MAD 1901

Design and Integration of a High-Powered Model Rocket-II

A Major Qualifying Project Report
Submitted to the Faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in Partial Fulfillment of the Requirements for the
Degree of Bachelor of Science
in Aerospace Engineering

____

Alexander Alvarez

____

Grace Gerhardt

____

Evan Kelly

____

John O’Neill

____

Jackson Whitehouse

March 2019

Approved By: ______________________________

Michael A. Demetriou, Advisor

Professor, Aerospace Engineering Program,

WPI

Abstract

Instability caused by factors such as wind or technical failures is a major concern within the field of model rockets and the aerospace industry as a whole. The purpose of this project was to model and enhance the stability of flight for a high-powered model rocket through the use of active control systems. This was accomplished by first creating a simulation of the rocket's flight path using a six degree-of-freedom mathematical model and subsequently by designing a feedback control system to stabilize the rocket's flight using an actuated fin system. Through the use of control systems and aerodynamics, a theoretical control law was proposed. Subsequently, an actuated fin system capable of controlling the flight path of the rocket was designed and built.

Table of Contents

List of Figures

List of Tables

Table of Authorship

Section	Author
Introduction	Grace Gerhardt and Jackson Whitehouse
Review	
Rocket Stability	Grace Gerhardt
Fin Shape	Alexander Alvarez
Active Control in Rockets	Grace Gerhardt
Euler Angles and Quaternions	Evan Kelly
Computational Fluid Dynamics	Alexander Alvarez
Simulation of Rocket flight	Evan Kelly
Avionics and Electronic Systems	Jack O'Neill
HPMR Program Goals	Common Section
HPMR Program Design Requirements	Common Section
HPMR Program Management and Budget	Common Section
MQP Objectives Methods and Standards	Common Section
MQP Tasks and Timetable	Grace Gerhardt and Jackson Whitehouse
Methodology	
Fin Design	Alexander Alvarez
Solving the Equations of Motion	Evan Kelly
Active Control Design	Evan Kelly
Creating the Simulation	Jack O'Neill
Electronics and E-Bay Fabrication	Jack O'Neill
Arduino Code for Flight Controller	Jack O'Neill
Launch Success Criteria	Grace Gerhardt
Results	
January Test launch	Grace Gerhardt
Final Product for the Fins	Jack O'Neill
E-Bay configuration and Capability	Jack O'Neill
Final Simulator	Evan Kelly
Conclusions	

Summary	Jackson Whitehouse
Summary of Fins	Jackson Whitehouse
Summary of E-Bay	Jackson Whitehouse
Summary of Flight Controller	Jack O’Neill
Summary of MATLAB Simulation and Control Law	Jack O’Neill
Broader Impacts	Jackson Whitehouse

1. Introduction

Model rocketry has been around since the late 1950's, starting in the United States. Model rockets can range from simple to extremely complex, since they can be bought as a small kit from a hobby store or thoroughly researched and built for competitions. The larger the rocket, the more complicated it becomes—while smaller hobby rockets might not involve electronic or recovery systems, larger rockets usually do. Larger rockets need a flight computer that is programmed to execute stage separation, which allows for a recovery system (typically parachutes) to eject from the rocket. Rockets are divided by the National Association of Rocketry and three basics classes. Class I allows for the use of H and I impulse class rocket motors. Class II may use J, K, and L class motors, and finally, Class III rockets may use M, N, an O class motors. Our rocket was a level one high powered rocked [28].

The baseline primary goal of the overall project was to design a Class I model rocket to fly to an altitude of 1500 feet and return safely to the ground. Through the collaboration of the entire MQP team, we outlined secondary goals to further challenge ourselves for this project. The secondary goals included the use of CO2 stage separation, electromagnetic booster separation, an autorotation recovery system and actuated fins.

To complete these tasks, the MQP team was broken in to three sub-teams: The Mechanical, Structural Analysis and Thermal (MSAT) team, the Propulsion, Staging, and Recovery (PSR) team, and our team, the Flight Dynamics and Stability (Controls) team. The MSAT team was in charge of designing the rocket as well as providing any structural and thermal analysis of the rocket body. The PSR team lead the design and development of the CO2 separation system, the electromagnetic booster separation, and the autorotation recovery system. The Controls team was in charge of the design and development of actuated fins as well as developing a simulation of the rocket's flight and electrically integrating avionics and electronic systems onboard the rocket.

The Controls team had many tasks at hand, the most challenging of which being the design and development of actuated fins. This involved optimizing fin design and developing a control law, so the fins reacted properly to disturbances in flight. Other tasks included using

MATLAB to build a flight simulation and motor failure plots and electronically integrating avionics and electronic instruments to the rocket to ensure proper timing of stage and booster separation.

This Major Qualifying Project (MQP) team was part of the High-Powered Model Rocket (HPMR) Program consisting of two additional MQP teams (*JB3-1901, MAD-1901)* The goal of the Program is to design, integrate, and test fly a high-powered model rocket capable of reaching an altitude of 457.2 m (1500 ft). The rocket design and built for the HPMR is a Class-2, based on mass, with design options that included two Level-1 motor configurations.

The objectives of this MQP team are: *perform mechanical design; perform structural, aerodynamic and thermal analysis; to acquire and fabricate components; integrate all components* of the HPMR shown in its final configuration in Figure 1.

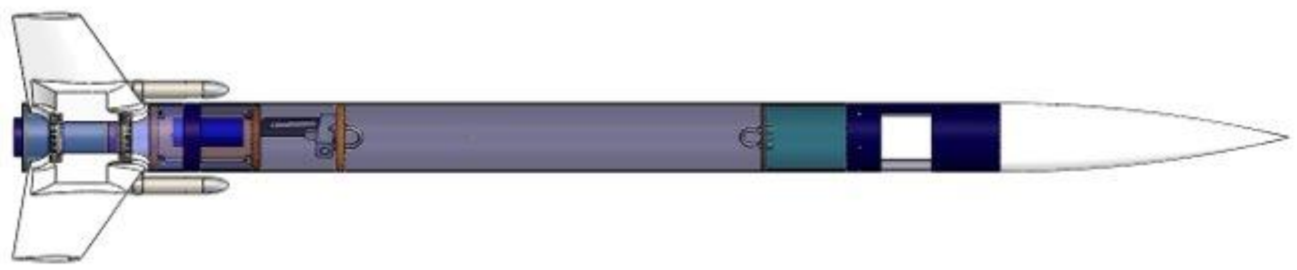


Figure 1 Rocket CAD Model.

1.1 Review

The science behind rocketry is very complex. The Flight Dynamics and Stability team focused on the control systems and stability of the rocket by utilizing MATLAB®, researching the equations of motion, and using computational fluid dynamics to analyze the actuated fins.

1.1.1 An Introduction to Control Systems

Control systems are used everywhere: they are used in a thermostat in a home, in cars on the road, and in spacecraft outside earth's atmosphere. A control system is "an arrangement of physical components connected or related in such a manner as to command, direct, or regulate itself or another system" [1]. Control systems can be broken down into two main categories: active and passive control.

Passive controls are control systems that do not have sensors or use power, thus their control actions do not depend on the output. An example of this would be a passively controlled launch vehicle such as a model rocket that relies on the aerodynamic torques acting on the body to orient itself during flight. Torque, as seen below in Equation (1) is an applied force (F) at a distance (r) and at an angle (θ) from the axis of rotation [2].

$$T = F \bullet r \bullet \sin(\theta) \tag{1}$$

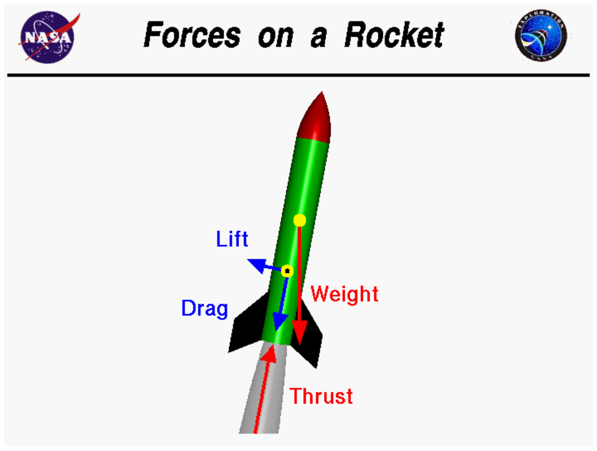
The passive control system on the rocket might include stationary fins or a boat tail. The fins use the aerodynamic torques on the body to assist in the orientation of the rocket, while the boat tail on the rocket helps with the stability of the rocket by reducing drag. Both the stationary fins and the boat tail do not use sensors or consume power and as such are considered passive control. In contrast, active control uses mechanical or electromechanical means to change the rocket's orientation based on input from sensors. An example of active control is the actuating fins of the rocket. While the rocket is flying, the sensors in the electronics bay (gyroscope, altimeter, radio frequency locator and inertial measurement unit) measure the rocket's position, angle, and acceleration. Since the goal of this project is to keep the rocket completely vertical during its flight, the sensors are used to measure if the rocket becomes off-track of its vertical path. Once the rocket deflects from the original vertical path, the actuated fins will rotate slightly, ultimately correcting for the rocket's deflection and putting the rocket back on track. The rotation angle of the fins is based on the feedback measurements of the sensors, making them an active control system. Active control systems can be broken down further into "closed-loop" or "open-loop" systems. An open-loop system is an active control system whose action is not dependent on the output. In contrast, a closed-loop system is an active control system in which the action is in some way dependent on the output of the sensors.

The main project proposed by the Flight Dynamics and Stability team is the construction and testing of actuated fins. Under the actuated fins are several responsibilities including designing optimal fin shape, designing the control theory and building the fins. Other responsibilities for the Flight Dynamics and Stability team include the electrical integration of the various other components from the other two teams such as the side boosters and auto rotation system into the electronics bay.

1.1.2 Rocket Stability

Basic rocket stability is a crucial part to any successful flight and focuses on the forces acting on the rocket during flight and how the rocket reacts to them. There are four general forces which act on a rocket during flight: thrust, weight, drag, and lift.

Thrust and weight act through the rocket's center of gravity and generally along its longitudinal axis of symmetry. Drag and lift act through the rocket's center of pressure perpendicular to each other and produce torques which causes rockets to rotate about their center of gravity. Figure 2 below shows how thrust, weight, drag and lift act on a rocket in flight [3].



[2] Figure 2: An illustration of forces on a rocket body., Copyright 2018, NASA.

Thrust is the total force produced by the rocket motor. It opposes weight during flight and as a result causes the rocket to accelerate. Thrust depends on the mass flow exiting the nozzle ($\dot{m}$), the velocity of the gas exiting the nozzle (V_e), the exit pressure of the hot gasses leaving the nozzle (p_e), the atmospheric pressure outside the nozzle (p_0) and the ratio of the areas between the throat of the nozzle and the exit (A_e). The rocket's weight is described simply by the rocket's mass multiplied by the acceleration due to gravity. The equations for thrust and weight can be found below in Equations (2) and (3):

$$F = \dot{\mathrm{m}} \bullet V_e + (p_e - p_0) \bullet A_e \tag{2}$$

$$W = m \bullet g \tag{3}$$

Lift and drag are mechanical forces created when there is relative motion between a solid body and a fluid. Lift is the force that makes objects fly and acts perpendicular to the object's motion. Drag acts opposite to the object's motion and is broken into three different kinds: skin friction, form drag, and induced drag [4].

Skin friction is the friction between the molecules of the rocket's surface and the molecules of the air it is travelling through. The magnitude of the drag caused by skin friction is dependent on the viscosity of the fluid that the object is travelling through as well as the roughness of the object's surface. Form drag is the aerodynamic resistance between an object and the fluid it moves through because of the object's shape. Changes in shape of an object cause pressure differences along its surface, ultimately causing a force to be applied across the surface of the object. The magnitude of this force can be found by integrating the pressure at points along the object throughout the entire surface area of the object. Induced drag is produced when an object produces lift in a non-uniform manner. In the case of a rocket, the fins produce more lift at the tip of the fin than the base, causing induced drag at the fin tips. Vortices are created at the tips of the fins during flight, which cause the local angle of attack to increase due to the induced drag. A long fin with a small relative chord length has a low induced drag, and a shorter stubbier fin will have a higher induced drag unless it is too short to be in the free stream flow an affected by the rocket body. A drag coefficient is used to describe the drag created by specific cross sections [4].

Coefficients of lift and drag are ways to model all the dependencies of lift and drag of an object, such as the object's shape, angle of attack, and flow conditions. Equations for lift and drag can be found in Equations (4) and (6), respectively. The lift and drag coefficients are a good way to determine the aerodynamic efficiency of an object. For example, the higher the coefficient of lift and the lower the coefficient of drag, the more aerodynamically efficient the object is. The coefficient of lift is dependent upon the lift force acting on the object (L), the density of the fluid (ρ), the velocity of the object (V), and the surface area of the object (A), as seen in Equation (5). The coefficient of drag (Cd_0) at zero lift is dependent upon the drag force (D), the object's surface area, the density of the fluid, and the velocity of the object, shown in Equation (7). In addition to the coefficient of drag, there is a coefficient of induced drag (Cd_i). As seen in Equation (8), the coefficient of induced drag depends on the coefficient of lift (Cl), the aspect ratio (AR) of the object which is the wingspan squared divided by the wing area, and the efficiency factor (e). The total drag coefficient of an object is shown in Equation (9) as the sum of the drag coefficient at zero lift, skin friction and the induced drag coefficient. The equations for coefficient of lift, drag and induced drag can be found below [5].

$$L = Cl \bullet \left(\frac{\rho \bullet V^2}{2}\right) \bullet A \tag{4}$$

$$Cl = \frac{2L}{(\rho \bullet V^2 \bullet A)} \tag{5}$$

$$D = Cd \bullet \left(\frac{\rho \bullet V^2}{2}\right) \bullet A \tag{6}$$

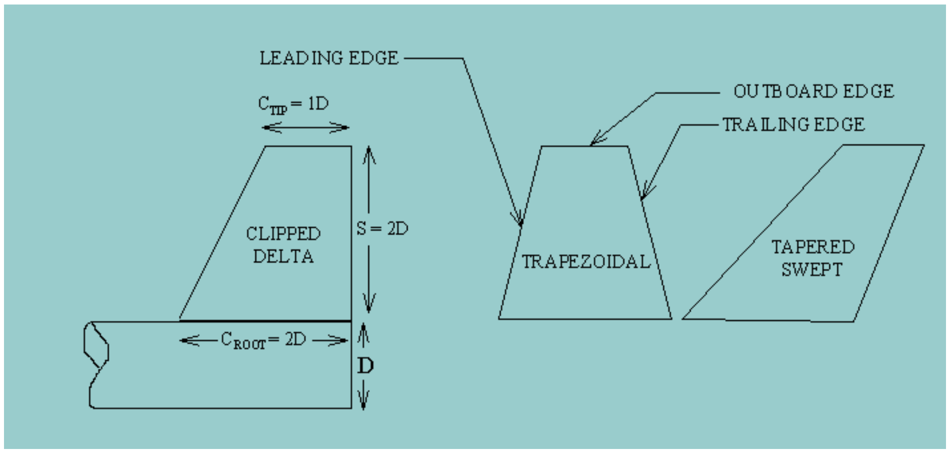
$$Cd_0 = \frac{D}{\left(A \bullet \left(\frac{1}{2}\right) \bullet \rho \bullet V^2\right)} \tag{7}$$

$$Cd_i = \frac{(Cl^2)}{(\pi \bullet AR \bullet e)} \tag{8}$$

$$Cd = Cd_0 + Cd_i \tag{9}$$

1.1.3 Fin Shape

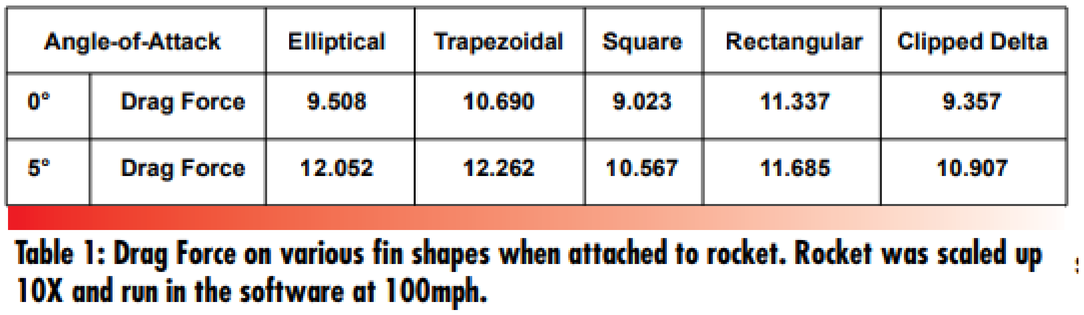
Rocket fins are a form of passive control on a rocket which produce lift and drag forces to stabilize the rocket. This force is called a restoring force and occurs so long as the rocket's center of pressure is lower than its center of gravity [3]. As the angle of attack of the rocket changes from zero, the fins begin to produce a lift force through the rockets center of pressure. This lift force exerts torque on the rocket's center of gravity causing the rocket to rotate back to an angle of attack of zero. Ideally, the fins will produce the necessary lift force required to restore the rocket back to an angle of attack of zero while being the most aerodynamically efficient. To optimize a fin, the size, shape, and profile must be considered [6].



[7] Figure 3: Fin sizing illustration. Copyright 2018, Richard Nakka.

For any fin shape and profile there is a general rule to be followed for size: the span and root chord length of the fin should be roughly twice the diameter of the body tube of the rocket. The chord length of the tip of the fin will vary depending on the fin shape. Figure 3 above shows the proper ratios for a clipped delta fin on a rocket of specific diameter, where the clipped delta fin shape is the chosen shape for the actuated fins [7].

Another important parameter to keep in mind is the overall shape of the fin. With each shape comes a specific aerodynamic efficiency. In theory, the most efficient fin shape is an elliptical fin, however experimental results from Apogee Rockets have proven otherwise, as shown below in Figure 4. It shows different fin shapes and the drag force produced by them at two different angles of attack. The fins tested had a rectangular cross section. It is important to note that the square fin shape did not protrude from the rocket as far as the other fins to the point of affecting the drag calculation [6].



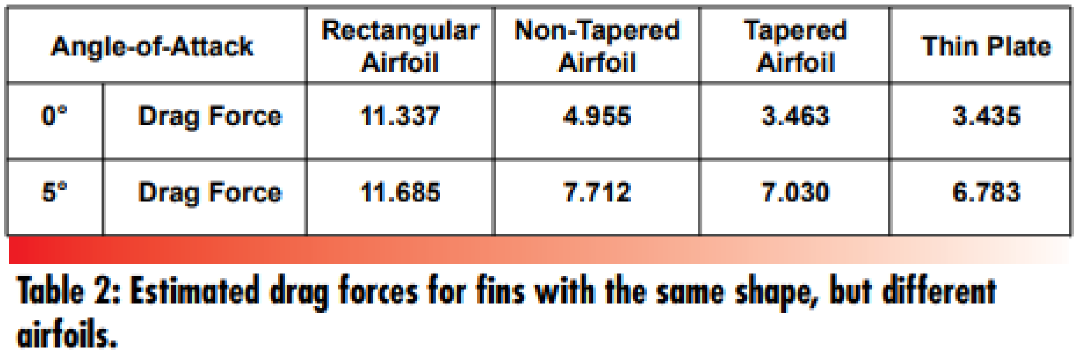
Angle-of-Attack		Elliptical	Trapezoidal	Square	Rectangular	Clipped Delta
0°	Drag Force	9.508	10.690	9.023	11.337	9.357
5°	Drag Force	12.052	12.262	10.567	11.685	10.907

Table 1: Drag Force on various fin shapes when attached to rocket. Rocket was scaled up 10X and run in the software at 100mph.

[6] Figure 4: Drag force by airfoil shape. Copyright 2018, Apogee Rockets.

From this data, the clipped delta shape proves to be the most efficient at low speeds. In addition, if the center of pressure needs to be moved further from the rocket's center of gravity to keep the rocket stable, the fins can be swept towards aft as in Figure 3 above labeled tapered swept [6].

