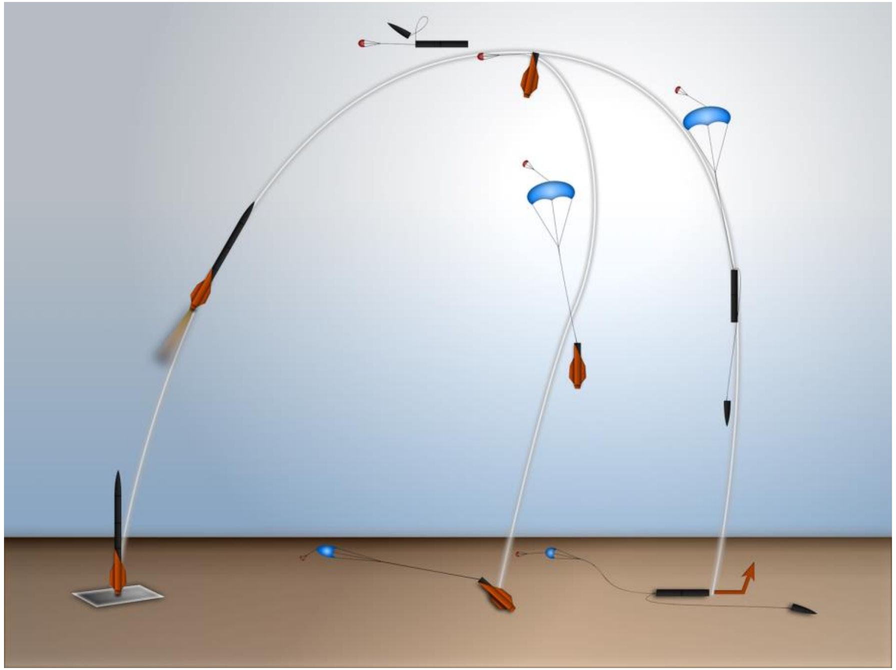
Group 33 Midterm Progress Presentation

02/10/2018

Project Overview



Projected altitude of 5199 ft.

↓

2 section separation at apogee

↓

Main deployment at 800 ft.

↓

Autonomous rover ejection

↓

Solar panel deployment

Project Overview: Software

- As the CS team for this competition, our responsibilities are the following:
  - Rover movement algorithm code
  - Creation, updating, and maintaining team website
  - Creation and formatting of LaTeX documents for CDR, FRR, LRR, and PLAR
  - Graphical representation of data collected via data logging module

LaTeX Presentation

- After the difficulties of using Word for the PDR document, the team decided to use LaTeX for all future deliverables, including the CDR, the FRR, the LRR, and the PLAR
- To help facilitate this, a presentation was created and given on adding content to LaTeX documents, including figures, tables, lists, equations, and various headings/subheadings

Preliminary Design Review

- For the Preliminary Design Review (PDR), a document, a presentation, and a flysheet had to be created
- Group 33's role was adding software-specific content to the PDR document and adding the final deliverables to the team website so NASA judges could easily access them
- Additionally, Group 33 assisted in editing the document and attended the presentation to answer any questions relating to software

Preliminary Design Review, Cont.

7.3.1. Payload Software

7.3.1.1. Software Requirements

The software running on the rover must, at a minimum, provide the rover with the intelligence necessary to:

- Move at least five feet from any part of the vehicle body
- Avoid obstacles the rover cannot drive over or past
- Deploy solar panels from its body

The software on board must be sufficient to instruct the rover when and how to perform each of these tasks since they must be performed entirely autonomously after the payload has been ejected from the vehicle's body.

To accomplish these goals in software, a software framework will be leveraged to help abstract away some of the implementation details of issues such as sensor communication and object mapping, an operating system will run on the rover's microcontroller to support said framework, and code will be written in a certain language to actually implement the rover's intelligence. For each of these, research was done to determine the best candidate for the team's purposes based on a variety of desired properties.

7.3.1.2. Software Design Alternatives

7.3.1.2.1. Software Framework

By utilizing a software framework, team members responsible for designing the rover's software can utilize existing and tested libraries for robotics fundamentals such as object mapping, motor control, and sensor communication. To this end, three different software frameworks were considered: Robot Operating System (ROS), Mobile Robotics Programming Toolkit (MRPT), and Microsoft Robotics Developer Studio (MRDS). These three were primarily considered due to their popularity and support relative to smaller robotics software frameworks.

When considering which of these frameworks would be the best fit for developing software for the rover, a number of factors were taken into account. Firstly, the learning curve for associated with utilizing the framework is a point of interest. While this aspect is not as critical as the core functionality of the framework, a framework that does not take a month to become familiar with is obviously preferable to one that takes a week. Secondly, the framework's support (or lack thereof) for various programming languages is a point of interest. Frameworks that support multiple languages are preferable; those supporting languages the team deems well-suited for rover software development even more so. In a similar vein, the framework compatibility with different operating systems and microcontrollers were also considered – a framework that does not support the best microcontroller (from a hardware perspective) would be a subpar choice, for example. Although difficult to quantify, the overall functionality of the framework

was deemed to be the most important factor when selecting a software framework. Broadly, this would include the breadth of libraries available for the tasks the team is interested in and how much abstraction the framework provides to the end user. Finally, due to hardware limitations of microcontrollers small enough to fit in a 5-inch diameter vehicle, the memory overhead of the framework in question was also considered.

