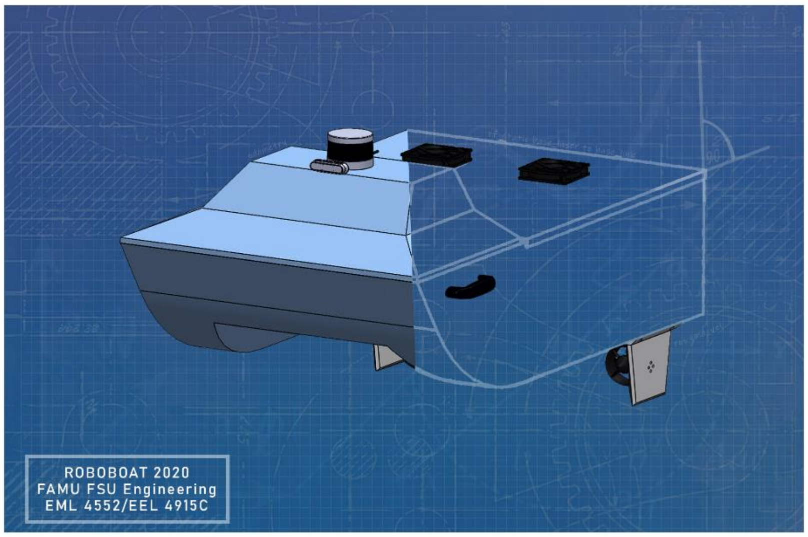
EEL 4911C/EML 4552

Final Report



Roboboat Development Team

FSU Panama City Mechanical & Electrical Engineering Seniors:
Mark Hartzog
Brandon Bascetta
Courtney Cumberland
Madison Penney
Peter Oakes
Toni Weaver

Professors: Dr. Damion Dunlap & Dr. Geoffrey Brooks
31 July 2020

TABLE OF CONTENTS

I. Summary of Final Report

A. Advisor Contact Information:

ECE Senior Design Coordinator:

Dr. Geoffrey Brooks
(850) 770-2247
gbrooks@pc.fsu.edu

MEE Senior Design Coordinator:

Dr. Damion Dunlap
(850) 770-2204
ddunlap@fsu.edu

Roboboat Technical Advisor:

Dr. Joshua Weaver
jnweaver@fsu.edu

B. Roboboat - Development Team

Mechanical Design Lead - Brandon Bascetta
Manufacturing Lead - Courtney Cumberland
Software Lead - Mark Hartzog
Software/Hardware Integrator - Peter Oakes
Hardware Developer - Madison Penney
Systems Lead - Toni Weaver

C. Project Summary

The overall objective of this project was to develop and manufacture a working boat complete with sensors and basic software that can compete in the Roboboat competition. This goal was intended to be achieved by completing three different subprojects. These included software development, hardware development, boat design and manufacturing. Figure 1, shown below, displays the functional decomposition of the project. This project focused primarily on the left three branches of the image.

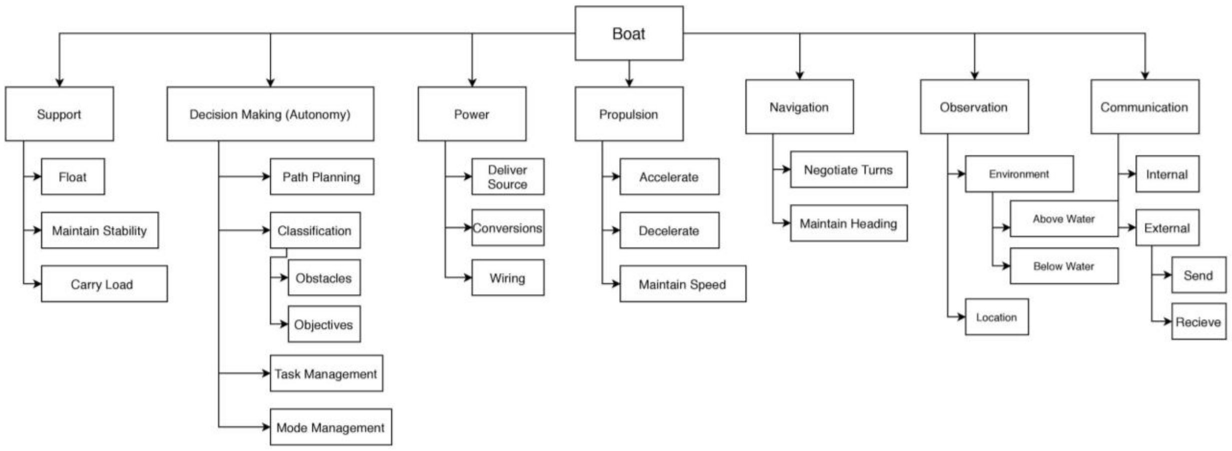


Figure 1. Functional Decomposition of the project.

