to perform reading :(");
    myFile.println(",");
    myFile.println(",");
    myFile.println(",");
    return;
  } else {

  //print to serial monitor
  Serial.print("Temperature = ");
  Serial.print(bmp.temperature);
  Serial.println(" *C");
  Serial.print("Pressure = ");
  Serial.print(bmp.pressure / 100.0);
  Serial.println(" hPa");
  Serial.print("Approx. Altitude = ");
  Serial.print(bmp.readAltitude(SEALEVELPRESSURE_HPA));
  Serial.println(" m");
  Serial.println();

  //print to myFile
  //temp in deg C
  myFile.print(bmp.temperature); myFile.print(",");
  //pressure in hPa
  myFile.print(bmp.pressure / 100.0); myFile.print(",");
  //altitude in m
  myFile.print(bmp.readAltitude(SEALEVELPRESSURE_HPA)); myFile.print(",");
  }
}
```

bmpSetup.ino

```
void bmpSetup(){

  if (!bmp.begin_I2C()) { // hardware I2C mode, can pass in address & alt Wire
    Serial.println("Could not find a valid BMP3 sensor, check wiring!");
    while (1);
  }
  // Set up oversampling and filter initialization
  bmp.setTemperatureOversampling(BMP3_OVERSAMPLING_8X);
  bmp.setPressureOversampling(BMP3_OVERSAMPLING_4X);
  bmp.setIIRFilterCoeff(BMP3_IIR_FILTER_COEFF_3);
  bmp.setOutputDataRate(BMP3_ODR_50_HZ);

}
```

bnoLoop.ino

```
void bnoLoop(){

  // Get a new sensor event
  sensors_event_t event;
  bno.getEvent(&event);

  //print to serial monitor
  Serial.println("bno connected");

  //set up vectors
  imu::Quaternion quat = bno.getQuat();
  imu::Vector<3> magnet = bno.getVector(Adafruit_BNO055::VECTOR_MAGNETOMETER);
  imu::Vector<3> gyro = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
  imu::Vector<3> euler = bno.getVector(Adafruit_BNO055::VECTOR_EULER);
  imu::Vector<3> accel = bno.getVector(Adafruit_BNO055::VECTOR_ACCELEROMETER);
  imu::Vector<3> linaccel = bno.getVector(Adafruit_BNO055::VECTOR_LINEARACCEL);
  imu::Vector<3> acc = bno.getVector(Adafruit_BNO055::VECTOR_ACCELEROMETER);
  imu::Vector<3> gravity = bno.getVector(Adafruit_BNO055::VECTOR_GRAVITY);

  //print to myFile
  myFile.print(quat.w());     myFile.print(",");
  myFile.print(quat.x());     myFile.print(",");
  myFile.print(quat.y());     myFile.print(",");
  myFile.print(quat.z());     myFile.print(",");
  myFile.print(bno.getTemp());myFile.print(",");
  myFile.print(magnet.x());   myFile.print(",");
  myFile.print(magnet.y());   myFile.print(",");
  myFile.print(magnet.z());   myFile.print(",");
  myFile.print(gyro.x());     myFile.print(",");
  myFile.print(gyro.y());     myFile.print(",");
  myFile.print(gyro.z());     myFile.print(",");
  myFile.print(euler.x());    myFile.print(",");
  myFile.print(euler.y());    myFile.print(",");
  myFile.print(euler.z());    myFile.print(",");
  myFile.print(acc.x());      myFile.print(",");
  myFile.print(acc.y());      myFile.print(",");
  myFile.print(acc.z());      myFile.print(",");
  myFile.print(linaccel.x()); myFile.print(",");
  myFile.print(linaccel.y()); myFile.print(",");
  myFile.print(linaccel.z()); myFile.print(",");
```

```
  myFile.print(gravity.x());  myFile.print(",");
  myFile.print(gravity.y());  myFile.print(",");
  myFile.print(gravity.z());  myFile.print(",");
  myFile.print(event.orientation.x, 4); myFile.print(",");
  myFile.print(event.orientation.y, 4); myFile.print(",");
  myFile.print(event.orientation.z, 4); myFile.print(",");
}
```

bnoSetup.ino

```
void bnoSetup(){
  // Initialise the sensor
  if(!bno.begin())
  {
    // There was a problem detecting the BNO055 ... check your connections
    Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
    while(1);
  }

  bno.setExtCrystalUse(true);
}
```

sdSetup.ino

```
void sdSetup(){

  // Open serial communications and wait for port to open:
  Serial.begin(115200);
  Serial.print("Initializing SD card...");
  // Retry mechanism for SD card initialization
  while (!SD.begin(config)) {
    Serial.println("initialization failed! Retrying...");
  }
  Serial.println("initialization done.");
  // make sure file opens
  myFile = SD.open("test.txt", FILE_WRITE);
  if (myFile) {
    Serial.print("File opened successfully");
    myFile.close();
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }

}
```

References

Albéri, M., Baldoncini, M., Bottardi, C., Chiarelli, E., Fiorentini, G., Raptis, K. G. C., ... & Mantovani, F. (2017). Accuracy of flight altitude measured with low-cost GNSS, radar and barometer sensors: Implications for airborne radiometric surveys. *Sensors*, *17*(8), 1889.

Altus Metrum. n.d. *TeleMega*. https://altusmetrum.org/TeleMega/.

Barrowman, J. S., & Barrowman, J. A. (1966). Theoretical Prediction of the Center of Pressure. *Apogee Components*. https://www.apogeerockets.com/downloads/PDFs/barrowman_report.pdf

Bolanakis, D. E., Kotsis, K. T., & Laopoulos, T. (2015). A prototype wireless sensor network system for a comparative evaluation of differential and absolute barometric altimetry. *IEEE Aerospace and Electronic Systems Magazine*, *30*(11), 20-28.

Bouten, W., Baaij, E. W., Shamoun-Baranes, J., & Camphuysen, K. C. (2013). A flexible GPS tracking system for studying bird behaviour at multiple scales. *Journal of Ornithology*, *154*(2), 571-580.

Brown, W., Wiesneth, M., Faust, T., Huynh, N., Montalvo, C., Lino, K., & Tindell, A. (2019). Measured and simulated analysis of a model rocket. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, *233*(4), 1397-1411.

Caballes, M. J. L., Ajuwon, M., Bunkley, J., & Chen, G. (2021, September). Rocket trajectory prediction using OpenRocket simulator and featherweight GPS tracker of a solid propellant rocket. In *10th Annual World Conference of the Society for Industrial and Systems Engineering, 2021 SISE Virtual Conference* (pp. 33-40).

Chew, I., Taylor, M., Thompson, M., Trine, K., Worosz, M., & Canino, J. (2023). Methods for modeling and Controlling the Flight Path of a High Power Rocket. In *2023 Regional Student Conferences* (p. 72413).

Federal Aviation Administration. (2025, February 20). *Procedures for Handling Airspace Matters*. https://www.faa.gov/air_traffic/publications/.

EggTimer. n.d. *Altimeters & AV Bay Stuff*. https://eggtimerrocketry.com/home/altimeters-av-bay/.

International Civil Aviation Organization. (2012, May 12). *The Eighth Meeting of the Southeast Asia and Bay of Bengal Sub-Regional ADS-B Implementation Working Group (SEA/BOB ADS-B WG/8)* (Item 5: Need for monitoring and improvement in compliance). https://www2023.icao.int/APAC/Meetings/2012_SEA_BOB_ADSB_WG8/WP06_HKG%20AI.%205%20-%20Use%20of%20Barometric%20Altitude.pdf.

JollyLogic. n.d. *AltimeterTwo*. https://jollylogic.com/products/altimetertwo/.

Kidwell, C., & Ash-Poole, J. (2004). *A comparison of altimeters and optical tracking*. https://narhams.org/library/rnd/Altimeters.pdf.

LaBudde, E. V. (1999). Extending the Barrowman Method for Large Angles of Attack.

MissileWorks. n.d. *RRC3+ Altimeter*. https://www.missileworks.com/store#!/RRC3-Altimeter/p/661985897.

Mustata, M. S., Negrea, P., Tiganiuc, D., & Niculescu, A. (2024, October). A Low-Cost Baro-Altimeter Model with Piezoresistive Sensor and Commercial Off-the-Shelf Components. In *2024 International Conference on Applied and Theoretical Electricity (ICATE)* (pp. 1-6). IEEE.

Rhudy, M. B., Fravolini, M. L., Gu, Y., Napolitano, M. R., Gururajan, S., & Chao, H. (2015). Aircraft model-independent airspeed estimation without pitot tube measurements. *IEEE Transactions on Aerospace and Electronic Systems*, *51*(3), 1980-1995.

Rhudy, M. B., Fravolini, M. L., & Napolitano, M. R. (2024). GPS-free roll and pitch estimation through pressure altitude aided inertial navigation system for a jet aircraft. *Aerospace Science and Technology*, *146*, 108975.

Schaub, T., Millon, A., De Zutter, C., Buij, R., Chadœuf, J., Lee, S., ... & Klaassen, R. H. G. (2023). How to improve the accuracy of height data from bird tracking devices? An assessment of high-frequency GPS tracking and barometric altimetry in field conditions. *Animal Biotelemetry*, *11*(1), 31.

Structure&Civil. (2017, March 24). *Make your own Altitude Measuring Device (Altimeter)* [Video]. YouTube. https://www.youtube.com/watch?v=1i6nB6yJkZI

Tripoli Rocketry Association. (2025, March). *Tripoli Rocketry Association Safety Code*. https://www.tripoli.org/safetycode.

Van Sickle, Jan. (n.d.) *Doppler Shift*. The John A. Dutton Institute for Teaching and Learning Excellence. https://www.e-education.psu.edu/geog862/syllabus.