7.3.1.2.2. Operating System

Overall, the operating system running on the rover's microcontroller was a secondary consideration. Although necessary to run any of the three software frameworks considered, whatever operating system is selected simply needs to support the desired framework, utilize minimal hardware resources, and be relatively straightforward to use. In light of these considerations, the alternatives were selected to be Ubuntu, Raspbian, and Windows 10 for IoT. Ubuntu was selected because it is the distribution of Linux with the most support and is arguably the easiest to use. Additionally, because it is Debian-based, it supports both ROS and MRPT. Raspbian was considered because it is an operating system specifically designed for Raspberry Pi, the microcontroller selected for the payload, and because it is lightweight. Being Debian-based, it is also supported by both ROS and MRPT. Finally, Windows 10 for IoT was considered because it is the only operating system capable of running on a Raspberry Pi that supports MRDS.

7.3.1.2.3. Programming Language

Beyond simply being supported by the selected software framework, the language the rover's software will be written in should have features that will make the development process easier and, at a minimum, not serve as a performance bottleneck. ROS primarily supports C++, Python, and Lisp, MRPT supports just C++, and MRDS only supports C# and a Microsoft proprietary visual programming language. In light of these restrictions, C++, Python, and C# were deemed to be the most viable alternatives since Lisp, being an archaic functional language, would be difficult to adapt to, and any visual programming language would force the microcontroller's OS to support GUI. The desired features for the programming language were determined to be:

- Team familiarity: using a language that the CS team members are already familiar with or least similar in syntax to one they are familiar would certainly help streamline the development process.
- Memory management: using a language with built-in garbage collection would help avoid a lot of runtime errors related to invalid memory access and prevent memory leaks from occurring.
- Multi-processing: since the microcontroller selected, a Raspberry Pi 3, uses a multi-core CPU, using a language that can leverage simultaneous use of multiple cores would help the team make the most of the microcontroller's capabilities.
- Ease of debugging: given the perceived complexity of the rover's software, the team anticipate a lengthy testing and debugging stage. Languages that are easier to debug are obviously preferable.

Preliminary Design Review, Cont.

- Memory overhead: in light of the limited memory available on the Raspberry Pi, languages that make efficient use of memory are desirable.
- Performance: although this property is generally inversely-correlated with automatic memory management, languages that can achieve the same functionality in a smaller number of clock cycles are preferable.

7.3.1.3. Software Design Decisions

7.3.1.3.1. Software Framework

After evaluating each of the potential software frameworks in relation to their desired features, the following design matrix was created:

Software Framework Options							
Design		ROS		MRPT		MRDS	
Requirement	Weight	Value	Score	Value	Score	Value	Score
Learning Curve	3	1	3	3	9	6	18
Language Support	6	10	60	1	6	2	12
OS Support	3	7	21	7	21	1	3
Functionality	10	10	100	6	60	6	60
Memory Overhead	6	4	24	7	42	1	6
Total		208		138		99	

Table 7.3.1.3.1.1

Overall, ROS is head and shoulders above both MRPT and MRDS in both language support and functionality, which are the two most important factors. ROS has full support for C++, Python, and Lisp and partial support for other languages like Java. In contrast, MRPT and MRDS only support C++ and C#, respectively. Additionally, ROS has by far the most community support of the three, with MRDS having its last update in 2012.

7.3.1.3.2. Operating System

After evaluating each of the potential operating systems for the microcontroller in relation to their desired features, the following design matrix was created:

Operating System Alternatives							
Design		Ubuntu		Raspbian		Windows 10 IoT	
Requirement	Weight	Value	Score	Value	Score	Value	Score
Ease of Use	3	8	24	5	15	10	30
Software Support	10	7	70	10	100	3	30
Memory Overhead	7	3	21	6	42	1	7
Total		115		157		67	

Table 7.3.1.3.2.1

Raspbian, being an operating system created specifically for the Raspberry Pi, is unsurprisingly the winner here. It can interface most effectively with the Pi's hardware, is lightweight, and is comparable to Ubuntu in terms of ease of use (besides its lack of GUI). Windows 10 for IoT scores poorly in both software support and memory overhead since it is only compatible with MRDS and it is weighed down by the full Windows 10 GUI. Ubuntu, although being easier to use than Raspbian, scores worse in the memory overhead department. This is because Ubuntu, even without its GUI, was designed to run on desktops, not microcontrollers.

7.3.1.3.3. Programming Language

After evaluating each of the potential programming languages in relation to their desired features, the following design matrix was created:

Programming Language Options							
Design		C++		Python		C#	
Requirement	Weight	Value	Score	Value	Score	Value	Score
Team Familiarity	3	8	24	6	18	4	12
Memory Management	6	6	36	10	60	10	60
Multi-Processing	5	8	40	1	5	8	40
Ease of Debugging	7	5	35	10	70	6	42
Readability	5	6	30	10	50	7	35
Memory Overhead	5	9	45	2	10	5	25
Performance	4	9	36	3	12	5	20
Total		247		235		239	

Table 7.3.1.3.3.1

Here, C++ is the winner. It is a mature language with extensive standard libraries and the recent version like C++11 and C++17 support the majority of modern programming language features like reflection, constant expressions, etc. Python, although being easy to read, debug, and write, is ultimately a subpar choice because its global interpreter lock (GIL) prevents it from fully leveraging multicore CPUs. While C# does support multi-processing, it loses to C++ in performance and would force the team to utilize the vastly inferior MRDS framework and Windows 10 for IoT operating system since it is not supported by either ROS or MRPT.

6.3 Leading Design (TO DO BRAD)

Critical Design Review

- Like the PDR, the Critical Design Review (CDR) required the submission of a report, a presentation, and a flysheet
- Group 33's responsibilities for this included:
  - Creating a template/folder structure in OverLeaf (collaborative site for LaTeX) for the report
  - Formatting the document in LaTeX (floating figures with text, converting Excel tables to LaTeX ones, adding an acronym glossary, formatting equations, etc.)
  - Editing the document for errors, aesthetics, and consistency of style
  - Participating in the presentation in order to answer any potential software-related questions
  - Posting the deliverables to the team website
- Over 30 hours of work for the CDR alone

Critical Design Review, Cont.