i. Boat Design and Manufacturing

Utilizing the engineering design methods to meet the customer's needs and comply with competition rules, a larger, more stable boat than the previous year's boat was designed for this year's competition. Therefore, a new boat was constructed. A fiberglass and epoxy resin composite were chosen as the primary material thus, the hand lay-up method of construction was implemented to manufacture the hull of the boat as well as the lids. The final CAD design of the boat was created in SolidWorks. The overall size and weight of the vessel is constrained by the Roboboat rules. This boat will have a length of 50", width of 30" and height of 30" and an estimated weight of approximately 22.67 lbs. excluding the electrical components.

ii. Hardware Development

The hardware design essentially took each respective sensor and wired and placed them in the most optimal position of the vehicle. Because of the nature of some of the sensors, it was imperative that they were calibrated and placed in strategic locations to be implemented properly so that they could generate helpful data.

iii. Software Development

The software team was responsible for bringing life to the hardware components to make them serve a functional purpose. Using the Robot Operating System (ROS), as our middleware platform, we tapped into pre-existing algorithms that are often tailor made for our sensors creators themselves. Using these algorithms and the tools in the lab we wrote our own software to create a functional system with each piece of software working in tandem to create a large and unique data set making the vehicle mobile.

D. Project Motivation

The Roboboat competition is an international robotics competition that focuses on allowing young engineering students to create solutions for some of the most difficult and challenging electrical, and computer engineering challenges. The tasks themselves include using custom algorithms to allow the boat to autonomously solve puzzles. For example, some of the tasks include navigating a channel of buoys, finding a path through a field of obstacles, or performing a speed test to exhibit the vehicle's power. The specific duties of the software team are to take the powered sensors and setup their respective firmware, and drivers, in addition to wiring them and using their data sets to produce logic solving with algorithms. These algorithms, as mentioned previously, will allow the tasks required by the competition to be solved autonomously. Using the experiences from last year, the team will optimize the algorithms to enhance their performance which will allow the vehicle to exhibit better run times. Algorithms, data processing and publishing will be implemented primarily through the ROS environment.

E. Roboboat Development Team Goals:

- Properly setup the drivers and various sensors and modules on the vehicle
- Create a functional data set generated by the various sensors
- Send the generated data to ROS (Robotic Operating System)
- Create data connections in ROS so the sensors can communicate to one another
- Import the data to custom executables and scripts to create logic solutions and data manipulation
- Create algorithms consisting of the modified data set
- Give the motors commands based upon the logic and algorithms being implemented

F. Roboboat Project overview

i. Spring 2020

a. Setup and integrate hardware using the PE's power box. This included driver installation, manufacturer packages and firmware.
b. After the first step was complete, the sensors data generation methods were calibrated, and the heat displaced by them regulated by placing them in strategic positions.
c. The IMU was placed in an area that caused the least magnetic interference on the test boat.
d. The LiDAR was placed on the top of the test boat to maximize the visible areas and ranges.
e. The camera was placed in the front of the test boat to maximize obstacles detection.
f. After these steps were completed, the data was further calibrated and optimized and then sent into ROS once more.
g. The boat hull design was finalized in CAD.
h. The boat size was finalized at 30" X 50" x 25".
i. The boat hull mold was finished using 1" and ½" foam, spray foam, modeling clay and packing tape.
j. A modular fin design was created to attach the thrusters to and mount on the bottom of the pontoons.
k. Software was developed to drive the boat using motor mixing.
l. Software was developed to allow the boat to be driven using the RC controller.

ii. **Summer 2020**

a. Test the power system to ensure all voltages are outputting correctly.
b. Each necessary sensor was connected to the power system.
c. The data collected in the first stage will be imported into ROS executable (nodes).
d. The LiDAR (Light Detection and Ranging sensor) and IMU (Inertial Measurement Unit) sensors were integrated into the ROS environment.
e. Code was created to complete the mandatory navigation channel task.
f. Sensor data was combined with navigation algorithms to allow the boat to perform basic obstacle avoidance.
g. The boat hull was manufactured using hand laid fiberglass.
h. Sensor mounts were created using CAD software.
i. The boat software, sensors and hull were tested in water.

II. Boat Design and Manufacturing Information

The following section will provide information regarding the design process as well as the manufacturing plan for the boat hull. The Mechanical team has been designing the boat since the Fall of 2019 by going through a set of decision matrices outlined in the Design Process section. The physical manufacturing of the boat is covered in the Manufacturing Process section.

A. Design Process

Starting in the Fall of 2019, the Mechanical Engineers completed a series of design matrices to aid in the decision-making process. The first process done was generating a customer requirements document based upon the needs of the previous year's Roboboat team. With their experience of being at the competition, they were able to provide valuable feedback on what was needed with respect to the physical design of the boat. From this information gained from the previous team, customer needs were generated from their feedback. These customer needs were put into a binary piecewise comparison chart to determine the weights of each criteria as seen in Table 1.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**Total**
Stability	-	1	0	1	1	1	1	**5**
Aesthetics	0	-	0	1	1	1	0	**3**
Maneuverability	1	1	-	1	1	1	1	**6**
Modularity	0	0	0	-	0	1	0	**1**
Deck Space	0	0	0	1	-	1	1	**3**
Manufacturability	0	0	0	0	0	-	1	**1**
Speed	0	1	0	1	0	0	-	**2**

Table 1. Binary piecewise comparison weights from the customer needs feedback.

