CS CAPSTONE SPRING FINAL REPORT

JUNE 13, 2018

30K ROCKET SPACEPORT AMERICA

PREPARED BY

JOSHUA NOVAK _____ Signature _____ Date

ALLISON SLADEK _____ Signature _____ Date

LEVI WILLMETH _____ Signature _____ Date

REVISION HISTORY

Name	Date	Reason For Changes	Version
Levi Willmeth, Joshua Novak, Allison Sladek	June 12, 2018	Initial document draft	1.0

Abstract

Final project report covering the design,technical challenges and solutions for the computer science portions of the Oregon State University (OSU) team's entry in the Spaceport America Cup 30k competition during the summer of 2018. The competition involves designing, building, and launching a student-made rocket to 30,000 feet.

CONTENTS

1 Introduction and Project Overview

This computer science capstone project designed and built software for the Spaceport America Cup 30k competition in the summer of 2018. The annual competition involves designing, building, and launching a student-made rocket to 30,000 feet. This year's Oregon State University team included 20 students from several disciplines including mechanical and electrical engineering, as well as computer science. This report contains the goals and progress of the computer science subteam.

1.1 Client Information and Role

The clients for this project are Professor Nancy Squires, and to some degree the entire OSU ESRA 30k team. Our client worked with us to set our initial goals, with a long term goal of achieving a safe and successful test launch in the spring, and competition launch at Spaceport America in the summer of 2018. Squires continued to work with us throughout the nine month project, to ensure that our progress stayed on track and adapted to the changing needs of the team. Her knowledge of the process and physics involved in launching rockets was very helpful, especially as we formulated our expectations of forces the rocket would experience at different stages of flight. In this way, Squires served a very helpful role as a supervisor, without contributing to the software development of the project.

1.2 Subteam Members and Roles

The computer science subteam included three computer science seniors named Levi Willmeth, Joshua Novak, and Allison Sladek. Three very different students who came together at the beginning of this project and learned to work together, draw from each others strengths, support each other's weaknesses, and ultimately build something together that none of us could have achieved on our own.

Levi Willmeth took on the development of the telemetry parser, using Python and Direwolf to translate audio signals into validated text. He also designed and implemented the SQL schema, wrote the ground station display using Curses, and was the primary developer of the flight avionics used on both the rocket and scientific payload, including the algorithms for calculating and controlling speeds for both motors on the payload. Levi worked with Joshua to write importable modules for each of the flight sensors.

Joshua Novak was the primary developer of our unit tests with nearly 90% code coverage, developed the live JavaScript graphs for the website, wrote calibration scripts for each of the hardware sensors, and wrote the R scripts to find outliers in our flight data, which could have helped detect any avionics problems between our test and competition launches.

Allison Sladek was the primary website developer, setting up our NodeJS site using handlebars, and integrating Levi's SQL queries together with Joshua's live graphs. She also developed the apogee detection algorithm and our networking configuration, as well as contributing heavily to the algorithm used by the avionics state machine.

1.3 Motivations and Overall Goals

The Spaceport America competition includes a scoring metric which rewards teams that provide avionics to control the rocket, receive live telemetry on the ground during the flight, and can visualize the results of an onboard scientific payload after the flight. This project's task was to write the software necessary to accomplish all of those goals.

This was the second year that a computer science team has participated in this competition, and we understood that one of our primary goals was to improve the quality and most importantly, reliability of the software components used

during the competition. Previous years have struggled to create reliable avionics and ground station software, partially due to being busy with other capstone goals such as developing electronic or mechanical components. By including a computer science subteam with the primary purpose of writing software, the entire team benefits in terms of software reliability, performance, and the ability to be more ambitious with our payload goals.

Therefor, our motivation was to produce a safe, reliable, and valuable software application that met the needs of our sponsor, performed well and earned a high score during the Spaceport America Cup competition, and ultimately justified the value of a computer science subteam on next year's rocket.

2 Technical Requirements Document (Original)

2.1 Project Overview

2.1.1 Introduction

The Spaceport America Cup is an international engineering competition to design, build, and fly a student-made rocket to 30,000 feet. The competition is scored on several criteria including software components like flight avionics, recording and displaying telemetry, and later displaying the results from an scientific payload.

2.1.2 Purpose

This document outlines the software requirements for the Spaceport America Cup 30k rocket competition in 2018.

2.1.3 Scope

The software described by this document will support the Oregon State University (OSU) American Institute of Aeronautics and Astronautics (AIAA) team's entry for the Spaceport America Cup 30k competition in the summer of 2018.

2.1.4 Definitions, acronyms, and abbreviations

AIAA	American Institute of Aeronautics and Astronautics
CS	Computer Science
ECE	Electrical and Computer Engineering
GUI	Graphical User Interface
GPIO	General Purpose Input Output pin
GPS	Global Positioning System
OSU	Oregon State University
PCB	Printed Circuit Board
SD card	Secure Data card
TBD	To Be Determined

2.1.5 References

Initial Project Description

http://eecs.oregonstate.edu/capstone/cs/capstone.cgi?project=340

Client Requirements Document assignment

http://eecs.oregonstate.edu/capstone/cs/capstone.cgi?hw=reqs

CS capstone group final Problem Statement

https://github.com/OregonStateRocketry/30k2018-CS-Capstone

2.1.6 Overview

This document contains the complete software requirements and specifications for the computer science portion of the Spaceport America Cup 30k rocket competition. The software can be organized into four parts:

1) Ground station - **Parser program**, takes an audio source from a radio and stores APRS fields in a database.
2) Ground Station - **Display program**, reads from the database and displays information to multiple users.
3) **Rocket Avionics**, reads and logs onboard sensors, triggers flight events on the rocket.
4) **Payload Avionics**, reads and logs onboard sensors, triggers flight events on the payload.

This document outlines the requirements for each of these programs organized as sets of user stories, characteristics, specific product functions, constraints, and early assumptions about program inputs and outputs. This document may be updated as the requirements change throughout the project.

2.2 Overall description

2.2.1 Product perspective

The Parser program will receive data from a file, or an external source (telemetry from the rocket) in the form of a radio signal. The specific data format will be agreed on by both the CS and ECE teams. The Parser program will run on a ground station and parse out the individual fields, then store them in a database. The database may be hosted on the same computer, or another computer on the same network.

The Display program will run on a client computer and graph or otherwise display telemetry or payload data from the database.

The Rocket Avionics program will use onboard sensors to detect and trigger the separation of the rocket near the apogee of flight, as well as deploy the main parachute closer to the ground. The optimal conditions for separation and deployment will be determined by team discussion and recommendations from experienced mentors.

The Payload Avionics program will use onboard sensors to signal a propeller to push the payload down in order to create a low gravity environment on the payload. It will also trigger deployment of a parachute before impacting the ground. The optimal conditions to deploy the parachute will be determined by team discussion.

2.2.1.1 System interfaces:

The Rocket Avionics software will interact with the rocket through the PCB designed by the ECE team.

The Payload Avionics software will interact with the payload through the PCB designed by the ECE team.

The Parser program will interact with an input source and database.

The Display program will interact with the same network and database as the Parser program.

2.2.1.2 User interfaces:

The Rocket and Payload Avionics software will allow for the use of an LED to indicate status.

The Parser program will provide a text interface that can be used for general debugging purposes.

The Display program will provide a graphical user interface allowing users to view information stored in the database. This will allow users to see active flights or view previously recorded flights or test data. They will also be able to view data recorded on the Rocket or Payload Avionics programs.

2.2.1.3 Hardware interfaces:

The Parser program will interact with a radio device to collect telemetry data. It may be necessary to run the Parser

program on multiple computers to interpret data from multiple audio sources, while avoiding interference between audio sources.

The Rocket Avionics software will interact with the rocket through a PCB designed by the ECE team, which will provide sensors that will probably include at least one accelerometer, gyroscope, and barometer, as well as multiple explosive charges to separate the rocket, and deploy drogue and main parachutes.

The Payload Avionics software will interact with the rocket through a PCB designed by the ECE team, which will provide sensors that will probably include at least one accelerometer, gyroscope, and barometer, a host of sensors for the zero-g experiments conducted on board, and at least one main parachute and a backup.

2.2.1.4 Software interfaces:

External libraries or software may be used to create graphs from information in the database.

The parsing program will read and write to a local database.

(stretch goal) Other computers or devices will be allowed to connect to the computer running the Parsing program, to view telemetry data and payload graphs via intranet connection. This will require configuring WiFi networks and generating dynamic graphs from the database.

2.2.1.5 Communication interfaces:

Telemetry for the rocket will be transmitted as a radio signal. The specific format and fields of the data will be determined by both the CS and ECE teams, and will at least include GPS, altitude, and timestamp fields.

2.2.1.6 Memory constraints:

Avionics software will need to execute on a computer system to be determined by the ECE team, which is likely to be a raspberry pi zero. All other software can be run on laptops with varied memory capacity.

2.2.1.7 Operations:

The Parsing program will offer a Monitor mode to listen for incoming data packets on the audio port.

The Parsing program will offer an Import mode to import formatted data from a file.

The Rocket and Payload Avionics programs may offer different flight modes and behave like state machines, depending on how the project develops and based on advice from industry mentors.

The Display program may offer a Monitor mode to view active flight data, if available. Active flight data will be indicated by data with timestamps within some number of minutes of the current time.

The Display program may offer a Review mode to view data from previously recorded or test flights.

2.2.2 Product functions

2.2.2.1 Parser program functions:

Import a data file from hard drive or SD card.

Listen to and parse data from a radio source. This will probably include parsing an audio input.

Validate the incoming data against an expected format.

Split the parsed data into individual fields and write them to a database.

2.2.2.2 Display program functions:

Read selected fields from a database.

Generate a display that may include graphs, charts, or text areas to summarize information.

2.2.2.3 Rocket Avionics functions:

Read data from individual sensors.

Store sensor data on a SD card.

Toggle a GPIO pin to trigger rocket separation.

Toggle a GPIO pin to trigger payload ejection.

Toggle a GPIO pin to deploy drogue parachute (if needed).

Toggle a GPIO pin to deploy main parachute (if needed).

2.2.2.4 Payload Avionics functions:

Read data from individual sensors.

Store sensor data on a SD card.

Measure current acceleration and calculate expected thrust to reduce acceleration.

Update a GPIO pin to control propeller speed.

Toggle a GPIO pin to deploy main parachute.

2.2.3 User characteristics

The intended users of this software will consist primarily of other team members and mentors. They will have technical knowledge related to operation and construction of the rocket and its sensors. These users are all seniors in college or beyond. The nature of the graphical data will be technical. The software team will also be present at launch to assist in the use of the software.

2.2.4 Constraints

2.2.4.1 Regulatory policies:

The project may be subject to competition restrictions that have not yet been published.

The radio transmissions must comply with Federal Communications Commission regulations.

2.2.4.2 Hardware limitations (e.g., signal timing requirements):

The software must comply with the requirements of the computer hardware selected by the ECE team.

The Parsing program requires an external radio to receive radio signals.

The Parsing program may require a network connection to any connected clients.

2.2.4.3 Interfaces to other applications:

The Parsing program requires an external data source such as a formatted audio source or file, and will write to a database on the same local network.

The Display program will read from the same database on the same local network.

2.2.4.4 Parallel operation:

Multiple copies of the Display and Parsing programs can run simultaneously on different machines, without interfering with each other.

2.2.4.5 Audit functions:

The Rocket and Payload Avionics programs will include a test suite covering at least 80% of the CS teams lines of code. The CS team cannot guarantee code coverage for software written by other groups.

The Parser program will include a test suite covering at least 80% lines of code written by the CS team. The test inputs may be in the form of files and/or audio sources.

2.2.4.6 Control functions:

Control functions are limited by the ECE and rocket hardware.

2.2.4.7 Higher-order language requirements:

The Parser and Display program may be written in a high level language.

The Rocket and Payload Avionics programs may be written in C or a higher level language. (TBD)

2.2.4.8 Signal handshake protocols (e.g., XON-XOFF, ACK-NACK):

The Parser or Display programs may or may not include authentication or signal handshakes. (TBD)

2.2.4.9 Reliability requirements:

The Parser program must reliably receive and store GPS data during flight.

The Display program must reliably compute and display data from the database.

The Rocket Avionics program must perform flight actions reliably and under the correct conditions, which will be determined over the course of the project. Expected flight actions include recording sensor data throughout the flight, detecting launch, detecting apogee, triggering separation, triggering main parachute deployment, and concluding the flight.

The Payload Avionics program must reliably and accurately meet the requirements of the experiment, which are TBD. Expected requirements include recording sensor data throughout the flight, detecting separation, controlling a propeller to reach a state of neutral gravity, triggering main parachute deployment, and concluding the flight.

2.2.4.10 Criticality of the application:

Displaying recent GPS data will be mission critical for recovery of the rocket. Interpretation and display of other data, including telemetry and payload is less critical, but still very important to the success of the launch. Separation at the right time is critical for recovery of the rocket. Triggering the main parachutes on both the rocket and payload are critical for the project.

2.2.4.11 Safety and security considerations:

Preventing accidental separation of the rocket is critical for human safety.

Preventing accidental deployment of main parachutes on both the rocket and payload are critical for human safety.

2.2.5 Assumptions and dependencies

The rocket will launch far enough away from spectators not to pose a safety threat.

The ECE team will provide the computer hardware necessary to run the Rocket and Payload Avionics programs, allowing us to record data for use by all programs.

The ECE team will create radio signals containing accurate and well-formatted telemetry data from both the rocket and payload components, which can be received by the Parser program.

2.3 Specific requirements

2.3.1 External interface requirements

The Parser program will need to be connected to an input source capable of receiving a properly formatted radio signal. Formatting may include audio signals in APRS format, as a serial string, or some other as-yet unknown but acceptable format.

2.3.1.1 User interfaces:

The Parser and Display programs will include either text-based or graphical user interfaces.

The Rocket and Payload Avionics programs may or may not include a user interface. It is possible the ECE team will provide one or more LEDs to indicate status.

2.3.1.2 Hardware interfaces:

The Rocket and Payload Avionics programs will interface with custom PCBs designed and manufactured by the ECE team.

2.3.1.3 Software interfaces:

The Parser or Display programs will be able to load a data file from an SD card.

2.3.1.4 Communications interfaces:

The Parser program will interface with the rocket through radio telemetry, and possibly to a database using WiFi.

The Display program will interface to a database using WiFi.

2.3.2 System features

2.3.2.1 Display program features:

The Display program will offer a GUI to display telemetry or recorded data as graphs, tables, maps, or individual fields. At least eight out of every ten intended users able to determine what telemetry information has been received.

The Display program will read information from a database. The database will be populated by the Parser program.

The user should be able to open the Display program and select either an ongoing launch or a previous launch to display. From there, they should view the most relevant data, including the location or path of the rocket displayed on a map, and a graph of altitude vs. time. The map will feature a small marker for each data point. The user will also be able to view other information including graphs of pressure, temperature, acceleration, or velocity vs. time, assuming those fields are available in the database.

If viewing data during a live flight, the Display program will update within five seconds of new information being received. The status of the flight, stating whether the rocket is still climbing, the payload has been dropped, or similar information, will be displayed within five seconds of change, assuming that information is available in the database.

Graphs generated by the GUI may or may not dynamically change their bounds to best fit the data. The location may require downloading an appropriate map of the area before the flight.

2.3.2.2 Parser program features:

The Parser program will interpret information received via radio signal, or from an input file. It will decode the audio source based on the transmitter selected by the ECE team. The decoded string will be split into individual fields based on the data they contain, and inserted into a database.

The Parser program will also verify the formatting of the message. If the message appears to be invalid, corrupted, or not possessing the required fields, it will be discarded.

The Parser program will include a test suite with at least 80% line coverage for the code the CS team writes. If the program uses an external program to translate audio signals to text, that program may or may not be tested.

The Parser program will write data to the database within five seconds of receiving a packet.

2.3.2.3 Avionics features:

The Avionics programs will be written depending on the needs of other teams working on the rocket.

They will likely be written in C or a higher level language, but this is dependent on the processor and sensors chosen by the ECE team.

Both the CS and ECE teams expect to contribute to the Avionics programs, although the details are still TBD.

The Avionics programs will be thoroughly tested. The test suite will have at least 80% of the lines written by the CS team. The test suites may execute using a set of input files.

2.3.3 Performance requirements

2.3.3.1 Avionics program requirements:

The Rocket and Payload Avionics programs will not require real-time accuracy. Mission critical tasks can be accomplished within several hundred milliseconds without creating risk. The program will perform any and all mission-critical tasks first, with a secondary priority of recording sensor data. Sensor data may be delayed while processing higher priority tasks.

The Payload Avionics will attempt to reach a state of null gravity by engaging a motor and propeller. It is recognized that the payload is likely to record some acceleration. Achieving null gravity is the goal, but that is not a requirement.

2.3.3.2 Parser program requirements:

The Parser program should process incoming radio packets within 5 seconds of receiving them. The CS team expects that a packet should be delivered roughly once per second.

2.3.3.3 Display program requirements:

If the Display program is in the appropriate mode, it should display new information within 5 seconds of the Parser program writing to the database. This means there may be up to a 10 second delay between sending telemetry data, and seeing the results on the screen.

The Display program will generate graphs from the database. Outlier data will be identified using mathematical methods. Fit lines or other mathematically appropriate formatting may be generated based on the data. Graphs will depend on data received, but include at least altitude vs. time.

2.3.4 Design constraints

The Parsing program will run on a Linux base operating system, requiring relatively inexpensive and commonly available hardware.

None of the programs will require an active internet connection.

The Rocket and Payload Avionics programs will run on a processor to be determined by the ECE team.

2.3.5 Software system attributes

The parsing and payload programs will run on a Linux based operating system.

2.4 Changes

Only one major change was made to the requirements. We will not be responsible for triggering separation, deploying parachutes, or triggering any flight events not related to motor controls on the payload. A draft including these changes was sent to Nancy Squires and approved by her via email.

2.5 Final Gantt Chart

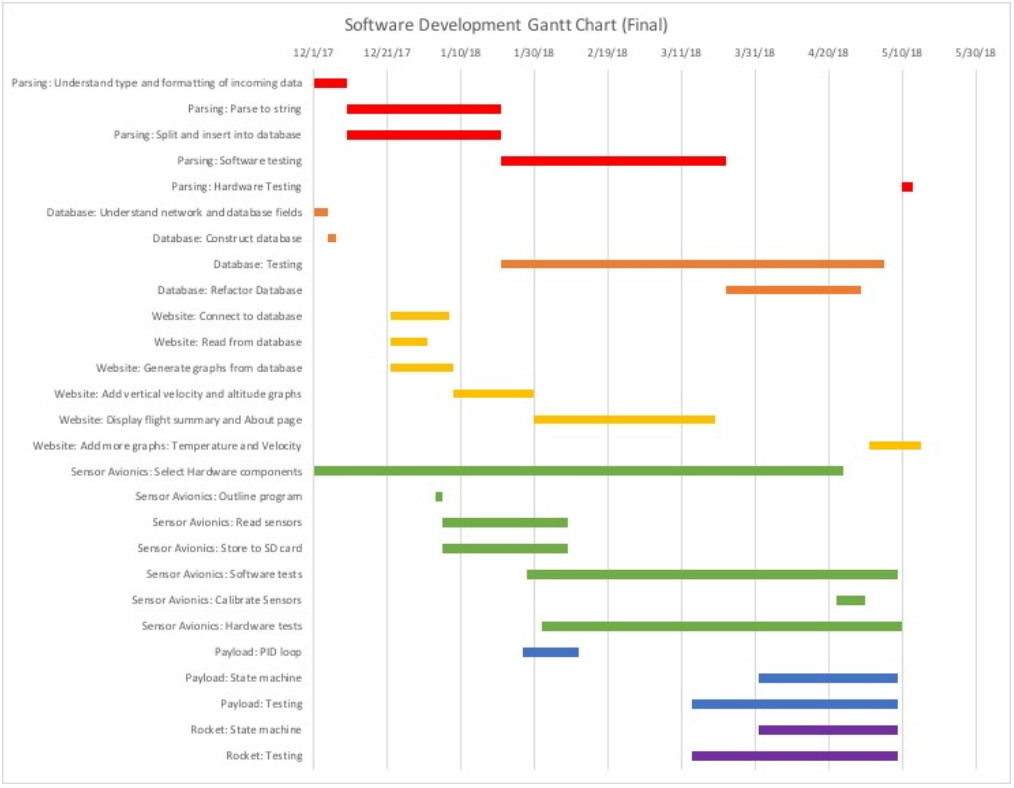


Fig. 1: Finalized Gantt chart, based on github commits

3 Proposed Design Document (Original)

3.1 Overview of design elements

This project requires that multiple pieces of software and hardware work together to accomplish our mission.

The rocket will carry at least two flight computers which will record on-board flight data, as well as multiple telemetry modules that will record and transmit flight data during flight. During the flight we will capture and display live telemetry data, which will be further improved after the flight by importing stored data from the flight computers and telemetry modules.

The on-board flight computers will also perform important tasks during the flight. The rocket avionics will use inertial measurement units and other sensors to detect apogee and trigger separation of the rocket, as well as deploy the drogue and main parachutes. The payload avionics will use accelerometers to measure gravity and engage a propeller to accelerate the payload downward, to create a micro gravity environment for the scientific experiment.

On the ground, a set of affordable computers will be connected to radios which will receive the live telemetry signals. These computers will parse out text from the incoming audio stream, and store that information over a local network and into a database.

Nearby team members and spectators will be encouraged to use their laptops, tablets and cell phones to connect to a WiFi network to access a web page containing a flight summary and graphs of acceleration, velocity, and altitude, among others. These graphs will be generated in nearly real time using information from the database.

After the rocket has landed and been recovered, we will use the parsers to import additional data from the on-board computers, into the database. This will provide a large amount of additional data and allow us to display additional graphs and information about the flight.

3.2 Avionics

3.2.1 Reading from Sensors

Joshua Novak will manage avionics code that handles reading from sensors.

There are a variety of sensors that will need to be read from for this project. These include accelerometers, altimeters, and GPS devices among other sensors. Avionics code will need to be written that reads from these sensors and can be called by functional avionics code for the rocket and paylod. The output of this code will need to represent the output of the sensor, interpreted according to the standards of the device.

The code will be written in C. This is because the libraries for most of the sensors will be in C, because the code will be running on a Raspberry Pi, and because the code will need to run as efficiently as possible. Libraries will be used when they are available and will not create conflict with the restrictions placed on the ECE team.

The code for reading from the sensors should be written as complete programs that can be called to target a particular sensor and which should be able to have their output piped into functional code for avionics. In some cases this will instead take the form of a program that reads all the inputs from an analog converter and feeds those into functional code, as there will be some analog sensors used as part of the project.

Alternatively, if this is found to have significant tradeoffs in terms of speed, memory, or stability, tests will be made using an object oriented solution in C++. If these tests find that there is no significant tradeoff in terms of speed, memory, or stability to use C++ with an object oriented solution, then we will go forward with C++. If the tradeoffs in terms of speed are too significant for both, we will look into different methods to improve encapsulation for testing purposes.

Some of the avionics for reading from sensors will be written by the ECE team, as there is a coding component to their capstone. Joshua will communicate with the ECE team to ensure that the code they write is compatible with that of the CS team, and will easily be able to feed into the functional avionics code. The CS team will also work alongside the ECE team to write this code, assisting them as necessary. Alternatively, the ECE team and CS team will write code separately, with CS avionics being used on the rocket and ECE avionics being written solely to satisfy their requirements.

Unit tests for the sensor avionics will be written alongside the code. Since Joshua is also responsible for testing avionics code, he will handle this as well, and ensure that any code written by the ECE team is thoroughly tested.

3.2.2 Rocket Avionics

The ECE team will be writing some of the avionics code, which will be coordinated between the two teams. Allison Sladek will manage writing the main rocket avionics on the CS side of this project.