Finally, the profile of a fin can also help to reduce the drag produced by the fin while maintaining effective lift forces. Rectangular airfoils have the highest drag at low speeds, while thin plates have the lowest as shown in Figure 5 below. However, thin plates are not effective for most rockets as they do not produce the lift necessary to restore the rocket back to its original orientation. Also, the thin plates' thickness can cause fin flutter during flight, significantly increasing drag force. The best fin profile is a tapered airfoil that is symmetric about the center of its chord [6]. It is important to note as well that for subsonic speeds the leading edge of the fin should be rounded while the trailing edge should come to a sharp, symmetric edge [7].



Angle-of-Attack		Rectangular Airfoil	Non-Tapered Airfoil	Tapered Airfoil	Thin Plate
0°	Drag Force	11.337	4.955	3.463	3.435
5°	Drag Force	11.685	7.712	7.030	6.783

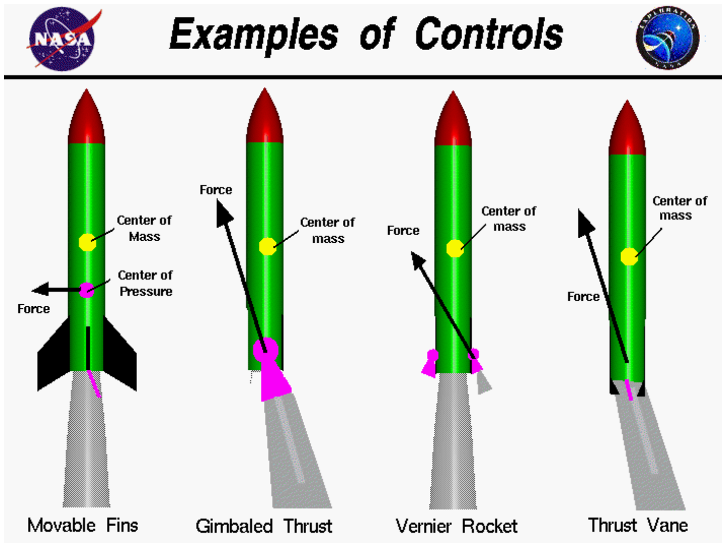
Table 2: Estimated drag forces for fins with the same shape, but different airfoils.

[6] Figure 5: Fin shape and corresponding drag force values. Copyright 2018, Apogee Rockets

In conclusion, a well optimized fin will have a span and root chord length of twice the diameter of the rocket's body tube. The fin should be a clipped delta shape with a symmetrical teardrop shaped tapered airfoil, and it should have a rounded leading edge and a tapered sharp trailing edge [6][7].

1.1.4 Active Control in Rockets

In addition to passive control with fins, rockets can be further controlled in flight using active control. Active control, or guidance, is one of the four main components in large rockets and is used to keep the rocket stable during flight and on its flight path. Active controls rely on rotating rockets about their center of gravity by changing the direction in which forces such as thrust and lift act on the rocket. This change in force produces a torque which acts on the rocket's center of gravity rotating the rocket and translating the center of gravity. There are four main types of active control and many variations of these four general control systems: actuated fins, gimballed nozzles, rocket vanes, and Vernier rockets, all shown in Figure 5 below [8].



[8] Figure 6: Active control for rockets. Copyright 2018, NASA.

Actuated fins are an older method of controlling rockets and is still used today for guided missiles. Actuated fins move to change the aerodynamic forces being placed on them producing a torque which rotates the rocket. Gimballed nozzles are used on most modern rockets and use the thrust to steer the rocket in its desired direction. To keep the rocket stable, gimballed exhaust nozzles rotate causing the thrust to change direction and moving the thrust vector out of line of the rocket's center of gravity thus rotating the rocket. Another form of redirecting thrust as used by the German V2 missile are rocket vanes. Rocket vanes are placed inside the exhaust nozzle of a rocket and rotate to deflect the thrust in different directions. Rocket vanes have a similar effect on the rocket's thrust vector as the gimballed nozzle. Finally, Vernier rockets can be used to guide rockets in flight by adding thrust to specific sides of the rocket. Instead of simply redirecting thrust to move the thrust vector out of line with the rocket's center of gravity, they add thrust. Vernier rockets are an older form of active control used in missiles such as the Atlas Missile [8]. In the case of model rockets actuated fins would be the most realistic option given the limited burn time of model rocket engines. With such short burn time the active controls using thrust would only be useful for a small portion of the flight.

1.1.5 Euler Angles and Quaternions

There are two main mathematical systems that fully describe the attitude of a three-dimensional rigid body in space. The first of these is the use of Euler angles, which describe the rotation of the object's body fixed x-y-z axis relative to a non-rotating ground fixed three-dimensional coordinate system. There is then one angle associated with each one of these rotations, commonly referred to as roll, pitch, and yaw respectively. The advantages to this method are that the system is both simple and intuitive to visualize and make sense of and the mathematics to rotate vectors from one frame to another can easily be accomplished by using the product of three 3-by-3 rotation matrices. However, Euler angles are often inadequate when working with multiple coordinate systems because there is a possibility of "gimbal lock", which mathematically represents itself with a division by zero when rotating a vector. To rectify this issue, another system of measuring angular position could be used in its place.

This system is known as quaternions and consists of one real component and three imaginary components and reflects the rotation of the body about a four-dimensional real and imaginary coordinate system. This system can be used similarly to Euler angles and removes the possibility of gimbal lock. The major downside to the use of this system is the added complexity that takes the form of one additional state and an additional equation of motion necessarily to model the motion of the rigid body in space. However, once implemented correctly, they are the best way to represent angular positions, and we have chosen to work with them throughout our project.

1.1.6 Computational Fluid Dynamics

To properly analyze the actuated fins, Computational Fluid Dynamics was used for a preliminary design of the fins. Computational Fluid Dynamics, or CFD, is a system of using numerical analysis and data structure to solve and analyze difficult fluid dynamics problems. It performs the calculations necessary to simulate the interaction between fluids and solids as defined by boundary conditions. CFD software, such as ANSYS® Fluent®, was used to analyze airflow over the rocket fins as the rocket moves and as the rudders actuate. A preliminary test of the fin shape in ANSYS® Fluent® can be seen in Figure 7 below:

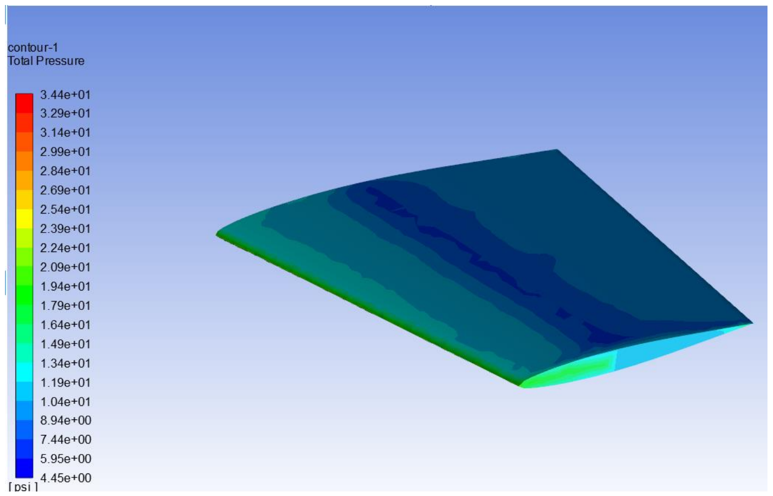


Figure 7: Preliminary CFD simulations for a fin at Mach 1. Copyright 2018, WPI.

For this purpose, the Finite Element and Boundary Element Methods were used to create a preliminary surface mesh of the fin and rudder and analyze the fluid elements as they interact with the solid fin shape. In doing this, the CFD software computed lift and drag coefficients as functions of flow speed and rudder angle. Using the coefficients, an estimate for maximum torque on the rudder was calculated and used to size the servos that are necessary to turn the fin rudder.

1.1.7 Simulation of Rocket Flight

To simulate the motion of the rocket within MATLAB®, we developed a system of equations of motion that govern the motion of the rocket. These equations allowed for the position and attitude of the rocket to be solved for numerically over the duration of its flight. We determined that the rocket could be modeled similarly to an aircraft using a 6 Degree of Freedom (DOF) model of rigid body dynamics [16]. We used two different right-handed Cartesian coordinate systems to model this flight, which consisted of a non-inertial body-fixed system that was fixed to the rocket with its origin at the rocket's center of mass. The sensor readings would be inputted as an inertial ground fixed system with its origin at the launch site of the rocket. Though Earth is not an inertial frame, we approximated it as one because over the time and height of the rocket's flight, the rotation and curvature of the Earth can be considered negligible

[16]. This model involves the use of 13 states of the rocket that describe its current position and orientation at a given time which are the three translational position coordinates, the three translational velocities, four quaternions, and three angular velocities all of which are taken from the Earth fixed system [17]. These states are then inputted into a series of 13 coupled non-linear differential equations, with the variables defined in Table 1, as shown below.

$$x = \begin{bmatrix} x \\ y \\ z \\ u \\ v \\ w \\ q_1 \\ q_2 \\ q_3 \\ q_4 \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{10}$$

$$\dot{x} = \begin{bmatrix} u \\ v \\ w \\ (Fx + \omega_y \bullet w - \omega_z \bullet v)/m \\ (Fy - \omega_x \bullet w + \omega_z \bullet u)/m \\ (Fz + \omega_x \bullet v - \omega_y \bullet u)/m \\ \frac{1}{2}(q_2 \bullet \omega_z - q_3 * \omega_y + q_4 \bullet \omega_x) \\ \frac{1}{2}(-q_1 \bullet \omega_z + q_3 \bullet \omega_x + q_4 \bullet \omega_y) \\ \frac{1}{2}(q_1 \bullet \omega_y - q_2 \bullet \omega_x + q_4 \bullet \omega_z) \\ \frac{1}{2}(-q_1 \bullet \omega_x - q_2 \bullet \omega_y - q_3 \bullet \omega_z) \\ (Mx - \omega_y \bullet \omega_z(I_{yy} - I_{zz}))/I_{xx} \\ (My - \omega_x \bullet \omega_z(I_{xx} - I_{zz}))/I_{yy} \\ (Mz - \omega_x \bullet \omega_y(I_{xx} - I_{yy}))/I_{zz} \end{bmatrix} \tag{11}$$

$$F = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} \bullet R_i^b + \begin{bmatrix} \frac{1}{2}\rho \bullet (u^2 + v^2 + w^2) \bullet A \bullet \frac{\frac{v}{w}}{\sqrt{\frac{v}{w}^2 + 1}} \bullet 2\pi \bullet \arctan\left(\frac{\mathrm{v}}{\mathrm{w}}\right) \\ \frac{1}{2}\rho \bullet (u^2 + v^2 + w^2) \bullet A \bullet \frac{v}{(u^2 + v^2 + w^2)^{\frac{1}{2}}} \bullet 2\pi \bullet \arctan\left(\frac{\mathrm{v}}{\mathrm{w}}\right) \\ \frac{1}{2}\rho \bullet (u^2 + v^2 + w^2) \bullet Cd \bullet Af \end{bmatrix} \bullet R_v^b + \begin{bmatrix} 0 \\ 0 \\ T(t) \end{bmatrix} \tag{12}$$

$$M = \begin{bmatrix} -\frac{1}{2}\rho \bullet (u^2 + v^2 + w^2) \bullet A \bullet C_{lu} \bullet u_1 \bullet L + \frac{1}{2}\rho \bullet (u^2 + v^2 + w^2) \bullet A \bullet C_{lu} \bullet u_3 \bullet L \\ -\frac{1}{2}\rho \bullet (u^2 + v^2 + w^2) \bullet A \bullet C_{lu} \bullet u_2 \bullet L + \frac{1}{2}\rho \bullet (u^2 + v^2 + w^2) \bullet A \bullet C_{lu} \bullet u_4 \bullet L \\ \frac{1}{2}\rho \bullet (u^2 + v^2 + w^2) \bullet A \bullet C_{lu} \bullet u_1 \bullet d \bullet (u_1 + u_2 + u_3 + u_4) \end{bmatrix} \tag{13}$$

Table 1: Equations of Motions Constants and Variables

Constants and Variables	Definition
x	x position coordinate
y	y position coordinate
z	z position coordinate
m	Mass of the rocket
g	Acceleration due to gravity
I_x, I_y, I_z	Moment of inertial tensor
F_x	X force balance
F_y	Y force balance
F_z	Z force balance
M_x	Rolling moment
M_y	Pitching moment
M_z	Yawing moment
ω_x	Roll rate
ω_y	Pitch rate
ω_z	Yaw rate
q_1, q_2, q_3, q_4	Unit quaternions
u	Airspeed relative to the atmosphere
v	Wind velocity relative to the atmosphere
w	Sideslip velocity relative to the atmosphere

R_i^b	Inertial to body rotation matrix,
R_v^b	Wind to body rotation matrix
u_1, u_2, u_3, u_4	Fin deflection angles

The first six of the above equations are the rotational equations that convert the velocities and angular velocities from body fixed to ground fixed coordinates using directional cosine matrices and Euler angles. The next six equations are Newton's laws in a non-inertial frame and allow for the calculation of the rocket's position and attitude which include the Euler equations in the final set of three equations.

The force balances in the above equations are equal to the sum of the aerodynamic forces and gravity along with the moments they apply to the rocket when taken in the body fixed coordinate system. Also included in these force balances is a composite thrust curve that was created from data published for the motors that are to be used in the rocket. The thrust curve was then generated by using a spline interpolation of this data and was then imputed as force terms in the X force balance relative to the rocket body.

These equations are then inputted into a MATLAB® script as a system of 13 coupled nonlinear ordinary differential equations which are then assigned initial conditions of 0 for all 13 states. This system is then solved using the function ode45 which then calculates the 12 states of the rocket over the flight time of the rocket. These states are then plotted as a function of time and the position and attitude states are then used to create a three-dimensional animation of the rocket flight by translating a cylinder and cone shape.

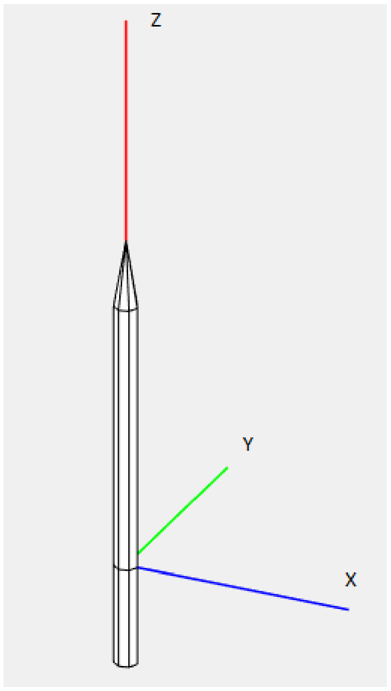


Figure 8: Body-fixed axes system of the rocket.

Shown above is a diagram of the body fixed axes system used for the simulation of the rocket's motion. The coordinate system was situated so that the z-axis extends out of the nosecone, and the x-axis extends to the right. In figure 8 above the blue bar is positive x, green is positive y, and red is positive z.

1.1.8 Avionics and Electronic Systems

There are multiple avionics and electronic systems used on the rocket: an inertial measurement unit (IMU), an altimeter, a radio frequency (RF) transmitter, an Arduino®, and four servo motors. The IMU and altimeter both communicate with the Arduino® using the popular Inter-integrated circuit (I^2 C) communication protocol. This is a communication style used between microcontrollers and integrated electronics which utilizes digital signals to pass communications.

Most of the avionics of the rocket reside in the rocket's electronics bay (e-bay). This includes the altimeter, a RF transmitter, an SD card writer, and an Arduino®. The altimeter determines the altitude of the rocket by measuring the atmospheric pressure and comparing it to the predetermined pressure value. In this project the altimeter used was the Adafruit MPL3115A2-I2C, which can measure barometric pressure within 1.5 Pascals and temperature within roughly 1 degree Celsius. This accuracy allows the altimeter to sense altitude within 0.3 meters [9].

The radio frequency transmitter transmits a specified signal from the Arduino® at 433MHz. A directional receiver is used in conjunction with the transmitter to pinpoint the rocket from its location at takeoff [10]. The transmitter is activated once the rocket detects apogee.

To store all the flight data collected a simple Micro SD breakout board was used. It connected with the Arduino® to seamlessly write all altimeter and IMU readings as well as IMU calibration states and Boolean statements describing whether launch, apogee, and parachute deployment were detected.

At the core of the e-bay lies the Arduino®. We chose the Mega 2560-CORE with an onboard ATMega2560 processor. We chose this Arduino® format since its physical size is small and the processor it holds is powerful. By choosing this Arduino® we could be completely sure that all the necessary electronics would fit in the e-bay and that the Arduino® would be powerful enough to interact with all the sensors and motors. The Arduino® was programmed with the Arduino® Software Integrated Development Environment (IDE) [11]. Powering the Arduino® and all integrated electronics was a three-cell 11.1-volt, 1000 mAh Lithium Polymer (LiPo) battery. This battery was chosen because of its small size and enough capacity to power the Arduino®, sensors, RF transmitter, ejection systems, and servo motors.

The Adafruit BNO055 9 degree of freedom (9DOF) absolute orientation sensor was used to accurately record the rocket's body-fixed acceleration and angular velocity during flight. It uses accelerometers in all three principle directions (x, y, and z axes) to measure the acceleration of the rocket during flight. The next 3 degrees of freedom are the angular rates that the rocket experienced during flight measured by gyroscopes. Finally, a magnetometer measured the magnetic field strength along the three axes. We chose this IMU because it includes an onboard microcontroller which processed all the raw data and outputted understandable readings in various formats through I2C. It can output absolute orientation data in Quaternions or Euler vectors, as well as angular velocity, acceleration, linear acceleration, heading (using the magnetometer), gravitational acceleration, and temperature [12].

In conclusion, all sensors worked directly with the Arduino ® with constant communication between the two. Each piece of the flight controller served a unique purpose and was vital to the mission. In some cases, sensors are simply for calculating data, while others will trigger flight events. All equipment was chosen for very specific reasons with flight requirements in mind.

1.2 HPMR Program Goals

The goals of the HPRM Program were shared among the three MQP teams involved (NAG-1901 , JDB-1901, MAD-1901). They are:

- Design, integrate, and fly a reusable, Class-2 high-powered model rocket capable of reaching an altitude of 457.2 m (1500 ft) using Level -1 motors.
- Provide the 21 members of the three MQP teams with a major design experience of a moderately complex aerospace system.

1.3 HPMR Program Design Requirements, Constrains, Standards and Other Considerations

The *design requirements* for the HPMR Program were shared among the three MQP teams involved (NAG-1901, JDB-1901, MAD-1901) and consisted of the following:

- Use on-board cameras to record video during flight.

- Use an autorotation recovery system to slow the descent and prevent damage upon impact.

- Use a CO_2 stage-separation system to eject the nose cone and deploy the recovery system.

- Use an electromagnetic stage separation system to separate boosters from the main rocket body.

- Use actively-controlled, actuated fins to control the trajectory of the rocket to insure vertical flight.

- Use single or clustered, Level-1 main motors, and boosters if necessary, to provide the necessary thrust-to-weight for a safe launch, while remaining below the total impulse limit.

The *design constraints* for the HPMR Program were shared among the three MQP teams and consisted of the following:

- The overall weight of the rocket must be minimized to ensure a high enough thrust-to-weight ratio to launch safely and meet project height requirements.
- The rocket must leave the launch rail at a high enough speed to ensure there is no chance of injury to those present at the launch site.
- Each motor must be able to individually provide a 5:1 thrust to weight ratio off the launch rail to provide an adequate safety factor.
- The dimensions and location of all internal subsystems must be compatible with constraints imposed by the height and width of the rocket body.

The *design standards* imposed by the National Association of Rocketry (NAR) [28] for high-powered model rockets applied to the three MQP teams and included the following:

- The rocket is built with lightweight materials (paper, wood, rubber, plastic, fiberglass, or when necessary ductile metal).

- Only certified, commercially made rocket motors are used to launch the rocket.

- Motors and rocket body materials used were purchased from reputable hobbyist sources.