A generation of 100 concepts were then made to have multiple ideas to work with when coming up with multiple concept assemblies. Four main concept assemblies were generated from the 100 concepts that the Mechanical Engineers thought would best work with customer needs. These concepts with the different combinations of concepts can be seen in Table 2.

	Hull	**Super Structure (Material)**	**Propulsion**	**Sensor Mount**	**Cooling System**	**Connection**
Concept 1	Cat/Mono	Same Material	Differential	Spider Rail	Active	N/a
Concept 2	Cat/Mono	Modular	Differential	Spider Rail	Active	Grenade Pins
Concept 3	Long Cat	Same Material	Differential	Spider Rail	Active	N/a
Concept 4	Long Cat	Modular	Differential	Spider Rail	Active	Snap Down

Table 2. Medium fidelity concept generation.

In this medium fidelity concept generation, the "Hull" section refers to the shape of the hull with "Cat/Mono" referring to a catamaran and mono hull hybrid and "Long Cat" referring to a long catamaran as the hull. The "Super Structure (Material)" refers to whether or not the component space was either modular or integrated into the hull; "Same Material" refers to the super structure being integrated into the hull while "Modular" refers to it being removable. The "Propulsion" section refers to the configuration for the propulsion system; "Differential" is a differential drive configuration. For the "Sensor Mount" section, this refers to the way the sensors were to be attached; "Spider Rail" refers to an adjustable and modular design to attach the sensors to the hull. "Cooling System" refers to the way in which the component space will be cooled off to avoid overheating; "Active" refers to a forced convection system that intakes outside air and expels internal air. "Connection" refers to the way a modular super structure would be attached to the hull; "Grenade Pins" and "Snap Down" are two connection types that would be used to connect the hull and the super structure. The four concept assemblies' sketches can be seen in the images below.

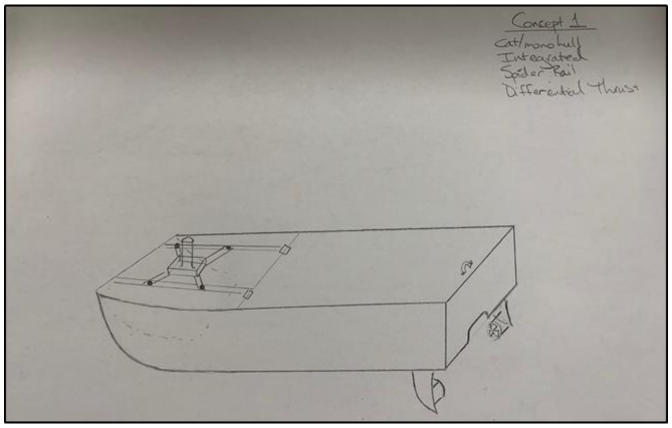


Image 1. Concept 1: A catamaran monohulled hybrid with an integrated super structure, spider rail sensor mount, and differential thrust.

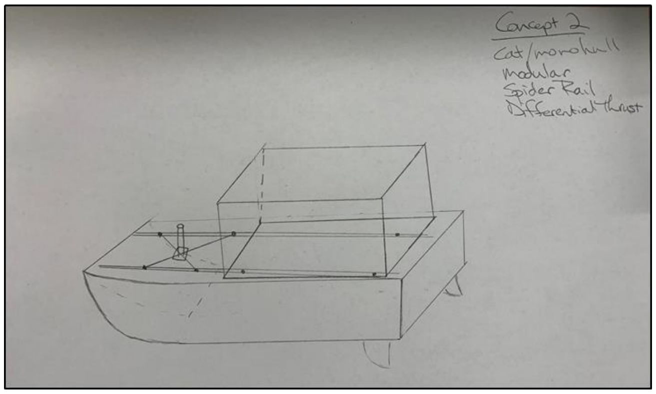


Image 2. Concept 2: A catamaran monohulled hybrid with a modular superstructure, spider rail sensor mount, and differential thrust.

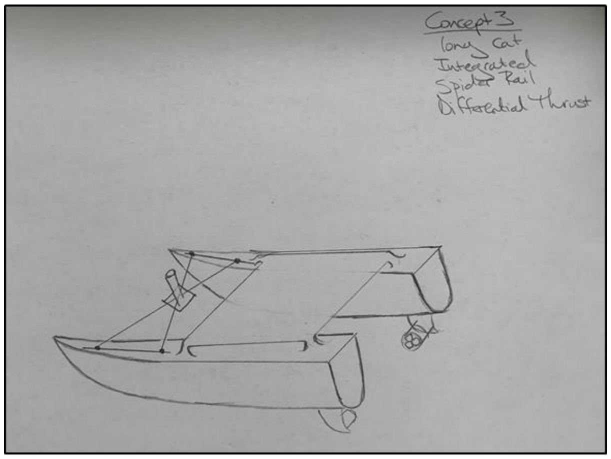


Image 3. Concept 3: A long catamaran hull with an integrated super structure, spider rail sensor mount, and differential thrust.