The main rocket hardware components will include a Raspberry Pi Zero B, Stratologger, Beeline GPS, and redundant sensors, including at least a barometer, thermometer, gyroscope, accelerometer, and altimeter. Redundant sensors can be averaged to achieve a much more accurate reading than could otherwise be achieved with fewer sensors. This will allow better analysis of the data, and more accurate apogee detection.

Separation of the main rocket at apogee to deploy both the parachute and payload is a mission critical operation. If it is triggered at the wrong time, the rocket may not reach the goal altitude, or could be in the wrong position for a successful separation. As such, it will be controlled by the Stratologger, a commercial product that measures altitude,

temperature, and battery voltage, and activates separation when apogee is detected. The main purpose of the Raspberry Pi will then be to record data logs of the main rocket sensors for later analysis, and for battery control. It will also be used as a comparison to the Stratologger. This will allow testing of the Stratologgers accuracy, and ensure that deployment will function as expected after pre-flight testing.

3.2.3 Payload Avionics

Levi Willmeth will manage writing the payload avionics for this project.

The scientific payload will carry a brushless motor and ESC, several IMU's, a Raspberry Pi Zero flight computer, a video camera, and a BeeLineGPS telemetry module. The flight computer will be turned on several minutes before launch and record up-close video of the scientific experiment throughout the flight. It will also read acceleration and attitude data from each of the IMU's to detect separation and measure the amount of gravity experienced by the experiment. After separation has been detected, the flight computer will begin increasing the speed of the motor to propel the payload downward, which will reduce acceleration felt by the experiment. The goal of the experiment is to achieve zero gravity for as long as possible, which we estimate will be 10-12 seconds.

Accurately detecting separation and ejection from the body of the rocket will be a critical task for the payload avionics. If the payload begins to spin the propeller before being ejected from the body of the rocket, it will immediately destroy the propeller and possibly damage other portions of the rocket. The motor is powerful enough that if held still while being powered, it could draw enough current to easily start a fire. It will be important to develop an accurate, not precise, method to determine when the rocket has separated and the payload has been ejected from the body of the rocket.

Because we are unable to add physical components like a manual switch or wire connecting the payload to the body of the rocket, we will need to rely on our sensors. The payload will carry multiple 6 degree of freedom IMU's which will give us acceleration and gyroscopic readings in the x, y, and z planes. We will be able to use these sensors to remain in an idle state during launch, glide, and finally to detect the rather large impulse forces created during separation and ejection.

We plan to modeling the avionics system as a state machine with pre-launch, launch, glide, separation, microgravity, and parachute states. This type of direct, forward progression allows us to design a state machine which prevents the payload avionics from inadvertently returning to a previous state, and gives us a solid idea of what behaviors to look for in order to transition to the next state.

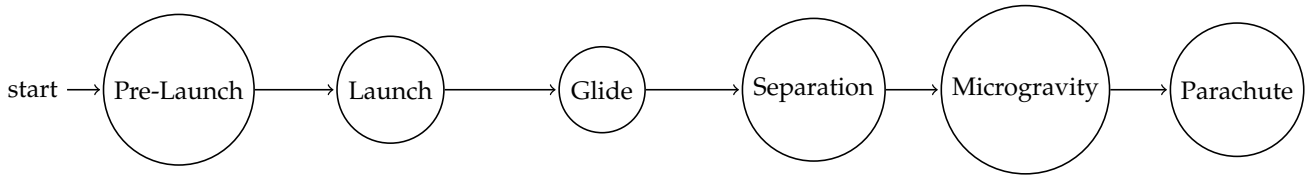


Fig. 2: Conceptual state diagram for payload avionics.

For example, during the pre-launch state we can effectively ignore all sensor readings until we see a very high acceleration on the z axis, which signals launch. We can begin logging all sensor values from that point on (or even retroactively record a set of rolling pre-launch values) until we see the z axis acceleration end, which signals that we have entered the glide state. Obviously we will need to look at more than a single sensor value to transition between states,

but the general idea is to understand what we can expect during each phase of launch, and encode those conditions as state transitions.

The major design concern for payload avionics are that accuracy is more important than precision. A false early start would risk damaging or completely destroying the rocket with the payload propellers, but a late start would only risk missing some or all of the experimental data.

3.2.4 Avionics Testing

The CS and ECE teams will be writing avionics for the rocket and payload which will be able to determine when separation should occur. Due to the high degree of precision that this requires and the significant negative consequences of failure for this code, ensuring the functionality of this code is of the utmost importance. It is also standard within the aerospace industry to very thoroughly test any and all avionics code, meaning that writing a thorough test suite will be good practice for working within the aerospace or related industries. There are three types of testing that the team has decided to carry out, hardware tests, unit tests, and simulation with robustness tests. The CS team will cooperate with the ECE team to perform the first and likely the second to some degree, and will manage the third.

Hardware tests will involve making certain that each sensor is functional, powered, and giving reasonable readings. This will largely be handled by the ECE team, but Joshua will take responsibility to ensure that the ECE team has thoroughly tested every piece of electronics on the Payload and the Rocket.

Unit tests will involve making sure that the basic functionality of the code is intact. This will mean ensuring that the programs and functions written to read from sensors give accurate output to a given input. These will mostly be used to test code reading from sensors, or that the code will trigger certain events when given input that should do so (such as triggering separation when inputs suggest the rocket has reached its apex). Some tests may be written to ensure a minimum degree of robustness, such as throwing out inputs that are invalid (such as an input that is a string of characters that should be a number). Joshua will oversee the writing of unit tests on avionics code. He will check to ensure that the goals for line coverage are met by the unit tests, and aim to have the unit tests ensure all code functions correctly to expected inputs.

The final aspect of testing the avionics code is to simulate a launch with robustness testing. This will involve encapsulating the functionality of the code for triggering flight events in some manner, and then testing that code against simulated inputs. These inputs will then be randomly altered in a variety of manners. If possible, this will be extended to include testing of code for sensor interaction, with the code interacting with simulated sensors rather than actual sensors. The randomization will likely take one of the following forms, but may include other forms if the code calls for it.

- Throwing a value that is out of bounds (this included the following)
  - A value that is of the wrong type
  - A value that is too large
  - A value that is too small
- Having a sensor cease responding
- Having a sensor continually send the same value
- Offsetting a sensors outputs by some set amount for the rest of the launch
- Having a sensor feed a value that is not out of bounds, but is not correct

- Having a sensor feed values for the rest of the launch that are incorrect and
  - in bounds
  - out of bounds
  - either in or out of bounds

The number of randomized failures will be able to be set at the start of the test, but not timing or the types of failures. Some failures may be weighted as being more significant, and therefore valued as a larger number of failures. The test will not be given a clear pass fail, but will instead push the outputs of the avionics code to a csv, which will be compared against the csv output of the code with no randomization as well as the inputs given by the simulation.

3.3 Telemetry and Parsing Flight Data

3.3.1 Transmitting Telemetry

The ECE subteam selected the BigRedBee BeeLineGPS telemetry module which cannot be used with additional sensors and only transmits using APRS formatting. This means that we cannot control which fields are being transmitted, or the packet formatting of the telemetry data. Each transmitter will send a packet once per second, containing the latitude, longitude, and altitude of the transmitter. These packets will be transmitted as audio data and can be decoded into a string in the APRS format.

There will be two telemetry transmitters in use during the flight. One will be in the nosecone of the rocket, and the other will separate with the payload. They will be on slightly different radio frequencies because the timing is not guaranteed, so the signals may overlap.

3.3.2 Receiving Telemetry

Levi Willmeth will manage receiving telemetry and parsing flight data for this project.

We will be using a software TNC called Direwolf to decode the audio tones into strings of text. Because the timing is not guaranteed and the signals may overlap, we will be using one primary receiver per radio signal, with an additional secondary receiver for redundancy. We expect to use four raspberry pi zeros to receive two incoming radio signals, to provide redundancy during signal processing.

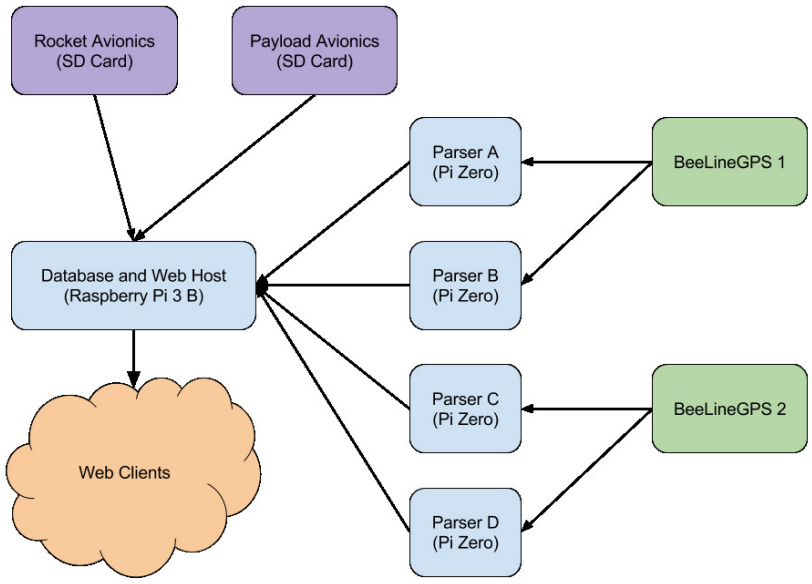


Fig. 3: Flow of information into the parser.

When the parsing computers boot up, they will immediately begin running the Direwolf software that allows them to receive an audio source and parse it into a string of text.[**?**] They will also begin running a Python script to take the input string from Direwolf, validate it for content, and insert individual fields into a database on the local network. The database will be described in section 3.4 of this document. These programs will be started as services so that they can be stopped, started, or automatically restart themselves as needed.[**?**]

```
2017-11-28  14:15:56UTC,ESRA-1,131812h3521.00N/07820.00W,496650,8642,446.875,PL
```

Fig. 4: Example APRS packet.

Each incoming radio packet may be from our rocket, or it may be from an unrelated transmitter on the same frequency. This is because we are using uncontrolled analog radio frequencies that may contain information from other, unknown users. We will use Python to parse the string, validate that it contains the correct radio call sign for our payload or rocket, and perform a checksum to test for corruption during transmittal.

If the packet does not pass checksum, contains an invalid call sign, or does not contain the correct data fields, a warning will be added to the parser log and the string will be inserted into a special database table for invalid inputs. This will allow us to inspect errors without discarding any data.

3.3.3 Parsing Data

The parsing program described in 3.3.2 will be configured to either process incoming strings from Direwolf, or to accept a well formatted file. This will allow the same parsing program that listens for radio inputs to parse the data recorded and stored on board the rocket and payload during flight.

After the flight, a human operator will be able to remove the SD card from the rocket and payload avionics flight computers, plug the SD card into a laptop, and point the parsing program at the SD card. The parsing program will

scan the directory (or individual file) and determine from the header which device created it. Using the appropriate table name and steps similar to those used while parsing radio inputs, the parsing program will split the strings into individual fields and insert them into the database over the network.

```
timestamp,acc_x1,acc_x2,acc_x3,acc_y1,acc_y2,acc_y3
2017-11-28 14:15:56UTC,27.2,56.7,32.2,234.2,235.7,463.4
2017-11-28 14:15:57UTC,27.1,56.7,32.2,234.2,235.7,463.4
2017-11-28 14:15:58UTC,27.2,56.8,32.2,234.2,235.7,463.4
```

Fig. 5: Example avionics output file.

3.4 Database

Levi Willmeth will set up and manage a MariaDB database for this project.

This project will collect several hundred thousand lines of sensor data, using several different sources. All of this data will need to be sortable and searchable in order to interpret the results of our flight. Furthermore, we want to be able to record and analyze different flights independently. Using a database to store our flight data will allow us to accomplish all of these goals.

Because the rocket avionics records different types of sensors than the payload avionics or the radio telemetry modules, we will use one table per type of data source. That means one table will hold all data received from a BeeLineGPS telemetry module, another table will hold all data imported from the payload avionics, and so on. These tables will also need to relate to each other in order to allow the display program to make appropriate connections between different sensors recorded on different hardware. This will be done using a Flight table with a primary key that will be used as a foreign key on each of the individual data tables.

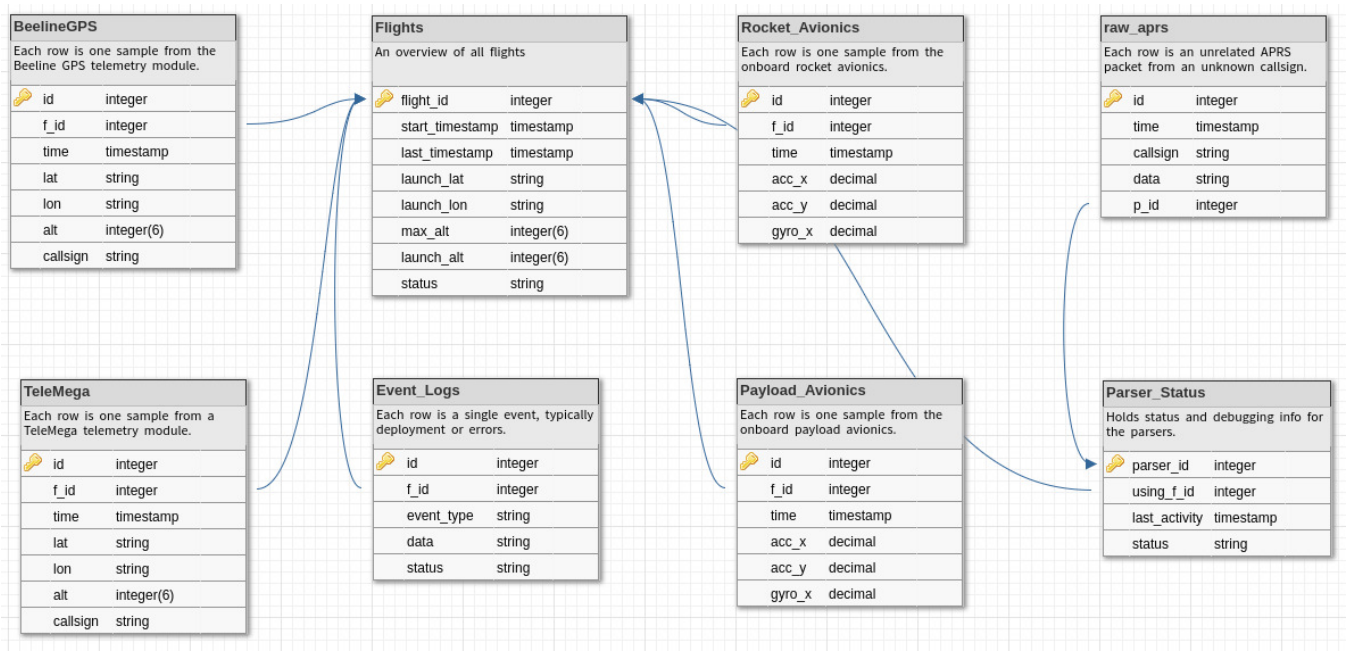


Fig. 6: Early draft of the database schema.

When the telemetry parsing Python script begins, it will check the Flights table of the database to see if there is a flight record with an Active status. This table and field will be used to synchronize the parsers so that records from one radio frequency can be related to records from another frequency at the same time. The Flights table has a primary key called 'flight_id' that is used as a foreign key on other related tables.

To eliminate duplicate rows caused by using redundant receivers, we will use a MySQL query that first checks if a row exists, before inserting a new row. This can be done as a single query: [**?**]

```
INSERT INTO
        BeelineGPS (f_id, time, lat, lon, alt, callsign)
SELECT
        * FROM (SELECT $cur_time, $cur_flight_id) AS tmp
WHERE NOT EXISTS (
    SELECT id FROM BeelineGPS WHERE time = $cur_time
) LIMIT 1;
```

There may be situations where a given timestamp contains data across multiple tables. For example, the rocket avionics may record some types of data at 100 Hz, and other types of data at only 10 Hz. Some of these records may share an identical timestamp, others may not. That is why we are using multiple tables and a foreign key instead of one larger table and assuming our timestamp is unique. While graphing this data, we are likely to want to relate data from different tables. For example if we recorded latitude and longitude in the BeelineGPS table, and acceleration in the Rocket_Avionics table, we may want to combine that data to see our acceleration at different points across a map. This is where the database becomes exceptionally useful compared to flat files. We can use a join query to select records from both tables, based on the knowledge that they will have identical f_id's and similar timestamps. The syntax would look something like this:

```
SELECT
        BeelineGPS.lat, BeelineGPS.lon, Rocket_Avionics.acc_x
FROM
        BeelineGPS
INNER JOIN
        Rocket_Avionics
ON
        BeelineGPS.f_id=Rocket_Avionics.f_id
WHERE
        BeelineGPS.flight_id=$cur_flight;
```

In this example the latitude may be sampled at only 1 Hz while acceleration could be sampled at closer to 100 Hz. We could choose to average out many acceleration values to find the best fit at the moment we recorded the latitude, or we could let MySQL find a single value using the Rocket_Avionics timestamp closest to the BeelineGPS timestamp. The syntax for that query could look something like this: [**?**]

```
SELECT
        BeelineGPS.lat, BeelineGPS.lon, Rocket_Avionics.acc_x
```

```
FROM
        BeelineGPS
INNER JOIN
        Rocket_Avionics
ON
        BeelineGPS.f_id=Rocket_Avionics.f_id
WHERE (
        SELECT
                timestamp
        FROM
                Rocket_Avionics
        WHERE
                BeelineGPS.timestamp >= (Rocket_Avionics.timestamp-100)
        AND
                BeelineGPS.timestamp <= (Rocket_Avionics.timestamp+100)
        ORDER BY
                abs(BeelineGPS.timestamp - Rocket_Avionics.timestamp)
        LIMIT 1
);
```

3.5 Display Components

3.6 Networking

The ground station will consist of a Raspberry Pi 3, several radio receivers, several Raspberry Pi Zeros, and a battery. The Raspberry Pi zeros will parse data received through the radios before sending it to the database and server hosted on the Raspberry Pi 3. The server will allow users to connect through an ad hoc network and display the graphical flight data on mobile devices.

USB OTG (On-The-Go) will be used to connect the four Pi zeros to the Pi three, which can then assign them IP addresses and power the other Pis. Setting up this connection is a relatively straightforward process that involves editing configuration files on the zeros to give them static IP addresses, and configuring the host Pi to forward network information and assign static IP addresses to each wired connection. [**?**]

The connected Pis will also serve the graphed flight data to mobile devices around the ground station. This will be accomplished through an ad hoc network. The Pi3 has an integrated wifi module and changing the interface configuration file at /etc/network/interfaces will allow the the Pi to operate in ad hoc mode. [**?**]

3.6.1 Web hosting

The dynamic graphing display will be hosted by an Apache server hosted on a networked Raspberry Pi 3. Apache's CGI module, the common gateway interface, can be used to serve dynamic content.[**?**] The flight data graphs generated by the dynamic graphing program will be served as HTML pages to connected mobile devices.

3.6.2 Dynamic graphing

Dynamic Graphing will be managed by Joshua Novak.

The dynamic graphs will be written in JavaScript, using the non-commercial version of CanvasJS. The graphs will be able to be displayed in an HTML page, and will update through queries to a database. These updates will be once a second, though new data may not be displayed if it is not available.

At least the following charts will be displayed in the form of dynamic graphs

- Location of the rocket on a map
- Altitude of the rocket
- Vertical velocity of the rocket
- Orientation of the rocket

The following will be displayed for launches where such data is available

- Temperture
- Velocity
- Acceleration
- Spin
- Barrometric pressure

The first kind of graph that will be displayed is the map. The map will be generated with basic JavaScript. An image of a map around the launch site will be used, with javascript overlaying dots representing the recorded locations of the rocket. The dots will have an on hover effect to display the GPS coordinates of that point, and the timestamp associated with it. Some additional effects may be added to make the graph more readable, such as gradually changing the hue of dots over time, so that newer points appear different from older ones. This will only be used to show the location of the rocket on a map. The points at which particular flight events occur may be represented with a different image, such as a broken circle for separation, or a half circle once the parachute is deployed.

The second kind of graph that will be used is an orientation graph. This will show the relative orientation of the rocket against a compass. The graph will be circular with an arrow overlayed on the graph. The point will be directed toward the bearing of the rocket relative to absolute north. This graph may use CanvasJS or other libararies, but should be able to be easily generated using basic JavaScript functionality. On hover, the graph will display a numerical value for the bearing. This graph will only be used to show the orientation of the rocket.

The final kind of graph that will be used is a basic value vs time graph. This graph will likely be generated using CanvasJS. It will display a maximum number of points relative to the size of the window, so that the graph scales well to different display sizes. These points will be the last several datapoints for the flight. The time will be determined by the timestamp. The relevant value to be graphed opposing time will generally be taken directly from the database query where it is available, and calculated from other data when it is not avaiable but can be calculated with a reasonable degree of accuracy. This will likely be used for all other fields.

3.7 Changes

The following changes to the original design document were approved by Nancy Squires.

3.7.1 Rocket Avionics

Due to hardware changes outside of our control, we needed to update the original design document to reflect that our flight computer was no longer connected to the parachute and recovery hardware systems. This change came from a competition rule change that required only using commercial off the shelf hardware for deployment and recovery systems.

3.7.2 Sensor Avionics

Code for sensors (and all other avionics code) was written in Python instead of C or C++. This was because all of our flight sensors share a single I2C bus, and initial tests showed that 95% of our program time was spent waiting for the hardware bus. Because the flight sensors were outside of our control, we could not fix this bottleneck, and we decided to use Python for ease of development given the extremely marginal performance difference.

All avionics code, including sensor avionics, was written by the CS team with limited to no input from the ECE team, though this was left open as a possibility in the original draft.

Finally, the final selection of flight sensors was outside of our control, and the only redundant sensors included were three inertial measurement units.

3.7.3 Avionics Testing

Our method for robustness testing was ultimately re-described as simulation with randomized error, as this is a more accurate description. While our implementation fits in the bounds of the description, and thus no edits were made, it is somewhat limited and the randomization mostly throws random in bounds failures. This is because input is interpreted from a stream of bytes, and thus incorrect typing is impossible, and almost any input for that stream of bytes would be considered in bounds for the particular sensor.

3.7.4 Networking

The parsers and other computers were connected via WiFi instead of USB-OTG, because the Raspberry Pi Zero only has a single USB port which was needed for the external audio card.

3.7.5 Display

It was decided that graphing the orientation of the rocket would not be particularly interesting or useful due to an expected rotational speed of around 2 Hz, and difficulties in finding a frame of reference. We also added the use of analysis in R to find outliers as well as its summary function, which can be used to find best fit lines.

4 Technology Review Document (Levi)

Abstract

A comparison of the technical features for several telemetry transmitters, methods of decoding APRS packets, and database solutions to organize and store flight data, as they relate to the Oregon State University entry into the 2017-18 Spaceport America Cup 30k challenge.

4.1 Project Overview

4.1.1 Introduction

Our project is to design, build, and test software that will fly on board the Oregon State University's entry to the Spaceport America Cup's 30k Challenge. The Spaceport America Cup is an international engineering competition to design, build, and fly a student-made rocket to 30,000 feet. The competition is scored on several criteria including software components like flight avionics, recording and displaying telemetry, and using a scientific research payload.

4.1.2 My personal role in the team

Our team has many members, including at least 12 mechanical engineering, three electrical engineering, and three computer science students, as well as several faculty and industry mentors. My primary role in the team is to work with my computer science subteam to design, build, and test the ground station software, as well as the flight software that will control avionics for the rocket, and a scientific payload that will be ejected from the rocket at apogee.

We have divided the programming aspect of the project into 4 groups of software: rocket avionics, payload avionics, parsing data, and displaying data. Our goal is to work together so that all three of us contribute to each piece, even though one of us may be called on to take the lead on different pieces at different times. Personally, my primary interests lie in the rocket and payload avionics components, but I also have previous experience in parsing APRS packets and graphing data.