- For flight tests, the motors are ignited electronically with commercial ignitors, purchased from reputable hobbyist sources.

- The rocket is launched with an electrical launch system, and with electrical motor igniters that are installed in the motor only after the rocket is at the launch pad or in a designated prepping area. The launch system includes a safety interlock that is in series with the launch switch that is not installed until the rocket is ready for launch and will use a launch switch that returns to the "off" position when released. The function of onboard energetics and firing circuits will be inhibited except when the rocket is in the launching position. The switch is installed and tested before launch.

- The rocket uses a recovery system to land the rocket safely and undamaged in such a manner that it can be flown again. Any wadding used in the recovery system is flame-retardant. For the test launch, this consisted of an appropriately sized parachute. An autorotation recovery system was designed for later launches.

The following *design considerations* for the HPMR Program were shared among the three MQP teams and included the following:

- Safety: A primary consideration during construction, integration, and launch, for both the MQP teams and the public.
    - Simulation of possible landing places to insure the safety of not only the project teams, but also the launch site.

    - Thrust-to-weight ratio: Designed to be relatively high, to insure safe levels and guarantee the rocket maintained a vertical orientation after leaving launch rail.

    - Proper disposal of partially burned motors to insure safety and minimize environmental impact.

- Social impact: The broader impacts of model rocketry as a hobby was researched by the individual teams with findings described in the individual reports.

- Environmental factors: Means of limiting potential environmental impact of model rocketry (e.g. material disposal, damage during launch and flight mishaps) was researched by the individual teams with findings described in the individual reports.

- Community outreach: considered to potentially engage those wishing to learn more about STEM related topics explored with this project.

1.4 HPMR Program Management and Budget

The HPMR Program consisted of three separate MQP teams, each responsible for different aspects of the Program.

The Mechanical, Structural, Aerodynamic, and Thermal (MSAT) MQP team (NAG-1901), with 8 members, was responsible for the physical assembly and mechanical integration of all subsystems designed by the other teams. The MSAT MQP had the responsibility of ensuring all other teams were aware of the spatial limitations inside the rocket that would affect their subsystem designs. The MSAT MQP also performed structural, aerodynamic, and thermal analysis on the various subsystems inside the HPMR to make sure everything worked cohesively, and to confirm that nothing would be damaged during a launch.

The Propulsion, Staging, and Recovery (PSR) MQP team (JB3-1901), with 8 members, was responsible for the design of the propulsion and recovery subsystems of the HPMR. The PSR MQP team performed analysis on motor sizing to choose the appropriate motors for the rocket and determined a parachute size that would return the rocket to the ground at a safe velocity. An autorotation recovery subsystem was also designed, which was meant to replace the parachute. The PSR MQP team also designed the systems that would separate the nosecone section from the rocket body (black-powder and eventually CO2) and the system that attaches/separates the boosters from the main body via electromagnets.

The Flight Dynamics and Control (FDC) MQP team (MAD-1901), with 5 members, was responsible for the design of the avionics for control and dynamic stability of the HPMR. For the first launch the FDC MQP team had to ensure parachute ejection at apogee as well as dynamic stability of fin design. While communicating with MSAT they were given maximum electronics bay dimensions to ensure sufficient volume for parachute and motors.

The three MQP teams met weekly with each of the faculty advisors involved as a conglomerate organization titled the Systems Engineering Group (SEG). Each week, the MQP teams presented an update of the past week's activities, discussed open action items between the teams, and sought input from the faculty advisors.

Funding for the construction of the rocket was provided by the WPI Aerospace Engineering Department. Per school policy, each student was allotted $250 for use in the project. With 21 students, the budget for the construction of the rocket totaled $5250. The total funds were split between the three MQP teams comprising the HPMR Program. The MSAT and PSR teams each had 8 members, corresponding to a budget of $2000 each. The Controls team received the remaining funds for its 5 members, with $1250. Overall, the SEG spent $3,828.84 in development of the rocket. The full cost breakdown can be seen in Appendix A.

The Code of Ethics for Engineers (National Society of Professional Engineers) states that

"Engineers, in the fulfillment of their professional duties, shall:

1. Hold paramount the safety, health, and welfare of the public.

2. Perform services only in areas of their competence.

3. Issue public statements only in an objective and truthful manner.

4. Act for each employer or client as faithful agents or trustees.

5. Avoid deceptive acts.

6. Conduct themselves honorably, responsibly, ethically, and lawfully so as to enhance the honor, reputation, and usefulness of the profession."

The first canon is especially relEvan Kellyt to this project, since model rocketry can be a dangerous hobby if certain regulations are not strictly followed. The HPRM Program took this canon very seriously, by adhering to all FAA and NAR guidelines and regulations throughout the design process, as well as by following all guidelines set forth by the executive staff at the launch site.

The second canon was addressed partially by placing students in each MQP team that they would be most interested and qualified for, thus creating a project wherein students are performing work in their area of expertise.

The third and fourth canons are less relEvan Kellyt to the HPMR Program, since there were no public statements to be issued; nor were there separate employers to speak of.

The fifth and sixth canons are covered by WPI's Academic Honesty Policy, which all three MQP teams (and all MQPs) must follow.

1.5 MQP Objectives, Methods and Standards

Objectives:

- Use active control to keep the rocket stable in flight and above the launch pad
  - Use SolidWorks® to design both passive and active fins
  - Analyze both sets of fins using ANSYS® Fluent® to find aerodynamic characteristics and the effect of the rudder's movement on the rocket
  - Develop an Active control law using a combination of MATLAB® and Simulink®
- Recover the rocket using a Radio Frequency locator
  - Program the Arduino® in Arduino® IDE to send an RF signal once the parachute was deployed
- Create a Rocket Simulation to predict the rocket's flight given specific inputs

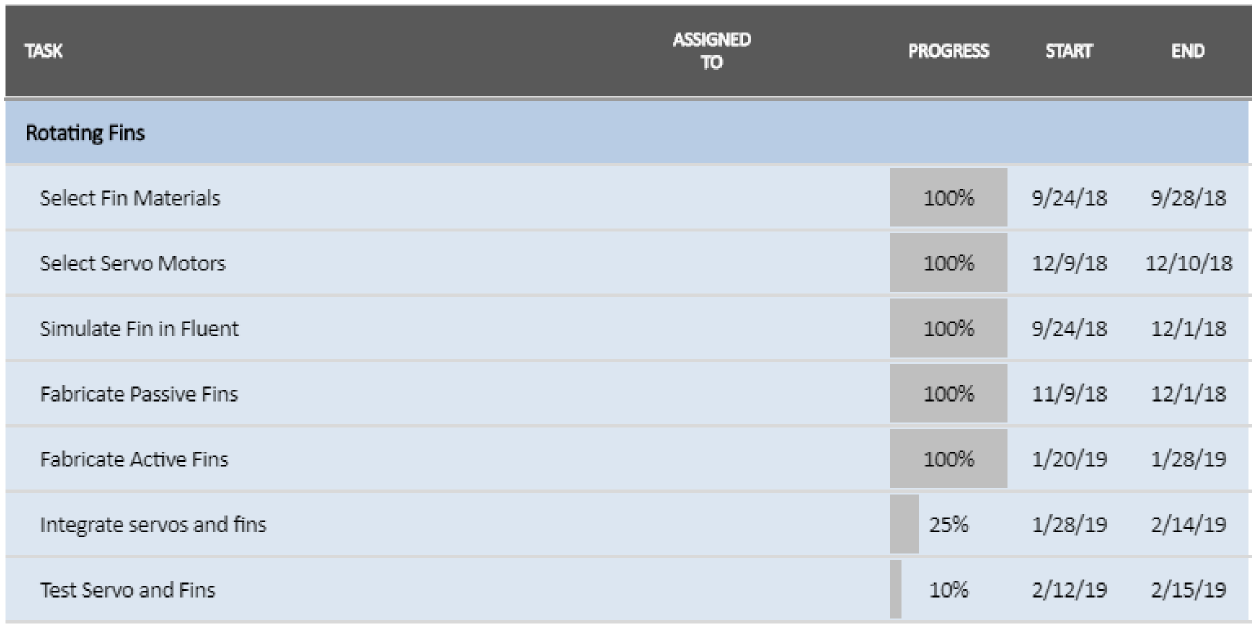
    - Use MATLAB® to design the full simulation
- Detect launch, apogee, and effectively trigger separation
    - Program the Arduino® using Arduino® IDE to detect these flight events

Common Engineering Standards Used:

- NACA 64A010 airfoil
- NACA 66-021 airfoil
- ANSI C18.3M for our lithium Ion battery
- NEMA ICS 16-2001 standard for our servomotors

1.6 MQP Tasks and Timetable

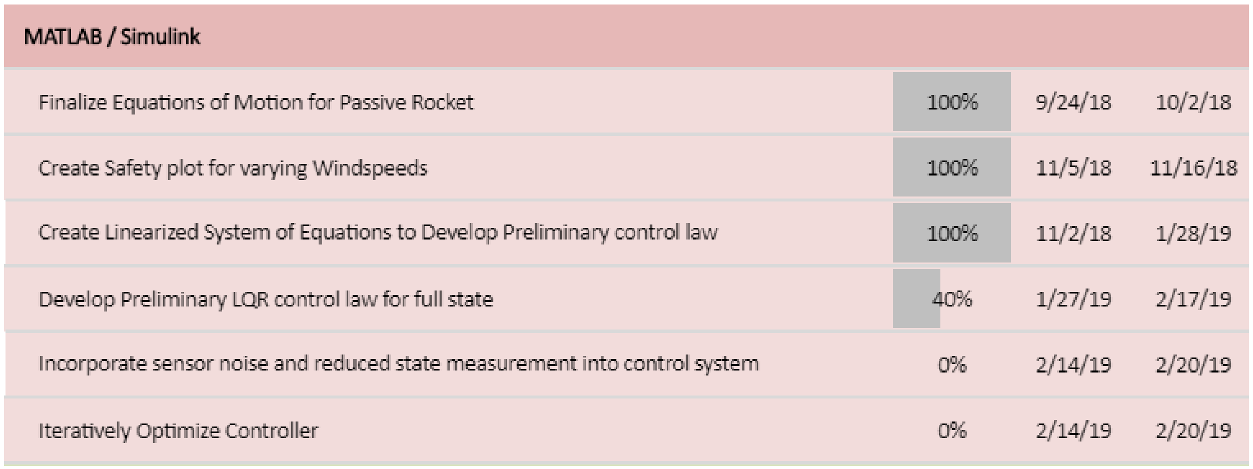
Rotating Fins



TASK	ASSIGNED TO	PROGRESS	START	END
Rotating Fins				
Select Fin Materials		100%	9/24/18	9/28/18
Select Servo Motors		100%	12/9/18	12/10/18
Simulate Fin in Fluent		100%	9/24/18	12/1/18
Fabricate Passive Fins		100%	11/9/18	12/1/18
Fabricate Active Fins		100%	1/20/19	1/28/19
Integrate servos and fins		25%	1/28/19	2/14/19
Test Servo and Fins		10%	2/12/19	2/15/19

Figure 9 Rotating Fins Gantt Chart.

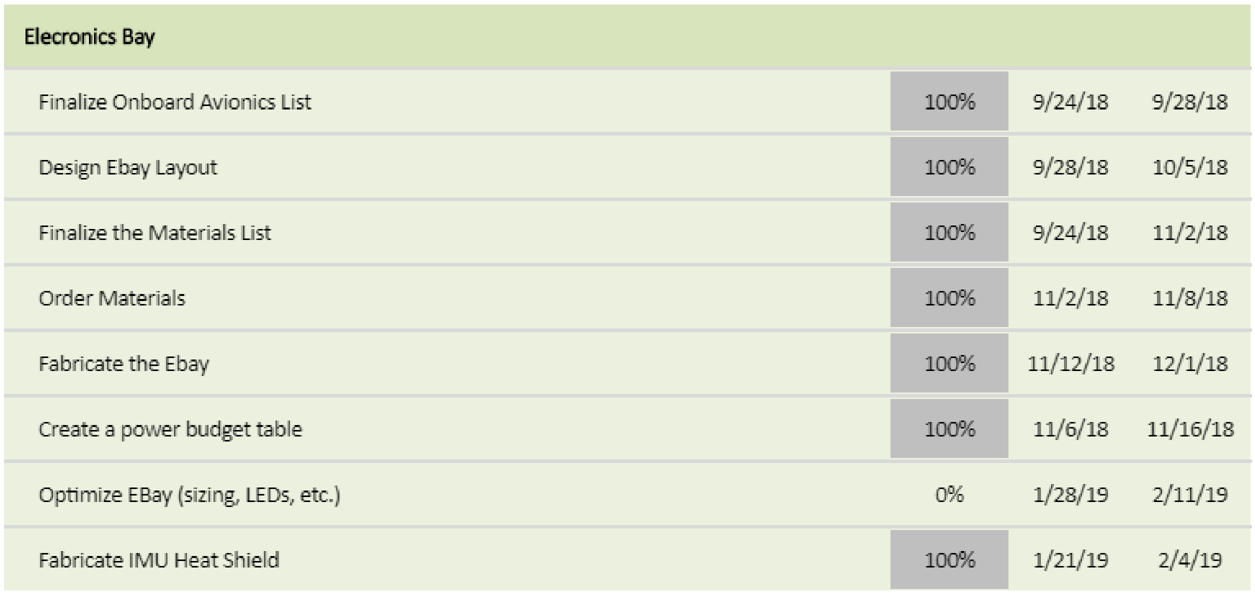
MATLAB ®/ Simulink ®



MATLAB / Simulink			
Finalize Equations of Motion for Passive Rocket	100%	9/24/18	10/2/18
Create Safety plot for varying Windspeeds	100%	11/5/18	11/16/18
Create Linearized System of Equations to Develop Preliminary control law	100%	11/2/18	1/28/19
Develop Preliminary LQR control law for full state	40%	1/27/19	2/17/19
Incorporate sensor noise and reduced state measurement into control system	0%	2/14/19	2/20/19
Iteratively Optimize Controller	0%	2/14/19	2/20/19

Figure 10 MATLAB®/ Simulink Gantt Chart.

Electronics Bay



Elecronics Bay			
Finalize Onboard Avionics List	100%	9/24/18	9/28/18
Design Ebay Layout	100%	9/28/18	10/5/18
Finalize the Materials List	100%	9/24/18	11/2/18
Order Materials	100%	11/2/18	11/8/18
Fabricate the Ebay	100%	11/12/18	12/1/18
Create a power budget table	100%	11/6/18	11/16/18
Optimize EBay (sizing, LEDs, etc.)	0%	1/28/19	2/11/19
Fabricate IMU Heat Shield	100%	1/21/19	2/4/19

Figure 11 Electronics Bay Gantt Chart.

2. Methodology

Many iterations went in to the responsibilities of the Flight Dynamics & Stability team. To properly design the fins, equations of motion for the rocket needed to be solved. Additionally, to design the control law and size the servos, we needed to have a flight simulation as well as aerodynamic analysis of the fins via CFD. To prepare for launch, we had to develop tests for the fins and success criteria for the flight.

2.1 Fin Design

The fins for the rocket were constructed based on the swept airfoil and clipped delta shapes. Each fin was designed to be 8 inches in both span and root chord length, to be twice the diameter of the rocket's body tube, and with a cross-section based on the NACA 64A010 airfoil, which was found to have the optimal shape and thickness to chord length ratio. The first iteration of the fin was designed and rendered using SolidWorks® to match the description and is shown below in Figure 13:

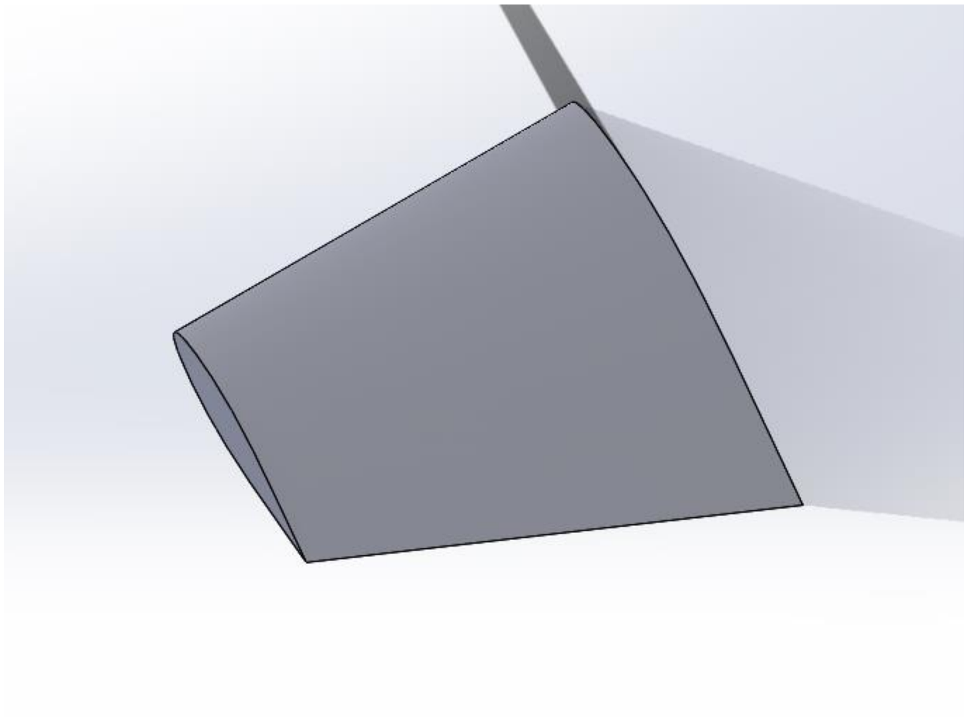


Figure 13: Preliminary stationary fin design. Copyright 2019, WPI.

From this fin, a section was cut to create an actuating rudder and provide active control. The rudder is cut 1 inch in from both the fin tip and the rocket body joining edge and 2 inches from the trailing edge at the corner of the rocket body joining edge and the trailing edge. The rudder is depicted in Figure 14 below:

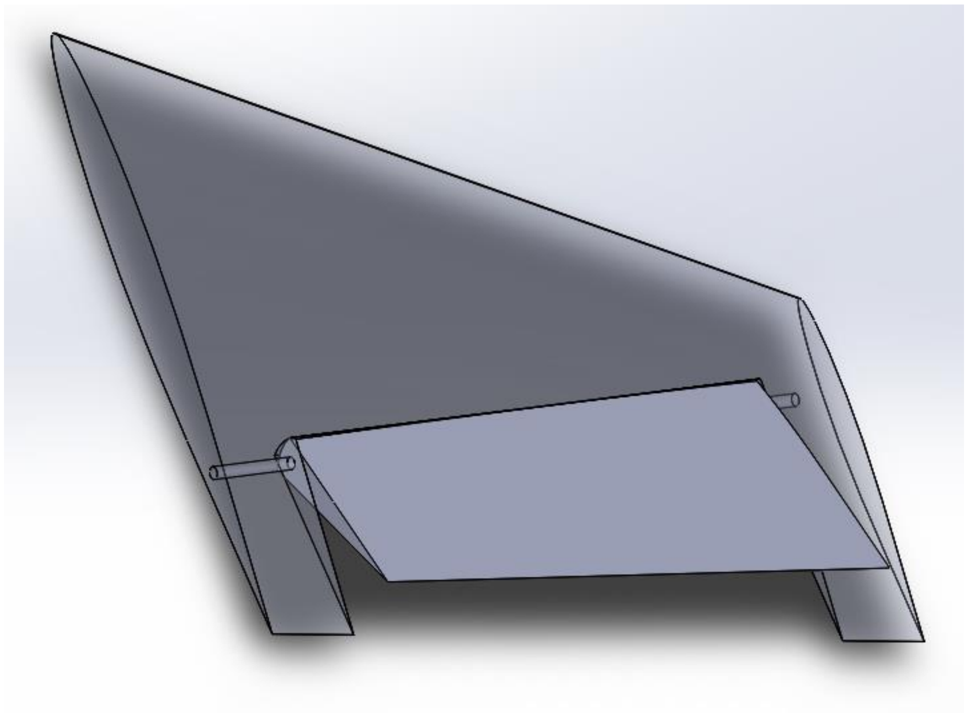


Figure 14: Preliminary actuated fin design. Copyright 2019, WPI.