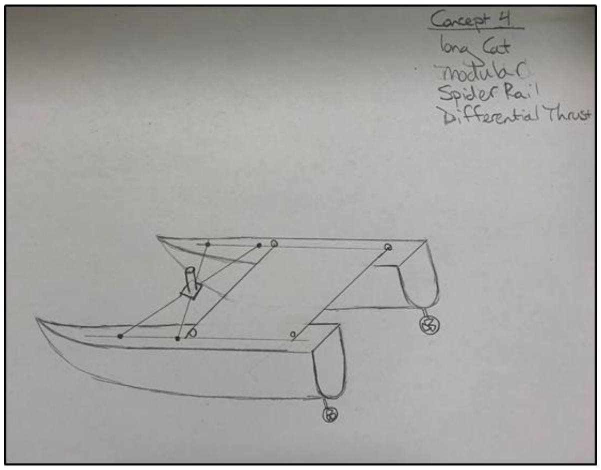


Image 4. Concept 4: A long catamaran hull with a modular super structure, spider rail sensor mount, and differential thrust.

Working from these four main concepts, they were compared to a datum, that datum being the competition boat from 2019. This can be seen in the Pugh charts in Table 3 and Table 4.

Selection Criteria	**DATUM (Wilson)**	**1**	**2**	**3**	**4**
Stability		+	+	+	+
Aesthetics		+	+	+	+
Maneuverability		+	+	+	+
Modularity		S	+	S	+
Deck Space		+	+	+	+
Manufacturability		+	+	+	+
Speed		+	+	+	+
Number of +'s		**6**	**7**	**6**	**7**
Number of -'s		**0**	**0**	**0**	**0**

Table 3. Pugh chart of concepts 1-4 compared to the 2019 competition boat.

Selection Criteria	**DATUM (Concept 4)**	**1**	**2**	**3**
Stability		S	S	S
Aesthetics		S	S	S
Maneuverability		+	+	S
Modularity		-	S	-
Deck Space		+	-	S
Manufacturability		-	-	-
Speed		+	+	+
Number of +'s		**3**	**3**	**1**
Number of -'s		**2**	**2**	**2**

Table 4. Pugh Chart of concepts 1-3 against the new datum, concept 4.

Based upon the results of the Pugh chart, concepts 1 and 2 both performed better than the new datum, being concept 4. To further analyze the best design combination, they were then put into a house of quality shown in Table 5.

Customer Requirements	**Importance Weight Factor**	Concept 1	Concept 2	Concept 3	Concept 4
Stability	5	3	3	3	3
Aesthetics	3	3	3	3	3
Maneuverability	6	1	1	3	3
Modularity	1	0	9	0	9
Deck Space	3	9	3	1	0
Manufacturability	1	3	3	9	9
Speed	2	3	3	1	1
Raw Score:	**189**	**66**	**57**	**56**	**62**

Table 5. The house of quality that scores each concept based upon the customer criteria weights from the binary piecewise comparison table.

Based upon the results from the house of quality, concept 1 became the design the team moved forward with. A higher fidelity model was generated in SolidWorks to create a 3-dimensional visualization which can be seen in image 5.

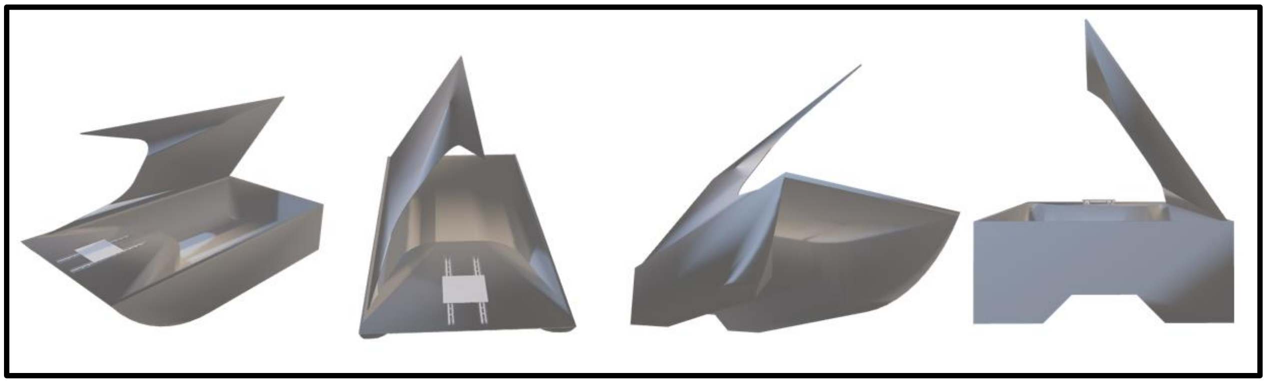


Image 5. High fidelity CAD drawings of the boat that are based upon concept 1's attributes.

With this design now created in SolidWorks as a baseline, the actual dimensions needed to be finalized. From the competitions guidelines of the boat having to be 3' x 3' x 6' and not weighing more than 140 lbs. Two cardboard prototypes of a tweaked design of the hull were made to see how much space was available for components and immediately it was noticed that the boat was too big being 32" x 60" for the hull's length and width. A new scale version of the boat was created with the dimensions being 30" x 50" x 25" and the team agreed that this was more suitable for the final scale of the boat. The boat was then recreated in CAD with the scale the team agreed upon. Other components were added to the CAD drawings, such as a modular fin-thruster attachment, latches and hinges for a lid access to the component space, fans for active cooling, and a gasket for waterproofing the interior around the lid. The final design is shown in Image 6.