4.1.3 My expected contributions towards our team goals

This project requires writing avionics software to control events on the rocket, another avionics software to control events on the payload, a ground station software to process incoming experiment or telemetry data and store the results in a database, and software to display information from the database in a browser. Additionally, each program will include a suite of unit tests.

These four goals provide a way of dividing the project into individual portions that can be developed simultaneously. Each of us intends to contribute to all of the portions, but we may also choose to each manage a portion in order to gain experience both leading, and following other team members.

I would like to manage one of the avionics programs, and possibly the parser program. For the rocket avionics, I would be excited to write a Kalman filter to exclude errant sensor data and prevent accidental early parachute deployment. For the payload avionics, I would like to write a closed loop PID controller to adjust the motor speed to provide optimum thrust to achieve micro gravity. The parser will also be interesting to work on because it may require parsing a wide variety of sensor data from csv files.

4.2 Transmitting telemetry data

4.2.1 Overview of telemetry

In this context, telemetry is the data transmitted down from the rocket during flight. This data will be used to calculate the rocket's maximum altitude, as well as locate the rocket after the flight. If something goes wrong during the flight and the rocket is not recovered, telemetry can be used to determine what went wrong so that future flights may avoid the same problem.

4.2.2 Criteria for telemetry

This project will use a commercial, off the shelf telemetry module to transmit data from the rocket to the ground. There are several commercial products that offer this capability. The important features of a telemetry module are:

- Radio frequency
- Sensor types and transmission rates
- Physical characteristics and costs

4.2.3 Considered Technologies

4.2.3.1 Bigredbee BeeLine GPS: The BeeLine GPS unit is a self contained GPS and telemetry transmitter that operates on the 70cm radio band. It is 1.25″ x 3″, weighs about 2 ounces, and commonly uses a 35cm long antenna. The BeeLine GPS can also record up to 2 hours of GPS data internally. The BeeLine GPS can log and transmit limited positional telemetry fields at 1Hz intervals at 100mW of power, with an advertised range of 40 miles.[**?**]

The BeeLine GPS transmits an audio signal that needs to be decoded on the ground, using a terminal node controller (TNC). There are several devices that can accomplish this, including using the sound card on a computer. The price varies by vendor, but most hardware TNC's cost over $80.[**?**] Software TNC's are limited to one signal per sound card, which makes exact pricing difficult to determine. Using a dedicated raspberry pi zero with USB sound card, the cost to process APRS packets with a software TNC should be around $25 per signal.[**?**]

It is unknown how many signals can be processed by a single computer, but it seems reasonable to assume that processing additional signals will eventually begin to reduce the rate of successful packet captures. In other words, processing each signal with a dedicated computer may add redundancy and performance.

The Bigredbee Beeline GPS costs $259 and requires a handheld radio, which costs around $40. It will also likely require a dedicated computer per signal, which could be a raspberry pi zero with usb sound card for an additional $25 per signal. This brings the total cost to around $324 per signal.

4.2.3.2 Altus Metrum TeleMega: The TeleMega is a self contained GPS and telemetry transmitter on the 70cm radio band, that can also function as a flight computer. It measures 1.25″ x 3.25″ and often uses a 35cm long antenna. The data is sent as an audio packet that is decoded by the receiver. The TeleMega can store several hours of data on board, and transmits many types of telemetry at custom intervals, using 40mW of power and an advertised range of 40 miles.[**?**]

TeleMega data packets are formatted in 32 bit packets to reduce errors, and are designed to be received and decoded by a TeleDongle ground station. The TeleDongle is an additional cost but also provides a convenient serial output over USB. This is important because it means a single computer can simultaneously receive multiple signals from different transmitters.[**?**]

The TeleMega costs around $280 and offers a 10% student discount. The TeleDongle is another $100, bringing the total to around $342 per signal.

4.2.3.3 RFD900: The RFD900 is a 900 MHz radio designed to work as a pair of serial modems. We could generate a signal on the flight computer, send that signal to a TX RFD900 on the rocket, pick up the signal on a RX RFD900 on the ground, and get whatever telemetry fields we want on the ground. The advantage is that the system would be very customizable. We could send any size packets at any rate we want. The disadvantage is reliability. The RFD900 transmits at a much higher frequency, which reduces rates and has more trouble with interference from atmospheric conditions such as humidity.[**?**]

The RFD900 costs about $300 for a transmitter/receiver pair, and requires additional processing on the ground, for a total of around $350 per signal. One significant downside to the RFD900 is that it is designed to work as a transmitter/receiver pair, which means we could not easily use redundant receivers. There is an open broadcast mode available that would offer that feature, but there are reasons to believe the RFD900 is not the most reliable choice.

4.2.4 Comparisons

Model	Radio Frequency	Range	Sensor fields	Transmission rate	Size	Cost
BeeLine GPS	100mW 70 cm	40 miles	Latitude Longitude Altitude	1 Hz	1.25″ x 3″, 2 ounces	$324
TeleMega	40mW 70 cm	40 miles	Latitude Longitude Altitude Ground speed Ascent rate Bearing Gps confidence Time	1 Hz	1.25″ x 3″	$342
			Device ID Flight number Configuration version number Apogee deploy delay Main parachute deploy Radio operator identifier Flight state Battery voltage Pyro battery voltage Calculated height	5 Hz		
			Angle from vertical 100g accelerometer on Z axis 3 axis accelerometer 3 axis gyroscope 3 axis magnetometer Barometer Temperature	10 Hz		
RFD900	1W 900MHz	26 miles	Customizable, any	Any	1.25″ x 2.5″	$300

4.2.5 Conclusions

Based on all factors, I believe the TeleMega is the best choice for the telemetry module. It provides far more fields of telemetry, offers similar range and reliability, could act as a redundant flight computer if needed, is competitively priced, and is designed and supported by one of our project mentors.

4.3 Receiving and parsing the telemetry packets

4.3.1 Overview of telemetry signal formatting

This subsection depends entirely on the exact model of telemetry transmitter used. Based on several discussions with the ECE subteam, we assume they will select the Bigredbee Beeline GPS telemetry unit, which is limited to sending a tone-based signal formatted as an APRS packet. That means we will need to use a HAM radio to receive the signal and convert it to an audio tone, which then needs to be parsed into a string using a process called Terminal Node Controller (TNC).

4.3.2 Criteria

The goal of this component is to parse an audio signal into text. As such, the main priority is accuracy and reliability. The parser should be able to correctly and reliably interpret the highest number of packets, even under sub optimal conditions. Multiple options will be judged based on performance (compare identical sample input packets across multiple parsers), reliability (what can go wrong), license or comprehension (open source vs black box), and cost.

4.3.3 Using a Hardware TNC

For a long time, this problem was solved by electronics hardware. Several HAM radio companies exist and sell units known as hardware TNC's, which process the audio signal into serial text. These have the advantage of being plug-and-play. They convert input to output and the user doesn't really need to know much about what happens inside.
There is an argument to be made that hardware TNC's are not all made using the same algorithm, and that modern software TNC's may be able to decode a greater proportion of 'messy' signals with some amount of noise or interference. (Will address this in the software TNC portion.)
Another downside to a hardware TNC is that it is essentially a black box to us. We will not be able to understand the circuit well enough to make any changes, fixes, or improvements.[**?**]
There are several hardware TNC's available for sale, listed from most to lease common.

- Kantronics KPC-3+ Packet Communicator, MSRP $199.
- Kantronics KAM, MSRP unknown, no longer commonly sold.
- Kenwood TM-D710A, MSRP $549.
- PacComm Tiny-2 MK-II, MSRP $99.

Another problem with hardware TNC's is that they are not as popular as they once were. It was difficult to find any units for sale to make this list, even using sites like ebay or amazon. Finding several identical and new hardware TNC units for our project may prove difficult or impractical.

4.3.4 Using a Software TNC

Software TNC's convert audio tones to text, which allow them to perform the same task as a hardware TNC without a physical circuit. Some software TNC's also include features to verify APRS packets, and even decode packets into individual fields. This is ideal for a situation like ours where individual fields could then be inserted into a database.
The downside to using a software TNC is that it requires one sound card per audio signal. Even if an audio splitter were used, multiple simultaneous audio sources would likely increase the number of errors even if additional software were used to attempt to separate the sources. Therefor it seems likely that one computer would be needed per audio source. Using several Raspberry Pi computers may be a good choice because they are small, cheap, and can run headless without the need for a keyboard or monitor.
4.3.4.1 Direwolf: The most popular software TNC is an open source project called Direwolf. Direwolf is a software "soundcard" AX.25 packet modem/TNC and APRS encoder/decoder. Because Direwolf is open source, it does not need to be a black box to us. We can look inside to understand how it works, make changes as needed, and use only the pieces we need for our project.[**?**]
Additionally, the developer of Direwolf proposed an experiment to compare it's new algorithm against a hardware TNC, using a pre-recorded 2 hour sample of APRS traffic from the city of Los Angeles. This allows others to run the same

experiment and compare their results against other hardware TNC's, or their specific computer hardware including sound card and processor. The results of his experiments showed that the Kantronics KPC-3 Plus successfully decoded 30% fewer packets, and the Kenwood TM-D710A decoded 33% fewer packets than the software TNC.[**?**]

4.3.4.2 AGWPE: AGWPE is another software TNC solution. Although it is free, AGWPE is not open source. Instead, it is distributed as an exe file and operates as a black box. It is not clear if it can run in Linux under wine or a virtual machine running Windows, but it is very likely that the additional complexity would present problems for using cheap, distributed redundant receivers using raspberry pi computers. There are no easily available statistics comparing the performance of AGWPE to similar technologies.[**?**]

4.3.5 Comparisons

All of these options also require purchasing a portable HAM radio unit and appropriate audio cable for each signal.

Product	Software/Hardware	Performance	Availability	Cost
Kantronics KPC-3+	Hardware	0.7	Limited, used	$199
Kenwood TM-D710A	Hardware	0.67	Limited, used	$549
PacComm Tiny-2	Hardware	?	Limited, used	$99
Direwolf+Pi Zero	Software	1	Open source	$25
AGWPE(Windows laptop)	Software	?	Closed source	$0

4.3.6 Conclusions

Using this information, I believe that using the Direwolf software TNC is the right choice for our project. Furthermore, I believe that using one Raspberry Pi per audio source is a good way to distribute the signal processing. If using a Pi Zero and USB soundcard, the cost would be around $25 per audio signal. Each Pi would process one telemetry signal and be networked to a central computer, which would collect and store data from each Pi.

4.4 Storing telemetry and flight data

4.4.1 Overview of the data to be stored

This project will generate telemetry data from the rocket, telemetry data from the payload, sensor data from the rocket, and sensor data from the payload. All of this data is useful and may be related. For example, using acceleration data from the rocket may clarify or improve results obtained from the payload.

Additionally, the data will need to be used by another program. We intend to view telemetry data and generate dynamic graphs while the rocket is still in the air, and after recovering the rocket we will import sensors and generate a more complete set of graphs using that data.

With these conditions, we believe a database is a good solution for storing all of this data.

4.4.2 Considered Technologies

4.4.2.1 SQLite: SQLite is an embedded, server-less, transactional SQL database engine. Records are stored in a single file, which can easily be copied across platforms. These features make it very lightweight and ideal for single-user applications.[**?**]

4.4.2.2 MySQL: MySQL is one of the most popular databases worldwide, especially for web based applications. It's freeware, receives regular updates, and is very reliable. MySQL could be considered the 'standard' for other databases to be compared against.[?]

4.4.2.3 PostgreSQL: PostgreSQL is another free database commonly used for web applications. It offers more features than MySQL, but at the cost of additional complexity and arguably worse documentation. One interesting perk for PostgreSQL is that it supports JSON.[?]

4.4.2.4 MariaDB: MariaDB is the newest database being considered. It offers some additional features such as server and application-level encryption, progress bars during especially large instructions, and claims improved stability and performance over MySQL.[?]

4.4.3 Comparisons

After researching several of the popular relational databases, the following comparison table was made[?][?][?]:

Database	Pros	Cons	Conclusion
SQLite	Easy to set up Stored as a single file	No user/access management Cannot easily function as server	Good for single-user applications
MySQL	Popular, lots of tutorials Fast writes	Not fully SQL compliant Queries using TIME lack precision	Not a bad choice
PostgreSQL	Fully SQL compliant Fast reads	More complex than others	More database than we really need
MariaDB	Active userbase Claims to be faster than MySQL		I like it!

4.4.4 Conclusions

An active user base and being open source are nice features for any piece of software. It would also be smart to use a popular database so that the skills we learn during this project are more likely to be applicable on future projects.

SQL lite lacks the server functionality we need for this project. PostgreSQL may complicate setup and usage by offering more features than we really need. I don't think it will matter for our scale of application, but PostgreSQL also seems to be optimized for read time, while we will probably want to optimize for write times due to the time-sensitive nature of our incoming data.

After comparing several types of databases, I believe MariaDB fits our needs the best. I like that it's open source and has modern, easy to use guides and support forums. It is SQL compliant so the skills we learn can also be applied to other, similar databases on future projects. MariaDB also supports precise queries for TIME or TIMEDATE fields, which may prove very useful for our type of data.[?]

5 Technology Review Document (Joshua)

Abstract

This document describes three technology choices that our team needs to make to complete our project, with three proposed options for each choice.

5.1 Graphical User Interface Language

5.1.1 Overview

The GUI will be in the form of either a web-hosted or local app that connects to an intranet database. It will display graphs of telemetry data, either live or recorded.

5.1.2 Criteria

The criteria for evaluation is based on whether there are pre-existing libraries that can accomplish the task, the ease with which it can be implemented alongside a database and in a web-hosted app, whether it runs server or client side, and how flexible it will be for adjusting the visual design of the application. Pre-existing libraries and ease of implementation are given a priority because any of the three should be able to be used to accomplish the task and efficiency of the code is not of significant importance since it will all be run on client computers such as laptops that should be more than well-equipped to handle the task, though performance issues may become significant if the script needs to run on the computer running the server.

5.1.3 Potential Choices

5.1.3.1 Python: Members of the team have used Python to create GUIs before, so the team is already aware of several libraries to use with this. Several are options are described in a blog on python libraries named 5 Python Libraries for Creating Interactive Plots[**?**] which also has links to the relevant libraries. They appear to be fairly easy to use and likely will fit our purposes, with some example plots being visually quite similar to our own end goal. In particular, HoloViews features a map similar to one we will need to generate from GPS data. Several of the libraries mentioned in the source have functions for outputting to HTML, so we should be able to use them to create our graphs. However, it is unlikely that these libraries will be able to dynamically alter the display as new data is made available since they generally output to HTML files, though this should be possible by overwriting HTML files when a library does not have the functionality to dynamically update the graphs and having the site automatically refresh some of its contents. In general, Python libraries will run on the server computer rather than the client, which may cause performance problems.

5.1.3.2 JavaScript: There are many libraries for JavaScript that the team should be able to use to generate or manipulate graphical content, including the CanvasJS[**?**]. This library accomplishes most of our goals for graphing data, explicitly describes methods for live updates to graphs, and has methods for generating graphs from csv files. This library is extremely suitable to our tasks. JavaScript generally runs primarily on the client side, which is true of this library, so performance should not be an issue. The only potential issue will be the map, for which either another library can be used, or the map image can simply be edited using standard JavaScript methods to place red dots or other markers over the map at the correct locations.

5.1.3.3 R: R is specialized for mathematical programming, which suits our needs. There are libraries for R which generate graphs that fit the desired output, such as Plotly[**?**]. These libraries will either be able to perform live updates (with the implementation of a Plotly server), or overwrite the currently existing HTML file to mimic this. Visually, the plots are very functional, and they include graphs that resemble the GPS map that needs to be created. However, it will need to run on the server computer rather than on the client computer, which may cause performance issues.

5.1.4 Compare

R (and the related libraries) has all of the graphs we will need and is specialized for much of this task, while also being very easy to use and code in. However, it will need to run on the server, and therefore may introduce performance issues. Python is not as specialized for this task or as easy to use, but is more flexible, and has a larger number of related libraries. It will also likely perform better. However, it also will run server side, and therefore may introduce performance issues. JavaScript has libraries that explicitly describe how to create live updates to dynamic graphs. It also runs client side, giving it the best performance.

5.1.5 Conclusion

JavaScript is likely the best candidate. It will be able dynamically generate our graphs, and runs client side to deal with performance issues. As it is the only option that accomplishes all of these goals, it is what will be used to generate the GUI.

5.2 Avionics Code - Language

5.2.1 Overview

A portion of the Avionics code for the rocket and payload will be written by the CS team, with some of the code also being written by the ECE team in order to satisfy the requirements of their Senior Capstone. This code will need to run on a Blackberry Pi.

5.2.2 Criteria

The criteria for selecting the language of the Avionics code is based on three major factors, the efficiency of the language, the compatibility of the language with the devices that are likely to be used, and the suitability/usability of the language for accomplishing the required tasks. As a minimum requirement to be used at all, the code must be compatible with the Blackberry Pi, narrowing the range for selection. So far, the only device that is known to be used in the rocket is BigRedBee's BeeLine GPS[**?**], which will not heavily interact with the avionics code and does not appear to have a related library. Looking to previous years, most sensors have C libraries for interacting with them, so languages that are compatible with C functions or can call C programs are more valuable. Therefore the secondary criteria breaks down to the language's compatibility with C.

5.2.3 Potential Choices

5.2.3.1 C: C easily satisfies all three criteria. As a low-level language, C is highly efficient. This makes it excellent for criteria one. It is obviously compatible with itself, satisfying the second criteria. As for suitability/usability for the task, C is the standard language for embedded code such as avionics code. Issues may be encountered due to the difficulty of coding in C compared to higher level languages but these should be able to be addressed. When creating the test suite, a higher level language would normally be preferred due to the ease of use and the compatible libraries, but the test should still be able to be written in C.

5.2.3.2 C++: C++ is a high-level object-oriented language, making it inefficient in many cases. While C functions and avoiding the use of objects can address this problem, additional inefficiencies will be introduced by using C++. C functions can be used in C++, so there is unlikely to be any compatibility issues. All of the necessary code should be able to be written in C++ without significant issue, and it's status as a high-level language will make it easier to use. Test suites will be easier to create and organize with the option to use objects.

5.2.3.3 Python: Python is a high-level object-oriented language with particularly issues with efficiency. For this reason, it largely fails criteria one, and may cause performance issues. Python can call C programs and collect their outputs, so it should be able to satisfy criteria two with minimal effort. Finally, it should be very easy to write code and tests in Python, and there are a wealth of libraries that exist to bridge any gaps we may encounter. However, such libraries are likely to be unnecessary. Test cases will be relatively easy to write in python.

5.2.4 Compare

None of the three languages will have significant compatibility issues, though Python may encounter minor ones. Python is by far the most inefficient of the three languages, and only brings minor benefits in usability and suitability. C++ is significantly more efficient and has similar benefits to Python in usability and suitability, as we are unlikely to need to use third-party libraries that may exist for Python. Finally, C is vastly more efficient than either of the other options, and while it may be somewhat more difficult to use, it is not a significant downgrade in this case.

5.2.5 Conclusion

C will be used for the avionics code due to the greater efficiency of the language with only minimal losses in usability and suitability. Tests may be written using C++, as the two languages are compatible with one another, but the avionics code itself will be written solely in C.

5.3 Avionics Code - Testing Methods

5.3.1 Overview

The client has requested that the CS team create a thorough suite of tests for the Avionics code. This is due to the many failed recoveries of rockets in the past, which may have been due to poorly written/tested code.

5.3.2 Criteria

The criteria for evaluation is whether the testing method will allow us to test against likely errors that we may encounter from the sensors. This will vary based on the sensor in question, but of particular importance is a sensor throwing a value that is outside of expectations, not an acceptable value, or failing to send a value at all. Of lesser but still significant importance is creating tests that ensure that the return values for a flight without sensor errors are accurate.

5.3.3 Potential Choices

5.3.3.1 Real Data: Testing against real data includes test launches/flights, testing against the sensors at rest, and testing against data recorded from previous flights, such as those carried out by earlier years or data found online. There are several limiting factors on these tests. Firstly, test launches/flights will not occur often, making their utility extremely limited. Also, we will most likely want our avionics code to be completely before any test launch. Second, previous flights and flights found online may have their data fixed to remove outliers or not contain errors. This makes this method less useful for satisfying our core criteria. However, they remain very useful for testing against the secondary criteria, as we should be able to test against flights/launches that reflect our own. Finally, tests against the sensors at rest will only be able to allow us to test within an extremely narrow range of possibilities that do not reflect the likely outcomes during a launch.

5.3.3.2 Simulated Launch w/ Robustness: Testing against a simulation tool or a collection of simulated data has a variety of advantages. It satisfies the second criteria very well, by testing against what should be the outputs of a typical launch or a launch within certain parameters. Robustness testing can be inserted in manner similar to how robustness testing is added to model-based testing, as outlined in the paper Model-based robustness testing for avionics-embedded software[**?**]. The article describes a manner of adding random changes to the outputs of different test objects representing sensors during a simulated launch. This ensures that the avionics code can handle incorrect outputs or failings in the hardware. This robustness testing would include randomly having the sensor throw a value outside of the expected

bound, a value that does not fit it's standard form, or having it shut down and turn on at random intervals. By setting this to occur randomly at some varying rate, the robustness of the suite can be dramatically increased, satisfying the second criteria.

5.3.3.3 Unit Testing: Unit Testing will ensure that the basic functionality of the code is there. It can be used to ensure that for very specific inputs the correct outputs are reached, and that every line of code and logic is checked. This helps satisfy the first and second criteria, and ensures the code is functional. It is also fairly straightforward to implement.

5.3.4 Compare

All three have particular advantages and disadvantages. Testing against real data is perhaps the best way to satisfy the second criteria, as it is definitely a test against a likely launch. However, it does not satisfy the first criteria to any significant degree, if at all. Unit Testing will assist in satisfying both criteria, but not as well as Simulated Launch with Robustness. However, it is much easier to implement than either other method.

5.3.5 Conclusion

Simulated Launch with Robustness is the best was to ensure that both criteria are satisfied, and therefore will be given the highest priority to complete. Unit Testing, due to the ease of implementing it as well as its significant returns, will be used alongside it. Testing against real data will be given the lowest priority, as it only satisfies the second criteria, and not significantly better than a Simulated Launch.

6 TECHNOLOGY REVIEW DOCUMENT (ALLISON)

Abstract

This document serves to illustrate the differences between the networking-related technology options for the CS 30K Rocketry team's ground station. This will include the network type used to connect the ground station modules to each other as well as the hardware and software used to host the database and server.

6.1 Introduction

6.1.1 Project Goals

The purpose of the 30k Rocket Spaceport America project is to design and build a rocket that can reach an altitude of 30,000 feet. This rocket must be recoverable as part of the competition guidelines. As part of the challenge, the payload should achieve at least 10 seconds of micro gravity in order to conduct a successful experiment. As the computer science sub-group on the project, the main goal of this team will be to display collected telemetry and payload data, as well as live GPS tracking of the rocket and payload for recovery of each after the launch through the ground station.

6.1.2 Group Roles

The project is comprised of six sub teams, including four mechanical engineering sub teams, one electrical engineering sub team, and one computer science sub team. Each of the mechanical teams are responsible for one of the following: aerodynamics and recovery, payload, propulsion, and structures. Rocket avionics are handled by both the electrical engineering and computer science teams, with the on board electronics primarily the responsibility of the electrical team, and the ground station software and electronics primarily the responsibility of the computer science team. There

may be some crossover where the avionics code on board the rocket is involved. The computer science team will also write unit tests for the software written by the team as a whole.

Within the CS team, I expect that each member will share a part of all pieces of the project, but each of us may take the lead for certain portions of it. Since all of my technology review pieces are related to networking technology, it may become my focus for the project, but I'd also be interested in helping with the other areas of the project, especially those not covered by this round of reviews.

6.2 Network Type

6.2.1 Overview

For the implementation of the ground station, some kind of network will be required to connect the devices displaying and graphing the flight data to the device hosting the database and to the devices parsing the radio signals. The goal is to allow team members to connect phones and laptops to the network, allowing them to view the live GPS data on their personal devices. Because the network does not need a connection to the wider internet, the focus of this subsection will be on infrastructure-less, or ad hoc networks.