To actuate the rudders, the rocket control system commanded servos to rotate each rudder as necessary. Originally, these servos were going to be placed inside the e-bay with a series of gears and belts running the length of the body tube to the fin actuation rod. Due to space constraints within the body tube, the servos had to be moved outside the rocket body. They were placed at the base of the fin, directly connected to the actuation rod. To reduce drag from the bulkiness of the servo motors, a shell was designed to cover the servos and was incorporated into the shape of the fin based on a NACA 66-021 airfoil. This fin shape iteration is expressed in Figure 15 below:

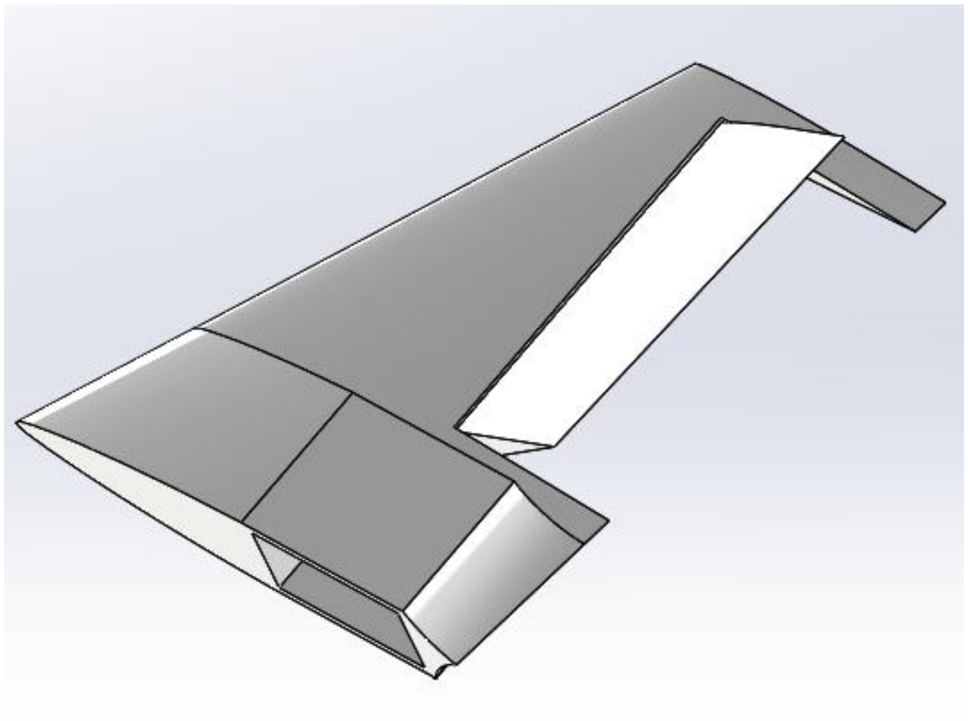


Figure 15: Final actuated fin design. Copyright 2018, WPI.

Lastly, to power the servos, wires ran the length of the body tube from the LiPo battery in the e-bay to the servos. The sizing of the servos themselves was based on a fluid analysis of the fins and the drag induced by the rudder at maximum deflection. The maximum deflection allowed for the rudder was 45 degrees, however it was assumed that a deflection of no more than 20 degrees is necessary, allowing for a Safety Factor of 2. Using a deflection angle of 20 degrees, the required torque for a servo can be calculated as follows, with aerodynamic constants and variables defined in Table 2:

$$F_D = \left(\frac{1}{2}\right) \bullet \rho \bullet v^2 \bullet A \bullet C_D \tag{14}$$

$$\tau = r \bullet F_D \bullet sin\theta \tag{15}$$

Table 2: Fin Aerodynamics Constants and Variables

Constants and Variables	Definition
F_D	Drag force
ρ	Density of air
v	velocity
A	Cross sectional area of rudder
C_D	Drag coefficient
τ	Torque
r	Radius of lever arm

Using Fluent®, the fin and rudder system at maximum rudder deflection was analyzed at a velocity of 150 meters per second, giving an average drag coefficient of 0.00977 for the rudder as shown in Figure 15.

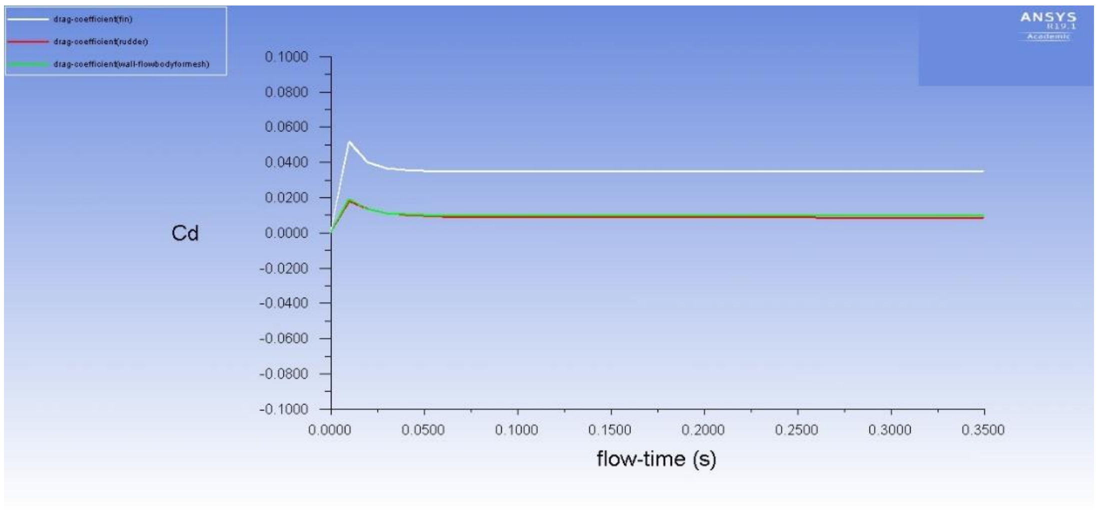


Figure 12: Plot of Drag Coefficient against Flow Time using ANSYS® Fluent®. Copyright 2019, WPI.

The white line in the plot above represents the drag coefficient of the fin, while the red line represents the drag coefficient of the rudder. The green line in the plot represents the drag coefficient of the rudder's mesh separation layer. Then based on the equations, the required torque was found to be 6.407 kilogram-centimeters or 0.6283 Newton-meters. The servo chosen to fulfill this requirement, with a Safety Factor of 2 in mind, was the Hitec Servos HS-755MG servo. This servo provided 14 kilogram-centimeters or 1.412 Newton-meters, enough for a Safety Factor of 2. As a result, this servo was integrated into each fin via the servo shell, allowing each rudder to be rotated accordingly.

2.2 Solving the Equations of Motion

Once we determined the equations of motion that govern the motion of the rocket, the next step in the analysis of its motion was to numerically solve the equations of motion. This was done by inputting them into MATLAB® as a system of equations. These equations were functions of the 13 states of the rocket and time. Additionally, we had to include the thrust due to the rocket motor's and did so by incorporating them as an additional term in the body-fixed x-force balance equation. These thrust values varied as a function of time and were determined by summing the forces from the individual rocket motors and interpolating them so that they had the same number of time stamps as the simulation so that at each iteration of the simulation a new thrust value was taken. Once all these parameters were inputted into the equations of motion MATLAB® function, we created another script that iteratively solved the equations of motion using the MATLAB® function ode45 for the time span of the simulation, which was from 1 to 10 seconds. The solutions to these differential equations were the states of the rocket's motion and were then graphically simulated and used to animate a graphic composed of a cone and cylinder.

2.3 Active Control Design

Once we had finished modeling the flight of the rocket mathematically, the next major task to creating the actuated fin system was to develop a feedback control law. This control law consists of a matrix, k, that when multiplied by the states vector x, yields the deflection angles of the fins, u, necessary to stabilize the flight of the rocket. In order to determine this k matrix, we employed a variety of approaches and modeled the effectiveness and of these different technique within MATLAB®.

The first approach that we investigated is a form of optimal control known as the Linear Quadratic Regulator, LQR. This method creates an optimal gain matrix in order to control a linear system and can be calculated by using the MATLAB® function lqr. However, this form of control design only applies to linear systems, whereas the rocket we are attempting to control is a highly nonlinear system. Therefore, the first step in this method was to linearize the rocket's equations of motion. This was done by expressing the system as a series of four matrices A, B, C, and D. The A matrix represents the linearized equations of motion of the passive rocket and was found by taking the Jacobian of the 13 equations of motion of the rocket with respect to its 13 states. This yielded a 13 by 13 symbolic matrix that, when multiplied by the states provides a linearized version of the full system. The B matrix represents how the rocket responds to its various control inputs in our case, the four fin deflection angles. It was determined by taking the Jacobian of the equations of motion with respect to the four input variables and yielded a 13 by 4 matrix. The C and D matrices were far more trivial as the C matrix reflects the measured states and the D matrix reflects the errors due to sensor measurements, neither of which are required to generate a control law assuming full access to states and no errors in measurement. These four matrices represent the linearized equations of motion and were evaluated at an equilibrium position where the rocket was traveling straight up with no rotations or angular perturbations. This condition reflects the states shown below where the vertical velocity value was equal to the maximum vertical velocity predicted in the flight simulation as to be conservative. Additionally, we had to specify two additional matrices for LQR to work, Q and R. Q represents weighting on states that are more imperative to control where higher values of Q specify higher amounts of control for those specific states, and R represents the costs of using the various available control inputs, which in our case were all equivalent. Once these six matrices were specified the lqr command in MATLAB® was then used with these matrices inputted as arguments to develop the closed loop feedback gain matrix k.

In addition to using LQR, we also investigated two other control design techniques. The first of these was by using non-linear controls to create an estimate of the k matrix. This was done by taking the transpose of the numeric B matrix and then multiplying it by negative 1 and using that as the gain matrix k. This therefore allows for the gain to be of the correct dimensions and allows for the control force to oppose the motion of the rocket as it desired. The final method for determining k was to create a series of educated guesses where the rows would all be identical, and the magnitude of each value would corelate to how vital the state that it corresponds to was to control. i.e. the quaternions and angular rates would have higher gain values associated with them.

Once we determined the gain matrices as detailed above, we altered the rocket simulation to model the effectiveness of each control law. We did so by inputting the numerical gain matrix as a constant in the code and then for every time interval calculating the required control inputs by multiplying the current state by the gain matrix. These variables were then included as terms in the moment balance as described in the equations of motion. The simulation then modeled the flight path of the actively controlled rocket and allowed for us to determine the corrected states and controlled flight path of the rocket.

Shown below is a block diagram of the control system that we proposed created in Simulink. It depicts the plant, actuators, sensors, and the control block.

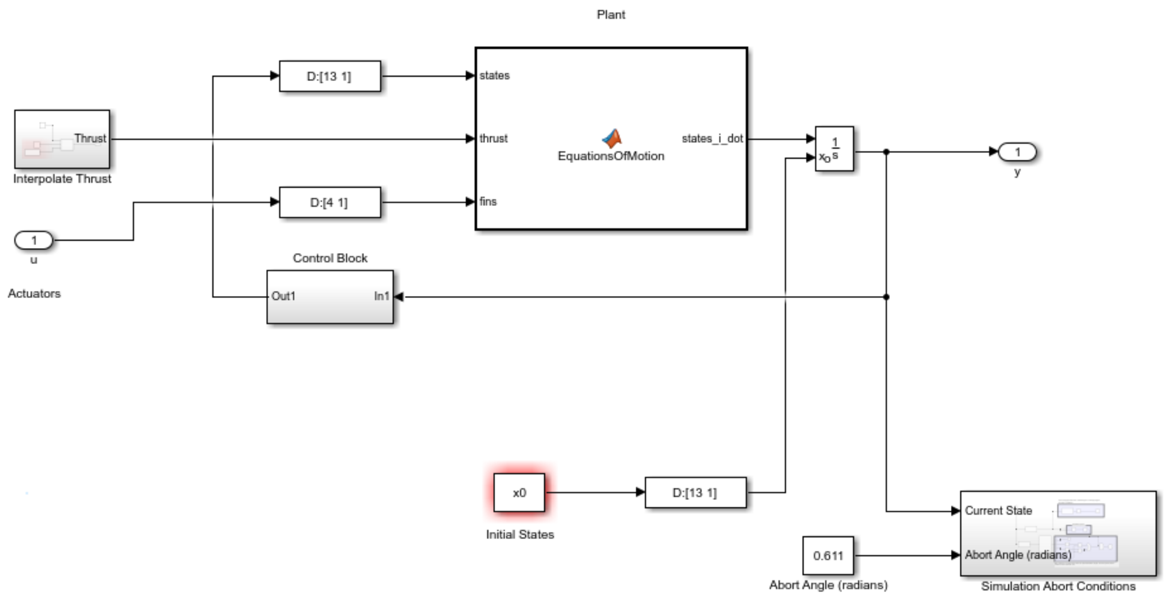


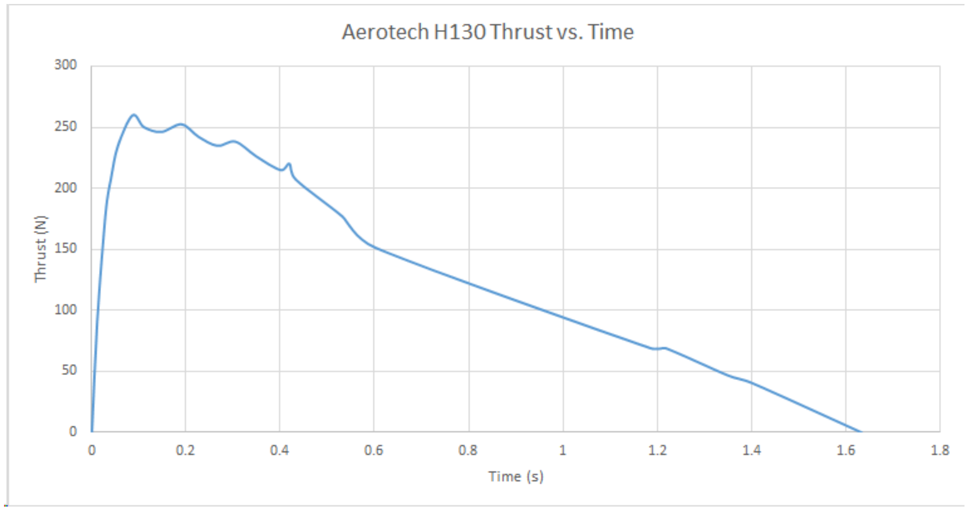
Figure 13: Block Diagram of Control System

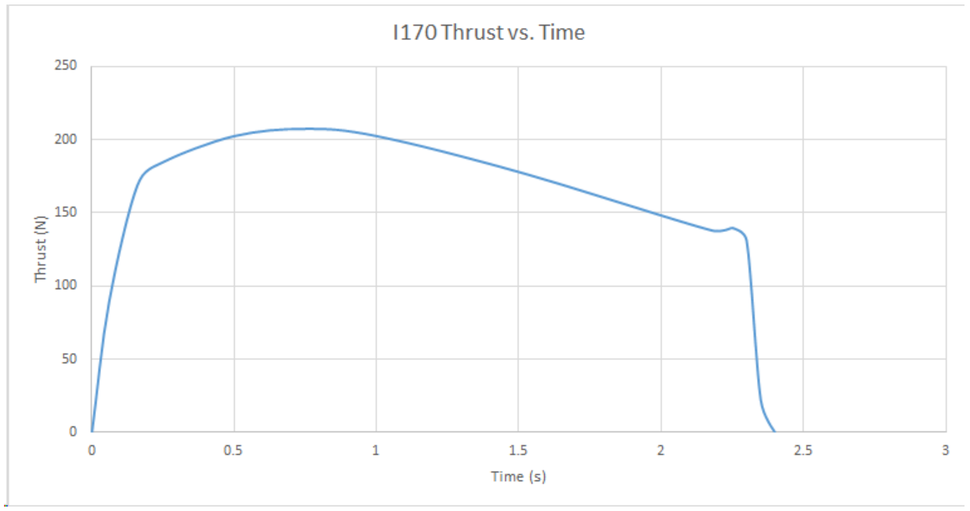
2.4 Creating the Simulation

After confirming the equations of motion for our system, we then moved forward with simulating the rocket's trajectory in MATLAB®. Due to the flight time being so short, we maintained a frame of reference in the inertial frame, neglecting any affects due to the Earth's rotation. This involved a lot of vector rotations using the MathWorks Aerospace Toolbox for quaternion rotations. Other frames of reference involved were the velocity-fixed frame for calculating aerodynamic forces on the rocket as well as the body-fixed frame for representing the thrust and for calculating torques imposed on the rocket body. The code was constructed such that the values of the rocket parameters were read through a data file, allowing for quick updates to the values when necessary. This proved to be quite useful since we began developing the code while the other two teams were designing the rocket and gathering CFD data.

Initial steps in the modeling process involved simulating projectile motion with only acceleration due to gravity, which allowed us to confirm the validity of the simulation through hand calculations. The next step involved incorporating lift and drag forces into the equations of motion. We calculated the lift and drag forces in the velocity-fixed coordinate system, then rotated the resulting aerodynamic force vector into the body-fixed frame, then the inertial frame. Since aerodynamic forces create a moment on the rocket body at the center of pressure, we used the body-fixed frame to calculate the moment due to the total aerodynamic force.

Initially, the Propulsion, Staging and Recovery team planned to use Aerotech H130 and Cesaroni I170 rocket motors, but eventually settled on the Cesaroni I218 motor. After acquiring the thrust curves for the Cesaroni I218 rocket motor from thrustcurve.org, we incorporated the force components into our simulation. The thrust curve for the Cesaroni I218 motor can be found in Figure 16 below.





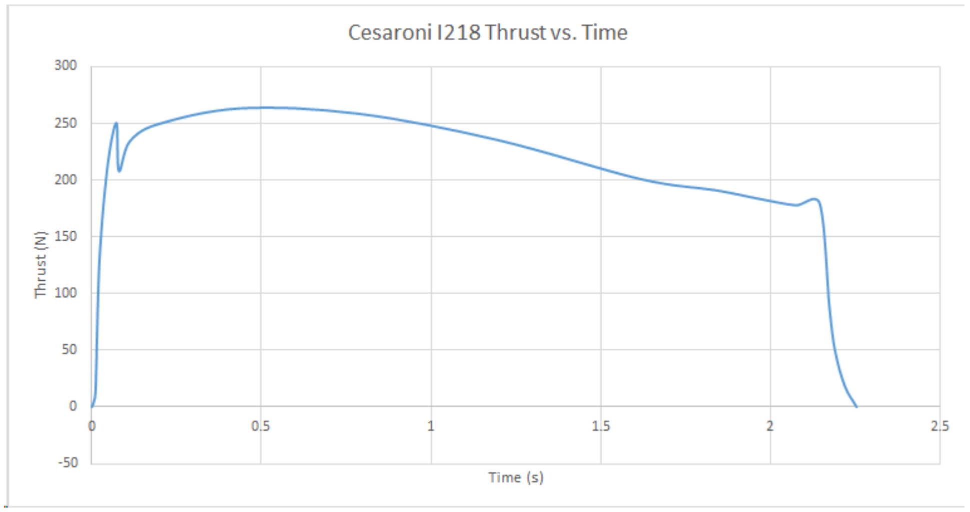


Figure 14 Rocket Motor Thrust Curves.

We developed a structure in MATLAB® which describes the motor name, physical location on the rocket's body, and the thrust over time. Using the interpolated thrust data, we then had the ability to simulate the force imposed on the rocket by the motor as well as the moments at any point in time during the launch. Although the moments (in theory) should all cancel each other out in normal operation, it was useful to include their calculation when simulating worst-case scenarios involving motor and fin failures.