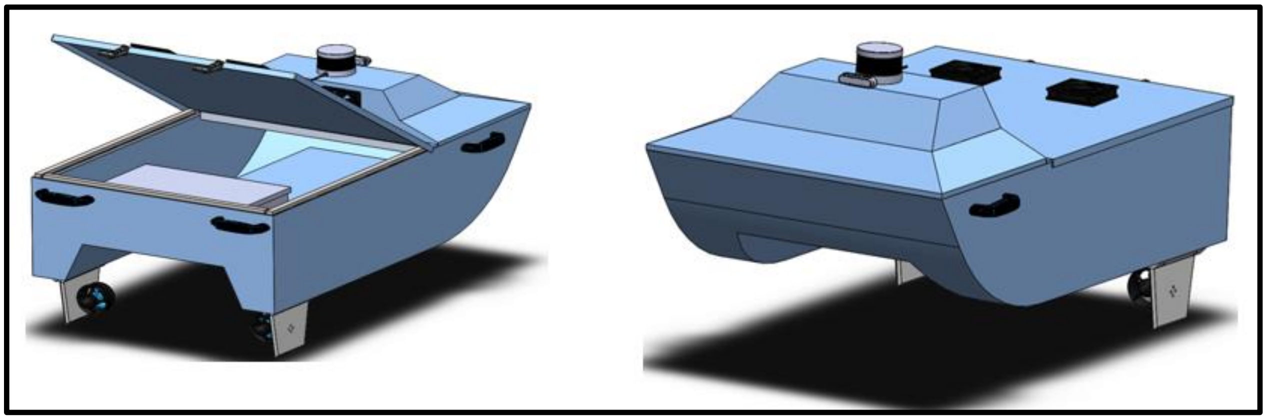


Image 6. Final CAD drawing of the boat hull design.

The next goal is to design sensor mounts for the CAD model in SolidWorks that are able to be modular, adjustable, and future proof. For the mounts to be modular, they need to be able to support different sensors in case other sensors want to be used. For the mounts to be adjustable, the design must be able to support varying translations and rotation of where it is mounted in case the sensors need to be moved. The sensor mounts also have to be future proof and in order for that to be accomplished, they must be easily manufactured in case if replacements are needed for any broken parts and also durable enough to withstand normal use by using higher strength materials. As far as manufacturing of the parts goes, the plan is to 3D print the parts with a combination of PLA and PETG since both are easily attainable and have great mechanical properties.

B. Manufacturing Process

Once the final boat hull was finalized and a SolidWorks CAD model was complete, the manufacturing process could begin. Initially, the material selection process was completed, and multiple materials were considered. A fiberglass/epoxy resin composite was chosen due to its low cost, easy manufacturability, anti-corrosiveness, and high strength to weight ratio. Specifically, a 6-ounce plain weave was chosen. “Because the fiber orientation directly impacts mechanical properties, it seems logical to orient as many of the layers as possible in the main load-carrying direction. While this approach may work for some structures, it is usually necessary to balance the load-carrying capability in several different directions, such as the 0°, +45°, -45°, and 90° directions.” [1] The Ship Structure Committee 403 was referenced to learn more about fiberglass and specifically how grainline orientation of the cloth is important to providing strength. Image 7 shows the grainline direction on the actual cloth while image 8 is an illustration of different grainline orientations and a proper lay-up configuration.

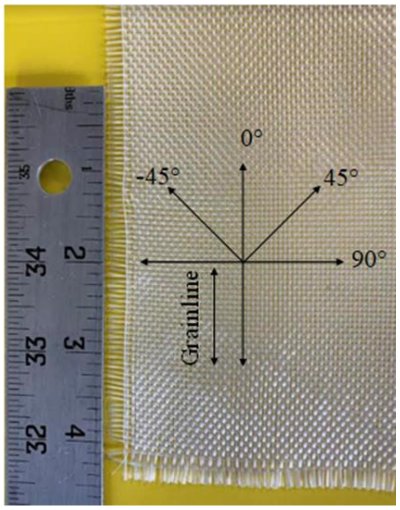


Image 7. Grainline orientation on actual cloth

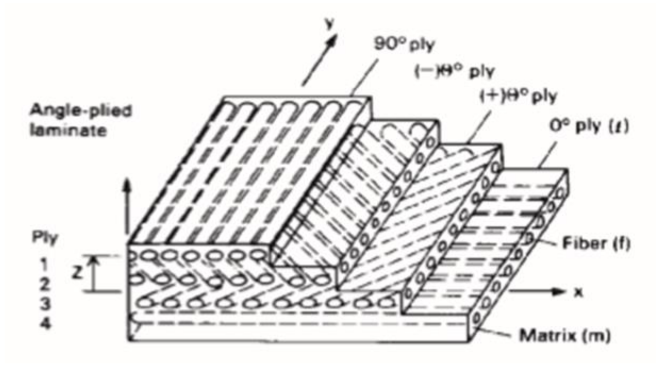


Image 8. Illustration of grainline orientation

Specifically, 3 layers of 6-ounce plain weave in varying grainline directions, 0°, 45°, 90° and one layer of fiberglass mat that is omnidirectional was selected. This layer configuration was selected after samples of differing layer numbers and grainline directions were produced. The selected lay-up composition achieved the best performance in practice tests.