6.2.2 Criteria

The network should be wireless to support connections with mobile devices. Because launches happen in remote areas, cell service may be poor where the system is deployed and the network cannot rely on the internet. The network should support devices frequently connecting and disconnecting, as the devices viewing the served data may change throughout the flight.

6.2.3 Potential Choices

6.2.3.1 Mobile Ad Hoc Network: A mobile ad hoc network, or MANET, is a type of decentralized network, meaning that it operates without any centralized administration. The network is formed from wireless mobile hosts that may leave or join the network freely. Each mobile node has relatively short range, which keeps battery consumption low and allows reuse of bandwidth. Connections within the network are made by using each node as a router, packet source, and packet sink. This means that any node may send, receive, or route packets to and from other nodes. The routing protocols are more complicated in a MANET because of their dynamic nature.[**?**]

6.2.3.2 Wireless Local Area Network: A wireless local area network, or WLAN, is a centralized network that typically deploys one or two access points to broadcast a signal in a 100 to 200 foot radius.[**?**] Depending on how it is configured, a WLAN may operate in an "infrastructure" or "ad hoc" mode.[**?**] Here, we would be configuring the network without access to the greater internet, so it would be run in ad hoc mode.

6.2.3.3 Wireless Personal Area Network: A wireless personal area network, or WPAN, is a centralized network that typically connects very close laptops, cellphones, and peripherals. These networks can connect devices up to 100 meters apart. There are several standards for this kind of network. Zigbee and Bluetooth standards operate over shorter ranges, while WiMAX has a larger range. [**?**]

6.2.4 Discussion

The major differences between types of wireless networks lie in centralization and infrastructure. There is a central unit in this network, the database and server host. The network needs no internet connection, so an infrastructure-less

network is ideal. Distances between nodes are expected to be quite close, because the team will be standing together at the launch site.

6.2.5 Conclusion

Because we already have a centralized server and distances will not be far enough to warrant a multi-hop system like a MANET, a wireless local area network should serve our purposes just fine.

6.3 Server Type

6.3.1 Overview

We plan on using a web based display program for graphing the flight data that is parsed into the database. This webpage will be be viewed over a local network on mobile devices. For serving the page over a network, server software will need to be selected.

6.3.2 Criteria

The server will need to handle enough connections to serve pages to each team member's mobile device, about 20. Updating the displayed graphs should be quick enough to display the live GPS data, which will arrive about once per second.

6.3.3 Potential Choices

6.3.3.1 Apache: Apache is the most commonly used web server software. It's open source and has a very supportive community that would make development easier.[**?**] This web server is favored for flexibility, power, and support. It features a system of dynamically loadable modules. Apache supports language interpreters for dynamic content.[**?**]

6.3.3.2 Node.js: As one of the servers covered by OSU curriculum (In CS 290), the team has some experience with node.js. Another open source server, it focuses on supporting the use of Java on the back end. This may be helpful if the team decides to make heavy use of Java for displaying the graphical flight data because a lot of support is built in. There are libraries that make this web server very easy to use with databases.[**?**]

6.3.3.3 NGINX: Another popular server, NGINX relies on an asynchrous, events-driven architecture and sports lightweight resource use. This web server focuses on being able to support many connections by passing dynamic requests off to other software. It excels at serving static content to many connections.[**?**]

6.3.4 Discussion

The content being served is dynamic in nature, and while NGINX boasts great scale-ability, this network will not have nearly enough connections to warrant using software that isn't as good at serving dynamic content. Both Node.js and Apache have libraries that support database use and make it relatively straightforward to serve dynamic content.

6.3.5 Conclusion

All of these web servers are open source and have supportive communities. Apache has the advantage of being able to dynamically include any modules we need, as well as being one of the more popular and widely supported servers. Because this web server is so common, it would be good for the team to gain experience with this technology for applications in the job market. This server also excels at serving dynamic content. All of our content is dynamic, so this is ideal.

6.4 Host Hardware

6.4.1 Overview

The database used to populate the display app must be served to other devices from some physical location. The server must also be hosted from some physical location. Both locations must be able to communicate with each other and with other networked devices.

6.4.2 Criteria

The hardware should be low cost to stay within budget. Both the server and the database must be hosted. The selected network type should be supported. The ground station would ideally be contained altogether in an enclosure including the parsers and host hardware.

6.4.3 Potential Choices

6.4.3.1 Raspberry Pi 3: The raspberrry Pi 3 has wifi capabilities and would be able to serve as a host. Using an rpi would allow the ground station to be contained in an enclosure with the parsers and receivers. The raspberry pi does not have a real time clock, but could be connected to one. Wireless LAN, Bluetooth, and Ethernet are supported. An external LED screen could be connected for debugging. These each cost $34.95. [**?**]

6.4.3.2 Laptop: A laptop would be more than adequate as a host for the database and server. However, this would prevent us from keeping the ground station in a compact box. Previous years have seen some degree of dust build up on laptops, which wouldn't affect immediate performance at launch, but isn't ideal. Most laptops support Bluetooth, wireless LAN, and Ethernet. A personal laptop could be used with our server and database software, which would keep the cost for this component at $0.

6.4.3.3 BeagleBone Blue: This beagle board is an open source linux-enabled robotics computer. It includes an on-board programmable real time unit. Both Bluetooth and WLAN are supported. These each cost $79.[**?**]

6.4.4 Discussion

For hosting the server and database, any of these options would be sufficient and would work with our network. The laptop would meet out needs, but remove some of the separation we wanted from the ground station and display. The beagle board was suggested due to the real time unit, but isn't strictly necessary for hosting the database and server. The raspberry pi is cheaper than the beagle board and would accomplish much of the same thing.

6.4.5 Conclusion

Based on the price and the general desire for a separated ground station, the Raspberry Pi 3 seems like the best choice for our system. It will provide minimum cost and meet all other criteria.

7 Weekly Blog Posts

7.1 Fall

7.1.1 Week 1

7.1.1.1 Levi Willmeth: Went to AIAA team meeting, let them know that I was working with Dr Squires on the project but that I did not know who the other CS team members would be. Also offered to help make a basic team website with teammate Matt Hoeper, to be hosted on github.

7.1.1.2 Joshua Novak: Summary

Made bids, emailed Nancy Squires about rocketry projects

7.1.1.3 Allison Sladek: :

Summary: Reviewed posted project proposals and submitted preferences.

7.1.2 Week 2

7.1.2.1 Levi Willmeth: All team members have been assigned. We met with our sponsor Nancy Squires, mentors Keith Packer and Joe (?), and the rest of the 30k subteams.

Went to the model rocket build on Thursday night, assembled my level 1 certification rocket, Allison and most of the 30k team did the same. Met with Joe Bevier who gave us some build advice including how to fold the parachute, and to use a longer piece of something like 5050 paracord instead of the short shock cord and elastic that came with the kit.

Met with Nancy Squires in Rogers 306. This was our first official meeting with her, as a subteam. We talked about the scope of the project. She said she would 'support our requirements' and work with our Capstone faculty whenever the rocket deadlines did not match up with the capstone class deadlines.

I met Keith Packer while he spoke with the 100k team this afternoon. Keith runs Altus Metrum and created the TeleMega avionics board that many teams use as one of a pair of triggers for separation and parachute deployment.

7.1.2.2 Joshua Novak: Plans:

Meet with EE team

Summary

Met with team, emailed client, met with client

7.1.2.3 Allison Sladek: :

Plans: Another meeting is scheduled next Thursday night with all subgroups on the project.

Problems: Project requirements are intentionally vague, so we'll need to meet with the other subgroups on the project to clear them up.

Progress Started writing the problem statement document this week

Summary: Assigned to 30k rocketry project, Attended meeting with all subgroups of rocketry team to identify focus areas and get updates on progress, Participated in team-building exercise, Met with sponsor to discuss expectations and requirements

7.1.3 Week 3

7.1.3.1 Levi Willmeth: Working to understand our role and responsibilities within the team, and our capstone requirements.

Met with the ECE team so that both subteams have the same expectations regarding who is writing software for the avionics and payload.

7.1.3.2 Joshua Novak: Plans:

Write Final Draft of Problem statement

Summary

Wrote and submitted rough draft of problem statement

Met with TA for the first time

Met with EE team, they do not currently know what computers will be used on the rocket

7.1.3.3 Allison Sladek: :

Plans: Meeting with all sub-groups for the 30k rocket every Tuesday Meeting with TA and group members every Friday Update AIAA membership for team roster by 10/16/17

Problems: At this point, we are still determining the scope of our project and how we will fit in with the other subgroups on the 30k project.

Progress: Talked to electrical engineering subteam and discussed possible data types to interpret at the ground station Met with TA

Summary:

During the weekly 30k all-team meeting, we checked in with the electrical engineering team and found that they were still deciding which boards to use to gather, store, and transmit rocket GPS and telemetry data. They are looking into using some Atlus technology along with something they build themselves. Once they have a better idea of how they plan on transmitting the data, it will be easier to design the software that interprets it. The TA meeting this week yielded a lot of recommendations that will have to be looked into.

7.1.4 Week 4

7.1.4.1 Levi Willmeth: Working to understand our role and responsibilities within the team, and our capstone requirements. Met with the ECE team so that both subteams have the same expectations regarding who is writing software for the avionics and payload. My current understanding of the CS capstone team requirements:

Avionics software: Detect launch, detect primary burn, detect roll, trigger separation, trigger drogue chute (?)

Payload software: Record data. (TBD)

Ground station: Select input source, parse inputs, display points on a map.

Visualizing onboard sensors: Parse files from multiple SD cards, combine or related, then graph data.

7.1.4.2 Joshua Novak: Plans:

Look for code from previous year's team

Summary

Wrote and submitted final draft of problem statement

7.1.4.3 Allison Sladek: :

Plans: Continue presenting progress at weekly 30k team meetings Look into old repositories from previous teams to determine how much code may be reused.

Problems: Need to consult with the electrical team to find out what platform they're planning on using and determine the best language to use on it. Draw up requirements document after consulting Dr. Squires.

Progress: Finished the problem statement and got approval from Dr. Squires to move forward.

Summary: The final draft of the problem statement was finished this week, and was approved by Dr. Squires. Because we are part of a larger team, a lot of our project is dependent on the work of other sub-teams. Until we know what platform the avionics electrical team has chosen, it will be difficult to design for.

7.1.5 Week 5

7.1.5.1 Levi Willmeth: Began using the "3 P's" method of notetaking: plans, progress, and problems.

Plans for next week: Decide if we want to expand on the previous year's ground station or start from scratch. If starting from scratch, begin deciding on a language for the ground station. Decide if we want to be able to use cell phones to

view the data, probably via intranet page. Improve our slides for the Tuesday meeting!

Progress since last week: Finished group problem statement and got it approved by Nancy.

Problems we've encountered: It feels like we're just waiting around for an assignment, let's make more progress!

7.1.5.2 Joshua Novak: Plans:

Finish requirements document, look into GUI options, look into work arounds for limitation on data, look for last year's code

Problems

ECE team appears to be planning to use a device that will severely limit what data we can receive

Progress

We should be able to design the GUI so that it can handle fields that are not available or generate some fields from other data. This will be our compromise if we aren't able to come up with an alternative

Summary

Checked online for code from last year's team, there was only ECE code. Looked into creating a web based GUI

Created outline and wrote multiple sections for requirements document. Met with other teams and discovered we may not have significant telemetry data for payload.

7.1.5.3 Allison Sladek: :

Progress Finished draft of requirements document Looking into reuse of previous teams' code from EE, CS teams Got telemetry board choice from electrical engineering team: BigRedBee

Plans: Continue to refine requirements document Meet with Dr. Squires to confirm requirements Further explore platform options this week Compare APRS to non-audible tones we would expect from the two possible telemetry boards Look into web-based GUI options

Problems: Board selection made by electrical engineering may reduce data received from rocket Transmitted data is reduced to latitude/longitude

Summary: This week, we finished the initial draft of the requirements document. More functional requirements will need to be added, and we still need to meet with Dr. Squires to ensure we have all the requirements she needs. The electrical engineering team selected the telemetry board to be placed onboard the rocket. They argued strongly for the bigredbee due to reliability of GPS data transmission and price. It will mean significantly less live data than we expected, but with some research into APRS transmission, it will work.

7.1.6 Week 6

7.1.6.1 Levi Willmeth: Plans: Nail down more details a bout the project. Ground station host, format for the client, etc.

Progress: This week I spent a lot of my efforts trying to pin down the specific telemetry unit. The ECE team wants to use a Bigredbee BeeLine GPS but I prefer the Altus Metrum TeleMega.

This is an important decision because the two units use different formatting (audio vs serial) and send different telemetry fields (lat/long/gps alt vs lat/long/gps alt/3 axis accel/3 axis gyro/3 axis mag/barometer/others).

By the end of the week, after multiple discussions between me and Drew, the ECE team decided to use the bigredbee beelinegps with no redundant transmitter. I am unhappy with the way this played out, because the decision was made without talking with us or considering how it would impact our portion of the project.

Problems: The ECE and CS teams need to work closely together on many decisions or both teams will suffer, so I hope that we can improve our communication before this happens again.

For my part, I am now being more proactive about talking with them about specifics such as which sensors and flight computer they want to use. I am also trying to better define what their expectations are regarding the CS team writing the avionics software.

7.1.6.2 Joshua Novak: Plans:

look into GUI options, look into work arounds for limitation on data, look for last year's code

Problems

ECE team appears to be planning to use a device that will severely limit what data we can receive

Progress

Research was done on alternative devices for sending the telemetry information, to show that there would not be any significant negatives to using the other options

Contact has been made with someone who may have access to last year's code

Summary

Finished requirements document

7.1.6.3 Allison Sladek: :

Plans: Begin dividing sections of project for technology review Draft due November 14th Review Software Engineering II notes to help with testing later on

Problems: Telemetry transmitter decision made by the ECE team is final, and may limit the data we receive during flight. Dr. Squires was out of town Thursday and Friday, and hasnt commented on the requirements document.

Progress Completed our Requirements Document

Summary: This week the requirements document was completed and sent to our sponsor on Tuesday. As of Friday at 3:00, we haven't received a response yet.

7.1.7 Week 7

7.1.7.1 Levi Willmeth: Plans: This week has been all about thinking about and outlining our project requirements. WHAT the software will do, but not HOW it will accomplish it.

We are beginning to better understand the big pieces (rocket avionics program, payload avionics program, parser program, display program).

Now, we are beginning to break those big pieces into smaller pieces. What will the parser program actually need to do? What are good technologies to accomplish that?

Progress: Talked amongst ourselves and with Nancy to define our project goals. Talked about the situation with the transmitter and were told to keep talking with the ECE's and trying to compromise.

Identified 6 different stages of flight for an eventual state machine.

Problems: I've been talking with the ECE's in slack about the avionics software, trying to understand where the line between our two teams are. I would like for our team to write the avionics software and a full hardware simulation for it, but the ECE's are also expressing a desire to write it.

I'm concerned that because the telemetry has already been massively cut back, and if the ECE's write the avionics, that the CS team will not have enough work to justify this being a capstone project.

We could write hardware simulation against the ECE's avionics, but based on the communication problems so far, I anticipate that their software would just materialize as a finished product, which would make testing very slow and difficult. Also, if we found test failures, would they accept and fix them, or how would that work?

I will bring these concerns to Kevin in case he has suggestions on how to improve the technical difficulty of our capstone project.

7.1.7.2 Joshua Novak: Plans: Read papers/articles found in research to find what is applicable to my tech review
Look into GUI options
look for last year's code
Summary
Wrote outline for tech review including criteria and overviews, researched, links below, met with sub-teams.
ECE team will definitely only be sending a small variety of data for telemetry
http://www.sciencedirect.com/science/article/pii/S1000936113001179 - Paper on Avionics testing
https://arxiv.org/pdf/1707.01466.pdf - Paper on Avionics testing
https://canvasjs.com/docs/charts/how-to/live-updating-javascript-charts-json-api-ajax/ - Javascript
Team meeting with TA

7.1.7.3 Allison Sladek: :

Plans: Tech Review ECE Communication Talk about sensors Reduce surprises in future A few typos were found in the requirements document Revision is highly recommended

Progress Still working on tech review

Problems: Links to old source code aren't working May need to talk to Nancy about finding old Repos

Summary: The focus for this week was dividing the specific pieces of the project between group members so that we cover the important sections of our project in the tech review, without overlap.

7.1.8 Week 8

7.1.8.1 Levi Willmeth: Plans: Continue working to improve communication with ECE subteam. We need to know what the hardware will be, and be able to influence those decisions.

Progress: Mixed. Currently we find out about hardware decisions after the fact, and the ECE subteam does not like to discuss or revisit their decisions.

I met with one of the 30k team members from 2017 at an OSGC event on campus today. We talked about their project and they showed me a screenshot of their ground station UI.

One of their ECE members told me that they used a Stratologger to trigger their payload, which failed before launch. They could hear it go into a 'failsafe mode' but didn't have enough time to take the rocket apart to reset it, so their payload failed. Something to think about.

Problems: We set up a meeting for Friday to go over sensors and redundant avionics. Last we heard they want to use a StratoLogger, which is the same part that failed during last year's launch. I will encourage using another part.

I think it's also useful to note that buying a BeelineGPS, Stratologger, ham radio, pi, and usb sound card will cost more than buying a TeleMega and TeleDongle. At this point we're spending more money to get a worse solution.

7.1.8.2 Joshua Novak: Plans:
Format Tech Review Bibliography
Email CanvasJS Sales Team
Reformat the rest of my OneNote
Problems
Potential license issue

Progress

Finished research and first draft of Tech Review

Decided on JavaScript for GUI, either CanvasJS or JCanvas

Summary

Found remaining sources

https://blog.modeanalytics.com/python-interactive-plot-libraries/ -python libraries

https://plot.ly/r/ - Plotly library for R

Met with sub-teams, began budget, finished budget

Researched CanvasJS, discovered potential license issue

Met with TA, decided to email CanvasJS sales team about licensing issue

7.1.8.3 Allison Sladek: :

Plans: Submit entry form for competition soon Need to fill out testing plans and "other" section of the entry form Plan to finish tech review by Nov 21st

Problems: Need to determine how much of avionics code will be written by EE team and how much will be written by CS team

Progress: Still working on tech review

Summary: This week primarily consisted of working on the tech review document. My pieces covered the network aspect of the ground station, including server, host hardware, and network type. This week, the competition guidelines and entry forms were posted. The link below contains links to all the competition doucments: http://www.soundingrocket.org/sa-cup-documents–forms.html

7.1.9 Week 9

7.1.9.1 Levi Willmeth: Plans: Look for screenshots or code from last year's 30k team. Create a budget for the ground station. Begin splitting our portion of the project into sections that each of us can focus on.

Progress: Talked with ECE's and mutually decided that the hand off boundary for the ground station will be the audio wire from the radio to the parser. Everything on one side of the wire is in their domain, everything on the other side is in ours.

Our budget is currently $343 to build a ground station with redundant parts.

Dan He on the ECE subteam says we'll have redundant sensors on at least the rocket, and possibly the payload. It still sounds like they are planning on using the Stratologger for a redundant flight computer. I informed them that last year's Stratologger failed on the launch pad and cost the team some points, but we'll see what they decide to go with.

Problems: I tried to organize a meeting with the ECE team over the weekend, but it didn't pan out. (Not much response from their team, or ours.)

7.1.9.2 Joshua Novak: Plans:

Write Design Doc

Problems

Spoke to ECE team, there is a new problem where our avionics code will not be triggering flight events for the rocket. This is because their mentors will not allow it.

Progress

Due to this, we will write avionics code that could be able to trigger flight events, but will instead simply record when

it would have triggered the flight events.

Summary

Finished final draft of tech review

We met with a member from last year's team and now have access to last year's code and ground control.

Emailed CanvasJS sales team, they sent me a link to a non-commercial package for the software.

Talked about plans for design document

Reformatted OneNote

7.1.9.3 Allison Sladek: :

Plans: Planning on starting design document next week

Problems: Using more off the shelf components than expected the team may decide to design software as if we were to control separation at apogee, receive live telemetry data, etc.

Progress Looking into different options for network design

Summary: This week, some of the other subgroups announced that we may need to cut the active experiment previously planned for the payload, and that the mentors have required that the EE team rely on more commercial components than previously expected. This will mean less required code for those portions of the avionics (apogee detection, and plotting related data from these onboard sensors), but we have discussed the possibility of writing some of this code anyway for a possible back up solution.

7.1.10 Week 10

7.1.10.1 Levi Willmeth: Plans: Re-evaluate individual management topics, determine how to merge CS and ECE code.

Progress: This was Thanksgiving week so we lost a few days of productive time. I found last year's CS github repo! https://github.com/chris-quenzer/ESRA-CS-Team-Post-Expo

I wrote out a list of long-term project pieces, we talked about which pieces each of us want to manage. After speaking with the ECE team, they will write their own code separately from us, but our code will be flown during the competition. I drafted our database schema this week.

Problems: Early in the week we learned that the ECE team did not want to connect the rocket avionics to the separation charges or parachute. This meant that there was no point in attempting to find apogee of the rocket, because we could not use that information.

We decided to write that functionality anyway, even though it would not be used. At this point we are very frustrated because this was the second major piece of the project that was cancelled by the ECE's. The first major loss was their decision to use a telemetry module with only lat, long, alt fields, instead of 20+ from another telemetry module. The second major loss was not being connected to the separation hardware. That leaves payload avionics and making the best of the onboard data.

We tried talking to the ECE's after the Tuesday meeting, with mixed results. We pointed out they did not have sufficient sensors for our side of the project, and they responded by saying they were still adding sensors. We said that we need multiple, redundant IMU's on the payload, and at least one IMU on the rocket. They said they understand, and will do that. We also said that we need to know when things change, because apparently they made the decision not to connect the rocket avionics to the separation charges almost 3 weeks ago, but we didn't learn about it until this Monday. This communication problem is really hurting us because we're making plans, and they are changing things without talking to us at all. They apologized but were unwilling to agree to connecting us.

So we're at a bit of a standstill. Our capstone project relies on hardware we have no control over, and communications with another team who are proving unreliable. We are trying to talk to our client Dr. Nancy Squires, but this week has been crazy and it hasn't happened yet.

7.1.10.2 Joshua Novak: plans Work on GUI, Network, Database, and hosting over break

Test and purchase hardware

Summary

We worked on and finished our progress report and design document.

7.1.10.3 Allison Sladek: :

Plans: Complete Presentation with the rest of the group.

Problems: Completed video was a little larger than expected, but there is room for it in the webspace.

Progress: Finished documents for the end of term.

Summary: Completed Fall progress report and Technical review documents. I edited the video together and removed awkward pauses.

7.1.11 Week 11

7.1.11.1 Levi Willmeth: Plans: Find or purchase a TeleMega to use while building the ground station. Finalize purchase order for ground station parts. Kevin McGrath suggested he may be able to help us with funding, because the TeleMega will not be used during the competition itself. Nancy Squires also offered to purchase the TeleMega, so I think we'll be covered one way or the other. I emailed Keith Packard asking for a total price including the 10% educational discount.

Progress: Asked about borrowing a TeleMega from the 100k team but it's not a good long-term solution.

Problems: No new problems this week. Just going through the work and making progress.

7.2 Winter Break

7.2.1 Levi Willmeth

Plans: Get as much done over winter break as possible!

Progress: I received the ground station parts and assembled the ground station using hot glue and some plastic sign boards to hold components in place. The yellow carry case is the perfect size and everything fits well.

I drafted the MPU6050 class in Python and started reading values. I want to compare Python against C or C++ to see if one language will be noticeably faster than another. One problem with the MPU6050 and I2C is that each MPU6050 can only respond to either 0x68 or 0x69. So we have 2 addresses and 3 sensors. We'll need to use the AD0 pin to isolate each sensor on a specific address before reading from it.