Oregon State University

2018 NASA SL Team

104 Kerr Admin Bldg. # 1011

Corvallis, OR 97331

Critical Design Review

January 12, 2018

systems connected to the regulators will draw no more than 650 mA, so the regulators will sufficiently be able to provide power to every system connected to the regulators. The regulators will be integrated into the custom PCB along with the battery level indication.

6.3.5 Wire Management

The permanent wire connections on the rover such as sensors and board to board connection are made using 20 gauge wires and through hole solder connections. Tension is taken off these connections by wire braiding and taping sections to the frame where possible. The temporary connections such as actuators and batteries are connected using screw terminals. Screw terminals provide a sturdy connection that doesn't require too much effort to connect or disconnect. The batteries will be connected to the custom PCB with the use of a wire harness, wire braiding, and screw terminals. An image of the battery wire harness can be seen below in Figure 80.

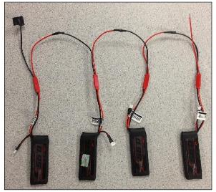


Figure 80: LiPo Battery Wire Harness

The wire harness is made of four Japan Solderless Terminal (JST) female connectors soldered together in series. The two bare leads are the leads which will be connected into the screw terminals on the PCB. Tension for the temporary connection is minimized using the same methods as the permanent connections.

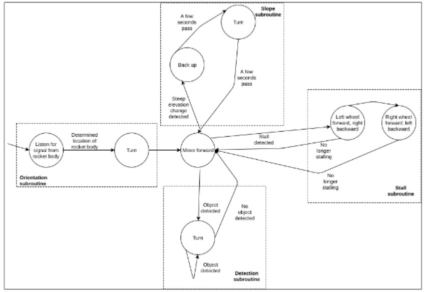
6.4 Rover Software

The software of the rover on the bottom level will be run on the Raspbian operating system (a Debian derivative), which will serve as a resource interfacing the rover software and ROS with the Raspberry Pi's hardware components. ROS will be used as the link between high level algorithmic computation and Inputs and Outputs (I/O) components, such as sonar sensors, microphones, and the IMU. ROS modules responsible for movement and path finding will subscribe to streams for these various sensors. Other ROS modules will be responsible for publishing sensor data as it arrives based on hardware interrupts to these streams. In the event of race conditions, data coming from more "vital" sensors, such as the motor driver or the IMU will take priority.

6.4.1 Movement Algorithms

Overall, the rover software will utilize three movement algorithms of varying complexity, The hardware needed by the less complex algorithms will be a strict subset of the hardware needed by their more complex counterparts, meaning that the rover can still function to some degree even if most of its sensors fail at a hardware level. Such redundancy is crucial when considering a rover than must function autonomously after experiencing extreme g forces during flight.

Figure 81: State Diagram for the Rover's Second Movement Algorithm



The three movement algorithms are as follows:

1) Simultaneous Localization and Mapping (SLAM): The rover will utilize a combination its magnetometer and its position relative to rocket frame parts (calculated by determining the phase shift because the same signals as recorded by the rover's two separate microphones) to triangulate its position. It will then utilize its front-facing sonar sensors combined with incremental scanning (via turning) to perform mapping of its surrounding area. Combined, this information will be used to determine an obstacle-free path for the rover away from the rocket frame that it will follow until it is at least five ft. away.
2) Obstacle avoidance: The rover will determine the location of any nearby rocket frame parts by listening for their signal on its dual microphones and then face away from them. From there, the rover will

Critical Design Review, Cont.

move forward, avoiding obstacles detected by its sonar sensors by turning away from them. For steep declines, which front-facing sonar sensors will not detect, the IMU will alert the rover of said decline and the rover will back up, turn away from the decline, and continue moving. At regular intervals, the rover will check it absolute direction against its initial one using its magnetometer in order to ensure that turning done during obstacle avoidance has not resulted in the rover now facing the landing site which it intends to leave.

3) Basic movement: As before, the rover will face away from the rocket if the microphones are functioning. From there, it will simply move forward continuously, only changing directions if it gets stuck, at which point it will alternate turning left and right at full power in an attempt to wiggle itself out of whatever trapped it. This strategy, corresponding to the stall subroutine in Figure 81, is also employed by the other two algorithms in the event of motor stall, but the ultimate goal of the more complex algorithms is to avoid getting stuck in the first place.

Figure 81 shows a state diagram of the second movement algorithm listed above. The overall diagram is split into four distinct subroutines. The first, the orientation subroutine, is run as soon as the rover is ejected from its housing. It involves listening for the rocket's signal on its twin microphones and then turning until the phase shift of the signals corresponds to facing away from the rocket. The second, the slope subroutine, occurs the rover determines via its IMU that the angle of descent is too steep. It then backs up and turns away from the hole. The third, the stall subroutine, occurs when the motors stall. Here, the rover alternates between full power left and right turns in an attempt to wiggle out of the trap. Finally, the detection subroutine occurs when the sonar sensors indicate that there is an obstacle ahead. The rover will then turn until there is no obstacle in front and continue to move forward.

6.4.2 Sonar Sensors

The sonar sensor data for the rover will need to be formatted in such a way that is understood by ROS's gmapping libraries. The digital formatting of this module will follow the ROS navigation stack `sensor_msgs/Range.msg`. This definition will be configured to the infrared radiation type to conform with examples provided by the ROS sonar tutorials. Additionally this definition will contain max and min ranges of the module as well as the fixed range of the calculated output.

6.4.3 Inertial Measurement Unit

The IMU will utilize ROS's navigation stack as a standard for analog output to digital data types. Specifically, we will use the `sensor_msgs/Imu.msg` message definition. This message contains a header for orientation, angular velocity, and linear acceleration for the X, Y, and Z axes.