Since a fiberglass composite was chosen, the construction method consisted of a hand lay-up process. For this process, a foam mold was constructed for the hull and the two lids separately using the dimensions from the CAD model. The boat mold can be seen in image 7. The fiberglass layers were individually applied to cover the mold and the epoxy resin was applied thus saturating the cloth, and the excess resin was squeegeed out. Image 8 shows the process of applying the resin to the cloth. Once the resin cured in about 12-15 hours, the rough edges were sanded down, and another layer was applied. When all layers had been applied the mold was removed. Two ½ inch thick layers of foam cut 4" x 30 " were placed in the pontoon portion of the hull. Finally, the exterior was painted with a moisture resistant paint and a moisture resistant tape was applied to the interior edges of the pontoon.

It is important to note that prior to producing the actual boat, the entire process was practiced by building small sample boats molds of foam and then the hand lay-up process was followed. Fiberglass construction includes many nuances that are only learned with experience. Technical details such as the working resin time, resin/hardener mixing ratios, how important it is to firmly secure the mold to a support and when to trim the excess composite materials were details that were improved with every sample. Having practiced on sample boats ultimately led to a superior final boat hull.

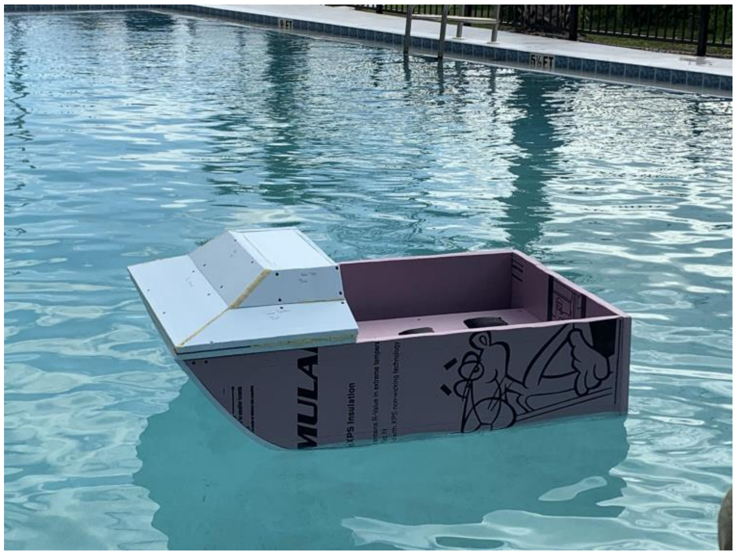


Image 9. The boat mold floating with 12 lbs. of weight applied

Image 10. Hand Lay-Up of Fiberglass

III. Hardware

The following section will provide information on the original plan for the wiring and integration of the hardware, in addition to the changes and actual work completed in the wiring and integration of the devices. An updated schematic and component list will also be presented to reflect the work done, along with the power requirements for the devices used.

A. Original Concept

In image 11 below, the original wiring and assembly diagram of the hardware is shown. The red lines represent the power connections, the blue lines represent the Ethernet connections, the green lines represent the serial connections, and the black lines represent data out. Four LiPo 4S batteries were to be connected to the power box, that was developed by a previous Senior Design Team, with 12 AWG wire, which in turn will supply the appropriate power requirements to each component through 20 AWG wire. The components were the Arduino MEGA board just above the power box, the ASUS router in the middle of the right side of the diagram, the LiDAR databox right above the router, the Intel Simply NUC computer on the top right, the NVIDIA Jetson Xavier on the top left, the Universal Serial Bus hub under the left computer, and the two electronic speed controllers (ESCs) on the left and right sides of the diagram. The router provides Ethernet connections to the two computers and LiDAR databox, and Wi-Fi to the ground station computer. Unlike the other components, the electronic speed controllers were to be connected to the power box through 12 AWG wire and also connected to the thrusters while receiving pulse width modulation (PWM) signal from the Arduino in order to control the thrusters.