I drafted a nodejs script that listens to GET requests from the Pi 3, and responds with database queries. A request that uses the 'time' field will be given the rows ¿= that timestamp, which allows us to query the database for recent values without receiving all values. This is how we can generate dynamic graphs for multiple clients without hammering the db. The 'Source' field is the table name, but we might want to change that or add another field for callsign? This allows us to plot the BeelineGPS and Rocket_Avionics instruments as different lines on the graph. It will be very easy to add new responses for different types of graphs by looking for other values in the 'graph' parameter. It might even be possible to specify certain fields as parameters, but that becomes a lot more complicated.

Together with Allison and Joshua, we got our first dynamic graph working! I wrote a python script that generates GPS data on a parabolic altitude curve and inserts it into the database, so we can test the dynamic graph.

Problems: Working with time zones is hard! The process of sending the timestamp via JSON is converting it to zulu time, which then breaks when we try to query the database for the most recent values. I want to know what time format the BeelineGPS uses, because we cannot change that. I would prefer to store the raw BeelineGPS values in the db, so that we are not corrupting any later debugging steps.

7.2.2 Allison Sladek

7.3 Winter

7.3.1 Week 1

7.3.1.1 Levi Willmeth: Plans: Check in with ECE subteam, who were not in Corvallis during the break. Finally place order for TeleMega, which has been in administrative limbo for a few weeks now.
Progress: Made contact with the ECE's online, vague plans never materialized. I prompted Kevin about the TeleMega order, who prompted the ordering department, and the order was finally placed on 1/8/18
Problems: This was our first week back from winter break and we didn't accomplish much. I'm not sure why. I think we were all just getting back into school mode after the break.
7.3.1.2 Joshua Novak: Plans:
Get Display to have all graphs for live flight working with handlebars, get sensors
Start writing avionics
Problems
Sensors may not be available for us to use, ECE team does not currently have all of them available for us
Progress
Worked on GUI, Network, Database, and hosting over break
Test and purchase hardware
Database and hosting mostly complete, GUI has all the algorithms for the graphs but is not currently implemented into handlebars which we are using to make the design
Summary
Met briefly to plan ahead, decided on meeting time with TA
Put together slides for meeting, talked with ECEs about what sensors we will need to purchase for ourselves, decided to purchase sensors for testing Met and worked on Display, made plans to meet on Tuesday
7.3.1.3 Allison Sladek: :
Plans: Determine best meeting time for TA AIAA general update meeting next week
Problems: Website still not in a testable state
Progress: Continue to add pages for displaying graphs to the web UI Make pages pull data from the database
Summary: Added preliminary partials to the webpage Met with group members to discuss division and direction of our project for the term

7.3.2 Week 2

7.3.2.1 Levi Willmeth: Plans: Figure out exactly what components will be in the ECE PCB.
Progress: This week we improved the audio parser, haggled with the ECE subteam about the sensors, and ordered our own set of sensors to continue working on the avionics software. I set a deadline for a finalized parts list from the ECE's by week 5 of this term.

I keep checking the order status, but the TeleMega has not been shipped yet. I contacted them on their website on Jan 21.

I refactored some early parser code I wrote over winter break. It can now read a CSV file and insert it into the database, as well as read an audio stream. I'm updating the database schema and determining the best way to store the resulting text fields.

Problems: We really need to know which components will be used so that we can begin programming the avionics software. I told the ECE subteam that if they haven't decided by week 5, we will make our own assumptions about the components, so that we don't continue to fall behind. I'm hoping this will pressure them to decide, and if not, well, the CS capstone project has to move on even if the ECE capstone project stalls.

One potential problem with the parser is that reading an input stream from the subprocess can cause the parser to hang, according to the docs. I'm not sure if my implementation prevents this, or if it just hasn't happened yet, but it's a possibility. More testing is needed.

7.3.2.2 Joshua Novak: Plans:

Add velocity graph

Improve visuals

Start writing avionics

Problems

There may be an upper limit to the number of simultaneous connection, though it may be that their devices were noticing the network provided no internet and did not connect for that reason

Progress

Got Display to have all graphs for live flight working with handlebars, ordered our own sensors

Summary

We completed the alpha for the display and tested with the full AIAA team.

7.3.2.3 Allison Sladek: :

Progress: Web UI nearing alpha version

Problems: The stress test revealed that the pi can't currently provide enough connections for our team. We'll have to look into this further, and possibly augment our setup with different hardware.

Plans: Competition Update Due Jan 26

Summary Worked primarily on web UI Website stress test showed connection issues with large groups

7.3.3 Week 3

7.3.3.1 Levi Willmeth: Plans: Breadboard the MPU6050's and BMP180, run our first end-to-end test of parser using the actual BeelineGPS module and HAM radio.

Progress: I set up a breadboard with 3x MPU6050's and 1x BMP180, and updated my python code to read from all of them and write to a text file. Initial tests show 90% of the time is spent waiting for I/O overhead, but the program is still running at 90 Hz.

Successfully ran a full end-to-end test using HAM radio and BeelineGPS module. The test was done at the weekly meeting and worked well.

Problems: We need ECE cooperation to access the BeelineGPS and HAM radio. I'd like to bring a radio home for additional testing, but it can wait. The ECE's are reluctant to let us use the radios by ourselves.

We tried stress testing the ground station at the weekly meeting and only 15 people were able to use it. Some people were able to connect but not able to see the website. I think we're overloading the WiFi on the Pi?

7.3.3.2 Joshua Novak: Plans:

Add velocity graph

Improve visuals

Copy database online

Start writing avionics

Problems

The database will only be accessible at OSU. This will be a major complication.

Progress

We were able to get a space on the OSU databases to upload our database for easier use

Summary

We decided to implement our database online so that the code for the display could be run apart from the local network of the raspberry pi

7.3.3.3 Allison Sladek: :

Plans: Work on Queries and Avionics

Problems: Telemega hasn't arrived yet

Progress: Brushed up GUI css and javascript

Summary: Continue GUI work and transition to avionics programming

7.3.4 Week 4

7.3.4.1 Levi Willmeth: Plans: Prototype software to spin the brushless motor, using PWM signals to the ESC. Copy the database off our esra pi, onto our new OSU database. Back up the entire ESRA SD card.

Progress: Successfully set up a breadboard to connect the ESC, and wrote a very simple program to set the input range, then cycle through the range using 1/20 steps. I worked on the PID loop and added a conversion to the PWM values that control the motor.

It took several days to get an OSU database set up on the osclass server, but Kevin eventually made it happen. Once I got the credentials, exporting and importing our database tables and rows was easy.

We had our weekly ESRA meeting today and talked with the ECE subteam about sensors. I showed them my sensor breadboard and told them we absolutely need to use pins 18 and 33 for the two ESC's, and 2 and 3 for the I2C communications. Pins 18 and 33 have hardware support for PWM signals, and 2 and 3 are reserved for I2C connections. That's when we learned one of the ESC's is actually a DC motor, NOT a brushless. I'm still not sure how that's going to affect us. DC motors don't use a signal wire, so the ECE's need something to convert a PWM signal into a large amount of current at a much higher voltage. Today was the first time we heard about this.

I told the ECE's which pins we're using to toggle the MPU6050's and reiterated that we need GPIO pins for those, but told them they could select other pins as long as they do not interfere with 2, 3, 18, or 33.

I agreed to loan the ECE's my breadboard for their electronics capstone checkoff. I'll get it ready by Monday, and they'll use it for a day or two then return it.

Problems: A full backup copies the entire SD card including whitespace, which means every copy is 16 GB. I would like to find a way to convert this to an ISO or smaller copy.

On the ECE slides they posted the part number for the BMP180 replacement, which uses the I2C address 0x68. This is a mild problem because it's the same address as the MPU6050's. But we can handle it by pulling all 3 MPU6050's to 0x69, reading from the RTC, then going back to the toggle method we've been using. (read from 0x69, ignore 0x68.)

7.3.4.2 Joshua Novak: Plans:

Improve visuals

Start writing avionics

Problems

Drift on clock each second is greater than our unit of measure, a millisecond

Progress

Vertical velocity implemented

Database was uploaded

Altitude for various sources was implemented

Summary

Implemented vertical velocity on Sunday

Implemented general case for altitude on Tuesday

Looked up runtime clock drift, it is roughly 0.0004 seconds per second

We should expect to power an upper limit of 15 microamps

7.3.4.3 Allison Sladek: :

Plans: Change address to not require port number. Add more GUI queries Calculate Apogee Draft Poster Soon

Progress: Motor tested this week

Problems: There was some concern about a sensor that required 5v instead of the 3.5v that all other sensors needed in avionics. This would've required some additional resistors and external power to limit current/voltage drops, but after talking to the ECE team again, it seems that the sensor in question did in fact require the standard 3.5v.

Summary: Started looking into apogee detection methods to compare to commercial parts May use magnetometer, accelerometer, and gyroscope in conjunction Found simple fix for displaying port number on website address Web database has been set up and will make writing additional queries easier to test

7.3.5 Week 5

7.3.5.1 Levi Willmeth: Plans:Compare sensor code in Python to C. We will compare the real and process loop times in python and c, to see if there's a significant difference. If the loop times are similar (which we expect), then we will proceed with writing the avionics in python.

Progress: Our tests show virtually no difference between C and Python. Both languages spend 90% clock time waiting for the I2C bus, so the rest of our program execution is largely insignificant. The project is hardware limited by the I2C bus.

Problems: We need to rewrite my Python sensor code in C.

At our weekly meeting, we heard the ECE's have ordered a replacement electronic speed controller (ESC) for the reaction wheel. It is unclear if that means they are replacing the DC motor with a brushless motor. A DC motor cannot use an ESC, so we're hearing conflicting answers. This affects the code that we use to control the motor.

7.3.5.2 Joshua Novak: Plans:

Improve visuals

Continue writing avionics

Problems

The bus is extremely slow, so we will work with python as there is no significant speed difference

There doesn't appear to be a way to reduce the size of our iso image

Progress

Wrote code for magnetometer and new altimeter

Started writing progress report, created rough draft of poster

We were able to make an image of our SD card, though it is quite large

Summary

Discussed bus as primary speed issue for sampling

Tested C vs Python, concluded that the bus will be the most significant factor and computation time will not be a major issue

Started writing progress report, created rough draft of poster

7.3.5.3 Allison Sladek: :

Plans: Complete poster for next week, Work on progress report for next week

Progress: Started progress report for capstone, Working on a solution for the online database

Problems: Minor roadblocks with database access solved with online database implementation

Summary: Working on integrating .config files into a singular format, Working on apogee detection, Working on small scale rocket, model Working on online database integration, Documentation for class in progress

7.3.6 Week 6

7.3.6.1 Levi Willmeth: Plans: Find a way to shrink our SD card backup image size. Try using mariadb to store avionics results instead of a CSV file. Try using the ECE's big motor and ESC with our code.

Progress: I picked up the motor and ESC from the ECE team and looked into default ESC signal durations are, I believe it will expect 40 Hz because that's a standard servo duration. I'm able to spin MY brushless ESC and motor with my avionics code, but the ECE model does not respond well.

I (mostly) implemented a comparison of the avionics program writing to a mysql database vs a csv file. Right now the CSV method is marginally faster, but I'm looking for speedups like batch processing, in my mysql code.

Problems: Right now every copy of our SD card takes as much space as the SD card, 16 GB per copy. This particular ESC uses a 5v logic symbol and 40 Hz signal length. The pi outputs 3.3v logic and some standard PWM servo signal length. Will they match up? We'll find out!

7.3.6.2 Joshua Novak: Plans:

Improve visuals

Continue writing avionics

Test Database implementation of avionics

Problems

Magnetometer was replaced with a new accelerometer

There may be issues communicating with the ECE team's motor due to differences in logic between the pi and the motor

Progress

Completed progress report

Summary

This week we mostly worked on the progress report.

7.3.6.3 Allison Sladek: :

Plans: Complete Documentation, Start using Github Issues

Progress: Completed Midterm report documentation, Wrote up pseudocode for apogee

Summary: This was a slow week for coding due to deadlines for the midterm progress report and poster due this week. Our group has signed up or is signing up for a poster critique session in a couple weeks (March 7th), and we have basic functionality implemented for the most part. See github issues for to-do list:

https://github.com/OregonStateRocketry/30k2018-CS-Capstone/issues

7.3.7 Week 7

7.3.7.1 Levi Willmeth: Plans: Finish testing the ECE propeller motor with my code.

Progress: Ben said we were the only group that followed the midterm submission guidelines. (Yay!)

This week I refactored the avionics code to write to a mysql database on the payload, instead of a csv file. This was a suggestion from our TA Ben, who thought it might run faster and give us some additional features such as easy export/import back on the ground.

Mysql is really nice in some ways. It can simplify problems with the timestamp. It can make exporting/importing data really easy. It is not faster than a CSV file.

Mysql runs at about 1000/13.074 = 76.5 Hz File.write() runs at about 1000/11.361 = 88 Hz

Is that a huge difference? Kinda, maybe. 12 Hz sounds like a lot to me, and these conditions already favor mysql. Using file.write with a larger buffer may widen the gap.

Problems: We should talk about how long we're comfortable holding data before committing it.

Today I met with the ECE subteam to troubleshoot the motor+ESC. The problem is that it's not responding smoothly to the PWM signal. It jerks and spins (kinda) but not well enough. The ECE subteam met with me, looked over my wiring and parameters, said they were ok, then used their own program, and got the same behavior. They weren't sure what the problem is, so they took it back to the ECE lab to troubleshoot.

7.3.7.2 Joshua Novak: Plans:

Test avionics on dummy payload

Improve visuals

Continue writing avionics

Problems

All accelerometers will be replaced with new accelerometer

Progress

Tested and finished implementation of altitude vs time with varied sources

Implemented and tested version of outputting sensor info to database rather than csv

Summary

Tested Database implementation of avionics, found it was slightly slower than outputing to csv, but would keep data intact, will likely go with outputting to csv for speed reasons, but may go with database implementation

7.3.7.3 Allison Sladek: :

Plans: Re-implement flight summary for GUI, Consolidate .config files for database, Write apogee detection function,

Upcoming Presentation for mechE capstone

Progress: Working on GUI fixes this week

Problems: Some issues with database connection were solved by double checking variable names

Summary: This week I found that some features implemented for the GUI were never pushed to github and have since been overwritten. I've been re-writing the flight summary page and working on a revised query to return relevant information for a given flight.

7.3.8 Week 8

7.3.8.1 Levi Willmeth: Plans: Write SQL query for flight summary page. Write new code for the new sensors. Write code to detect apogee.

Progress: I wrote a SQL query that calculates the data needed, but it needs to be converted into the node.js app.

Joshua added code for the pressure sensor but it needs to be refactored to fit the rest of the avionics API format and to run faster. Right now it has two unnecessary 50 ms delays that are just killing the sample rate.

Allison is working on the apogee detection algorithm. We worked together as a group to identify some conditions we expect the rocket to see at each state transition boundary.

Problems: This was a slow week for me personally. Or rather, I was busy with other assignments. I did write a pretty complicated mysql query to summarize a flight (min max altitude, start/end locations, etc) based on data logged during the flight. I'm pretty pleased with how it came out.

7.3.8.2 Joshua Novak: Plans:

Refactor Altimeter into more efficient format

Work on avionics

Improve visuals

Problems

Clock will likely need to be written from scratch, public libraries all require too many other libraries and programs

Altimeter runs too slowly

Progress

Performed refactorization of altimeter

Researched Clock code

Summary

I found some implementations of the clock and the altimeter online, but they were not particularly fast and the implementation of the clock involved installing way too many libraries.

7.3.8.3 Allison Sladek: Plans: Re-implement flight summary for GUI, Consolidate .config files for database, Write apogee detection function, Upcoming Presentation for mechE capstone

Progress: Continuing GUI fixes: query selection

Problems: Query for flight summary doesn't select on flight ID, which will have to be done either in the query, or in the javascript that displays the summary fields on the front end.

Summary: Continuing to work on github issues, especially the GUI fixes

7.3.9 Week 9

7.3.9.1 Levi Willmeth: Plans: Write SQL query for flight summary page, refactor MPU6050 into MPU9250, refactor MPL3115A2 code to fit our avionics API, write new code for PCF8523 altitude sensor.

Progress: I refactored the MPL3115A2 code and got it running faster without the extra delays. Joshua is working on the PCV8523 code.

I integrated my mysql summary query into the nodejs site, and made an improvement by selecting max and min altitude in a more efficient manner. There are probably other improvements to be made here as well.

Problems: We have a winter poster coming up soon and have been spending time working on that.

7.3.9.2 Joshua Novak: Plans:

Improve visuals on Display

Write tests for sensors

Problems

Magnetometer readings on the accelerometer only return zero. We will need to do more research into this if we plan on using the magnetometer, but we currently do not

Progress

All sensors implemented and demoed (Sunday)

Summary

Tested Accelerometer code to show it will work for new Accelerometer

Had peer review for poster

The accelerometer code is the same as the 6050, as we have no plans to use the additional magnetometer values

The altimeter is put into barometer mode so we can make sure that the sensor values it derives altitude from (temperature and pressure) are in a reasonable range during flight. The other modes do not return this, and toggling from one mode to another would introduce complexity and take more time. To receive values as quickly as possible we can toggle the sensor's control register to have it update its values on command, taking an amount of time based on the oversampling rate in the control register.

7.3.9.3 Allison Sladek: :

Plans: Re-implement flight summary for GUI, Consolidate .config files for database , Write apogee detection function, Upcoming Presentation for mechE capstone

Progress: Continuing GUI fixes: query selection

Problems: Query for flight summary doesn't select on flight ID, which will have to be done either in the query, or in the javascript that displays the summary fields on the front end.

Summary:

Continuing to work on github issues, especially the GUI fixes

7.3.10 Week 10

7.3.10.1 Levi Willmeth: Plans: Fix a timestamp problem caused by sampling at faster than 1 Hz.

Progress: This week I updated the BeelineGPS table to use decimal seconds. This was important because we're recording at 80 Hz so many samples had identical timestamps. I also learned that we can do datetime to string conversions in our select statements, instead of handling that in python or javascript. That might be a nice way to synchronize all of our conversions by doing them in the same place, with the same language, to get the same results.

I improved unit tests for some of the ground station files:

/git/Capstone/Database$ coverage report DWParser.py parser.py Mariadb.py

Name Stmts Miss Cover

———————————

DWParser.py 33 1 97%

Mariadb.py 58 3 95%

parser.py 97 25 74%

———————————

TOTAL 188 29 85%

Problems: Working with dates and times is a pain.

7.3.10.2 Joshua Novak: Plans:

Update visuals

Refactor parser to be easier to test

Problems

Our parser appears to be very difficult to test due to large number of uses that it needs to fulfill, as well as there being a number of checks for credentials and the like that are mostly superfluous and are just there to see if something is broken. It will either need to be refactored to be easier to test, or we will need to put a large amount of effort into writing tests for it that check all uses that it fulfills.

Progress

Wrote tests for avionics

Wrote R code for finding outlier data

Wrote a basic test for the parser's main function

Summary

Worked on Progress report, wrote a lot of tests as we had a requirement for 80% line coverage on much of our code that we had not satisfied since we thought that this would only be necessary by the end of the project, not the end of this term.

7.3.10.3 Allison Sladek: :

Plans: Re-implement about page for GUI, Look over poster again after critique session, Write up documentation for end of term

Progress: Continuing GUI fixes: About page and flight summary

Problems: Don't have the hardware to test avionics, but can test it when we get the other breadboard back from the EE team

Summary: Continuing to work on github issues, especially the GUI fixes and those needed before first practice launch.

7.4 Spring Break

7.4.1 Week 1

7.4.1.1 Levi Willmeth: Plans: Improve the database schema by replacing redundant text like callsign with a M:M relationship. Update database module to use a config file to make it easier to switch between test and production databases.

Progress: Refactoring the database schena to use more M:M relationships was easy. Updating every aspect of the project that broke because of this change was not. I just had to put my head down and power through one bug after another until the ground station worked. It took a solid 2 days of refactoring, but I think it was worth the effort.
I refactored and fixed many of the website queries broken when I overhauled the database schema. I found the website difficult to work with because it uses things like partial views and it's been a couple weeks since I've done anything with the website, but I got through it.
I also wrote an installer script which can run on a fresh rasbian lite image to install all of the necessary libraries to get the payload avionics program running.
I also modified the old unit tests to work with any file changes I needed to make, and added new unit tests for the mainPayload and payloadState modules. The latest coverage report is up to 93% coverage, but of course it still needs better tests:
pi@esra-p: /Payload $ coverage report -m ESCMotor.py MPL3115A2.py MPU9250.py PCF8523.py mainPayload.py payloadState.py
Name Stmts Miss Cover Missing

———————————————

ESCMotor.py 56 2 96% 98, 100
MPL3115A2.py 23 0 100%
MPU9250.py 63 0 100%
PCF8523.py 25 4 84% 20, 38-46
mainPayload.py 41 7 83% 5, 10, 49, 90-96, 107-108
payloadState.py 93 8 91% 10, 14, 19, 96-97, 225-228

———————————————

TOTAL 301 21 93%

Problems: On 3/28/18 I met with Dan He from the ECE team to pick up the new ESC and payload motor for testing. I asked how the payload PCB was going and he said there wasn't going to be a payload PCB. I asked what he meant, and he said there wasn't room, so they dropped it. I asked if that meant the sensors were changing and he asked me to post in the slack channel to discuss it.
In slack, I learned that the ECE team changed the sensors without talking with us (again) and we will no longer have access to an MPL. That means we won't know the altitude of the payload. That means our current apogee detection methods won't work, and that some of our capstone requirements will need to be updated.

7.5 Spring

7.5.1 Week 1

7.5.1.1 Levi Willmeth: Plans: Plan out remaining tasks.
Progress: We met and whiteboarded out the major remaining tasks. I added them as github issues so we can track and check them off as we go.
This week I designed and built the first-gen of the AIAA certification flight avionics PCB. The PCB connects a raspberry pi, MPU6050 and MPL3115A2, powered by a tiny 1S, 150 mAh lipo battery. (predicted run time ¡ 45 minutes) This will

fly in my personal model rocket and collect IMU and altitude data during the test launch on 4/20. The data can be used to test our assumptions about different stages of flight.

We collected the ECE's first-draft PCB to test with.

Problems: The ECE PCB only has a single MPU and the RTC. It's missing two MPU's and the altitude/pressure sensor. They are working on a new version.

7.5.1.2 Joshua Novak: Plans:

Work on State Machine

Add on hover to map

Add different symbols to map

Problems

Found bug in update functions for the graphs

Progress

Fixed bug, added color changes to map graph

Summary

Over Spring break state machine was added to implementation of avionics

New implementation for altimeter reading just altitude, as the altimeter's own method for finding altitude is accurate to within a few feet and the reading will not vary much with temperature. This simplifies the implementation, which is preferable

7.5.1.3 Allison Sladek: :

Plans: Test launch coming up in the next two months.

Progress: Website working.

Problems: Need integration tests soon.

Summary: It's the first week back from break and we're working on improving test cases and the website theme.

7.5.2 Week 2

7.5.2.1 Levi Willmeth: Plans: We need to do a full integration test before the practice launch. That means we need the completed payload, with motors and batteries, in it's final flight configuration. The practice flight is on 4/20 and we need at LEAST 1 week between our integration test, and the flight, to ensure we have time to fix any problems.

Progress: I met with the ECE team but it didn't go so well, see below.

Problems: It was a rough week. I met with Joshua Novak, and the ECE and Payload teams to test the reaction wheel. I brought a pi zero but the wifi wasn't working, so I had to connect to it using usb otg. (mistake #1) Dan from the ECE team connected the motor driver to the raspberry pi incorrectly. (mistake #2) When he plugged in the battery, it fed 24V into the PCB, which fried the pi AND my laptop. After getting it home and looking at some repair docs, my the motherboard is toast. I might have had better luck if I had used the USB ports on the right hand side of the laptop instead, because that's a breakout board. Live and learn I guess. Dan offered to help pay for replacement parts and we agreed to split the cost.