6.4.4 Motor driver

The controls for the motor itself will be abstracted away from ROS and implemented in a separate class outside of the framework. For the driver itself we will use Pololu's library for the Pololu Dual MC33926 Motor Driver, which is currently implemented in Python. If needed, we will convert this driver into a C++ implementation using the C++ version of the Wiring Pi Library.

6.5 Ejection System

As described within the mission profile, the airframe will descend in two separate sections: motor and payload. The scientific payload will be housed within the payload section which is tethered to the launch vehicle nose cone. Following separation the payload section will have one structural end fully open to the environment as the coupler will descend along with the motor section. During descent the payload will be retained within the launch vehicle by a Kevlar harness which is released upon landing. Following successful ground recovery of the payload and motor sections, a signal is initiated which will release the payload utilizing black powder charge ignition from an e-match.

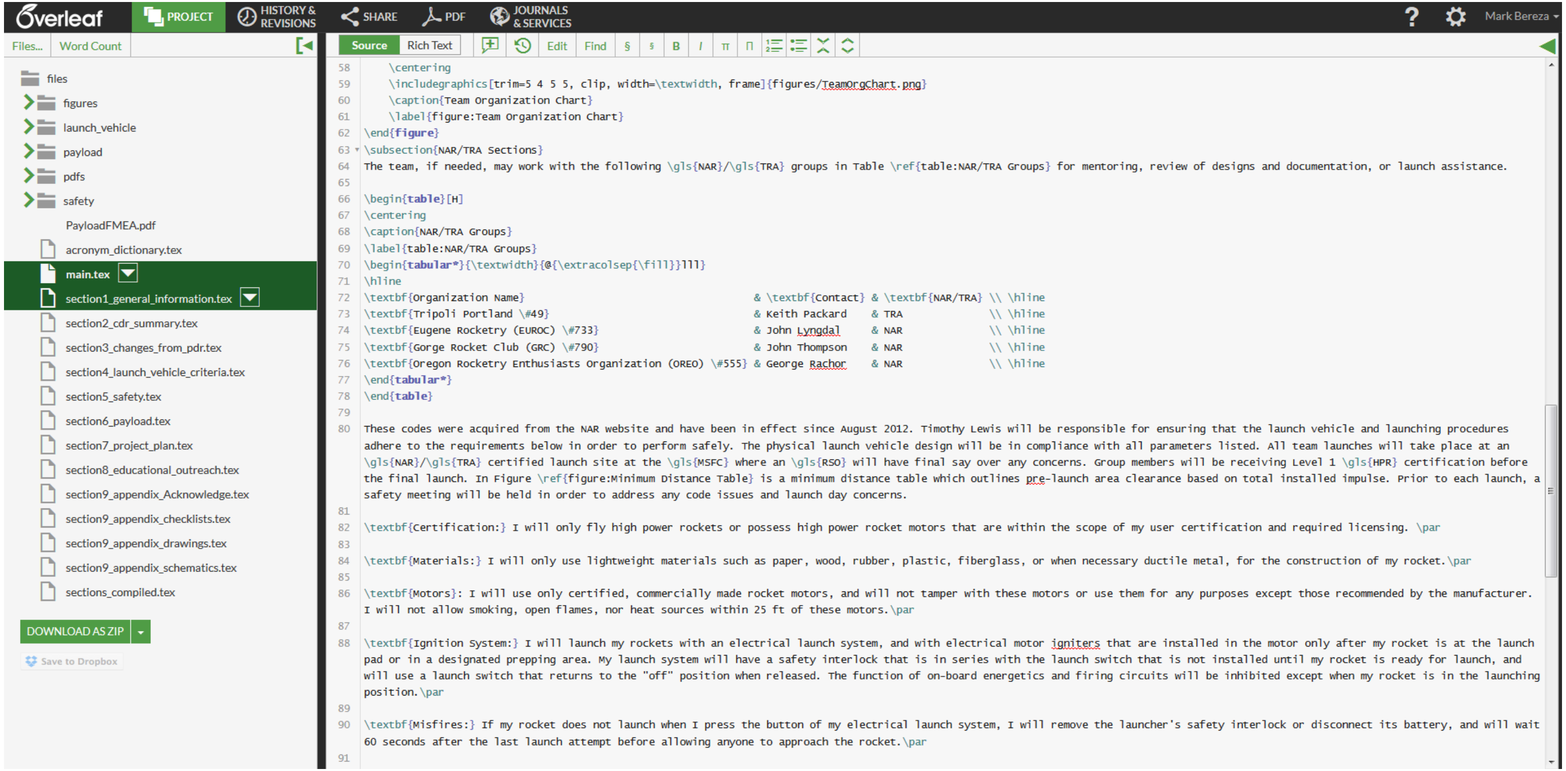
Within the PDR, design alternatives were evaluated for their ability to complete the stated payload ejection requirements. Decisions were made based off of a scoring of the alternatives against engineering specifications. The results of the analysis are summarized below in Table 17.

Table 17: Ejection System Decisions

Task Solution Evaluated	Final Decision	Reasoning
Overall Ejection Method	Open end ejection	Ejecting out an exposed end of the payload section makes for a relatively simple design for the structures team. Additionally it has no structural impact on launch vehicle performance.
Retention	Primary: Tender Descender	Reliability to secure payload throughout entire flight sequence. Both release devices initiate from a small black powder charge.
	Backup: ARRD	Both release devices initiate from a small black powder charge.
Linear Motion Ejection	Black Powder charge	Black powder is extremely reliable and durable. Additionally the technology is flexible to meet the changing needs of the payload team if necessary.

A complete CAD mock-up of the payload bay can be referenced below in Figure 82. The design will be reviewed at a sub system level.

Critical Design Review, Cont.