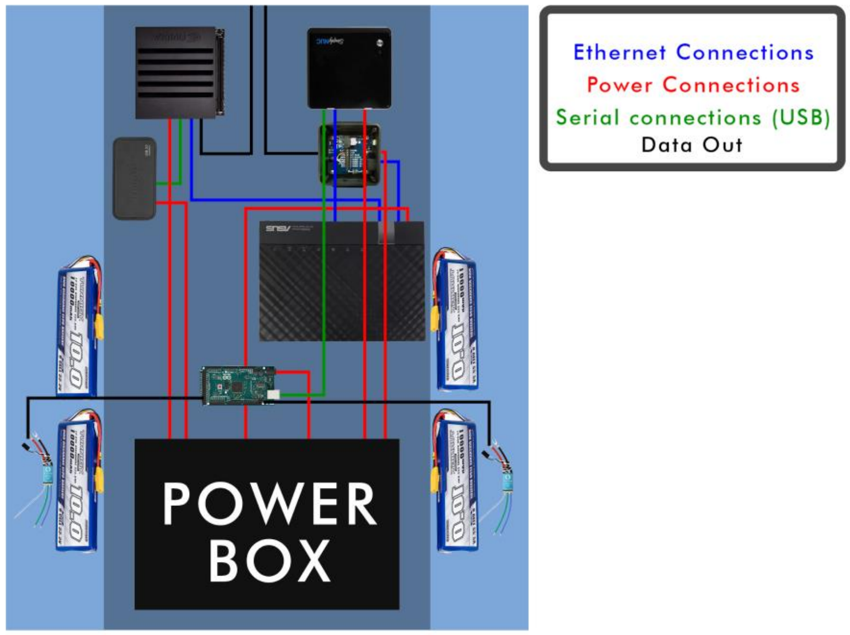


Image 11. The original wiring schematic and layout of the components and sensors.

B. Final Layout & Design

Working up to testing, certain components were changed or taken out of the system. To help reflect these changes, image 12 is provided below to illustrate the final wiring and assembly of devices. The red lines still represent the power connections, blue lines are the Ethernet connections, green lines are the serial connections, and the black lines represent data out. Image 13 shows the original list of components and the final list of components used, while also marking the changes between the two lists. One of the changes made was the switch from using a Jetson Xavier to another Simply NUC for one of the onboard computers (both onboard computers were Simply NUCs). To power both computers, instead of using the power box, the two Simply NUCs were each connected to a 12V to 19V converter, which were each powered by a 4S LiPo battery. The USB hub and RealSense camera were not integrated into the system and power circuit like originally planned. The second Arduino Mega board, inside the power box, was intended for digitally monitoring voltages and currents in the power box. However, due to more space being needed and this being beyond the scope of the project, this board was removed. This process included removing the board's respective step-down voltage converter and wires connecting to the busbar terminal inside of the power box.

To ensure each device would be powered and connected safely, certain types of connectors were used and soldered together, and wire ends were also tinned. XT60 connectors were soldered, and covered in heat shrink tubing, to the three sets of wires serving as the power inputs from the batteries to the power box. Wire ends from the two 24V outputs, for the ESCs and thrusters, were soldered to XT60 connectors and covered in heat shrink tubing. Two more XT60 connectors each had a set of wires soldered to it and covered in heat shrink tubing. These connectors were then plugged into the XT60 connectors from the two 24V outputs. The other ends of these wires were put into screw terminals, which were also connected to the ESCs' red and black wires. Screw terminals were used again to connect the ESCs to the thrusters, connecting the white, green and blue wires of each. The wire ends from the last 24V output on the power box were tinned and screwed into a barrel connector. This connection was used for the LiDAR data box. The wires for the 9V and 12V outputs on the box were also tinned and each screwed into their own barrel connector. The 9V connection was used for the Arduino Mega and the 12V connection was for the router. Input wires on the two 12V to 19V converters were soldered to extra wire to extend the length and this connection was covered with heat shrink tubing. The input and output wire ends of both converters were then tinned and screwed into barrel connectors. The 19V output on both are each for a Simply NUC computer.

Four Turnigy High Capacity 10000 mAh 4S LiPo Batteries were used to power the whole system, however a fifth battery could have been used (specifically for the third input on the power box). The batteries were placed in the pontoon of the test boat during testing, with two batteries in each pontoon. Two batteries were connected to the power box to power it and the other two batteries connected to the two 12V to 19V converters to provide power to the two computers. The components sat inside the plastic bin portion in the middle of the test boat, connecting to their respective connectors and outputs. The LiDAR itself was mounted to the front of the boat. The power box sat in the front of the bin while the Arduino Mega board, LiDAR databox, and two 12V to 19V converters were on top of the power box lid. The router sat at the back of the bin with the two computers and two ESCs on either side of it.