7.5.2.2 Joshua Novak: Plans:

Work on State Machine

Problems

ECE team is being very inconsistent on what sensors will be available on the final hardware. This is worrying as we

need certain sensors to detect apogee and control the motors

Motor test with the ECE team ended with Levi's laptop being fried, though they offered to help him pay for repairs

Progress

Added on hover and different symbols for different callsigns to map

Summary

Mostly worked on graphs as of late

Thought about how to implement apogee check

7.5.2.3 Allison Sladek: :

Plans: Hoping to test sensors during upcoming certification launch.

Progress: Working through github issues.

Summary: Revisiting our github issues this week.

7.5.3 Week 3

7.5.3.1 Levi Willmeth: Plans: On 4-21-18 we learned that the ECE sensor hardware has changed (again) and we need to either refactor, or ignore the new hardware. Solving this emergency became our new plan for the week.

Progress: Today I met with Joshua and Allison to talk about our current status and plans. We printed out and went over the payload state machine's transition boundaries and decided to make some changes, based on conversations with Dr Squires.

Problems: During the weekly team meeting we learned the ECE team has re-added 3 MPU9250's to the payload. 2 are oriented in the same direction and the third is rotated 180 degrees. So x=x, y=-z, z=y.

This is a bit of a problem for us because we have not planned for rotating only one of the IMU's. All of our discussions with the ECE's have assumed that all three IMU's would have the same orientation.

We will need to write something that allows each sensor to be oriented based on it's final launch configuration. Also, we need to add back the other 2 IMU's into the software, which means updating the DB and import scripts again, etc, etc.

7.5.3.2 Joshua Novak: Plans:

Work on state machine

Problems

Altitude data is very noisy, so using it to find apogee may require some changes

Progress

Made changes to new unit test suite to suit changes to MPL

Summary

I've been very concerned about the implementation of the state machine. I think that we can use altitude, but pressure spikes cause major issues. We are looking into doing a launch with the smaller rocket with just sensors to see how it will work.

7.5.3.3 Allison Sladek: :

Plans: Hoping to test sensors during upcoming certification launch and test launch on May 5th.

Progress: Updated vertical velocity display, working on poster for expo.

Problems: Still need an end-to-end test of sensors, transmitters, receivers, database, and website.

Summary: Working on apogee state machine transitions and updates to website.

7.5.4 Week 4

7.5.4.1 Levi Willmeth: Plans: Get our code working on the latest flight hardware, update our state machine, be ready for a practice launch.

Progress: This week I updated the payload state machine transitions, based on notes from our last group meeting. The biggest change was looking at how we move from SecondaryEnginePhase into ExperimentPhase. I don't like the idea of using acceleration here because if the payload is spinning quickly, we may see too many G's on the net acceleration. We COULD look only at the Z axis but there's no guarantee it will roll over exactly at apogee.

I also pushed new code for converting axis and axis directions, but because I haven't been able to test it with a breadboard yet, I pushed it to a branch called levi/fixOrientation.

We also had our first full integration test, combining all of the rocket components into a single physical shape. There wasn't a lot for the CS subteam to do but I did get to see how the various pieces fit together, and clarify a few things with the ECE team.

Problems: We need the flight hardware again to test the latest code on the flight hardware. It's been challenging to synchronize with the ECE team and we're struggling to find a time that works for everyone. They need the flight hardware, and so do we.

7.5.4.2 Joshua Novak: Plans:

Work on state machine

Finish orientation script

Work on tests

Problems

Payload ladder will have MPU's in varied directions, this will cause issues as we will have to map from that orientation to another, meaning we will have to do this uniquely for each sensor

The offsets for MPU's vary by series, and some of the sensors we are working with for testing are of a different series than the ones on the payload or rocket

Progress

Fixed a bug in the display where it was displaying the wrong time format on graphs

Summary

Worked on poster and turning in document revisions to be approved by Nancy

Added a velocity check to apogee detection on the state machine

Added a simulator tool for the payload

Worked on tool for calibrating the accelerometers

Had a nervous breakdown over stress from the project, I am getting help but I know this is going to be a difficult few weeks going forward

7.5.4.3 Allison Sladek: :

Plans: Test launch moved to May 12th and 13th with an integration test this weekend

Progress: Finished expo poster, revised outdated documentation

Problems: Still haven't done an end-to end test

Summary: Spent a lot of time this week on updating our poster and capstone documentation. We still have to do some testing before the practice launch

7.5.5 Week 5

7.5.5.1 Levi Willmeth: Plans: Tie up some loose ends, Progress: This week I finished updating the CSV import program to work with the latest database schema.

I also cleaned up some old github issues that were finished but never closed online. Problems:

7.5.5.2 Joshua Novak: Plans:

Test software on ECE hardware

Problems

Some of the hardware has issues with disconnecting randomly, Levi is assisting with debugging the wiring and solving this issue

Progress

Finalized test suite

Finalized calibration script

Added cubic and square implementation of R scripts

Finalized state machines

Summary

Worked on our midterm report

Did a lot of work on the state machines and the tests, finalizing both

Added a filter for out of bounds values to the altimeter, as I noticed it behaved strangely for values less than 0, which could cause a false positive for apogee detection.

Oriented and calibrated the rocket

Added an offset to the altimeter so that we can set the base altitude before launch to some value Added graphs for acceleration and temperature

7.5.5.3 Allison Sladek: :

Plans: Getting ready for test launch next weekend

Progress: Display graphics changed

Problems: Still haven't done an end-to end test

Summary: Display theme has been changed to a dark theme similar to our meeting slides. Comments have been improved throughout my sections of the website. Working on capstone documentation.

7.5.6 Week 6

7.5.6.1 Levi Willmeth: Plans: Progress: Problems:

7.5.6.2 Joshua Novak: Plans:

Test launch this weekend

Problems

Bug in parser where it uploads a positive value for longitude

Motors were damaged during integration testing, so they will not be running during the launch

Progress

Fixed bug in parser

All software ready for launch!

Summary

Prepped all software to work on ECE hardware during launch

Setup SD cards to use one ECE hardware during the launch

Setup SD cards for parsers and server/monitor during launch

Set all scripts to run on boot.

7.5.6.3 Allison Sladek: :

Plans: Test launch this weekend

Progress: Made our code freeze commit this week.

Problems: Trouble with ordering the needed SD cards before our practice launch, but we should have enough for the weekend.

Summary: Finished the progress report last weekend. This week was spent preparing for our upcoming test launch. Hope it goes well!

7.5.7 Week 7

7.5.7.1 Levi Willmeth: Plans: Reflect on the test launch, decide what went well, what needs work.

Progress: The rocket avionics performed beautifully. The state machine and transitions worked as expected and we can see that it moved through all 6 stages at reasonable intervals. We assume that wherever the payload is, it probably went through the same steps because it uses the same algorithm and sensor code. However, because it was not recovered, we don't know for sure.

The telemetry parsing system worked well and we were able to successfully track the rocket from launch to it's landing location. The landing area was farther away than we anticipated and it ended up leaving the visible map. We would like to fix this bug before Spaceport.

Problems: Before launch, we registered both the rocket and payload telemetry on the launch pad. The altitude and locations looked good. At launch, we continued receiving telemetry from both rocket and payload for 45 seconds. However, while the rocket climbed and fell, the payload still showed it's location as being on the launch pad, unmoved. We believe it suffered from some sort of hardware failure on launch, but it's unclear what happened. One situation that could cause the result we saw, is if the BeelineGPS telemetry transmitter lost it's satellite lock, possibly by losing it's GPS antenna. The payload was successfully ejected but never recovered, so we can't learn what went wrong.

7.5.7.2 Joshua Novak: Plans:

Setup cameras for new payload

Problems

Payload was lost, new implementation is mostly pi cameras

Display website is hammering the database, likely due to the map trying to get an update each second

Offset script in MPL has an issue because the first value generated by the MPL is always 0, and the first attempt to get a value is done by the offset script

Progress

Got cameras

Identified root of database issue

Summary

EXPO

Fixed an issue with the summary script so that it now displays all sources

Confirmed scripts for payload would have activated after falling 300 feet from apogee, meaning that had the payload not been lost and had there been no electronics failing or failing of the motors, the experiment would have been a success

7.5.7.3 Allison Sladek: :

Plans: Start new avionics for replacement payload soon

Progress: Our first test launch this weekend was a success. The rocket was recovered in great shape.

Problems: The pi was not able to support the database and serve the website at the same time with large amounts of data in the database. This weekend, we moved the database over to Levi's laptop, where it was able to display graphs throughout the test flight.

Summary: The test flight this weekend was a success overall. The rocket recovery went well and our website displayed the exact location of the rocket. However, the payload was lost and a new cheaper replacement will have to be designed. Expo preparation was this week.

7.5.8 Week 8

7.5.8.1 Levi Willmeth: Plans: Tidy up the github repository, clean up our comments and outstanding issues. Figure out what we'll be using for a payload during the Spaceport competition.

Progress: Because the payload was completely lost, we won't be able to use it during the Spaceport competition. That's sad because we worked really hard on it, but there just isn't enough time (or money) to rebuild it before the end of the term. This is a consequence of waiting too long for a practice launch.

The payload and ECE teams got together and announced they will build a new payload with telemetry but no sensors. Instead, it will include two raspberry pi cameras. The CS team was not involved in this decision and did not even find out about the discussion until the regular weekly meeting.

Problems: At least half of our project just got cut from the competition.

7.5.8.2 Joshua Novak: Plans:

Come up with ideas for payload cameras

work on final documentation

Problems

With the implementation planned by the ECE team and Payload, there is likely not much we can do that would be interesting

Progress

Got cameras from Kevin to use to setup camera scripts

Summary

Brainstormed ideas for payload cameras

Helped as much as I could with the documentation for the team

Went to ME final presentation and emphasized importance of communication

7.5.8.3 Allison Sladek: :

Plans: Look into ways to use an NIR and an RGB pi camera on the new payload

Problems: Lost old payload, and looking into new uses for one with cameras as the primary sensor.

Summary: This week was pretty quiet. There wasn't much to do immediately after expo. We'll have to start on more capstone documentation soon.

7.5.9 Week 9

7.5.9.1 Levi Willmeth: Plans: Prepare an SD card for each of the new payload flight computers, that runs on boot and records a video for 1 hour.

Progress: Because the payload has been dramatically simplified, there aren't many remaining pieces to work on. Telemetry worked well during our practice launch. The rocket avionics worked perfectly. The payload would probably have needed some debugging but since it's gone, there is no way to know exactly what needs to be fixed, or test any fixes we made.

I finished making two SD cards for the payload.

Problems: Boredom.

7.5.9.2 Joshua Novak: Plans:

Talked to my brother about a video expert, will contact him further to figure out what advice he has Work on final documentation

Test how cameras will react to spin

Problems

Cameras appear to not be particularly well aligned, this will make any useful editing impossible

7.5.9.3 Allison Sladek: :

Plans: Work on capstone documentation. Record videos next week. Progress: Outline capstone documentation

Problems: Need to set up a camera spin test for the new payload.

Summary: Our subteam met up to divide up the workload for the final presentation and report this week. I'll edit the video together next week.

7.5.10 Week 10

7.5.10.1 Levi Willmeth: Plans: Perform a spin test to find the best camera settings to use at Spaceport.

Progress: There are only two good options: 1080 @ 30 fps, or 720 @ 60 fps. After doing some rotations and looking at the video, I believe 1080 will not look good while the rocket is spinning.

Problems: None.

7.5.10.2 Joshua Novak: Plans:

Finish Documents

Contact video guy

Spin test cameras

Implement some fix to database hammering (or keep update disabled, will work in either case)

Go to competition

Progress

Presentation was completed fairly easily

7.5.10.3 Allison Sladek: :

Plans: Finish final report next week and prepare for final integration and competition.

Progress: Finished final presentation video this week.

Summary: This week was mostly spent on recording and editing the final presentation. Our final integration will be next Wednesday, and then the competition will be the week after finals. Hope it goes well!

8 Final Poster

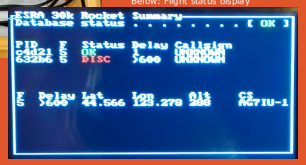
Our Mission

Our goal for the Spaceport America Cup is to learn and experience working together as a team to build and launch a rocket to 30,000 ft.

The rocket will also carry a 10 lb scientific payload which will be ejected at apogee. This payload carries a microgravity experiment designed by local high schools, and will use a propeller and reaction wheel to accelerate downwards for 10 seconds to create an extended microgravity environment.

Above: The ground station box

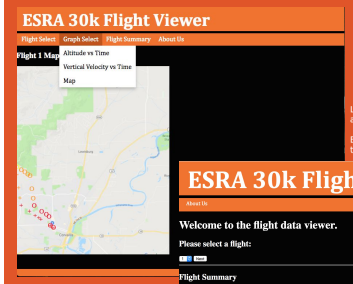
Below: Flight status display



Ground Station and Telemetry

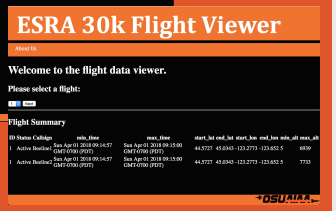
Live telemetry from the rocket will be received and parsed by several affordable micro computers in our ground station.

Our WiFi network and web page allow users to track the rocket and preview experimental results in nearly real-time. This data is further improved by importing sensor logs after recovery.



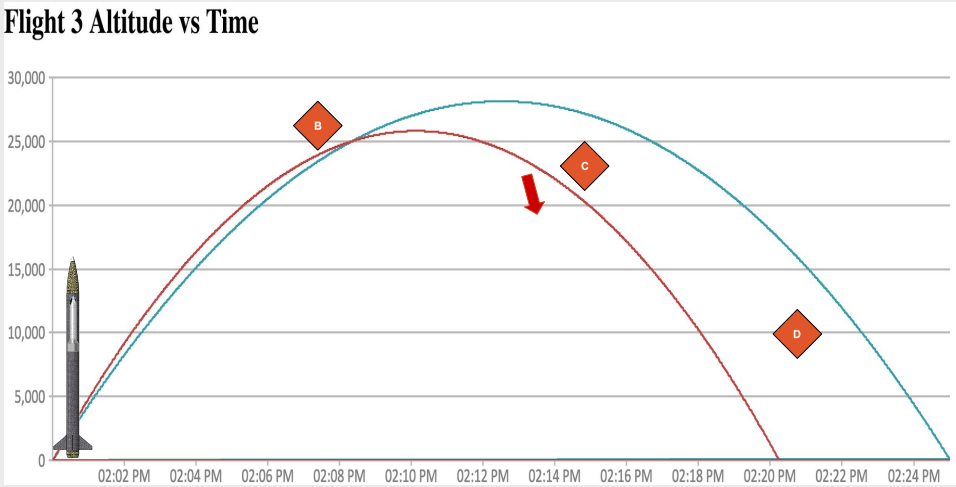
Left: View of the live map and graph selection

Below: Home screen of the flight viewer



Oregon State University

30K Spaceport America Cup



Avionics Flight Stages

- **A** Ready to Launch, the system conserves power and waits for high acceleration, then a coast phase after main engine burn.
- **B** At apogee the rocket splits apart. The rocket nose cone and tail remain connected, while the experimental payload is ejected.
- **C** The main rocket ejects a drogue parachute and the payload begins accelerating toward the earth as the microgravity experiment begins.
- **D** When the experiment ends, the payload ejects a parachute. Both sections fall gently to earth and are recovered.

Recording Flight Data

- **E** Avionics in the rocket's nose cone record ~5,500 data points per second. These include three, 9 DOF inertial measurement units, temperature and barometric pressure sensors, and a precision clock to synchronize data between computers. In addition, live GPS position and altitude data are transmitted via radio.
- **F** Avionics in the payload record another ~4,500 data points per second, as well as control the rate of rotation and descent of the payload during the experiment. When the avionics are recovered, the data recording during flight will be imported and combined with the telemetry data to better understand the flight.

Team Members
(Pictured from left to right)
Joshua Novak
Levi Willmeth
Allison Sladek

Technical Advisor
Dr. Nancy Squires

Competition Sponsor
Experimental Sounding Rocket Association (ESRA)

About our team

We like to write software for things that go fast.

Specifically, we wrote flight avionics software for both the rocket and payload, as well as designed and built a ground system capable of receiving and displaying flight logs and live telemetry data.

The AIAA 30k team includes 18 senior capstone students studying fields including mechanical and electrical engineering, and computer science.

9 DOCUMENTATION

9.1 Overview of Structure

Our project consists of several major components. First are the avionics systems, which measure and record sensor data and control motors in the case of the payload avionics. This recorded data can later be uploaded to a databese. Second are the parsers, which parse incoming telemetry data and upload them to a database. Third is the database monitor, typically hosted on a Pi with a small LED screen. Finally there is the web server, which collects data from the database and serves a web application which can be viewed by users. In general, data is recorded from parsers or avionics systems, uploaded to the database, and served via the web application or the monitor.
There are a handful of other components somewhat outside of this structure. There are the unit test suites, which test the avionics and parser code. There are the two avionics simulation utilities, which take a csv input and simulate a flight against components of the avionics systems with some degree of randomization, outputting another csv. There is the calibration script, which calibrates the accelerometers for the avionics system for a specific orientation. And there is the R analysis code, which takes a csv input and plots best fit lines while checking for outlier data. The test suites interact with the avionics and parser systems as basic unit test suites. The simulation tools are typically run using csv outputs generated by the avionics systems, and are used to test changes to the avionics state machines. The calibration script runs a modified version of the accelerometer code and outputs a csv of calibration values for the accelerometers. Finally, the R code is tested against csv outputs from the avionics or parser systems, typically the avionics systems.

9.2 Avionics

9.2.1 Software Requirements

- Rasbian OS
- Avionics folder of https://github.com/OregonStateRocketry/30k2018-CS-Capstone/
- Coverage Library
- Python 3
- pymysql
- pyyaml
- pigpio
- python3-pip
- python3-smbus

9.2.2 Hardware Requirements

The avionics suite is designed to work for a specific set of hardware designed by the ESRA 30k ECE team. This hardware consists of a Raspberry pi and several sensors connected via I2C bus, specifically an MPL3115A2 altimeter, a PCF realtime clock, and three MPU9250 acclerometers connected on pins with gpio number 17, 22, and 27. Hardware details specific to each subsystem will be detailed in the related section.

9.2.3 Installation

After downloading the Avionics folder to a Raspberry Pi, navigate to the Avionics folder with an account which has root access and run the following commands.

- chmod +x installPayload.sh
- sudo ./installPayload.sh

This will install the software requirements listed below the coverage library. Next, install the coverage library with the command ”pip install coverage.”

9.2.4 Rocket Avionics

For the rocket, the orientation of the accelerometers is assumed have the sensors rotated a quarter turn counter clockwise about the x axis such that the positive y axis is mapped to the z axis.
To run the rocket avionics system, run the script mainrocket.py. This script runs in an endless loop. It will output a csv file with values for the various sensors as well as the time and state. The rocket goes through five states, corresponding to the numbers zero through four. Zero is the grounded state, which it remains in until a launch is detected. One is the launch state, which it remains in until acceleration forces on the rocket decrease to below one g. Two is the coasting state, which it remains in until apogee is detected. Three is the post-apogee state, which it remains in until the rocket lands. Four is the landed state, which upon reaching the pi shuts down.

9.2.5 Payload Avionics

For the payload, the orientation of the accelerometers on pins 22 and 27 is assumed have the sensors rotated a quarter turn counter clockwise about the x axis such that the positive y axis is mapped to the z axis. The accelerometer on pin 17 is assumed to be the same as the others, but rotated one half turn about the z axis. There should be an ESC and a DC motor connected to pins 18 and 13 respectively.
To run the payload avionics system, run the script mainpayload.py. This script runs in an endless loop. It will output a csv file with values for the various sensors, the current power for each motor, as well as the time and state. The payload goes through six states, corresponding to the numbers zero through five. Zero is the grounded state, which it remains in until a launch is detected. One is the launch state, which it remains in until acceleration forces on the rocket decrease to below one g. Two is the coasting state, which it remains in until apogee has been detected and the payload has fallen some three hundred feet below it. Three is the experimental state, which it remains in until the payload falls below ten thousand feet, the deployment of the drogue parachute is detected, or it has been in the experimental state for 12 seconds. Four is the post-experimental state, which it remains in until the payload lands. Five is the landed state, which upon reaching the pi shuts down.

9.2.6 Calibration

The calibration script assumes that there are three accelerometers connected and that all three have their z-axis pointed away from the ground, against the pull of gravity. The acclerometers should be on a level surface and should not be moving.
To run the calibration script, navigate to the folder Avionics/Calibration and run the script calibration.py. The script will generate a csv titled ”calibration_values.csv” which you can copy the calibration values from. For each sensor and type (acceleration and gyro) the vaules printed in order are the x value, y value, then the z value.

9.2.7 Unit Testing

In the Avionics folder, use the command ”coverage run unitTests.py.” By using the command ”coverage report” you can see the full report of the coverage for each library, including those not written by our team.

9.2.8 Simulation Tools

First you will need to construct an input csv. For this, create a file named input.csv with data in each column as follows. The first three columns should be the x, y, and z values of the acceleration. The next three should be the x, y, and z values of the gyro. The next should be altitude. Each column should be of the same length, and you should not have headers to each column, just data.

Now, edit the corresponding simulator you wish to use to have the correct settings. This will be either payloadSimulate.py or rocketSimulate.py. At the bottom of each script is function for if the script is an ”if name = main” function. In that there is a call to construct a Simulator object. The first argument is the sampling rate you are simulating, or the time between each datapoint. The next is the degree of randomness for your simulation, which for values zero to 100 randomizes zero to 100 percent of all inputs. The third argument is the initial state for your simulation. For details on what each state corresponds to, check the section for that avionics system.

Finally run the simualtion script you want to, either payloadSimulate.py or rocketSimulate.py. This will output a csv with the same formatting as input.csv, except that the csv will have an eigth column for the state corresponding to that datapoint.

9.3 Web Display

9.3.1 Software Requirements

- Node
- NPM
- Display folder at https://github.com/OregonStateRocketry/30k2018-CS-Capstone/

9.3.2 Installation and Running

After installing Node and NPM, rename the config.example file to config.js and edit it to point to your database. This is explained in the General Instructions section. Now, open a terminal and navigate to the Display folder. Run the following commands.

- npm install
- node app.js

Now, open a browser on the same computer to localhost:3000, or if on a computer other than the server but on the same network to ”servername”:3000

9.3.3 Using the Web Display

After opening a browser to the index as explained above, use of the web display is very user friendly. First, select the flight number you are interested in looking at. A brief summary is on the index page, which can assist in making this decision. Next, select which graph you are interested in looking at. This can be the map, the altitude, the velocity, the acceleration, the summary, or the temperature. You will then find yourself at the corresponding graph for that flight. You can open a drop down at the top of any page to switch your flight or graph. There is also a link to a page describing the team and their goal.

9.4 R Scripts

9.4.1 Software Requirements

- R
- Display folder at https://github.com/OregonStateRocketry/30k2018-CS-Capstone/

9.4.2 Instructions

First you will need your csv for input. This should be two columns, one for your variable of interest, and the other for time. The variable can be a measure of acceleration, temperature, or altitude. The time column should have a heading "time." The variable column for acceleration, temperature, or altitude should have a heading of Accel, Temp, or Alt respectively.

Open a session of R and select to open and run a script. Navigate to the Display/R folder and select the script that you desire to run. They are all fairly self descriptive, including the variable of interest and the type of fit line. If no fit line type is specified, it is a linear fit line. Run the script. When prompted to select an input, use the input csv generated above. The session of R should now have a list of the outlier points and a summary of the atributes of the best fit line.

9.5 Parsers

9.5.1 Software Requirements

- Rasbian OS
- pyyaml
- pymysql
- Direwolf
- Database folder at https://github.com/OregonStateRocketry/30k2018-CS-Capstone/

9.5.2 Installation as a Service

Rename config.yml.example to config.yml and edit to point to your own database as described in the General Instructions section.