The final component to the model was to model the torques due to fin deflection. Before we had obtained accurate values for the fin aerodynamic coefficients, we were able to incorporate estimates of these values to confirm the validity of the force and moment calculations due to the fin deflections. In the model, each fin was assigned a body-fixed coordinate, so the input fin deflection angles would produce moments about the center of pressure and thus adjust the rocket body's rotational orientation. We omitted translational forces imposed by fin deflection since their magnitudes were negligible.

The simulation would detect apogee by "deploying" the parachute when the inertial-frame vertical velocity is negative, (the rocket begins to descend). A second set of equations of motion, provided to us by the Propulsion, Staging and Recovery team described the behavior of the rocket as it falls with the parachute. With the wind model, the resulting aerodynamic forces applied to the body are calculated, giving us an estimate of how far away the rocket will land. The following plot describes the simulated wind model specific to our launch site.

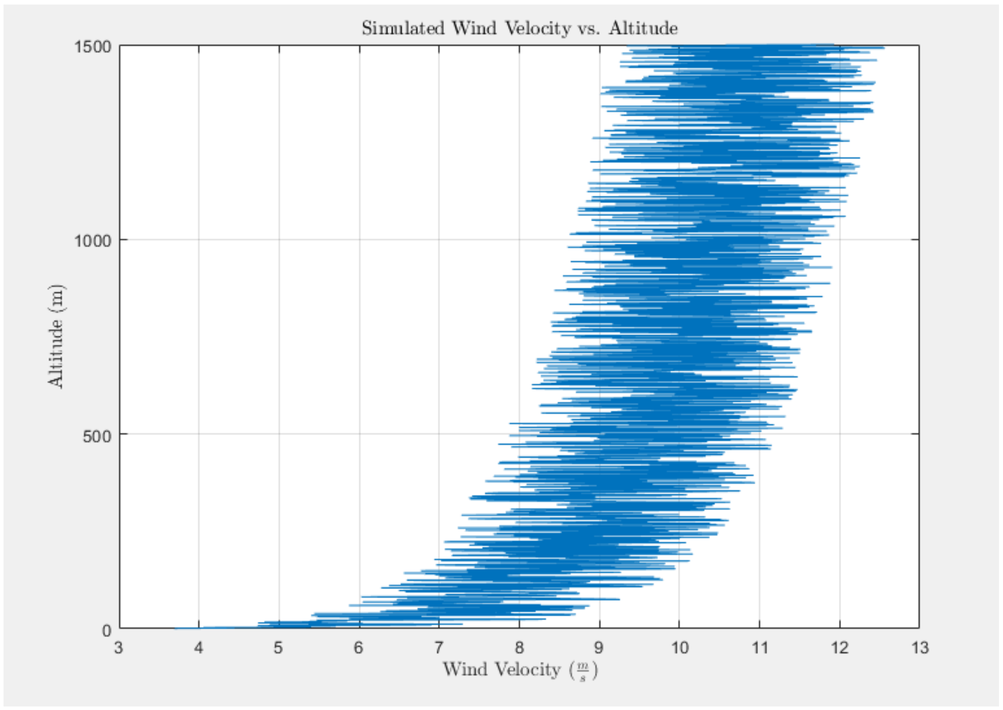


Figure 15: Simulated Wind Velocity vs. Altitude. Copyright 2019, WPI.

The plot above was created by using a mathematical model to calculate wind speed as a function of height. This model stated that the total wind speed at any given point is equal to the mean wind speed for this area plus a turbulence or gust component [13]. This gust component was approximated as a zero-mean random process using a model called the Davenport spectrum as illustrated in the equation below [13].

$$S_w(\omega) = 4800 \bullet \text{vm} \bullet \kappa \bullet \beta \bullet \omega\big(1 + (\beta \bullet 2 \bullet \omega^2)\big) \tag{16}$$

$$\beta = \frac{600}{\pi \bullet vm} \tag{16 a}$$

$$\kappa = [2.5 \bullet \ln(\frac{z}{z_0})]^2 \tag{16 b}$$

$$\text{vm} = \text{mean wind speed}$$

$$\text{z} = \text{height}$$

$$z_0 = \text{surface roughness}$$

This spectrum was then used to determine the velocity of wind as a function of altitude and this velocity was then used to calculate a wind force that was assumed to be unidirectional and perfectly horizontal as shown in the equation below [13].

$$F = .002 \frac{N}{\left(\frac{m}{s}\right)^2} \bullet V_n^2 \tag{17}$$

This wind term could then be added to the force balance in the x- and y-directions of the rocket to simulate the effects of wind on the rocket.

The equations of motion were all defined in the inertial frame then run through an ODE solver in MATLAB®. The resulting 13 states (position, velocity, quaternions, and angular velocity) were plotted using two forms of animation: a “stationary” view which viewed the rocket launch from a distance, and a “following” view which followed the rocket along its trajectory and allowed the user to visualize how it rotates during flight. Examples of these respective plots are shown below:

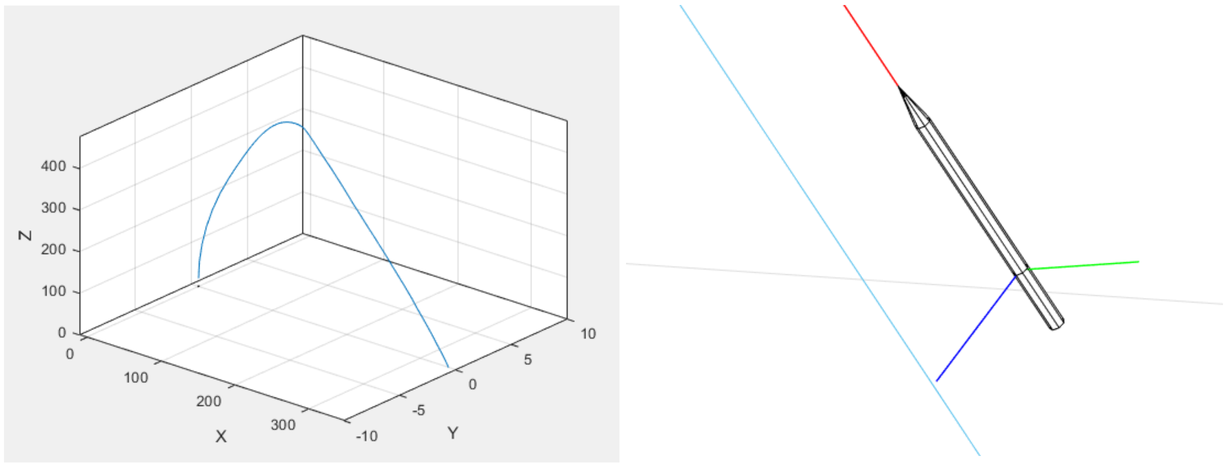


Figure 16: "Stationary" and "Following" visualizations of the rocket's trajectory. Copyright 2019, WPI.

For the test flight this simulation tool proved useful in estimating the rocket's landing range at various wind speeds. By changing the magnitude of the average wind speed, we plotted the distance from the launch site the rocket landed at. The plot, shown in Figure 15, shows that with increasing wind speed, landing radius for the rocket gets larger:

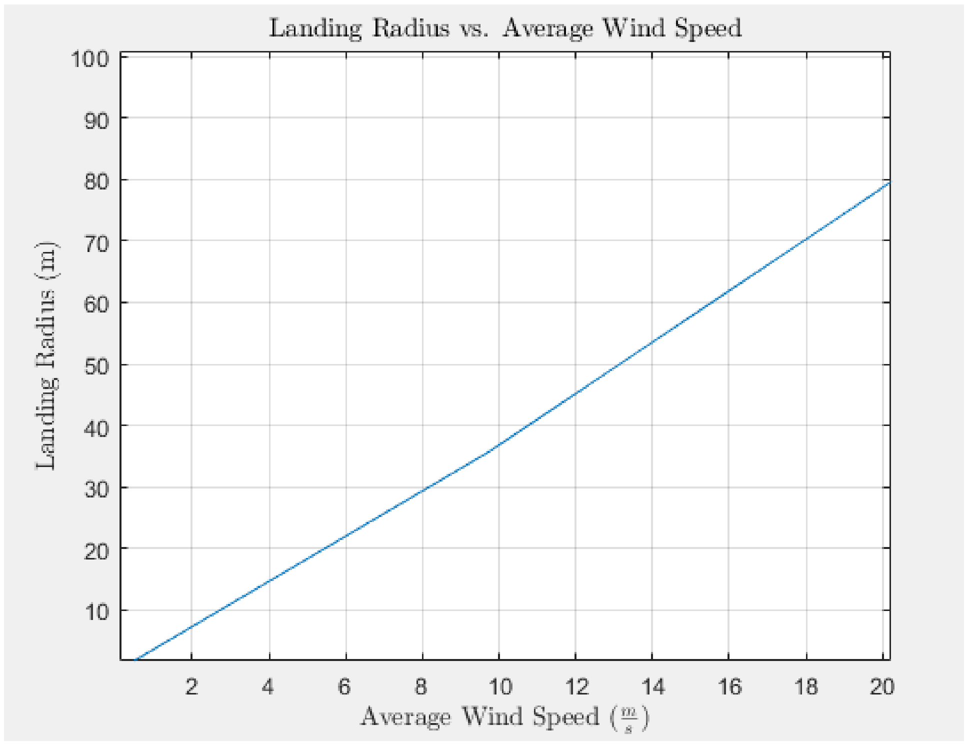


Figure 17: Landing Radius vs. Average Wind Speed. Copyright 2018 WPI.

There was a chance during flight that one or more of the motors could fail or not ignite at all. Because the motors were all situated away from the vertical body-fixed axis, a motor failure would cause unwanted torque on the rocket body, resulting in a curved trajectory. Our simulator was designed to allow us to disable motors, ultimately helping us visualize different trajectories if a motor failure were to occur. The following plots in figures 20 and 21 visualize several motor failure cases during flight:

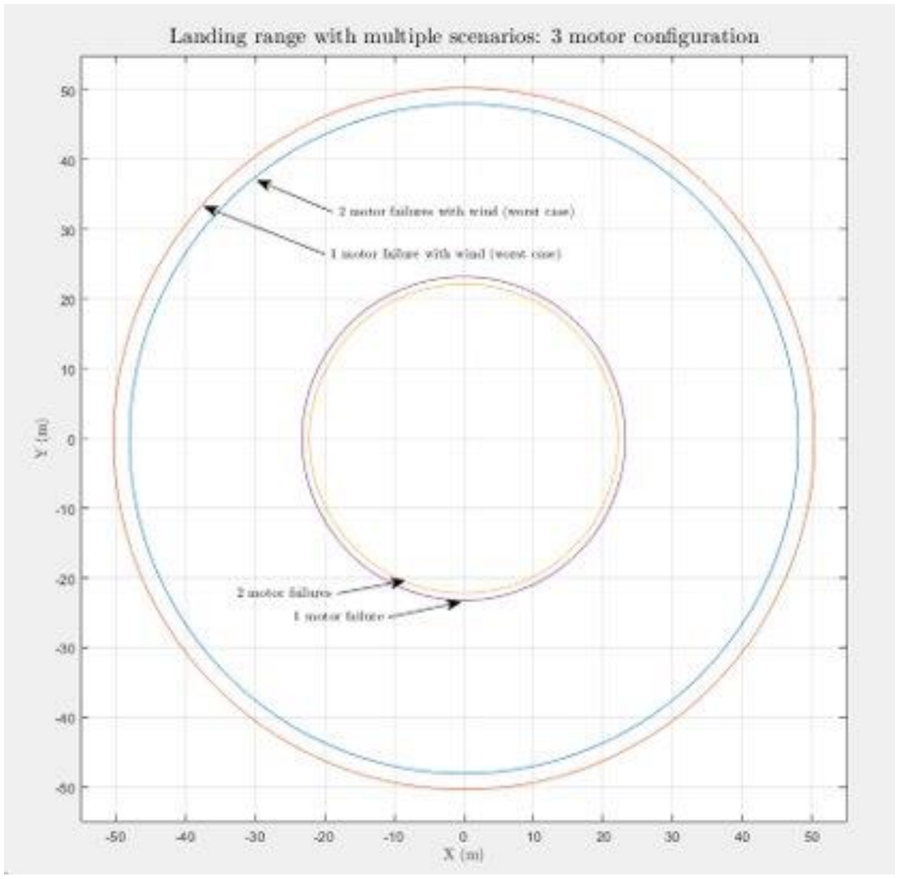


Figure 18 Landing Range with Multi-Motor Failure.

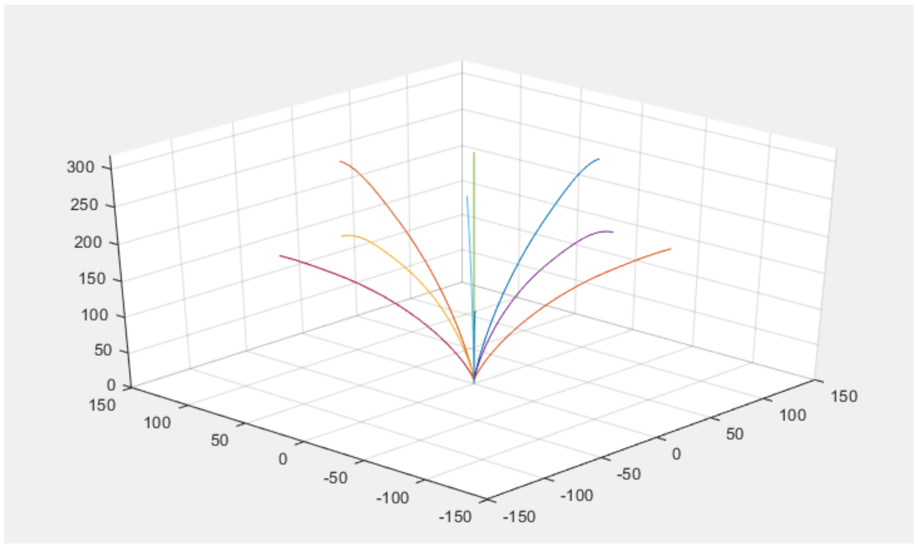


Figure 19: Various trajectories with motor failures and no wind.

In preparation for the first launch we created safety plots to estimate the trajectories and landing radii of various motor failure cases. Figure 20 shows estimates of maximum landing radii in four motor failure cases, assuming the three-motor configuration of Aerotech H130 motors. The inner two circles show the simulated landing radii of single and double motor failures with no wind, assuming the rocket launch angle was vertical. The outer two circles are the simulated results of the worst case of a single and double motor failure case with the wind model. Since wind blows in one direction, the "worst case" motor failures would be those which produce a moment on the rocket body in the same direction as the wind. This leads to the largest landing radius out of the six possible motor failure configurations. This means that figure 20 predicts a minimum and maximum landing radius with motor failures, depending on wind velocity and direction relative to the failed motors.

Figure 21 was created using the original seven-motor configuration (three H130 motors with four Aerotech I170 boosters). It depicts trajectory rather than landing radius in different motor failure cases. This was helpful in determining whether this motor configuration was safe since we could visualize roughly how angled the rocket's trajectory would become in any motor failure case. We ended up not using this seven-motor configuration for our first launch.

2.5 Electronics and E-Bay Fabrication

The electronics bay was the heart of the rocket. It was crucial that the e-bay be constructed properly not only to control the fins, but also to send the firing signal for separation and for parachute deployment.

The construction of the e-bay included two metal rods that serve as runners to the top and bottom bulkheads. Attached to the rods using zip-ties was a thin piece of plywood with the avionics and electronic systems secured to it and powered by a LiPo battery.

The e-bay was constructed using ½” and ¼” plywood, ¼” aluminum rods, ¼” hex nuts, and zip-ties. To build the e-bay, we first designed it in SolidWorks®. We then laser cut the bulkheads and the center piece out of plywood. The top bulkhead was cut out of ½” plywood, while the center of the e-bay and the bottom bulkhead were cut out of ¼” plywood. The top bulkhead needed to be thicker because it experiences a strong force from the shock cord when the parachute is ejected. The center piece had a ½” diameter hole laser cut from the bottom to account for wires traveling from the avionics to the LiPo battery.

After being laser cut, the pieces were sanded down, so they were smooth to the touch and to better fit inside the body tube of the rocket. Two ¼” holes were drilled in to both the top and bottom bulkheads for the aluminum rod runners. Two smaller holes were drilled in to the top bulkhead for the U-bolt and the U-bolt was secured using hex nuts. The top bulkhead was sealed to the body tube using epoxy resin.

Using a Dremel rotary grinder, the steel rods were cut to 6.75” in length and filed to have a smooth edge. To have the center piece be removable for wiring and debugging, we drilled 1/8” diameter holes in the center piece and loosely attached zip-ties to the holes. This way the center piece was still attached to the runners for stability purposes but could also be slipped off the rods if needed.

For the avionics, we soldered the altimeter and SD card reader to a PCB board, which integrated them with the CPU that was “plugged” into headers on the same piece of PCB. Since the IMU needed to be placed at the rocket’s center of gravity for accurate calculations, it was placed on a separate piece of PCB board and tied into the main board with a wire ribbon and headers soldered into the board. We raised the main PCB board from the center plywood using screws and aluminum spacers to give a half inch of space between the two. To keep the IMU from overheating and to mount it at the center of mass, a small housing was designed, and 3D printed using PLA plastic and mounted inside the rocket using Velcro.

Finally, the e-bay was finished by securing the aluminum rods to the top bulkhead, sliding on the center piece with the electronics, and securing the bottom bulkhead to the steel runners with hex nuts.

2.6 Arduino® Code for Flight Controller

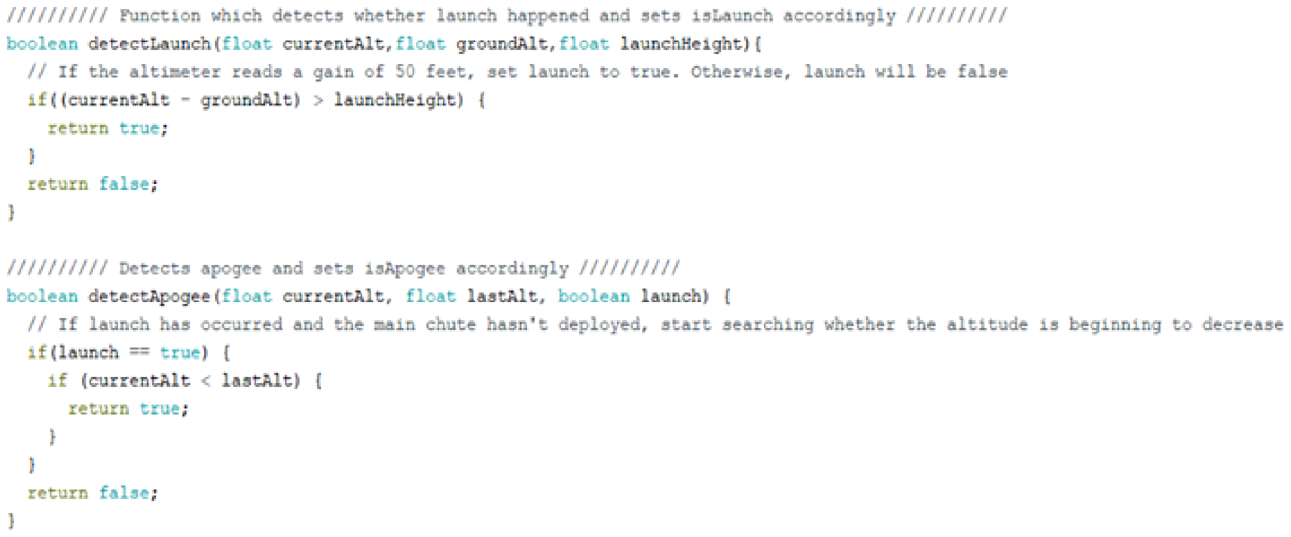
Each of the sensors in the e-bay needed to be able to send data to the Arduino®. We wrote code to communicate to the IMU and altimeter using I2C communication protocol. For the first launch, the e-bay had three mission-critical tasks: to detect launch and apogee, and to send the parachute ejection signal. To detect launch, the Arduino® solved for the following criteria: the linear acceleration in the vertical direction is more than $5m/s^2$, and the gained altitude exceeded 5m. The Arduino® only begins solving for apogee after launch was detected, making it critical that it correctly detects launch. This is also why there were two criteria for detecting launch, since if it incorrectly detected launch on the launchpad it could have led to an ejection misfire and thus unsafe conditions.