Flight Readiness Review

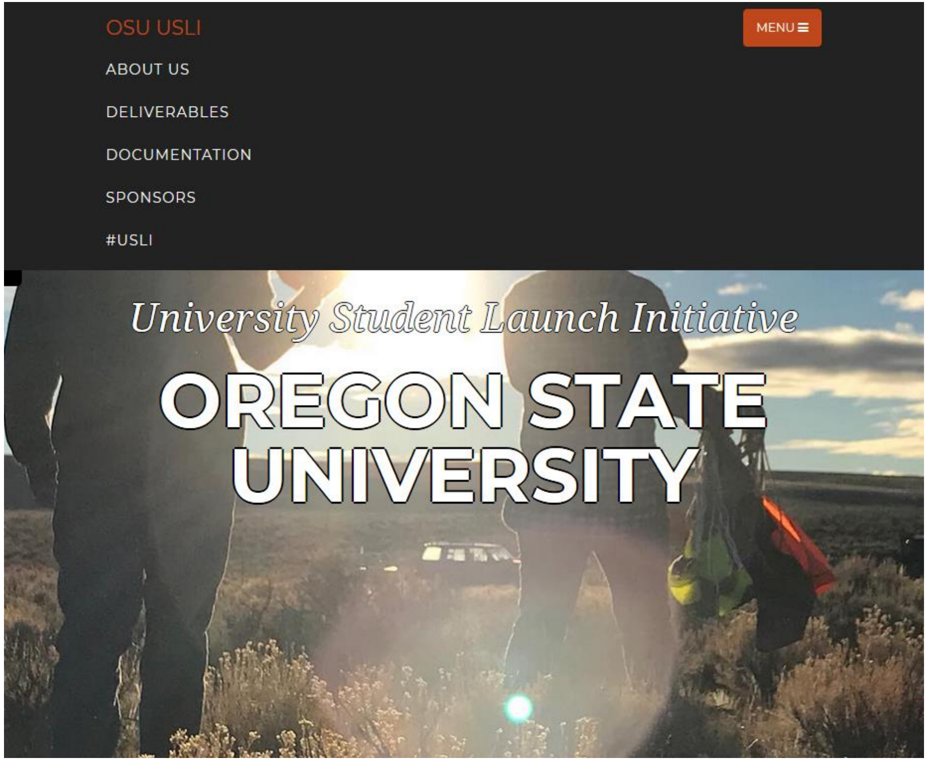
- Although the FRR won't be due for another few weeks, Group 33 has already contributed some work towards its completion:
  - Created a LaTeX template in Overleaf with figure/table/list/equation example code
  - Presentation given to team regarding Overleaf use, LaTeX syntax, and deadlines for content for document

Website

- Purchased domain name osuusli.com for team website
- Created website on osuusli.com, hosted via GitHub
- Raspberry Pi connected to osuusli.com domain to facilitate remote development

Website: Homepage

- Homepage featuring slideshow background and navigation bar



Website: Project Overview

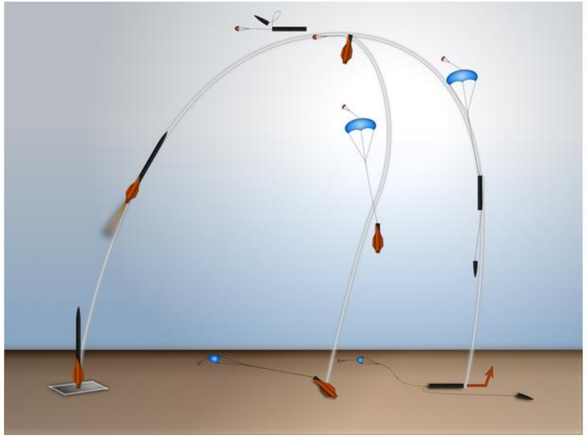
- Project overview section with image of mission profile

OSU USLI

ABOUT US DELIVERABLES DOCUMENTATION SPONSORS #USLI

WHAT IS IT?

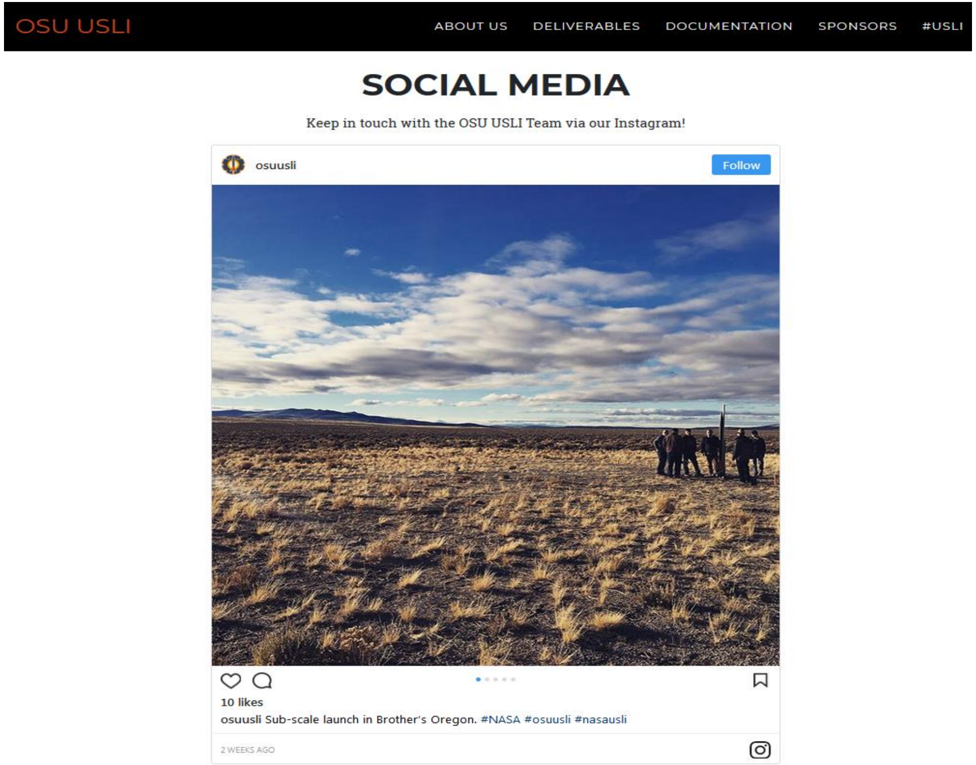
The University Student Launch Initiative is a research-based, competitive, experiential exploration activity. It strives to provide relevant, cost-effective research and development of rocket propulsion systems. This project offers multiple challenges reaching a broad audience of middle and high schools, colleges, and universities across the nation (NASA.gov). The competition consists of three possible experiments: Target Identification, A deployable rover, or landing coordinates via triangulation. OSU's team has chosen to compete in the deployable rover experiment. This is OSU's first year participating in this challenge, and the team is comprised of Mechanical, Electrical, and Software Engineering students all working toward a successful rocket launch and payload deployment in April of 2018.