Edit three fields in the file esra-parser.service inside of the Database folder. The User field should be equal to a user with root access. The WorkingDirectory field should be the base of the Database folder, wherever you have saved it. The second value of the ExecStart field should be the full name including directories of the parser.py script, which is inside of the Database folder. Now, move this file to the folder /etc/systemd/system/ and run the following commands.

- sudo systemctl start esra-parser
- sudo systemctl enable esra-parser

Now the parser will run on boot and attempt to parse incoming signals.

9.5.3 Installing DireWolf

On one of the parsers, input the following commands.

- cd
- git clone https://www.github.com/wb2osz/direwolf
- cd direwolf

- make
- sudo make install
- make install-conf

Now that Direwolf is installed, you need to configure the settings. Follow the DireWolf User Manual at https://github.com/wb2osz/direwolf/blob/master/doc/User-Guide.pdf to configure your audio device.

9.6 Monitor

9.6.1 Software Requirements

- Rasbian OS
- pyyaml
- pymysql
- Database folder at https://github.com/OregonStateRocketry/30k2018-CS-Capstone/

9.6.2 Installation as a Service

Rename config.yml.example to config.yml and edit to point to your own database as described in the General Instructions section.
Edit three fields in the file esra-monitor.service inside of the Database folder. The User field should be equal to a user with root access. The WorkingDirectory field should be the base of the Database folder, wherever you have saved it. The second value of the ExecStart field should be the full name including directories of the monitor.py script, which is inside of the Database folder. Now, move this file to the folder /etc/systemd/system/ and run the following commands.

- sudo systemctl start esra-monitor
- sudo systemctl enable esra-monitor

Now the monitor will run on boot and attempt to monitor incoming signals.

9.7 Database

9.7.1 Software Requirements

- Rasbian OS
- pyyaml
- pymysql
- Database folder at https://github.com/OregonStateRocketry/30k2018-CS-Capstone/

9.7.2 Create Tables

After installing the mariadb server and client, run ”$ mysql -h username@host -p dbname <create_db.sql”

9.7.3 Delete Tables

To reset or drop the database, run ”$ mysql -h username@host -p dbname <drop_db.sql”

9.7.4 Upload CSV Data to Database

Rename config.yml.example to config.yml and edit to point to your own database as described in the General Instructions section.

Move the csv output from either the rocket or payload to the Database folder. Do not rename it. A rocket output should be named ”av_rocket.csv,” a payload output should be named ”av_payload.csv.” Run the script importCSV.py.

9.8 General Instructions

9.8.1 Edit Database Config Files

The Host field should be set to the address of wherever the database is being hosted. For a database hosted locally, use localhost. User should be set to a user with access to all necessary tables of the database. Pass or Password should be set to the password for this user. DB or database should be the name of the database.

9.8.2 Running Scripts at Boot

Edit the file /etc/rc.local to include a call to run your script just above the line reading ”exit 0.” This will make the script run as soon as it turns on. Some warnings, it will run in the background, so it cannot output to the terminal, and it will run at the root directory, though it will import local libraries if necessary.

9.8.3 Cloning SD Cards

In Linux:

Without the SD card, on a host machine run df -h to see normal drives.

Insert the SD card into a USB reader, insert into the host and run df -h again.

Notice there are new lines indicating your SD card, probably something like /dev/sdb or /dev/sdb1 and /dev/sdb2.

It’s normal to have two or more lines if the SD card is already partitioned.

Copy the entire SD card into a file with sudo dd if=/dev/sdb of= /Downloads/sd_card_bu.img

The dd tool gives no progress bar or feedback, and you should understand it will take awhile because it copies the ENTIRE SD card, even blank space.

So if you have a 16 GB SD card, your image will be 16 GB.

Remove the original (source) SD card and insert the new (target) SD card.

Unmount the new SD card before writing to it:

```
sudo umount /dev/sdb1
sudo umount /dev/sdb2
Write the image onto the target SD card:
```

sudo dd bs=4M if= /Downloads/sd_card_bu.img of=/dev/sdb

Verify the write is complete:

```
sudo sync
```

9.8.4 Set Remote Time From Local Time

```
ssh pi@p1.local sudo date -s @`( date -u +"%s" )`
```

(Warning, use ssh-copy-id first, or the timestamp will be off by as long as it takes you to enter your ssh password!)

9.8.5 Clear SD Card Partitions

Find your drive with: fdisk -l

Add/remove/change disk partitions: sudo fdisk /dev/mmcblk0

10 Recommended Technical Resources for Learning More

10.1 Generally Useful Urls

2018 ESRA Team Website

https://oregonstaterocketry.github.io/30k2018/

Spaceport America Cup home

http://www.soundingrocket.org/sa-cup-home.html

Our project documents and reports

https://github.com/OregonStateRocketry/30k2018-CS-Capstone/tree/master/Documentation/Assignments

Some usage and setup instructions

https://github.com/OregonStateRocketry/30k2018-CS-Capstone/tree/master/Documentation

10.2 Avionics

Control GPIO pins using pigpio

http://abyz.me.uk/rpi/pigpio/python.html

Component Data Sheets

https://github.com/OregonStateRocketry/30k2018-CS-Capstone/tree/master/Documentation/Components

10.3 Telemetry Parsing

Beeline GPS User's Guide

https://aiaaocrocketry.org/AIAAOCRocketryDocs/SLI2011-2012/Manuals/beelineGPS_13.pdf

TeleMega Documentation

https://altusmetrum.org/TeleMega/

AX.25 Amateur Packet-Radio Link-Layer Protocol

https://www.tapr.org/pub_ax25.html

Direwolf User Guide

https://github.com/wb2osz/direwolf/blob/master/doc/User-Guide.pdf

10.4 Ground Station

Documentation for Curses on display LCD

https://docs.python.org/3/library/curses.html

Tips for using Curses

http://gnosis.cx/publish/programming/charming_python_6.html

PyMySQL documentation

https://pymysql.readthedocs.io/en/latest/

11 Conclusions and Reflections

11.1 Levi Willmeth

11.1.1 Technical Information Learned

For me, the greatest technical challenges came from designing the telemetry parser, and avionics programs.

The telemetry parser takes an audio signal and converts it into strings using the APRS protocol. This was challenging because it required learning an open source tool called Direwolf to analyze and interpret the audio signal. Then I needed to validate the resulting strings using a checksum and by comparing the fields, before inserted them into the database. The combination of steps required thinking and planning ahead to prevent problems such as what to do with an audio signal if the database cannot be reached, or if the resulting strings cannot be validated. I wanted to use the same database class for the parser, csv import script, monitor program, and website, so I spent a lot of effort writing a useful database API.

The avionics program also had several interesting technical challenges. We had no control over the final hardware selection, which led to multiple rewrites as the flight hardware changed several times throughout the year. While outlining the avionics program, I wrote each hardware component as an individual class and this made it easier to swap out one component for another, as long as we used similar functions to return the data.

Working with the motors was another interesting task. The project used a brushless motor to spin the propeller, but a brushed motor to spin the counterweight. This meant that both motors needed to use a similar PID loop to prevent overshooting the optimal speed, but the methods for controlling each motor was very different. I solved this problem by writing the motor class using the same function names for both types of motor, and handling the details internally. This made it possible to use a single PID class for both of these very different motors.

11.1.2 Non-Technical Information Learned

I'm finishing this project with a better understanding of what it takes to build an entire rocket from scratch. As part of this project I was able to attend several assembly and fabrication meetings in which I helped mix and pack solid rocket fuel, watched the fabrication of carbon fiber and fiberglass components, observed the development of 3d printed ladders for electronics, and formation of all manner of attachment points and rigging. I listened to problems caused by insufficient lubrication on cotter pins, frustrations on how inaccurate sheer pins can be, and watched two rocket motors turn into firey balls of metal shrapnel and smoke. None of these will help me write better software, but they do contribute to my understanding of what it takes to build and launch a rocket to 30,000 feet.

11.1.3 Project Work Lessons

We began this project with a goal of under promising and over delivering. Thanks to speaking with our technical advisor Behnam Saeedi and graduating students, we had a good idea of what to expect for a project timeline. We knew that removing features would be more difficult than adding new ones, so our initial documents outlined a fairly minimal set of expectations that we hoped would satisfy all of our goals and leave room for expansion as the project developed.

In general, we did a good job of starting early and taking advantage of the winter break to make significant progress on the project. By the time winter term began we had a functioning parser, database, and had prototyped sensor breadboards and were beginning to work on avionics code. I believe the website was on it's way as well.

Which is good, because at some point we began to slow down. As I recall we were very productive through winter, then sort of stalled out sometime near spring term. I believe some of this was due to waiting until one piece was finished

to integrate it with another. We were also aware that flight sensor changes were in process and were waiting to find out which sensors would be available. This led to an uncertain work environment as we could not make that decision ourselves, but needed an answer before we could make progress.

In hindsight, I think that we should have done a better job making and following a Gantt chart, or some form of organized task scheduling. Something like an agile development method would probably work well for this type of project. We probably could have saved ourselves a few late nights by planning ahead and more evenly distributing tasks during each sprint.

11.1.4 Project Management Lessons

We made several important decisions at the beginning of the project, without fully understanding what the ramifications would be. One such decision was each of us claiming specific portions of the final project. I believe this is generally a good idea because each person can 'own' a specific task and work towards their goal more quickly without stepping on each other's toes. However, the weight of each task was difficult to judge so early in the project.

One project management lesson for me was that our initial decisions were not as set in stone as we treated them. We could have adjusted or added new areas of responsibility more freely. Indeed, I wish that we had used something like an agile development method where ownership could even change hands between sprints. That may have encouraged each of us to experiment with more of the total project, instead of remaining experts of certain domains.

I also believe that we could have done a better job of working together. I would have liked to have done more group code review sessions, but I recognize that it can be hard to find the time.

11.1.5 Teamwork Lessons

One of the greatest challenges throughout this project, for me, was working with the ECE subteam and agreeing on a final design. This led to two problems: a divide in subteam priorities, and a failure to build redundancy into the system. At the beginning of the project we met with the ECE team to discuss what types of sensors and telemetry would be used on the payload and rocket. I asked for redundancy in sensors and telemetry, and to use a telemetry transmitter with useful values. The ECE subteam was very resistant to discussing hardware components with us. They wanted to fly the same BeelineGPS unit with very limited telemetry data, that had failed the previous year, and did not want to consider alternatives.

I suggested compromising by using the BeelineGPS unit they wanted, as well as including a redundant TeleMega transmitter which had the extensive telemetry fields the CS subteam wanted. I was told they had already decided to fly a single BeelineGPS in the rocket, and another in the payload. They were not interested in redundancy, and I was the only one arguing for any sort of redundancy, so eventually I let it go. That was my mistake. I did bring it up with Dr Squires but I should have stayed with it, because I knew this was important and using a single telemetry module ended up costing us the payload during our test launch.

I bring this up because I believe there was a non-technical failure of communication between the CS and ECE subteams. The ECE subteam had complete control over the flight hardware, and the CS subteam's needs and concerns were largely marginalized and ignored.

11.1.6 What I Would Have Done Differently

I believe the ECE and CS subteams should have had regular meetings with our mutual mentor Dr Squires, to ensure that both teams needs are reflected in the final project. There should have been a very clear set of expectations and

responsibilities between the two subteams, including deadlines for making hardware decisions, and a clear process for changing hardware. It was not enough for the subteams to resolve this ourselves, because there was no accountability when the agreements were not kept.

To be clear, both subteams bear some responsibility here. I realize the CS subteam may not understand the implications of certain decisions, and that unexpected developments may necessitate changing flight hardware with little notice. However, the ECE subteam should also be aware the CS subteam has goals, priorities, and deadlines as well. The CS subteam could have done more to resolve this problem at the beginning of the project. Instead, the problem continued through the entire project, to the detriment of both subteams, and the project as a whole.

11.2 Joshua Novak

11.2.1 Technical Information Learned

I learned a small amount about a lot of different things that go into a rocket and a lot about how to talk to sensors through an I2C bus on a Raspberry Pi. The former was a lot more interesting to me personally than the latter. I learned a little bit about propellants and how they have grains and characterizations. I know our propellant is a solid one, and perhaps a bit too fast. This is important. Genuinely so. Because it burns so rapidly, we have an absurd degree of thrust force, which is necessary to propel our very heavy rocket. However, the thrust lasts for a very short duration, and has a tendency to chuff (read as ”burn inconsistently” or ”stop for a bit then go again”, it’s similar enough for the average person) under certain conditions. Also, the motor we have is absurdly large, the largest OSU has ever had. I learned how materials are layered with their fibers in different directions in order to strengthen the final product. I learned that toroidal parachutes are different from the typical spherical ones. I learned that a stratologger is an altimeter which determines altitude based on barometric pressure run through a Kalman filter, and can trigger flight events by sending a signal at apogee or a certain time after apogee. I learned you should not put an antenna in a metal box, and that antennas are an advanced sorcery. These are all very interesting lessons.

11.2.2 Non-Technical Information Learned

I learned the requirements for the ECE team are absurd and ought to be changed. Too much is put on too few people and it leads to them having issues getting everything done which they need to. These guys are putting together the electronics systems for a rocket and attending a competition which they intend to perform quite well at. They are working absurdly hard to accomplish that. Putting additional burdens on their shoulders to satisfy some arbitrary definition of what a senior project should be is ridiculous. They are building a rocket. That is hard enough. Anything they do as a requirement for their capstone which does not contribute to them building a better rocket is a waste of their energy.

I learned a bit about the business and regulations of rocketry, as most of the mechanical engineers are aiming for careers in aerospace and they handle a lot of interactions with sponsors. Rocketry is very expensive, and our team is on a far, far tighter budget than most teams attempting to accomplish the same goal.

11.2.3 Project Work Lessons

Aim low, stretch for more. If we had aimed as high as we had dreamed at the start, I don’t know if I would be here today. That being said, don’t be afraid to stretch. Often I was saying that something we were trying would wind up

being impossible, and I was often wrong. You have to have a realistic understanding of what you really can do with a project.

Pace yourself. I know that I ran into some issues with this, doing most of my work in short bursts, then moving on to work on other classes for a week before going back to working on the project. It led to me occasionally forgetting how to fix something, or leaving myself with too little time to put something together.

11.2.4 Project Management Lessons

Be flexible with responsibilities. Some people will show themselves to be better suited to handling one problem than another, and perhaps its best to just let them handle it. Don't use this as an excuse to do nothing, or to let anyone do nothing, but do understand that roles are going to change over the course of a project. Perhaps this can't be done all the time, but when you can, be flexible.

While this may not be true in general, for myself, working on a project with others at the same time was better than working individually. I feel very rewarded spending time solving problems alongside others. It helps keep up morale to do things together rather than alone in front of an isolated screen.

Spend time together on things other than the project. I learned this observing the ME teams, who worked together extremely well. I think that spending a lot of time doing fun activities together helped form a bond between them. While I don't feel our team failed to gel, I can really see the benefits of camaraderie.

11.2.5 Teamwork Lessons

Just about nothing is more important than communication. Nearly all our struggles through the year boiled down to communication. Minor defficiencies in communication grew into much larger problems than they ever needed to be. Assigning blame for these kinds of failures is also pointless, it's better to just find a solution.

11.2.6 What I Would Have Done Differently

I would have met with the ECE team more often. I would have met with the other members of my sub-team more often. I would have spent more time with my other team members at the AIAA lab. I would have gotten authorization to help out in the propulsion lab and helped them mix or pack propellant.

In general, I think the thing I would do the most differently is I would spend more time talking to the rest of the team and getting to know them better. They are all hardworking people who have accomplished an amazing thing in putting together this rocket. I wish I knew them all better than I do, not merely because I think it would have made the project easier and better, but because they are interesting people in their own right.

11.3 Allison Sladek

11.3.1 Technical Information Learned

Working on this project helped me get a better understanding of networking, NodeJS, databases, sensor control, LaTeX and IEEE documentation. A surprising portion of the class was spent on technical documentation and on using LaTeX.

11.3.2 Non-Technical Information Learned

Over the course of this project, I learned a lot more about rockets in general than I knew previously. Every week, I got updates on how the other subteams were designing and building their rocket subsystems from scratch. It was an interesting look at how other engineering disciplines go through their build processes.

11.3.3 Project Work Lessons

It's easy to get lost in a project of this scope and with a team of this size. It was interesting to see how our pieces fit in with the rest of the team, but it was also stressful until we knew it would work. I think incorporating testing into development earlier on would have added some peace of mind and made us more confident in our work.

11.3.4 Project Management Lessons

Good planning is very important. I did a lot of my work for this project in short bursts, and I wasn't always sure what had to be done until it was due very soon. As the project evolved, some pieces ended up being larger or smaller than expected, and it would've been more fair to redistribute the tasks among our subgroup. Having more frequent subteam meetings may have helped with workload redistribution.

11.3.5 Teamwork Lessons

This capstone group had a much larger team than most. Comprised of six subteams of three capstone members each plus undergrads, coordination itself was a massive undertaking. At the beginning of the project, our weekly team meetings seemed a little excessive, but by the end of the year it was clear that they did more to keep our team on track than any of the other steps we took. Communication was an issue at times, especially when we would only get updates pertaining to our subgroup at the weekly meetings. This mostly solidified the lesson of how important good communication is between people on a project together.

11.3.6 What I Would Have Done Differently

More frequent meetings between the computer science team and the electrical engineering team would've made the term go a lot smoother and eliminated some surprises on both sides. I think having a more defined schedule for testing and development (and following it) would have also improved the project. We had a rough dependency chart at the beginning of the project, but it was never updated.

12 Appendix 1: Essential Code Listings

12.1 Avionics Code

12.1.1 mainPayload program

These code segments are from the mainPayload program. This segment shows the translation used to reorient and calibrate each inertial measurement unit. Reorientation was necessary because one sensor was mounted in a different direction than the others, and calibration was necessary because of sensitivity differences in the sensors.

```
# Adjust orientation of certain payload sensors
orient_A = {
'gyro_x': ('gyro_x', -1),
'gyro_y': ('gyro_z', -1),
'gyro_z': ('gyro_y', -1),
'acc_x' : ('acc_x',  -1),
'acc_y' : ('acc_z',  -1),
'acc_z' : ('acc_y',  -1)
```

```
}
orient_BC = {
'gyro_x': ('gyro_x', 1),
'gyro_y': ('gyro_z', 1),
'gyro_z': ('gyro_y', -1),
'acc_x' : ('acc_x',  1),
'acc_y' : ('acc_z',  1),
'acc_z' : ('acc_y',  -1)
}

# Use the GPIO number, NOT the pin number!
self.mpuA = MPU9250.MPU9250(pi=self.piggy, gpio=17, orient=orient_A)
self.mpuB = MPU9250.MPU9250(pi=self.piggy, gpio=27, orient=orient_BC)
self.mpuC = MPU9250.MPU9250(pi=self.piggy, gpio=22, orient=orient_BC)

# Get these IMU values from the Calibration/Calibrate.py script
accel_17 = [ 4876, 6161, 8462]
accel_27 = [-7058, 6056, 9173]
accel_22 = [-3771, 7372, 9001]
gyro_17  = [   40,  -13,    9]
gyro_27  = [  -22,  -12,  -11]
gyro_22  = [  -90,  -21,   46]


# Apply calibration offsets into registers on the IMU
self.mpuA.set_accel_calibration(accel_17[0],accel_17[1],accel_17[2])
self.mpuA.set_gyro_calibration(gyro_17[0],gyro_17[1],gyro_17[2])
self.mpuB.set_accel_calibration(accel_27[0],accel_27[1],accel_27[2])
self.mpuB.set_gyro_calibration(gyro_27[0],gyro_27[1],gyro_27[2])
self.mpuC.set_accel_calibration(accel_22[0],accel_22[1],accel_22[2])
self.mpuC.set_gyro_calibration(gyro_22[0],gyro_22[1],gyro_22[2])
```

12.1.2 ESCMotor.py module

These code segments are from the ESCMotor.py module, which calculates and controls the motor speed of the payload's brushless and dc motors.

```
def pid_to_pwm(self, pid):
    '''Converts from one range of numbers to another'''
    pwm_min, pwm_max = self.ESC_MIN, self.ESC_MAX
    pid_min, pid_max = -self.Kp*2, self.Kp*2      # Are PID values always positive?
```

```
    if pid<pid_min: pid = pid_min
    if pid>pid_max: pid = pid_max

    pid_range = (pid_max - pid_min)
    pwm_range = (pwm_max - pwm_min)
    return int((((pid - pid_min) * pwm_range) / pid_range) + pwm_min)


def set_motor(self, speed):
    '''Set the motor speed to a PWM value'''
    if self.ESC_MIN <= speed <= self.ESC_MAX:
        # Two types of motors use this library:
        if self.TYPE_DC:     # DC motors
            # Remember: the PWM range is 0 to (MAX - MIN)
            # speed will be between MIN and MAX, so subtract MIN here
            self.pi.set_PWM_dutycycle(self.PIN, speed-self.ESC_MIN)
        else:                # Brushless motors w/ESC
            self.pi.set_servo_pulsewidth(self.PIN, speed)
        return True
    return False


def update_motor(self, current_value):
    '''Takes acceleration, updates PID and motor'''
    PID = self.update(current_value)
    PWM = self.pid_to_pwm(-PID)
    self.set_motor(PWM)
    return (PID, PWM)


def update(self, current_value):
    """
    Calculate PID output value for given reference input and feedback
    u(t) = K_p e(t) + K_i \int_{0}^{t} e(t)dt + K_d {de}/{dt}
    """
    # How wrong is our measurement?
    error = self.set_point - current_value
    delta_error = error - self.last_error
```

```
        # How long has it been since our last adjustment?
        current_time = time.time()
        delta_time = current_time - self.last_time

        self.PTerm = self.Kp * error

        # Prevent runaway I gain which causes overshoot
        self.ITerm += error * delta_time
        if self.ITerm < -self.windup_guard:
            self.ITerm = -self.windup_guard
        elif self.ITerm > self.windup_guard:
            self.ITerm = self.windup_guard

        self.DTerm = delta_error / delta_time

        # Save results for next loop
        self.last_time = current_time
        self.last_error = error

        return self.PTerm + self.Ki * self.ITerm + self.Kd * self.DTerm
```

12.1.3 payloadState.py module

This segment is from the payloadState.py module, which controls the boundary conditions between states. The payload uses this module to decide what happens during each phase of the rocket's flight, and to determine when to move to the next state. The payload uses 6 different states for pre-launch, primary engine burn, coast and apogee, experimental, descent, and finalizing logs before shutting down. For brevity, only the firsst two states are displayed here.

```
class State(object):
    def __init__(self):
        self.stateNum = None

    def monitorPhase(self, sensors):
        ''' Implemented by the specific phases '''
        pass

    def __repr__(self):
        """Represent this object as a string"""
        # print("__repr__ = ", self.stateNum)
        return self.__str__()
```

```
    def __str__(self):
        """Describe this object by it's class name"""
        return self.__class__.__name__


class PreLaunchPhase(State):
    """
    The state before launch.
    Current conditions:
        The rocket is motionless on the launch pad.
    Duration:
        Between ~30-120 minutes.
    Goals:
        Conserve energy while waiting for launch event.
    """

    def __init__(self):
        self.stateNum           = 0
        self.duration_start     = None
        self.duration_threshold = 0.2
        self.acc_threshold      = 1.5

    def monitorPhase(self, sensors):
        """
        Move to the next phase if:
            Over 1.5 G acceleration on Z axis for over 0.2 seconds.
        """
        if abs(sensors['acc_z']) > self.acc_threshold:
            if not self.duration_start:
                # First tick of acceleration sets the duration timer
                self.duration_start = time.time()
            elif (time.time() - self.duration_start) > self.duration_threshold:
                # Conditions were met, so advance phase
                return PrimaryEnginePhase()
            # Enough G's but not enough time has passed, keep checking
            return self
        else:
            # Not enough G's should reset the duration timer
```

```
            self.duration_start = None
            return self


class PrimaryEnginePhase(State):
    """
    The state during main engine burn.
    Current conditions:
        The rocket is accelerating quickly.
    Duration:
        ~10 seconds.
    Goals:
        Log sensor data.
    """

    def __init__(self):
        self.stateNum           = 1
        self.duration_start     = None
        self.duration_threshold = 0.2
        self.acc_threshold      = 1.5

    def monitorPhase(self, sensors):
        """
        Move to the next phase if:
            Under 1.5 G acceleration on Z axis for over 0.2 seconds.
        """
        if abs(sensors['acc_z']) < self.acc_threshold:
            if not self.duration_start:
                # First tick of acceleration sets the duration timer
                self.duration_start = time.time()
            elif (time.time() - self.duration_start) > self.duration_threshold:
                # Conditions were met, so advance phase
                return SecondaryEnginePhase()
            # Enough G's but not enough time has passed, keep checking
            return self
        else:
            # Not enough G's should reset the duration timer
            self.duration_start = None
            return self
```