After detecting launch, the Arduino® would begin searching for apogee. The primary apogee detection mechanism was to trigger the ejection charge when the current altitude reading was less than the last altitude reading by one meter. It was critical that the ejection charge was deployed so we added fail-safes in the code that would ensure the signal was sent, and at the right time. To ensure the ejection charge was deployed properly, we added a minimum and maximum time after launch that the Arduino® would allow the ejection charge to be sent. If for some reason the Arduino® detected apogee right after launch, this minimum time constraint would prevent the charge from being sent. Likewise, if the Arduino® never detects apogee even after the rocket begins its descent, the ejection signal would be sent after a maximum time after launch. These time constraints allowed us to experiment safely with our apogee detection code. The times were determined using our launch simulation MATLAB® code.

Once the Arduino® detected apogee, it would send a signal to the MOSFET which would ignite the ejection charge for the main parachute. At this point it would also deploy the RF transmitter. This would allow us to determine from the ground whether the ejection signal was sent even if we could not see the rocket. The Arduino® code was made to detect and manage all important flight events as well as assist in the recovery of the rocket.

2.7 Testing and Success Criteria

Without a properly functioning flight controller, the rocket would quickly become an uncontrolled projectile incapable of recording useful flight data. To ensure that the integrated



```
////////// Function which detects whether launch happened and sets isLaunch accordingly //////////
boolean detectLaunch(float currentAlt,float groundAlt,float launchHeight){
  // If the altimeter reads a gain of 50 feet, set launch to true. Otherwise, launch will be false
  if((currentAlt - groundAlt) > launchHeight) {
    return true;
  }
  return false;
}

////////// Detects apogee and sets isApogee accordingly //////////
boolean detectApogee(float currentAlt, float lastAlt, boolean launch) {
  // If launch has occurred and the main chute hasn't deployed, start searching whether the altitude is beginning to decrease
  if(launch == true) {
    if (currentAlt < lastAlt) {
      return true;
    }
  }
  return false;
}
```

Figure 20: Arduino®® IDE code sample using MATLAB. Copyright 2019, WPI.

electronics worked properly, several tests were needed for avionics and electronics.

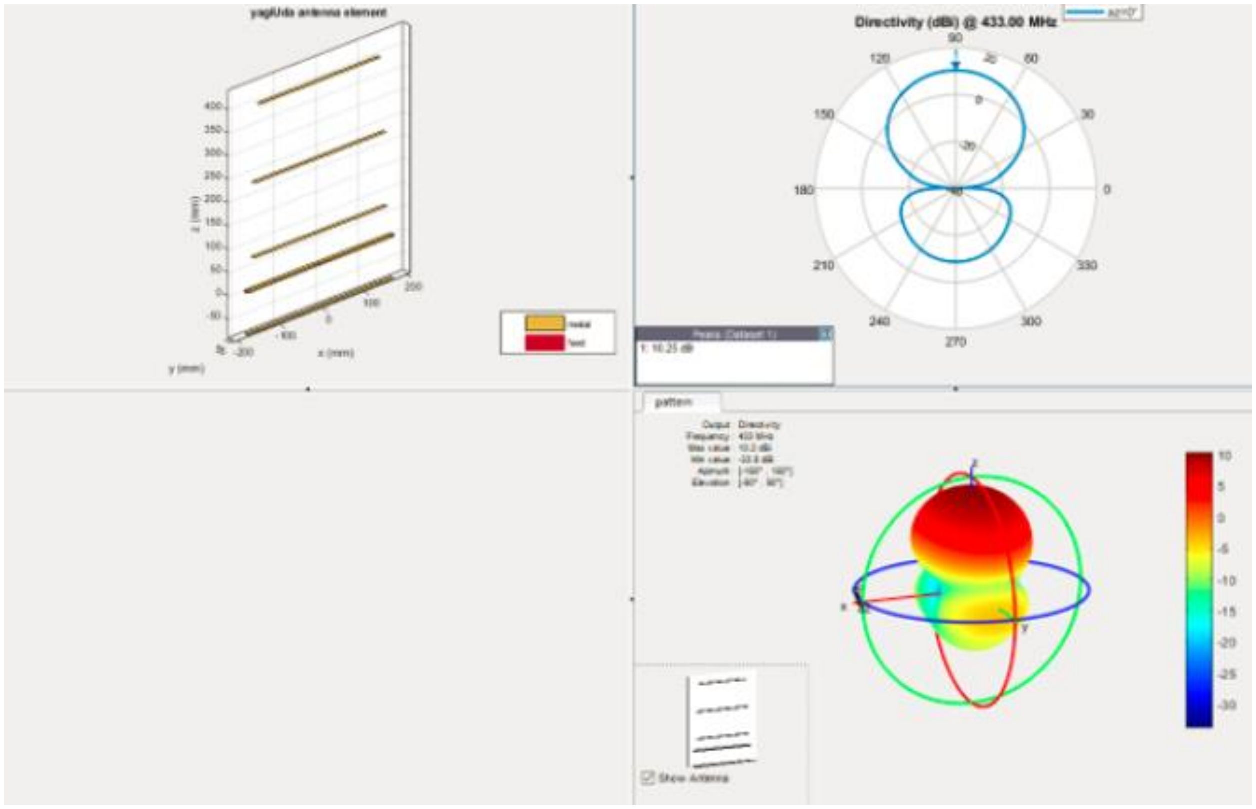
To test if the avionics were working properly before launch, a fully constructed e-bay was taken up and down six flights of stairs. This basic test was used to ensure that the altimeter, IMU, RF transmitter and SD card writer were working properly. Additionally, this test was used to ensure that the Arduino® could detect launch and apogee, as well as send the signal to deploy the parachute. The test was used to determine that the sensors were sending data to the Arduino® which was writing the data to the SD Card as well as executing the proper flight events at the correct time. After conducting the “Stair Test” with the e-bay, it was connected to the computer. The data was recorded, and the flight events were checked to see if apogee was detected and the parachute was deployed. The test was run numerous times to troubleshoot any issues with the code and fine-tune all flight events.

The second test required was to check that the proper voltage was being sent to the separation charges. The Arduino® IDE was used to artificially detect apogee (without throwing the e-bay) triggering the parachute deployment event. First, a voltmeter was used to measure voltage coming from the MOFSET once the parachute deployment was triggered. The voltage ranged from 0 volts before parachute deployment to 11.1 volts (battery voltage) after deployment. Once it was confirmed that the voltage was being sent from the MOFSET, pyrogen-coated wires were used in place of the voltmeter and the test was run again. If the proper voltage was sent from the MOFSET, the pyrogen-coated wires would ignite, making a successful test. This test gave us confidence that the separation charges would light during flight once the Arduino® detected apogee, therefore deploying the parachute. In addition to parachute deployment, the RF transmitter began to transmit a signal which could be heard over a radio tuned into 433 MHz, so we could track the rocket as is descends.

Regardless of the success of these tests, a fail-safe was built into the flight code in case the altimeter failed during flight. Based on the simulated rocket flight, an average flight time to apogee was calculated and used in conjunction with the Arduino®'s internal clock to ensure the parachute was deployed in flight. The code was written such that, in the event of an altimeter failure, the Arduino® would deploy the parachute 5 seconds after the calculated time to apogee. It was decided that adding a five second margin to deploy the parachute from the calculated time to apogee would ensure the rocket had started its descent before deploying the parachute. To test this system the altimeter "shut off" in the code simulating a failure and again a voltmeter was used detect voltage from the MOFSET.

2.8 Directional Antenna Design

The RF transmitter is only half the battle. The challenge for us was to create an antenna that would not only detect the transmitter signal but would be directional. A Yagi-Uda directional antenna is a commonly used directional antenna in the amateur radio community. We used the Mathworks® Antenna Toolbox to design a Yagi-Uda antenna for our needs. The following figure shows some of the results of our design:



Output Directivity
Frequency 433 MHz
Max value 10.2 dBi
Min value -33.8 dBi
Azimuth [-180°, 180°]
Elevation [-90°, 90°]

Figure 21 Antenna Performance.

The figure in the top left shows the position of the three director elements as well as the driven and reflector elements. The figures on the right show the antenna gain at different angles. They show that the gain is maximized at the front of the antenna at about 10 dB. This antenna was specifically designed to receive signals at 433 MHz which is the RF transmitter frequency. The following figure 24 shows the completed Yagi-Uda antenna. It was quite simple to construct. Knowing it is made from a cardboard tube and steel wires it was remarkable this functioned quite well when testing it. However, as seen from the previous figure the antenna's field of view was very wide, so following iterations of this design could be improved to narrow it down. A narrower field of view would give us a more precise estimate of the rocket's location during its descent. This is an area of improvement we would have made during a second launch attempt. B

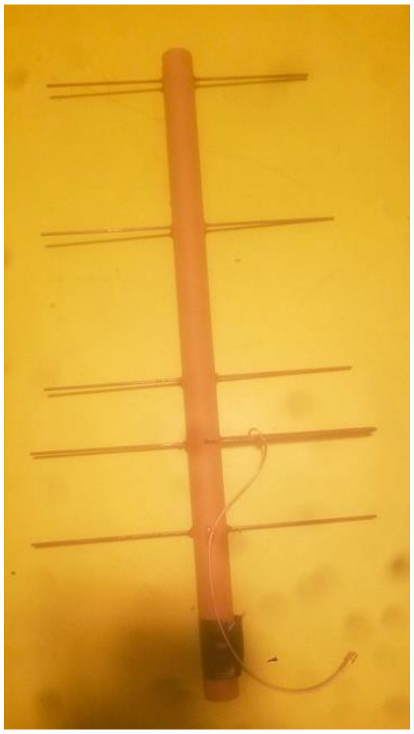


Figure 22 Directional Antenna.

2.9 Launch Success Criteria

In preparation for the preliminary test launch on January 19th, 2019, a list of success criteria from a dynamics and controls standpoint was determined. The main objectives for the flight were to execute all in-flight events properly and record all flight data from the altimeter and IMU. The criteria, except for the results column, is detailed in the Launch Test Criteria in Table 3 below:

Table 3: Launch Test Criteria

Test Objective	**Success Criteria**		**Verification**	**Results**
Record flight data	**Pass**	Arduino® successfully stores all data to SD card	- Data log has stream of data with all specified flight data	
	Fail	Arduino® does not store data and the Data log does not show data stream		
Detect Launch	**Pass**	Arduino® detects launch and sets variable isLaunch to true	- Data log will show value of isLaunch throughout flight	
	Fail	Arduino® does not detect launch and the Data log does not show values during flight		
Detect Apogee	**Pass**	Arduino® detects apogee and sets variable isApogee to true at apogee	- Data log will show value of isApogee throughout flight	
	Fail	Arduino® does not detect apogee and the parachute is not released		
Flight controller deploys signal for ejection charge	**Pass**	Arduino® successfully deploys the ejection signal to the ejection charge at apogee	- Data log shows it has deployed the signal for the ejection charge	
	Fail	Arduino® does not send the ejection signal and the stage separation does not occur		
RF Transmitter	**Pass**	RF signal transmitted and received after apogee	- Can hear the signal and receive a direction from the transmitter once the rocket lands	
	Fail	RF signal is not transmitted, and the song cannot be heard		

The execution of flight events was significantly more critical than recording data. This is because the flight events included detecting launch and apogee and deploying the parachute. If launch was not detected, then apogee would not be either, which meant the parachute would not deploy. If the parachute did not deploy, the impact of the rocket hitting the ground would be detrimental to the rocket's structure and the electronics inside. Thus, writing the code to detect flight events took precedence over writing the code to record flight data.

3. Results

The results of our methodology and overall project are described below in detail. We also provide in depth results of any testing that was done on the various subsystems that we worked on.

3.1 January Test Launch

The test launch took place on January 19th, 2019 in Durham, Connecticut. Although the overall launch was unsuccessful since the rocket never left the launch pad, there were some successes for the Flight Dynamics and Stability team. The criteria and their outcomes can be found in Table 4 below

Table 4: January Launch Test Criteria

Test Objective	Success Criteria		Verification	Results
Record flight data	**Pass**	Arduino® successfully stores all data to SD card	- Data log has stream of data with all specified flight data	**Pass**
	Fail	Arduino® does not store data and the Data log does not show data stream		
Detect Launch	**Pass**	Arduino® detects launch and sets variable isLaunch to true	- Data log will show value of isLaunch throughout flight	**Pass**
	Fail	Arduino® does not detect launch and the Data log does not show values during flight		
Detect Apogee	**Pass**	Arduino® detects apogee and sets variable isApogee to true at apogee	- Data log will show value of isApogee throughout flight	**N/A**
	Fail	Arduino® does not detect apogee and the parachute is not released		
Flight controller deploys signal for ejection charge	**Pass**	Arduino® successfully deploys the ejection signal to the ejection charge at apogee	- Data log shows it has deployed the signal for the ejection charge	**N/A**
	Fail	Arduino® does not send the ejection signal and the stage separation does not occur		
RF Transmitter	**Pass**	RF signal transmitted and received after apogee	- Can hear the signal and receive a direction from the transmitter once the rocket lands	**N/A**
	Fail	RF signal is not transmitted, and the song cannot be heard		

While the rocket never left the launch pad due to the lack of proper lubrication of the motors prior to installation, there were still two successes at the launch site: storing data and reading that launch did not occur.

The Arduino® successfully stored acceleration data to the SD card while the rocket was being mounted on the launch pad. As such, the first criterion receives a passing grade. The data for acceleration (in meters per second squared) versus time (in seconds) can be seen below in Figure 26.

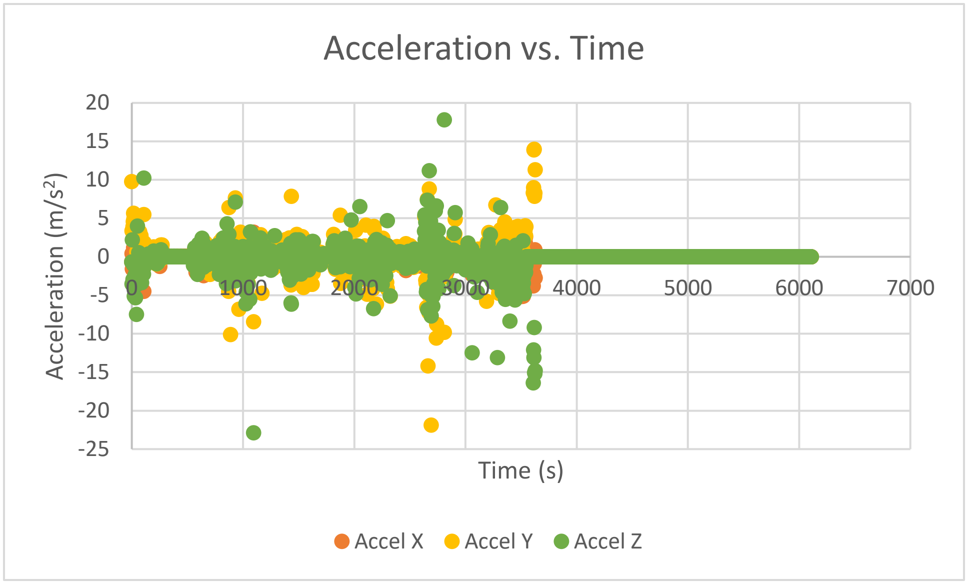


Figure 23 Rocket Acceleration vs. Time Copyright 2019, WPI.

The orange, yellow, and green dots on this graph represent body-fixed accelerations of the rocket in the x-, y-, and z-axes respectively. We speculate that the main cluster of points between 600 and 3,500 seconds is setting up the rocket on the launch pad and the failed launch. The straight green line along the time-axis was the team walking the rocket back to the site and turning off the Arduino®.

The second success in the test launch was the fact that launch was not detected. The criterion dictates that launch be detected, but since there was no launch to detect, the Arduino® was successful in not detecting launch. As such, the second criterion gets a passing grade.

The last three criteria do not receive a result, since they depend on the launch happening.

3.2 Final Product for the Fins

There were two rounds of 3D printing for the fins: passive fins and active fins. The first set of fins printed were the passive fins. We printed four fins, each of which was only one part. The passive fins were configured incorrectly in the 3D printing queue, so instead of printing from the base of the fin to the tip, the fin was printed from the bottom of the airfoil to the top. This created jagged transitions between the layers that needed to be sanded in order to create smoother layers. Once sanded, the fins were covered with a layer of epoxy. After the epoxy dried, the fins were sanded again to create a smooth finish. The fins were then ready to be mounted to the rocket to use for the test launch on January 19th 2019. The passive fins mounted to the rocket body can be seen below in Figure 27.

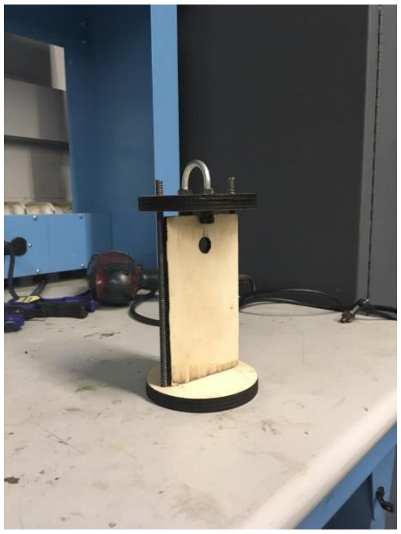


Figure 24 Electronics Bay Skeleton.

After the test launch, we were ready to print the active fins. The active fins were printed in two parts: the airfoil and the rudder. The airfoil was the same shape as the passive fin except there was a cutout for the rudder to fit into. The 1/8" diameter rod would be inserted through the airfoil and the rudder and then connected to the servo motor to actuate the rudder. This time, the fins were oriented correctly in the 3D printing queue and were printed from the base to the tip. This ultimately created smoother transitions between the layers and the fins did not need to be sanded. The airfoil, full active fin, active fin without the rudder, and the rudder can be seen below in Figures 28, 29, 30, and 31 respectively.

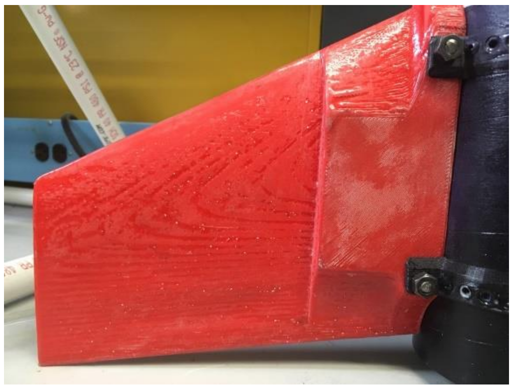


Figure 25 Passive Rocket Fin.

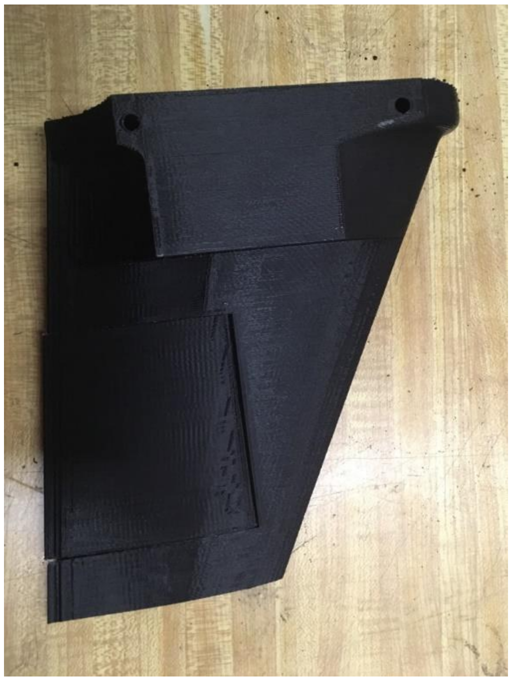


Figure 26 Active fin with Rudder.