Mission profile

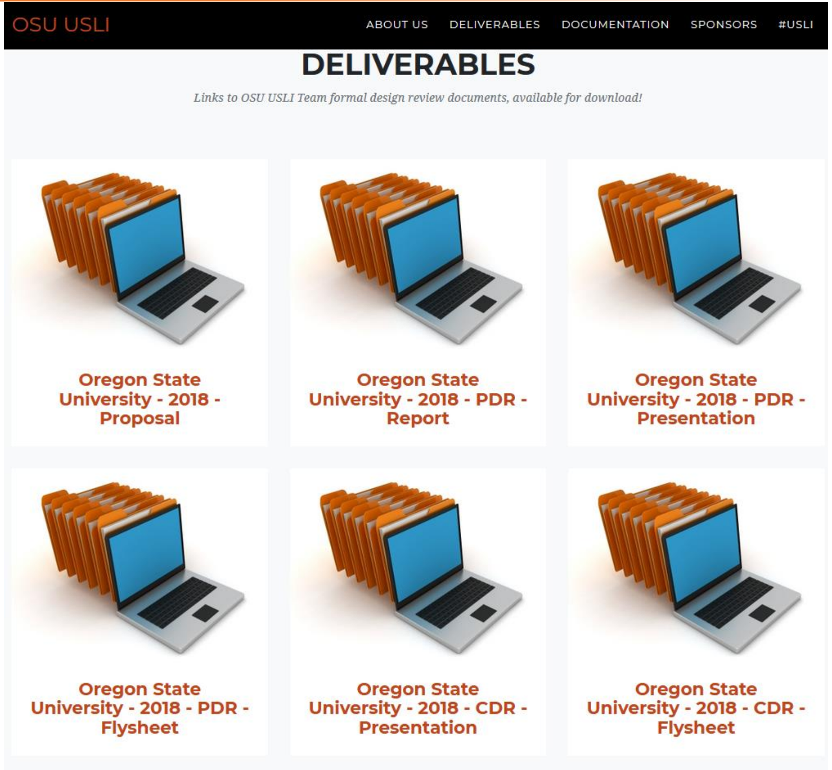
Website: Instagram

- Social media section linking to team's Instagram



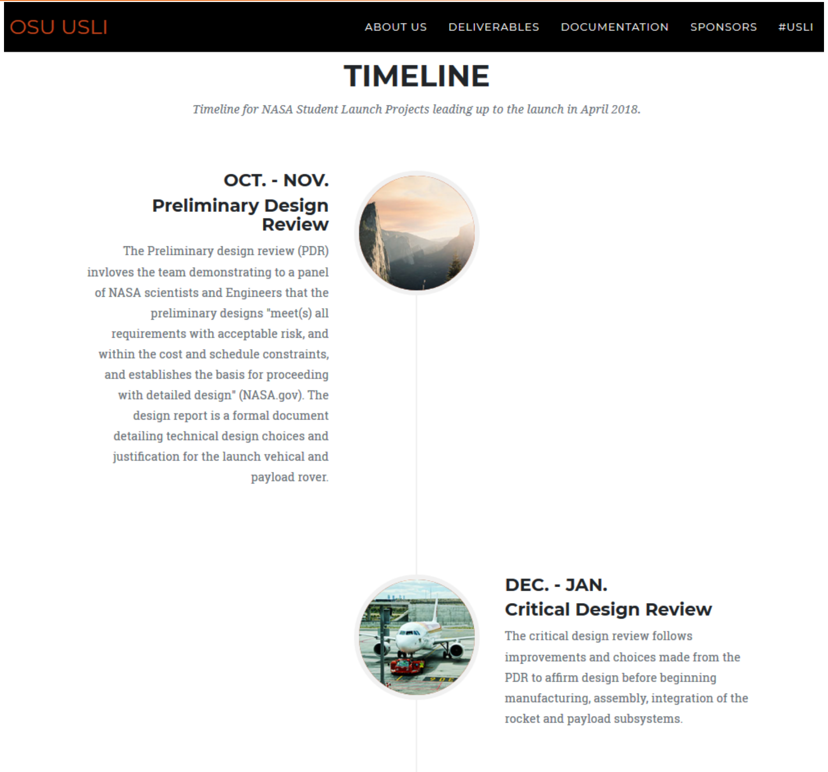
Website: Deliverables

- Deliverables section featuring documents, presentations, and flysheets for NASA