12.2 Ground Station

12.2.1 LCD Display

The ground station's LCD display is limited to 240x320 pixels, meaning space was at a premium. We used Python's Curses library to better control the placement and refresh rates of different portions of the screen.

```
def connectToDatabase(stdscr):
    db = None
    stdscr.addstr(1,33,"[    ]")
    while not db:
        try:
            db = Mariadb('config.yml')
            stdscr.addstr(1,35,"OK",curses.color_pair(2))
            screen.refresh()
        except Exception as e:
            stdscr.addstr(2,1,str(e))
            stdscr.addstr(1,35,"ERR",curses.color_pair(4))
            screen.refresh()
            time.sleep(5)
    return db

def displayStaticElements(stdscr):
    ''' Runs once to display the static screen elements '''
    stdscr.addstr(0,1,"ESRA 30k Rocket Summary")
    stdscr.addstr(1,1,"Database status . . . . . .")

    # stdscr.addstr(3,1,"Parser status table:")
    stdscr.addstr(4,1,"{:6}{:3}{:7}{:6}{:10}".format(
        "PID", "F", "Status", "Delay", "Callsign"
    ))

    # stdscr.addstr(10,1,"Currently active flights:")
    stdscr.addstr(11,1,"{:6}{:8}{:9}{:7}{:6}".format(
        "Delay", "Lat", "Lon", "Alt", "CS",
    ))

def updateParserSection(stdscr, db, dbtime):
    ''' Updates the section displaying parser statuses '''
    rows = db.getParserTable()
    for i,r in enumerate(rows):
        # Find out when the parsers last connected (in seconds)
```

```
        delay = int(
            (dbtime - r['last_activity']).total_seconds()
        )

        stdscr.addstr(5+i,1,"{:6}{:3}{:7}{:6}{:10}".format(
            str(r['id'])[-5:],
            str(r['using_f_id']),
            '',
            str(delay) if delay < 600 else '>600',
            str(r['callsign'])
        ))
        if delay < 10:
            stdscr.addstr(5+i,10,'OK',curses.color_pair(2))
        elif delay < 30:
            stdscr.addstr(5+i,10,'SLOW',curses.color_pair(3))
        else:
            stdscr.addstr(5+i,10,'DISC',curses.color_pair(4))
```

12.2.2 Mariadb module

This project included several different programs that all need access to the database. We wrote a single Python module that simplified the chaos by reading credentials from a configuration file, and providing an API to access the database.

```
class Mariadb:
    ''' A class to interact with the ESRA mariadb database '''

    def __init__(self, configFile='config.yml'):
        try:
            with open(configFile, 'r') as cf:
                cf = yaml.load(cf)
            self.connection = pymysql.connect(
                host        =   cf['database']['host'],
                user        =   cf['database']['user'],
                password    =   cf['database']['pass'],
                db          =   cf['database']['db'],
                charset     =   'utf8mb4',
                cursorclass =   pymysql.cursors.DictCursor,
                autocommit  =   True
            )
        except Exception as e:
            raise EnvironmentError
```

```
        self.cf = cf
        self.last_connected = None
        self.valid_callsigns = {}


    def createNewActiveFlight(self, fid):
        ''' Create a new flight ID while limiting duplicates '''
        with self.connection.cursor() as c:
            sql = """
                INSERT INTO Flights(id, status)
                VALUES({}, 'Active')
            """.format(fid)
            c.execute(sql)
            return c.lastrowid


    def getCurrentPosition(self):
        ''' Returns info about the current location of the flight '''
        with self.connection.cursor() as c:
            sql = """
            SELECT B.latest, B.lat, B.lon, B.alt, C.callsign FROM Flights AS F
            JOIN (
                SELECT f_id, c_id, lat, lon, alt, MAX(time) AS latest
                FROM BeelineGPS
                GROUP BY f_id, c_id
            ) B ON B.f_id = F.id
            JOIN ( SELECT id, callsign FROM Callsigns ) C ON C.id = B.c_id
            WHERE F.status = 'Active'
            """
            c.execute(sql)
            return c.fetchall()
```

12.2.3 Example config.yml file

A simple configuration file allowed us to easily switch between production and testing databases, or to move the database between hosts, as we did during the test launch.

```
database:
    host: localhost
    user: myusername
```

```
pass: mypassword
db:   dbname
```

12.2.4 NodeJS website app.js

The app.js file is the beating heart of the website. It controls what happens when the user reaches a specific url, and runs the database queries used by each page's handlebars template. This file contains many different pages and queries, here are just two examples.

```
app.get('/summary', function (req, res) {
  // Display a summary of all recorded flights
  sql = '
      SELECT
          B.f_id, B.id, C.callsign, F.status,
          MIN_T.min_time, MIN_T.start_lat, MIN_T.start_lon,
          MAX_T.max_time, MAX_T.end_lat, MAX_T.end_lon,
          ALT.max_alt, ALT.min_alt
      FROM BeelineGPS B
      JOIN (SELECT id, status FROM Flights) F ON F.id = B.f_id
      JOIN (SELECT id, callsign from Callsigns) C ON C.id=B.c_id
      JOIN (
          SELECT f_id, c_id, lat AS end_lat, lon AS end_lon, time AS max_time
          FROM BeelineGPS WHERE time IN (
              SELECT DISTINCT MAX(time) FROM BeelineGPS GROUP BY f_id, c_id
              )
          ) MAX_T ON MAX_T.f_id = F.id
      JOIN (
          SELECT f_id, lat AS start_lat, lon AS start_lon, time AS min_time
          FROM BeelineGPS WHERE time IN (
              SELECT DISTINCT MIN(time) FROM BeelineGPS GROUP BY f_id, c_id
              )
          ) MIN_T ON MIN_T.f_id = MAX_T.f_id
      JOIN (
          SELECT id, MIN(alt) AS min_alt, MAX(alt) AS max_alt FROM BeelineGPS
          GROUP BY f_id, c_id
          ) ALT ON ALT.id = B.id
          GROUP BY B.f_id, B.c_id
  '
  db.query(sql,function(err, results) {
      // console.log('Results: '+results['time']);
      //res.render('index-page', summary : result);
```

```
      res.send(JSON.stringify(results));
  });
});

app.get('/q', function(req, res){
    var get = req.query.get;
    var f_id = req.query.f_id;
    var limit = (req.query.limit) ? ' LIMIT '+req.query.limit : '';
    var time = (req.query.time) ? " AND time >= '"+req.query.time+"'" : '';
    var sql = null;

    switch(get){
        case 'altVTimeSources':
            // Returns a list of the unique callsigns for a flight id
            sql = `SELECT DISTINCT
                    callsign FROM BeelineGPS WHERE f_id=${f_id}`+time+`
                    UNION
                    SELECT 'Rocket'
                    FROM Rocket_Avionics WHERE f_id=${f_id}`+time+`
                    UNION
                    SELECT 'Payload'
                    FROM Payload_Avionics WHERE f_id=${f_id}`+time+`
                    ORDER BY source DESC`+limit;
            break;
        case 'altVtime':
            // Plots altitude vs time
            sql = `SELECT time, alt, callsign AS Source
                    FROM BeelineGPS B
                    JOIN Callsigns C ON C.id = B.c_id
                    WHERE f_id=${f_id}`+time+`
                    UNION
                    SELECT time, alt, 'Payload'
                    FROM Payload_Avionics
                    WHERE f_id=${f_id}`+time+`
                    UNION
                    SELECT time, alt, 'Rocket'
                    FROM Rocket_Avionics
                    WHERE f_id=${f_id}`+time+`
                    ORDER BY time ASC`+limit;
```

```
        break;
    case 'acc_zVtime':
        // Plots altitude vs time
        sql = `SELECT time, acc_z, 'Rocket'
               FROM Rocket_Avionics
               WHERE f_id=${f_id}`+time+`
               ORDER BY time ASC`+limit;
        break;
    case 'tempVtime':
        // Plots altitude vs time
        sql = `SELECT time, temp, 'Rocket'
               FROM Rocket_Avionics
               WHERE f_id=${f_id}`+time+`
               ORDER BY time ASC`+limit;
        break;
    case 'receptionVtime':
        // Plots reception vs time
        sql = `SELECT time
               FROM BeelineGPS WHERE f_id=${f_id}`+time+`
               ORDER BY time ASC`+limit;
        break;
    case 'flightSummary':
        // Returns flight summary info
        sql = `SELECT B.*
               FROM Flights AS F
               INNER JOIN BeelineGPS AS B
               ON B.f_id = F.flight_id
               INNER JOIN (
                 SELECT callsign, MAX(time) AS latest
                 FROM BeelineGPS
                 GROUP BY callsign
               ) AS M
               ON M.callsign = B.callsign AND M.latest = B.time
               WHERE F.status = 'Active'`;
        break;
    case 'map':
        // Plots location v time as 2d or 3d map
        sql = `SELECT time, lat, lon, alt, callsign AS Source
               FROM BeelineGPS B
```

```
                    JOIN Callsigns C ON C.id = B.c_id
                    WHERE B.f_id=${f_id}'+time+'
                    ORDER BY time ASC'+limit;
            break;
        case 'fid':
                //return list of flight IDs
                sql = 'SELECT id FROM Flights';
                break;
        default:
            // Invalid or missing get field
    }

    if(sql){
        console.log('SQL = '+sql);
        db.query(sql,function(err, results) {
            res.send(JSON.stringify(results));
        });
    } else {
        console.log('No SQL');
        res.end('No SQL');
    }
});
```

12.2.5 Live graphs using CanvasJS

Our incoming telemetry data was a constant feed, so we wanted our website to constantly update as well. We used the CanvasJS library to grab the latest results from the database and re-draw our graph at 1 second intervals. Grabbing only the latest results reduced the load on our server, and we matched our polling delay to the delay of incoming telemetry packets.

```
<script type="text/javascript">
    window.onload = function () {
    function line() {
      this.source = "";
      this.dps = [];
    };
    var lines = [new line(),new line(),new line(),new line(),new line(),new line()];
    var pause = true;
    var counter = 0;
    var lasttime = "";
    $.getJSON("/q?get=altVtime&f_id={{fid}}", function(data){
```

```
    $.each(data, function(key, value) {
            var n = 0;
            while(lines[n].source != "" && n<6){
                if(value['Source'] == lines[n].source){
                    lines[n].dps.push({x : new Date(value['time']), y : value[
                    break;
                }
                n++;
            }
            if(lines[n].source ==""){
                console.log(""+new Date(value['time']))
                lines[n].source = value['Source'];
                lines[n].dps.push({x : new Date(value['time']), y : value['alt']
            }
            lasttime = value['time'];
        });
        pause = false;
});

var chart = new CanvasJS.Chart("chartContainer",
{

    title:{
    text: ""
    },
    axisX:{
        valueFormatString: "MM-DD-HH:mm:ss"
    },
     data: [
    {
        type: "line",
        name: lines[0].source,
        dataPoints: lines[0].dps
    },
    {
        type: "line",
        name: lines[1].source,
        dataPoints: lines[1].dps
    },
```

```
    {
        type: "line",
        name: lines[2].source,
        dataPoints: lines[2].dps
    },
    {
        type: "line",
        name: lines[3].source,
        dataPoints: lines[3].dps
    },
    {
        type: "line",
        name: lines[4].source,
        dataPoints: lines[4].dps
    },
    {
        type: "line",
        name: lines[5].source,
        dataPoints: lines[5].dps
    }
    ]
});
waitForGraph();

function waitForGraph(){
    if(pause) {
        setTimeout(function(){waitForGraph()}, 100)
    } else {
        pause = true;
        updateChart();
        chart.render();
    };
};

function updateChart() {
  console.log("Last time: "+lasttime);
    $.getJSON("/q?get=altVtime&f_id={{fid}}&time=" + lasttime, function(data){
      $.each(data, function(key, value) {
              var n = 0;
```

```
                        while(lines[n].source != "" && n<6){
                            if(value['Source'] == lines[n].source){
                                lines[n].dps.push({x : new Date(value['time']), y : valu
                                break;
                            }
                            n++;
                        }
                        if(lines[n].source ==""){
                            lines[n].source = value['Source'];
                            lines[n].dps.push({x : new Date(value['time']), y : value['alt
                        }
                        lasttime = value['time'];
                        console.log("values are" + JSON.stringify(key[0]) + "," + key[1] + "
                    });
                    // pause = false;
                });
                waitForUpdate();
                function waitForUpdate(){
                    if (pause){
                        setTimeout(function(){waitForUpdate()}, 100)
                    } else {
                        pause = true;
                        chart.render();
                        setTimeout(function(){updateChart()}, 1000);
                    };
                }
            }
        }
```

12.2.6 Handlebars

Handlebars is a library that helps template websites, allowing many pages to share similar elements. This segment from our main handlebars layout shows how we used the same CSS and script files, as well as header and footer elements across our entire website.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="shortcut icon" type="image/x-icon" href="/Patch.png">
    <meta charset="utf-8">
    <title>ESRA 30k Flight Viewer</title>
```

```
    <link rel="stylesheet" href="/style.css" media="screen">
    <script src="./jquery-3.2.1.min.js"></script>

  </head>
  <body>

{{{body}}}
{{>footer}}
  </body>
</html>
```

12.2.7 Telemetry Parser

The telemetry parser is responsible for taking an audio string and inserting text into our database. It uses the same config.yml file and database module as the other programs, but has some additional checks such as ensuring incoming data is paired to an active flight.

```
def getSerial(self):
    ''' Read the serial number for the current pi '''
    try:
        with open('/proc/cpuinfo','r') as f:
            return f.readlines()[-1].split()[-1]
    except:
            return '0000'


def createBeelineLog(self, filename):
    # Create log file if not exists
    try:
        f = open(filename, 'r')
        f.close()
    except IOError:
        f = open(filename, 'w')
        f.write('BeelineGPS\n')
        # Write header to log
        f.write('timestamp,callsign,audio_level,lat,lon,alt,alt_units,f_id,serialNum\n')
        f.close()


def listen(self, numLoops=True):
```

```
# Ensure we have a few necessary things
assert(self.f_id)
assert(self.serialNum)
assert(self.callsign)
print("\nListening for packets...")
# Ensure the log file exists and is ready
self.createBeelineLog(self.logFile)
with open(self.logFile, 'a+') as log:
    while (numLoops):
        if (numLoops is not True): numLoops -= 1
        if (datetime.datetime.now() - self.db.last_connected).total_seconds() > 3:
            self.db.updateParserTable(
                self.f_id, self.serialNum, self.callsign
                )
        data = self.wolf.checkAudio()
        data['f_id'] = self.f_id
        data['serialNum'] = self.serialNum
        self.insertBeelineGPS(data)      # write to database
        log.write(
            str(datetime.datetime.now())+','+   \
            ','.join([str(value) for key, value in data.items()])+'\n'
            ) # write values to local log
return True
```

13 APPENDIX 2: PHOTOS

13.1 Ground Station

All our main ground station components are in this weatherproof hard plastic box. This will protect the raspberry pis from the sand and dirt of the launch site. The glowing LEDs show that the parsers are connected to power. The screen on the main raspberry pi shows the status of each connected parser.

Fig. 7: Ground station box with essential components for parsing, hosting, and serving

Fig. 8: Closeup of the Raspberry Pi Screen

13.2 Website

Our website is themed after our weekly meeting slides in spirited black and orange school colors. Below are images from the site displaying data from our first test launch.

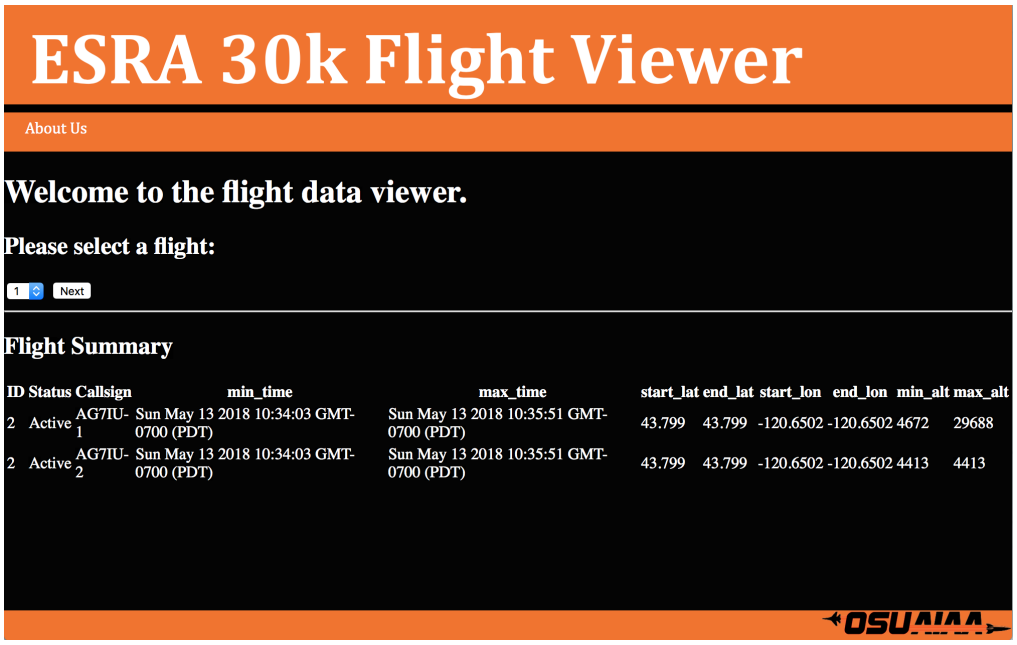


Fig. 9: Website Home Screen

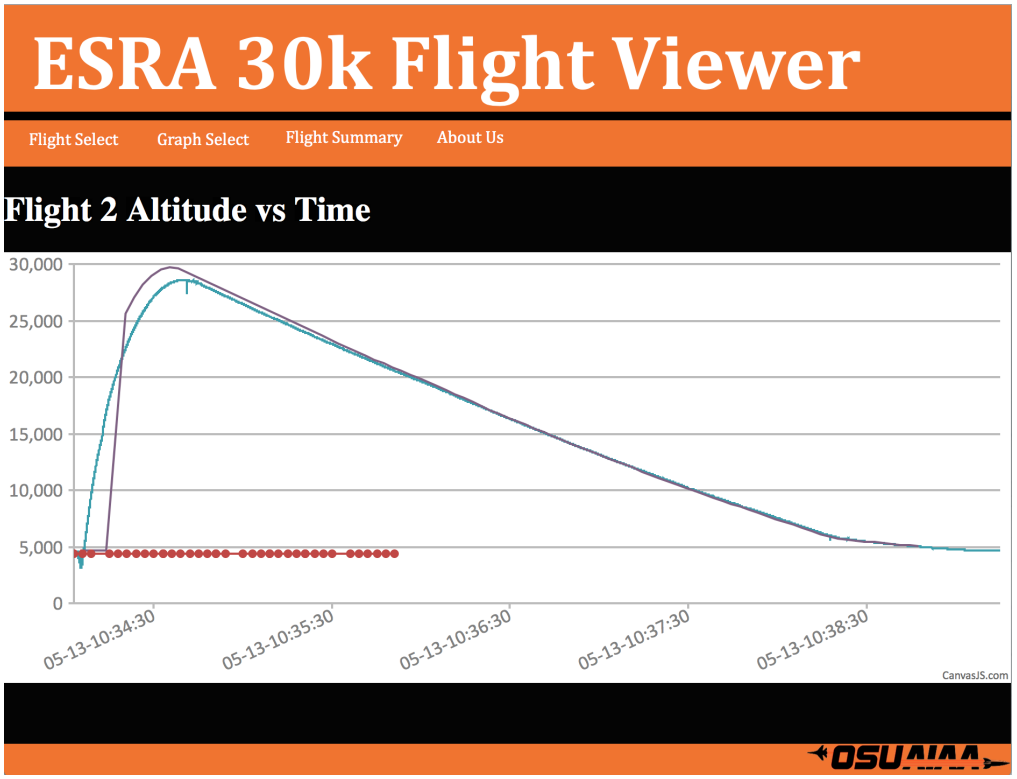


Fig. 10: Altitude graph from first test launch

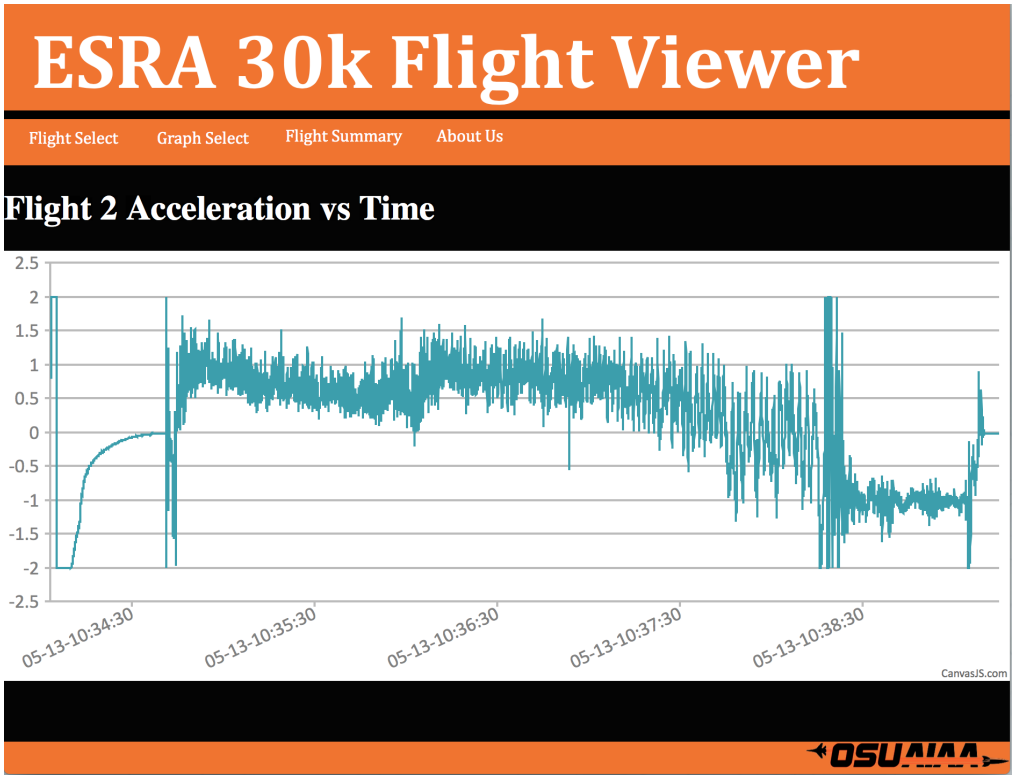


Fig. 11: Acceleration graph from first test launch

13.3 Payload

Our payload software was installed and run on this payload, which was launched for the last time on May 13th.

Fig. 12: Original Scientific Payload

13.4 Rocket

Our rocket avionics were installed and run on the rocket pictured below. Our first test launch saw it beautifully recovered, and we look forward to seeing fly again at the competition.

Fig. 13: Rocket on the launch rail at its first test fire