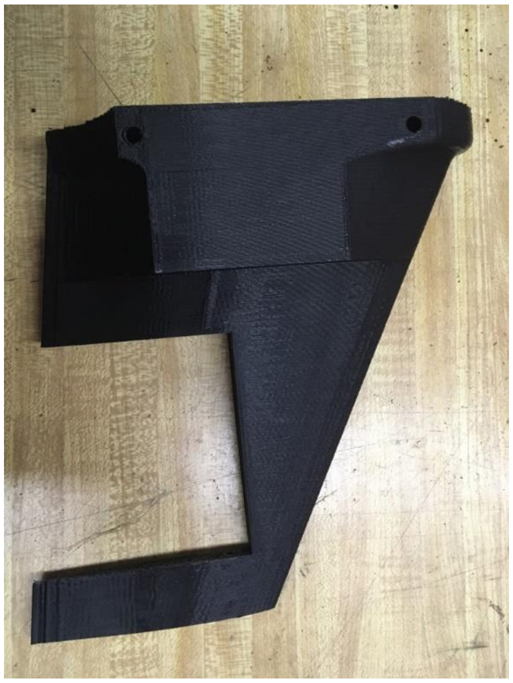


Figure 27 Active Fin without Rudder.

Figure 28 Active Fin Rudder.

After assembly of the fins, the actuation code was developed further using MATLAB®.

3.4 E-bay configuration and Capability

The final e-bay configuration consisted of two bulkheads attached by two ¼" threaded steel runners with a center piece of plywood. Surrounding this skeleton was a 6-inch-long blue tube piece that is four inches in diameter. On the bottom of the tube the runners, and plywood bulkheads were attached by both nuts and epoxy. The top bulkhead is secured with nuts only to allow for easy access to the electronics within. On the top of the top bulkhead there was also a U-bolt for the parachute shock chord. Located on the plywood centerpiece was the main PCB board which housed the Mega 2560-Core microprocessor, an Adafruit MPL3115A2-I2C altimeter, a Micro SD Card breakout board, a 433 MHz RF transmitter, as well as an 11.1v 1100 mAh Lithium Polymer battery located on the opposite side of the plywood centerpiece. Located at the rocket's center of gravity was the IMU, an Adafruit 9 degree of freedom absolute orientation sensor. Since the IMU sat away from the main e-bay a hole had to be drilled in the top bulkhead for the wires which ran from the main PCB board to the IMU.

As detailed in Table 4, the avionics of the e-bay were successful in detecting launch, apogee, and when all sensors were calibrated. In addition to being able to detect these things, the e-bay was also successful in its ability to send current to the separation charge and trigger the separation for the parachute. To detect launch and apogee we used the altimeter readings. The Arduino® sketch compared the current altimeter readings to the ones before it and determined when the rocket had launched and reached apogee. The Arduino® detected launch when the measurements it was taking were ten meters more than the original launch height which was reset constantly to void measurement creep. It detected apogee once the measurements taken were three meters lower than the measurement before it. Once apogee was detected the parachute was deployed. Using a difference in measurements of three meters was to ensure that the rocket was falling before the parachute was deployed. There was some sensor noise, and the difference in measurements had to be large enough to overcome the sensor noise. Finally, the Arduino® was capable of writing to the SD card when all three sensors were fully calibrated, the altitude measurements, accelerometer measurements, when launch was detected, when apogee was detected, and when the parachute was deployed. This information proved useful for post flight analysis especially during the testing of the sensors.

3.5 Final Simulator

The MATLAB® simulator was one major success of our group's contribution to this project. Countless hours were spent creating, modifying, and debugging this simulator. Eventually we were able to validate its evaluations by comparing its calculated flight trajectories alongside hand calculations as well as Open Rocket® (a commonly used hobbyist rocketry tool) simulation results. Throughout the year it proved to be quite useful in providing all three sub-teams with needed information about the predicted flight outcomes and in this regard was indispensable.

The main function of this simulator is to output an array of all thirteen states over a time span, given the rocket's physical parameters, wind velocity, the rocket's motor configuration and its fin deflection angles. The end goal for this simulator was of course to act solely as a non-linear plant model for the design of the fin active control system. However as mentioned previously it has proven to be useful in many more ways throughout the span of this project.

The simulator solves two differential equations throughout its flight: `equations_of_motion.m` and `main_chute_equations_of_motion.m`. These two functions hold the equations of motion of all thirteen states during flight and during the parachute descent, respectively. They employ a combination of custom quaternion rotation functions and Mathworks Aerospace Toolbox® functions to perform the numerous quaternion rotations.

The ODE solver function, `run_passive_control.m` solves the two sets of equations of motion. As shown in the code the time span is set to an arbitrarily high value. The function detects when the rocket reaches apogee and when it lands thus allowing it to automatically stop iterating through the equations of motion. This feature helps the function run seamlessly and allows for efficient simulation results. When the function is called it will begin solving `equations_of_motion.m` from the user-specified initial conditions, physical parameters, and desired motor configuration. Throughout this iterative ODE solving process the function will solve for apogee by detecting whether the inertial-fixed vertical velocity becomes negative. At this point the function switches over to begin solving `main_chute_equations_of_motion.m` with the current states as the initial conditions. The function then begins detecting whether the rocket's vertical position becomes zero. Once the rocket "lands" the ODE solving process is complete, and the overall time array and the corresponding state matrix is retained. Next, the function `animate_rocket.m` is called. This function inputs the state and time arrays as well as a specified plot type. There are four plot types: 'plot', 'plot_circle', 'stationary' and 'follow'. The 'plot' option will simply plot the rocket's calculated trajectory as a three-dimensional plot. This option is the simplest option and is useful for generating simple visualizations of the rocket's trajectory. The second option, 'plot_circle', plots a circle with a radius equal to the rocket's landing distance. The 'follow' option produces a three-dimensional representation of the rocket body along with its body-fixed axes oriented at the rocket center of gravity. This rocket will be translated and rotated over time, allowing the user to visualize the rocket's rotational motion throughout the flight. This plot option was very useful for us throughout the project in visualizing the rocket's flight stability and in debugging the equations of motion code.

4. Summaries, Conclusions, Recommendations, and Broader Impacts

4.1 Summary

Outlined below are the descriptions of progress made in all major areas of this MQP. We have described the progress made on implementing our methodology and the degree to which they were successful.

4.1.1 Summary, Conclusion, and Recommendations of Fins:

Two sets of fins were designed for this project. The first set of fins was a passive design created for the first launch. The second set of fins were active with a rudder powered by an on-board servo. Both sets of fins were designed in SolidWorks® and theoretically analyzed using ANSYS® Fluent® Software. The passive fins were based off a NACA 64A010 symmetric airfoil and the active fins were based off NACA 66-021. After being designed in SolidWorks® they were 3d Printed out of PLA plastic. The passive fins were tested in the wind tunnel to obtain experimental lift and drag coefficients to be compared to the analytical values found in ANSYS® Fluent®.

While the active fins were never tested in flight our team was able to construct them and wire the servos to the flight computer. We wrote an Arduino® file which told the servo to move multiple times at different rudder angles. The control law was not integrated into the active fins, but we were able to build a working model of that active fins ready to be used in conjunction with a completed control law for flight.

We cannot consider the active fins a complete success for two reasons. First, although we were able to successfully integrate the servos with the flight computer, we were not able to implement the control law to use the active fins to stabilize the rocket. Second, we were unable to test their effectiveness neither in the wind tunnel nor in actual flight. Time was a large factor in the lack of success for this portion of the project. With the complexity of the control law we ran out of time before being able to finish it.

4.1.2 Summary, Conclusion, and Recommendations, of Electronics bay and Design

The electronics bay was composed of a Mega 2560-Core microprocessor, an Adafruit 9 degree of freedom absolute orientation sensor, an Adafruit MPL3115A2-I2C altimeter, a Micro SD Card breakout board, a 433 MHz RF transmitter, and an 11.1v 1100 mAh Lithium Polymer battery. All, except for the IMU, are contained within blue tube casing. The shell for the E-bay was a blue tube body 6 inches long supported by two steel threaded rails with plywood caps on the top and bottom, and finally the PCB board which held most of our sensors and the Arduino® rested on a plywood plate. The entire assembly was tightened down with nuts on the top and epoxied together on the bottom. The IMU, which was placed at the center of gravity of the rocket, sat within a 3d printed shell for thermal protection from the separation charge.

The electronics bay design and configuration were a success. All sensors were integrated effectively with the microcontroller and the configuration fit within the six-inch space given for it within the rocket. Although there were some challenges with the soldering and organization of components of the flight computer it functioned properly in the end. The E-bay could have been improved by creating a custom-made PCB board which would have made the organization of components easier and neater.

4.1.3. Summary, Conclusion, and Recommendations of Flight Controller:

We had to build the flight controller from scratch. All components of the flight controller were ordered separately and soldered to the PCB board which our Arduino® plugged neatly into. There were some significant challenges in organizing all of electronics on the PCB board while ensuring that nothing was shorted out, and all sensors were connected to the proper pins for the Arduino®.

Once all sensors were properly soldered to the PCB the next challenge came with getting them to all work together and send data to the Arduino®. We were able to create an Arduino® sketch which can detect when the rocket launched, and reached apogee using altimeter data, triggered the signal for the separation charge, read out calibration data for all sensors, transmitted an RF signal once the parachute deployed, and recorded flight data to an SD card through the SD card reader. For the first test flight the E-bay functioned properly.

The flight controller was a success in functionality but was not flight proven. Although it worked properly in the initial test flight, since the rocket did not leave the launch rail and therefore not all functionalities of the flight controller were tested. In laboratory tests the flight controller can complete all tasks it was programmed to do which are stated in the Summary. One challenge that we faced with the flight controller was programming it to accurately detect launch and apogee. Since we only used altitude to detect them the flight controller it would fail to do so if the changes in altitude were not great enough to overcome the measurement error due to sensor noise. This program design could be improved by incorporating the accelerometer in some way to work in conjunction with the altimeter. The flight controller could also have been improved by using LED’s to show when all sensors were fully calibrated, as there was no way to tell that they were calibrated until after the flight data was recorded. LED’s would allow for us to easily be able to determine if the sensors were calibrated. Overall, the flight controller was a successful undertaking.

4.1.4 Summary, Conclusion, and Recommendation of MATLAB® Simulation and Control Law:

Overall, the construction of the MATLAB® simulation was a success. The simulation was able to successfully evaluate the rocket’s full trajectory given different motor configurations, wind velocity, starting conditions, and fin angles. It generated several result plots which helped visualize various aspects of the flight path and gain useful information while debugging the script. In the future this script could be given a graphical user interface (GUI) which would enhance the user experience and make much simpler to use. It could be designed to allow the user to create and test any rocket configuration and control law they choose with relative ease.

One shortcoming of our project was the failure to develop a completed control law. We were unfortunately unable to develop a feedback control that met our expectations for our project. A major setback for us the linearization of the rocket system which we were ultimately unable to achieve due to its complexity. Later in C-term we experimented with non-linear controller which did produce input fin deflection angles. We eventually ran out of time developing this controller and trying to linearize the system, ultimately leading to an incomplete stability controller.

4.2 Overall Project Broader Impacts

Building an actively controlled can be effectively used for three general purposes. Efficiently launching from the surface of Earth and into space, guiding missies to their destination, and for rocket enthusiasts to challenge themselves. In all cases, they are being done today and with some success.

From a social perspective, active guidance in rockets increases the efficiency and likelihood that a space mission will be a success garnering a higher likely hood for scientific advancement or improved technology in space which helps Earth such as communication and

GPS satellites. Active guidance can make for a fun project for enthusiasts as well. It must also be considered from a social perspective when thinking about war. Actively guided missiles are far more effective than “dumb bombs” and therefore are used over them in modern war. They are both expensive and deadly. Building technology like this comes with the understanding that it may be used to wage war and take lives, all at the taxpayer’s expense. Determining whether this is right or wrong is not for this paper, but it is a social consideration none the less. Actively guided rockets can bring about social improvement due to advances in science, can be a good challenge for enthusiasts and can be the cause of destruction.

Guidance is very important economically for space missions as it ensure mission success. Space programs are extremely expensive and without utilizing the technology at hand these missions would be less successful and in turn more expensive. Guided missiles have a direct economic impact because they are fired by countries, not by companies. For a missile to be fire from a country it is typically the taxpayer, or government money which would otherwise be used in different ways. In addition to this, both systems can have a large economic impact because of their complexity. There are multiple steps to the manufacturing process of building a guided rocket, and each step requires skilled workers. The manufacturing process effects the economy by creating a multitude of jobs in various fields.

Finally, from an environmental perspective, depending on the type of propellant, the exhaust from a rocket motor could be spewing greenhouse gases and other pollutants into the air negatively affecting the environment and more so the local population. Missiles are used to ensure destruction. The explosion from a missile negatively effects the environment by killing everything within a certain radius and destroying what humans and animals may call home. They change the landscape dramatically in a short period of time, that alone is not healthy for a local ecosystem. In general, actively guided rockets can be considered bad for the environment.

5. References

[1] J. Distefano, A. Stubberud, and I. Williams, *Feedback and Control Systems*, 2nd ed. Los Angeles: McGraw Hill Education, 20 (Coker, 2018)12.

[2] NA, “Torque and Equilibrium.” [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/torq2.html. [Accessed: 11-Oct-2018].

[3] T. Benson, “Rocket Stability,” 2015. [Online]. Available: https://spaceflightsystems.grc.nasa.gov/education/rocket/rktstab.html. [Accessed: 11-Oct-2018].

[4] N. Hall, “What is Drag?,” 2015. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/drag1.html. [Accessed: 11-Oct-2018].

[5] N. Hall, “The Drag Coefficient,” 2015. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/dragco.html. [Accessed: 11-Oct-2018].

[6] T. Van Milligan, “What is the best fin shape for a model rocket?,” *Peak Flight*, vol. 445.

[7] R. Nakka, “Fins for Rocket Stability,” 2001. [Online]. Available: https://www.nakka-rocketry.net/fins.html. [Accessed: 11-Oct-2018].

[8] T. Benson, “Rocket Control,” 2015. [Online]. Available: https://www.grc.nasa.gov/www/k-12/rocket/rktcontrl.html. [Accessed: 11-Oct-2018].

[9] F. Semiconductor, “Freescale Semiconductor Document Number: MPL3115A2 Xtrinsic MPL3115A2 I 2 C Precision Altimeter,” 2011.

[10] “TWS-BS RF MODULE Series Wireless Hi Power Transmitter Module (RF ASK) 3 nd. Edition,” 2008.

[11] “DOC ID: Arduino®-CORE DataSheet Arduino®-CORE DataSheet,” 2014.

[12] “BNO055 Intelligent 9-axis absolute orientation sensor,” 2014.

[13] W. Gawronski, “Modeling wind-gust disturbances for the analysis of antenna pointing accuracy,” *IEEE Antennas and Propagation Magazine*, vol. 46, no. 1, pp. 50–58, 2004.

[14] A. LAfflito, *A Mathematical Perspective on Flight Dynamics and Control*. Cham: Springer International Publishing, 2017.

[15] S. C. Prodduturi, “A Six-Degree-Of-Freedom Launch Vehicle Simulator For Range Safety Analysis,” M.S. Thesis, 2007.

[16] G. M. Siouris, *Missile guidance and control systems*. New York: Springer, 2011.

[17] P. Zarchan, *Fundamentals of Kalman Filtering: A Practical Approach, Fourth Edition*. American Institute of Aeronautics & Astronautics, 2015.

[18] A. Walker-Horn, “FLUENT - 3D Transonic Flow Over a Wing - SimCafe - Dashboard,” 2015. [Online]. Available: https://confluence.cornell.edu/display/SIMULATION/FLUENT+-+3D+Transonic+Flow+Over+a+Wing. [Accessed: 01-Oct-2018].

[19] H. Ashrafiuon, “Guidance and attitude control of rockets with fixed impulsive forces,” in *Proceedings of the American Control Conference (ACC)*, 2017, pp. 2237–2242.

[20] F. Trevisi, M. Poli, M. Pezzato, E. Di Iorio, A. Madonna, N. Bressanin, and S. Debei, “Simulation of a sounding rocket flight’s dynamic,” in Proceedings of *IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, 2017, pp. 296–300.

[21] N. Patra, K. Halder, A. Routh, A. Mukherjee, and S. Das, “Kalman Filter based Optimal Control Approach for Attitude Control of a Missile,” *ICCCI-2013*, 2013.

[22] Airfoil Tools, “NACA 63A010 AIRFOIL (n63010a-il),” 2018. [Online]. Available: http://airfoiltools.com/airfoil/details?airfoil=n63010a-il#polars. [Accessed: 01-Oct-2018].

[23] D. Wyatt, “An Actively Stabilized Model Rocket,” 2006.

[24] S. M. Kaplan Brian J Carlson and Professor A. Michael Demetriou, “Missile Attitude Control System Design , liAtt.41,” 2002.

[25] “Early Space Age! Launch Vehicle Dynamics! Launch Vehicle Flight Dynamics&quot; Political Rains and First Fruit: The Cold War and Sputnik ! 2!,” 2015.

[26] C. R. Garcia, “Project Hermes - MIT Rocket Team - MIT Wiki Service,” 2018. [Online]. Available: https://wikis.mit.edu/confluence/display/RocketTeam/Project+Hermes. [Accessed: 01-Oct-2018].

[27] “Proposal for Entry to NASA Student Launch Project U,” 2016.

[28] “Model Rocket Safety Code,” 2012. [Online]. Available https://www.nar.org/wp-content/uploads/2018/08/Model-Rocket-Safety-Code.pdf. [Accessed: 27-Feb-2019].