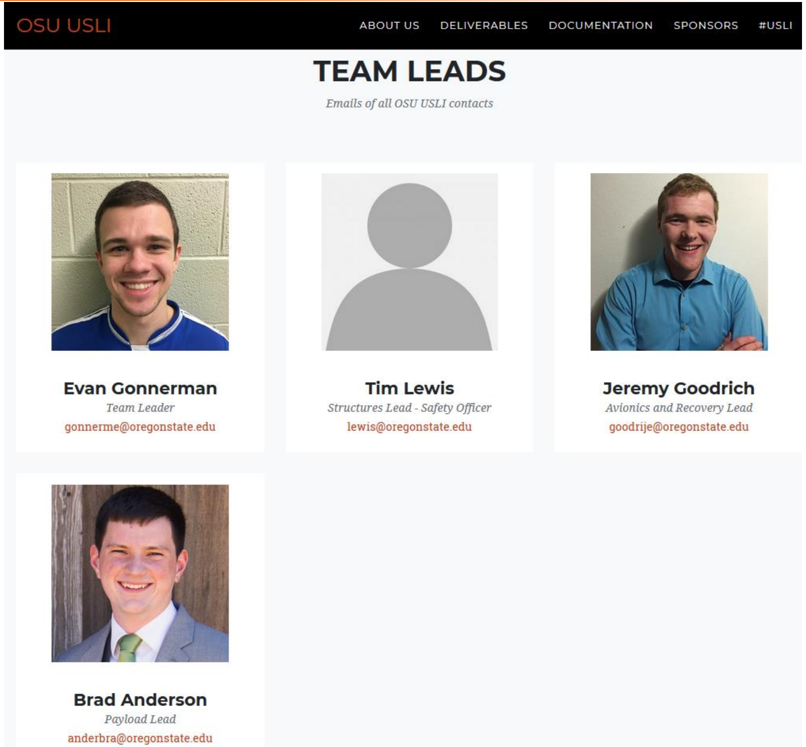
Website: Timeline

Timeline featuring critical dates for the competition



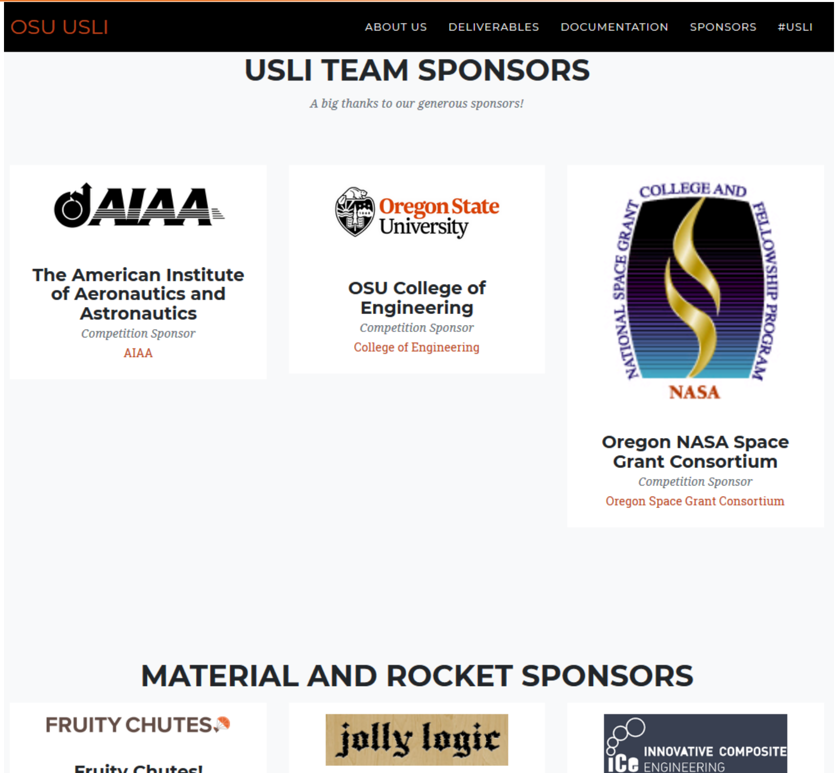
Website: About Us

- About Us section featuring team members’ names, photos, roles, and contact info



Website: Sponsors

- Sponsors page featuring monetary and materials sponsors w/ links to their websites



Website: Features

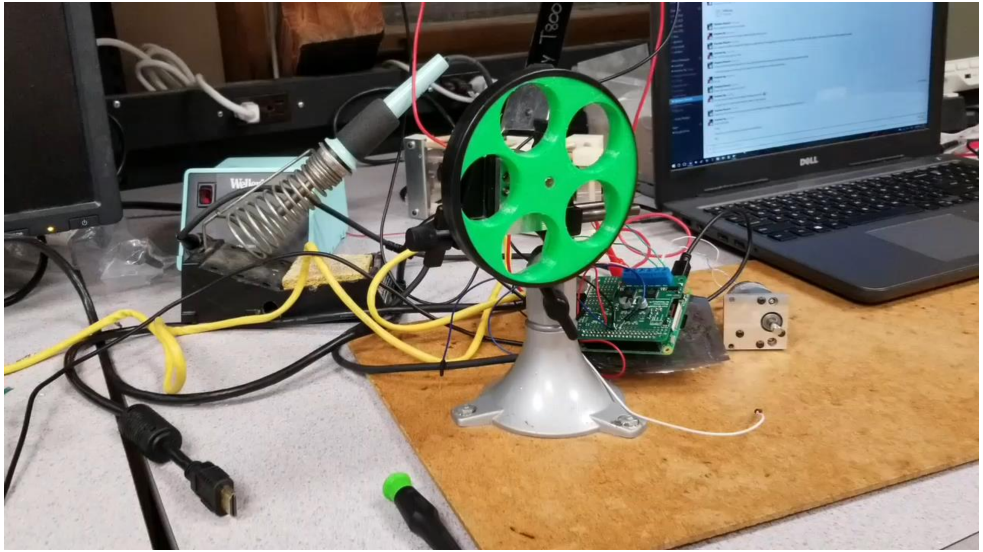
- Slideshow background on top of page
- Navigation bar linking to each section
- Accessibility features
  - Tab navigation
  - Alt-text for all images
  - Color contrast in accordance WCAG 2.0 Accessibility Standards
- Black and orange color scheme
- Dynamic scaling of contents based on window size

Rover: Raspberry Pi Setup

- Two Raspberry Pi's: one attached to testbed and one connected online to the osuusli.com domain
- SD card purchased for Raspberry Pi
- Raspbian installed on Raspberry Pi
- ROS installed on Raspberry Pi from source
- WiringPi installed on Raspberry Pi

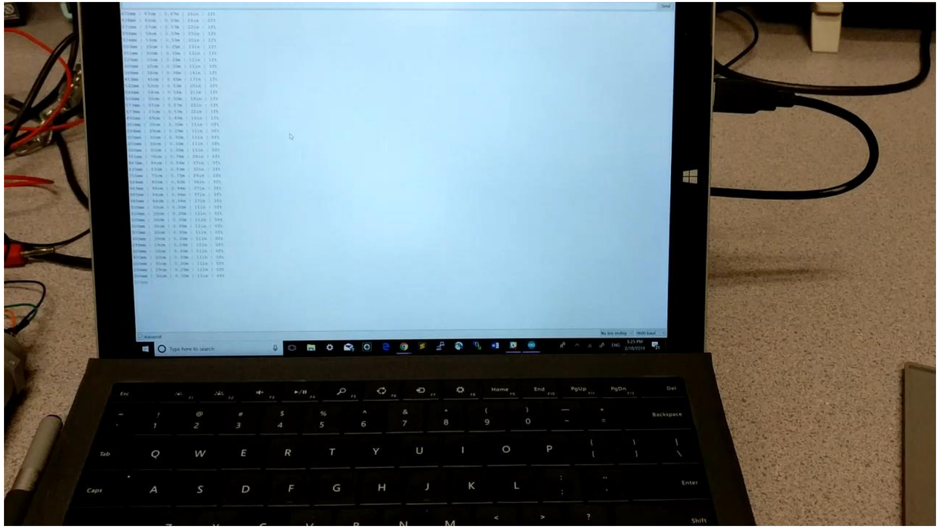
Rover: Motor Drivers

- Wrote motor driver code in Python
- Verified that motor driver code and motors worked:



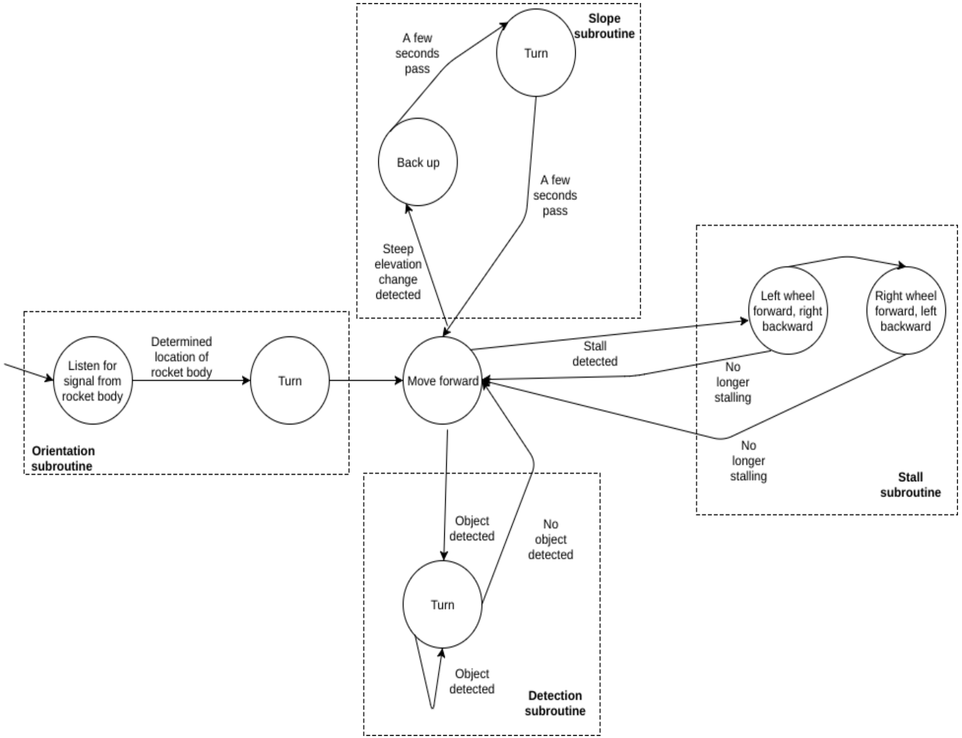
Rover: Sonar Drivers

- Started work on sonar driver code in Python
- Successfully tested sonar sensor and its driver code:



Rover: Movement Algorithm

- Created state diagram outlining high level movement algorithm
- Implemented algorithm in ROS and ran it on ROS's RVIZ 2D simulator



Rover: Networking and SSH

- Were able to connect testbed Raspberry Pi to internet
- Found a way to determine the IP address of the Raspberry Pi without connecting a monitor
- CLI script created utilizing nmapping to bind IP address to bridged connection to facilitate easy SSH connections

Rover: ADC Code

- Currently, code can print out values corresponding to analog readings from directional microphones

Educational Outreach

- Four Outreach Events Completed
  - Silver Crest Middle School - 19 students
    - Model Rocket Launch
  - Philomath Middle School - 21 students
    - Model Rocket Launch
  - Sprague High School - 191 students
    - Aerodynamics, Matchstick Rockets, & Electromagnetism
  - Walker Middle School - 191 students
    - Electromagnetism & Mousetrap Cars
  - 422 students reached!