6. Appendices

Appendix A: Project Budget Breakdown

HPR MQP Budgeting Management				
	Total Budget	Order #	Description	Cost
MSAT	$ 2,000.00	1	ARR- Tube	$ 161.29
		2	West Systems	$ 100.84
		3	McMaster Carr	$ 23.45
		4	GiantLeap	$ 33.06
		5	Amazon-Cameras	$ 69.30
		Con.1	McMaster pt2 (With Controls)	$ 36.68
		6	DigiKey	$ 112.72
			Remainder:	$ 1,462.66
PSR	$ 2,000.00	1	Evike- Magazine	$ 27.00
		2	Apogee	$ 144.67
		6	McMaster	$ 32.69
		7	ServoCity	$ 9.99
		5	Apogee2: Fire items	$ 89.62
		MSAT 6	DigiKey Magnet Stuff	$ 31.11
		9	Mouser Transistors	$ 9.90
		11	Adafruit Motor Drivers (really Digikey)	$ 7.50
		13	McMaster: Garolite + Magnet Wire	$ 39.54
		8	H73J Motors - Apogee	$ 152.55
		10a	MotoJoe End Closures (x3 + x1) aft	$ 159.77
		10b	OffWeGo Aft Closure (x1)	$ 45.20
			Remainder:	$ 1,250.46
Controls	$ 1,250.00	1	McMaster-Wire/Rod	$ 26.03
		4	CopperHill-MegaCore	$ 40.95
		3	Adafruit-SD and IMU	$ 62.35
		2	Sparkfun-RF Transmitter	$ 4.95

			Remainder:	$ 1,115.72
			Total Spent	$ 1,421.16
Total	$ 5,250.00		Total Remainder:	$ 3,828.84

Appendix B: MATLAB Code for Equations of Motion

```
function states_i_dot =
equations_of_motion(states,rocket,motorCluster,thrust,windSpeed,u)
%% Define the quaternions
q           = states(7:10);
%% External Forces (inertial frame)
% Aerodynamic forces on rocket body
vel_i       = states(4:6);
vel_b       = quaternion_I_to_B(q,vel_i);
speed       = norm(vel_b);
alpha       = -atan2(vel_b(1),vel_b(3)); % Angle of attack
beta        = -asin(vel_b(2)/speed);     % Side-slip

Cx_v        = rocket.Cla * sin(alpha) * rocket.area;
Cy_v        = rocket.Clb * sin(beta)  * rocket.area;
Cz_v        = rocket.Cd  * rocket.frontArea; % Estimating the drag to be
directly opposite the nose
C_v         = [Cx_v; Cy_v; -Cz_v];
F_ad_v      = 0.5*(1.225)*(speed^2).*C_v;
q_v         = angle2quat(0,-alpha,beta);
F_ad_b_quat = quatrotate(quatconj(q_v),F_ad_v')';
F_ad_i          = quaternion_B_to_I(q,F_ad_b_quat);

F_g_i           = [0;0;-9.8*rocket.m];

%% Wind Force
%     windVel_b   = quaternion_I_to_B(q,[windSpeed;0;0]);
%     alpha_wind  = -atan2(windVel_b(1),windVel_b(3)); % Angle of attack
%     beta_wind   = -asin(windVel_b(2)/windSpeed);     % Side-slip
%
%     Fx_wind        = -0.5*(1.225)*(windSpeed^2)*rocket.Cla * rocket.area
* sin(alpha_wind);
%     Fy_wind        = -0.5*(1.225)*(windSpeed^2)*rocket.Clb * rocket.area
* sin(beta_wind);
%     Fz_wind        = -0.5*(1.225)*(windSpeed^2)*rocket.Cd *
rocket.frontArea;
%     Fwind_v        = [Fx_wind; Fy_wind; Fz_wind];
%     q_v            = angle2quat(0,-alpha_wind,beta_wind);
%     Fwind_b_quat   = quatrotate(quatconj(q_v),Fwind_v')';
F_wind_i           = [0.5*1.225*(windSpeed^2)*rocket.Cla; 0; 0];
%% External force
F_external_i     = F_g_i + F_ad_i + F_wind_i;

%% Thrust force
[~, motorCount] = size(motorCluster);
T_motors     = [zeros(2,motorCount);thrust]; % Creates a column vector for
each motor
T_net_b      = sum(T_motors,2);
T_net_i      = quaternion_B_to_I(q,T_net_b);
%% Net Force on the rocket body
F_net_i      = F_external_i + T_net_i;

%% External Moments (inertial frame)
```

```
% Rotate the distance from CP to CG into the inertial frame
cp2cg_b     = rocket.dcp-rocket.dcg;
cp2cg_i     = quaternion_B_to_I(q,cp2cg_b); % distance from the cp to cg
in inertial frame
% Moment due to aerodynamic force of rocket body
M_ad_i       = cross(cp2cg_i,F_ad_i);

%% Moments due to rocket motors
thrustCGOffset  = rocket.L; % Motor offset from the CG
for i = 1:motorCount
    M_motors_b(:,i) =
cross([motorCluster(i).location(1:2);thrustCGOffset],T_motors(:,i));
end
Mmotors_net_b = sum(M_motors_b,2);
M_motors_net_i = quaternion_B_to_I(q,Mmotors_net_b);
%% Moments due to Fins
% Fin locations (body fixed)
d = 0.2159;                        % Flipper offset from z-axis
l  = -(rocket.dcg(3) + rocket.L); % Flipper offset from the CG
f1 =
0.5*(1.225)*rocket.A_rudder*speed^2*sign(u(1))*2*pi*min(abs(u(1)),deg2rad(
30));
f2 =
0.5*(1.225)*rocket.A_rudder*speed^2*sign(u(2))*2*pi*min(abs(u(2)),deg2rad(
30));
f3 =
0.5*(1.225)*rocket.A_rudder*speed^2*sign(u(3))*2*pi*min(abs(u(3)),deg2rad(
30));
f4 =
0.5*(1.225)*rocket.A_rudder*speed^2*sign(u(4))*2*pi*min(abs(u(4)),deg2rad(
30));
% %
r1_b = [ d  0   l];
r2_b = [ 0  d   l];
r3_b = [-d  0   l];
r4_b = [ 0 -d   l];

f1_b = [ 0   f1  0];
f2_b = [-f2   0  0];
f3_b = [ 0   -f3 0];
f4_b = [ f4  0   0];

M1_rudder_b =  cross(f1_b,r1_b);
M2_rudder_b =  cross(f2_b,r2_b);
M3_rudder_b =  cross(f3_b,r3_b);
M4_rudder_b =  cross(f4_b,r4_b);

%  M_rudder_b = (M1_rudder_b + M2_rudder_b + M3_rudder_b + M4_rudder_b)'
M_rudder_b = [l*(f1-f3); l*(f2-f4); -d*(f1+f2+f3+f4)];
% intertial-frame moments due to the flippers
M_rudder_i      = quaternion_B_to_I(q,M_rudder_b);

%% Net moments on the rocket body
M_net_i       = M_ad_i + M_rudder_i + M_motors_net_i;
```

```
M_net_b       = quaternion_I_to_B(q,M_net_i);        % Body-fixed net
moments
w_b          = quaternion_I_to_B(q,states(11:13)); % Body-fixed angular
velocity
%% Omega Matrix and angular momentum to calculate qdot
Omega       = [[0            states(13) -states(12) states(11)];
    [-states(13)  0           states(11) states(12)];
    [ states(12) -states(11)  0          states(13)];
    [-states(11) -states(12) -states(13) 0         ]];
H           =  (rocket.I)'.*states(11:13);                   % angular
momentum

%% Define states_i_dot as the solutions to the equations of motion
posdot_i    = vel_i;                 % inertial frame velocity
veldot_i    = F_net_i / rocket.m;         % inertial frame acceleration
q_shifted   = circshift(q,-1);           % need to move the scalar part
back to the 4th index
q_dot       = circshift(0.5*Omega*q_shifted,1);   % time derivative of
quaternions
w_dot       = ((M_net_i-cross(states(11:13),H))' ./ (rocket.I))'; %
angular acceleration

%% Output term (dx)
states_i_dot    = [posdot_i; veldot_i; q_dot; w_dot];

%
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% end of Equations of Motion
%
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%% quaternion_I_to_B
    function B = quaternion_I_to_B(q,A)
        % Rotation from Inertial fixed to Body frame
        B = quatrotate(q',A')';
    end
%% quaternion_B_to_I
    function I = quaternion_B_to_I(q,A)
        % Rotation from Body fixed to Inertial frame
        I = quatrotate(quatconj(q'),A')';
    end
end
```

Appendix C: MATLAB Code for the Rocket Simulation

```
% Main launch simulator
% Authors: WPI Flight Dynamics and Stability MQP team
% Date:    02/28/2019
clc; clear; close all;
addpath('../Functions');
addpath('../Data_Files');
options = odeset('JConstant','on', 'RelTol',1e-4, 'AbsTol',1e-4);
load rocket;
load wind;
load parachutes;
%% Define your initial conditions and user inputs (inertial reference
frame)
initial_pos      = [0.0 0.0 0.0];  % Initial translational position
initial_euler    = [0.0 0.0 0.0];  % Initial euler angles (roll, pitch,
yaw)
initial_vel      = [0.0 0.0 0.0];  % Initial velocity
initial_omega    = [0.0 0.0 0.0];  % Initial rotational velocity
load motorCluster_v3;              % Choose motor configuration. (v1, v2,
v3)
motor_enable     = [1];            % Enable array (length must = # of
motors)
simulate_landing = 1;              % 1: simulate descent 0: break at
apogee
include_wind     = 1;              % 1: include 0: do not include
plot_type        = 'follow';         %
'plot','plot_circle','follow','stationary'
%% Initialize
states = zeros(1,13); % initialize state matrix
states(1:3)   = initial_pos';
states(4:6)   = initial_vel'; states(6) = states(6)+0.0001;
states(7:10)  =
angle2quat(initial_euler(3)+0.001,initial_euler(2),initial_euler(1),'ZYX')
;
states(11:13) = [0.0 0.0 0.0];

parachuteDeployed = false;
t0       = 0;       % Initial Time
tf       = 300;     % Final Time (not necessary to adjust)
nip      = 2;       % Number of integration points
nsteps   = 10000;
t        = t0;      % initialize t
stepSize = tf/nsteps;
tspan    = [t0:stepSize:tf]';       % Total time span
currentStates = states';            % State array used inside the loop
motorCluster = create_thrust_curves(motorCluster,tspan);
numMotors      = size(motorCluster); % size is the number of rocket motors
for i = 1:numMotors(2)
motorCluster(i).enable = motor_enable(i);
end
%% Begin solving for flight trajectory
for i = 1:nsteps
    t1 = stepSize*(i-1);
    t2 = stepSize*i;
    temp_tspan = t1:(t2-t1)/nip:t2;
%% Find the thrust at current time
    [~,index] = min(abs(t2-tspan));
    for j = 1:numMotors(2)
      currentThrust(j) = motorCluster(j).enable *
motorCluster(j).thrust(index); % Thrust in the inertial z axis. the Enable

% component sets the thrust for that motor

% to zero if enable is set to 0.
    end
%% Find the wind velocity at current altitude
    [~,index] = min(abs(currentStates(3)-wind.altitude));
      currentWindVel = include_wind*wind.velocity(index);
%% Solve the ODE
switch parachuteDeployed
    case false
    [tNew,tempStates] = ode45(@(tNew,currentStates)
equations_of_motion(currentStates,rocket,motorCluster,...

currentThrust,currentWindVel,...

[0;0;0;0]),...

temp_tspan,currentStates,options);
    case true
    [tNew,tempStates] = ode45(@(tNew,currentStates)
main_chute_equations_of_motion(currentStates,rocket,mainChute),...

temp_tspan,currentStates,options);
end
    t(i) = t2;
    currentStates = tempStates(nip+1,1:13)';
    states(i,:) = currentStates';
%% Break conditions
    % break if apogee is reached
    if (states(i,6) <= 0) && (parachuteDeployed == false)
        fprintf("Apogee reached at %0.1f meters \n",states(i,3));
        pause(1);
        parachuteDeployed = true;
        if simulate_landing == 0
            break;
        end
    end
    if (parachuteDeployed == true) && (states(i,3) <= 0)
        fprintf("Rocket has landed %0.0f meters away from launch site at
%0.1f m/s\n",norm(states(i,1:2)),states(i,6));
        break
    end
fprintf("time = %0.1f\n",t2);
end
%% Animate the resulting state array
animate_rocket(t,states,rocket,50,plot_type);
```

```
fprintf("Animation Complete \n");
```

Appendix D: Arduino ® Programming for Flight Controller

```
//
//
        //  High-Powered Rocket MQP 2018 - 2019   //
        //  Test Launch flight controller         //
        //                                        //

        ////////// Include all necessary libraries //////////
        #include <Adafruit_MPL3115A2.h>
        #include <Adafruit_Sensor.h>
        #include <Adafruit_BNO055.h>
        #include <utility/imumaths.h>
        #include <Wire.h>
        #include <SPI.h>    // SPI communication for SD card
        #include <SD.h>     // SD card library
        #include "TMRpcm.h" // Music player library


        #define BNO055_SAMPLERATE_DELAY_MS (100) // Time (ms) delay between loop iterations

        ////////// Define all digital pins here //////////
        const int SDpin        = 30; // Pin for SD card (change this to whatever it is set
        to)
        const int ejectionPin  = 33; // Pin number for the ejection charge pin
        const int RFpin        = 32; // Pin number for the RF transmitter data out

        ////////// Define the Strain Gague Pins here //////////
        const int strain1      = A0;
        const int strain2      = A2;
        const int strain3      = A4;

        ////////// Definable Variables Critical to Mission //////////
        unsigned long emergencyEjectionTime                  = 11000; // Milliseconds after
        launch where the ejection charge will go off regardless
        unsigned long minimumEjectionTime                    = 6000;  // Minimum time of
        ejection
        double        launchDetectHeight                     = 5;     // Height where
        launch is detected (meters)
        double        linearAccelerationLaunchDetectThreshold = 10;    // Linear
        acceleration threshold (ms^-2) to help detect launch
        double        apogeeAltDifferenceThreshold           = 1.5;   // Altitude
        difference for apogee detection (m)
```

```
////////// Define variables here //////////
String        dataString;                          // String of data which is written
to SD Card
unsigned long adjustedTime;                        // Time read by arduino. "Resets" to
0 at launch using timeDifference variable
unsigned long timeDifference        = 0;         // Difference in time which "resets"
time to 0 at launch
double        linearAccelerationX   = 0;         // Z component of linear
acceleration

////////// logic variables describing events during flight //////////
boolean launch    = false;
boolean apogee    = false;
boolean mainChute = false;

////////// Variables read from sensors //////////
float altDifference;  // Subtraction number for altitude creep
float correctedAlt;   //
float measuredAlt;    // Current altitude
float lastAlt;        // Previous altitude
float groundAlt = 0;  // Altitude measured at the ground

////////// Define the sensors here //////////
Adafruit_MPL3115A2 altimeter = Adafruit_MPL3115A2();
Adafruit_BNO055    IMU       = Adafruit_BNO055();
TMRpcm tmrpcm; // Music player object
/*-------------------------------------------------------------------------------------
------------------------------------------------
 *
 *                                                         SETUP
 *
 *-------------------------------------------------------------------------------------
------------------------------------------------*/
void setup() {
  Serial.begin(9600);
// Begins Sensors
  altimeter.begin();
  IMU.begin();
  SD.begin(SDpin);
  Serial.println("Sensor and SD card driver setup complete");

  pinMode(ejectionPin, OUTPUT);      // Sets the ejection pin to be an output
  pinMode(RFpin, OUTPUT);            // Sets the RF pin to be an output

  IMU.setExtCrystalUse(true);        // Use external crystal for better accuracy
  Serial.println("Acquiring Ground Altitude...");
  altDifference = altimeter.getAltitude();       // Uses an average of 1000 readings
to find the ground altitude
  lastAlt       = altDifference;                 // Sets the "last" altitude to the
ground altitude
  Serial.println("Setup Complete");
}
/*-----------------------------------------------------------------------------------
------------------------------------------------
 *
 *                                                           LOOP
 *
 *----------------------------------------------------------------------------------
------------------------------------------------*/
void loop() {
  ////////// Get a new sensor event for the IMU //////////
  sensors_event_t event;
  IMU.getEvent(&event);

  ////////////////////////////////////// Read data from sensors
/////////////////////////////////////////////////////
  float measuredAlt          = altimeter.getAltitude(); // Function from altimeter
library which aquires current altitude in meters

  imu::Quaternion quat       = IMU.getQuat();
  imu::Vector<3> accel       = IMU.getVector(Adafruit_BNO055::VECTOR_ACCELEROMETER);
  imu::Vector<3> mag         = IMU.getVector(Adafruit_BNO055::VECTOR_MAGNETOMETER);
  imu::Vector<3> gyro        = IMU.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
  imu::Vector<3> linearAccel = IMU.getVector(Adafruit_BNO055::VECTOR_LINEARACCEL);
  imu::Vector<3> gravity     = IMU.getVector(Adafruit_BNO055::VECTOR_GRAVITY);

  ////////// Aquire Calibration States //////////
  uint8_t system, gyroCal, accelCal, magCal = 0;
  IMU.getCalibration(&system, &gyroCal, &accelCal, &magCal);

  ////////// Create the data string //////////
  String   dataString  = "Launch: " + (String)launch+" Apogee: " + (String)apogee+"
Main Chute: "+(String)mainChute+"\t";                             // Event
Booleans
```

```
            dataString += "Time: " + (String)((double)adjustedTime/1000)+"  \t";
// Current Time
            dataString += "Corrected alt (m): "          +(String)correctedAlt+"
\t";                                                                    // Current
Corrected altitude
            dataString += "Cal: Sys: " + (String)system+ " Gyro: " +
(String)gyroCal+" Accel: " + (String)accelCal+ " Mag: " + (String)magCal+"  \t";
// Calibration values
            dataString += "Quat: "                +(String)quat.w() +"
"+(String)quat.x() +" "+(String)quat.y() +" "+(String)quat.z() +"  \t";
// Quaternions
            dataString += "LinAccel: "            +(String)linearAccel.x()+"
"+(String)linearAccel.y()+" "+(String)linearAccel.z()+"  \t";                //
Linear Acceleration
            dataString += "Accel: "               +(String)accel.x()+"
"+(String)accel.y()+" "+(String)accel.z()+"  \t";
// Accelerometer
            dataString += "Mag: "                 +(String)mag.x()  +"
"+(String)mag.y()  +" "+(String)mag.z()  +"  \t";
// Magnetometer
            dataString += "Gyro: "                +(String)gyro.x() +"
"+(String)gyro.y() +" "+(String)gyro.z() +"   \t";
// Gyroscopic sensor
            dataString += "Measured Alt: "        +(String)measuredAlt+"   \t";
            dataString += "Strain Gague: "                +
(String)analogRead(strain1)+"\t"+(String)analogRead(strain2)+"\t"+(String)analogRead
(strain3);  // Strain gague readings

  ////////// Write the data string to the SD card //////////
  writeToSD(dataString);

  ////////// Adjust the time and altitude for when rocket is sitting on launch pad
//////////
  correctedAlt = measuredAlt - altDifference; // Every loop this subtracts the
measured altitude from the altimeter by the difference variable

  ////////// Set launch event triggers according to states measured from sensors
//////////
  linearAccelerationX = (double)linearAccel.x();
  if (launch == false){
  launch = detectLaunch(measuredAlt, groundAlt, correctedAlt, launchDetectHeight,
linearAccelerationX);
  }
  adjustedTime = millis() - timeDifference;   // Every loop this subtracts millis()
by the time difference variable
  apogee = detectApogee(measuredAlt, lastAlt, adjustedTime, launch);

  //If apogee is acheived, deploy the main chute

  if (apogee == true) {
    mainChute = true;
    deployMainChute();
  }
  if(mainChute == true){
    deployRF();
    deployMainChute();
  }
  ////////// Write everything to the SD card, and set the current height measurement
to the "last" measurement
  lastAlt = correctedAlt; // Sets the "memory" altitude
delay(BNO055_SAMPLERATE_DELAY_MS);
}

/*---------------------------------------------------------------------------------
-------------------------------------------------
 *
 *                                                         FUNCTIONS
 *
 *---------------------------------------------------------------------------------
-------------------------------------------------*/
////////// Function which detects whether launch happened and sets isLaunch
accordingly //////////
boolean detectLaunch(float measuredAlt,float groundAlt, float correctedAlt, float
launchDetectHeight, double linearAccelerationX){
  if((linearAccelerationX < linearAccelerationLaunchDetectThreshold) && (launch ==
false)){
    altDifference = measuredAlt;
    groundAlt     = correctedAlt;
  }
  if(((correctedAlt - groundAlt) > launchDetectHeight)) {
    timeDifference = millis();  // Sets the time difference to current time -
"resets" the clock to 0sec at time of launch
    return true;
  }
  return false;
```

```
}

////////// Detects apogee and sets isApogee accordingly //////////
boolean detectApogee(float measuredAlt, float lastAlt, unsigned long adjustedTime,
boolean launch) {
  // If launch has occurred and the main chute hasn't deployed, start searching
whether the altitude is beginning to decrease
  if(launch == true) {
    if ((adjustedTime > minimumEjectionTime) && (((lastAlt - correctedAlt) >
apogeeAltDifferenceThreshold) || (adjustedTime > emergencyEjectionTime))) {
      return true;
    }
  }
  return false;
}

////////// Function for all events necessary to deploy the parachute //////////
void deployMainChute(){
  // This is where we code whatever the servo motor does to deploy the main
parachute
  // add printout for calling this
  digitalWrite(ejectionPin,HIGH);
  digitalWrite(13,HIGH);
}

////////// Function which turns the RF transmitter on //////////
void deployRF(){
  // Code for activating the RF transmitter goes here
  int delayTime = 350 + (int)(correctedAlt*0.25);

  tone(RFpin, 691/2,delayTime);
  delay(delayTime+5);

  tone(RFpin, 691/2,delayTime);
  delay(delayTime+5);

  tone(RFpin, 750/2,delayTime);
  delay(delayTime+5);

  tone(RFpin, 800/2,delayTime);
  delay(delayTime+5);
}
```

```
////////// Function writes all data to the SD card when called //////////
void writeToSD(String stringToWrite){
  // open the file. note that only one file can be open at a time,
  // so you have to close this one before opening another.
  File dataFile = SD.open("tstlnch.txt", FILE_WRITE);
  // if the file is available, write to it:
  if (dataFile) {
    Serial.println(stringToWrite);
    dataFile.println(stringToWrite);
    dataFile.close();
  }
  // if the file isn't open, pop up an error:
  else {
    Serial.println("error opening datalog.txt");
  }
}
```