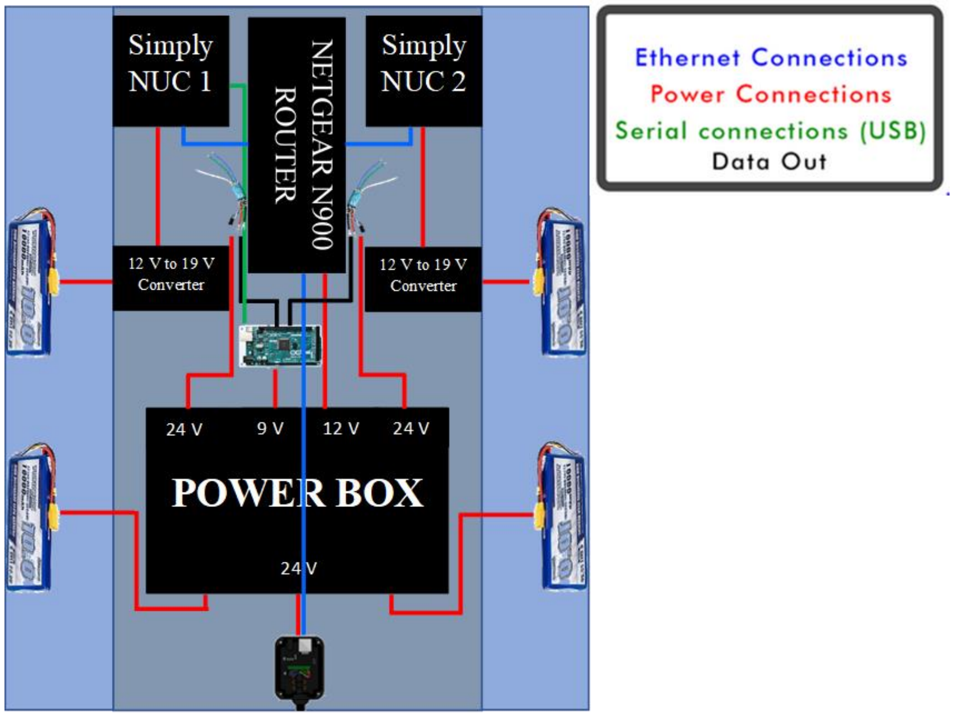
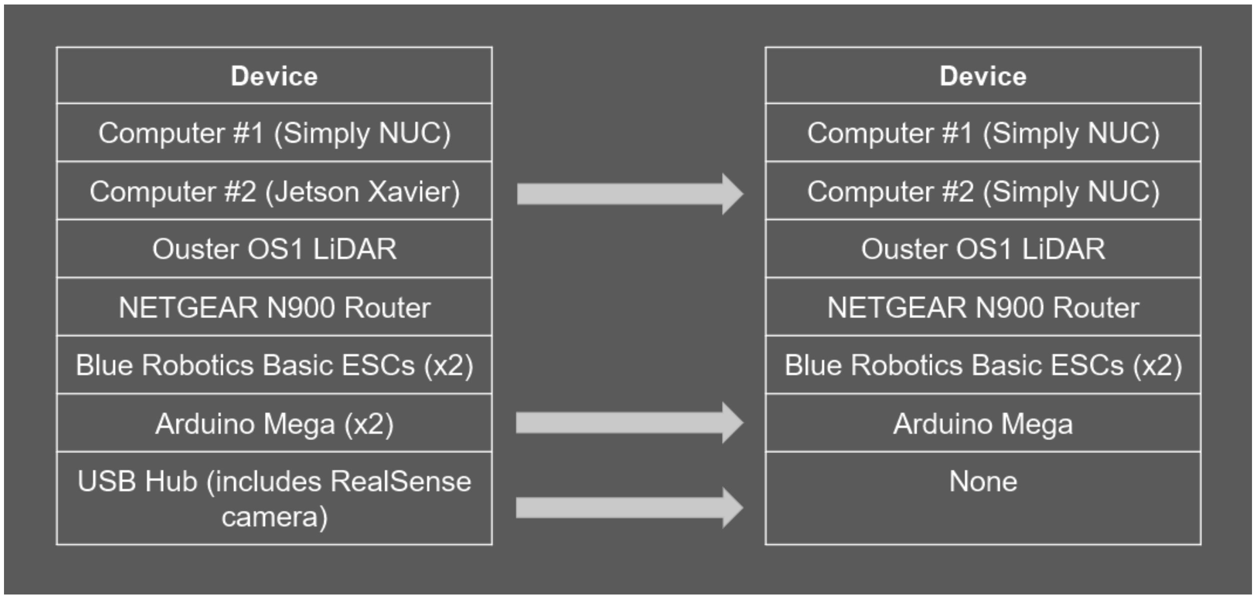


Image 12. The final wiring schematic and layout of the components and sensors.



Device	Device
Computer #1 (Simply NUC)	Computer #1 (Simply NUC)
Computer #2 (Jetson Xavier)	Computer #2 (Simply NUC)
Ouster OS1 LiDAR	Ouster OS1 LiDAR
NETGEAR N900 Router	NETGEAR N900 Router
Blue Robotics Basic ESCs (x2)	Blue Robotics Basic ESCs (x2)
Arduino Mega (x2)	Arduino Mega
USB Hub (includes RealSense camera)	None

Image 13. The original and final component list. The original list is on the left, while the final list is on the right; the arrows are indicating the changes.

C. Sensor Mount Design

Over the course of this semester, sensor mounting was split into two portions. The first portion focused on mounting the sensors to the newly fabricated vessel. Since this was a brand-new boat, any mounts that were previously made would not work or were ruined from the previous competition vehicle when the boat capsized. The new mounts that were to be made needed to fill a certain number of criteria when being designed.

Given the insight observed from the previous year's competition, the winning teams' featured mounts that were adjustable. Whether it was their Lidar, camera, or any other sensor, each team had some sort of articulation for adjusting. Image 14 shows the Roboboat 2019 competition winner's boat which features multiple adjustable mounts. This became a big inspiration and goal for the design of the new boat's sensor mounts.

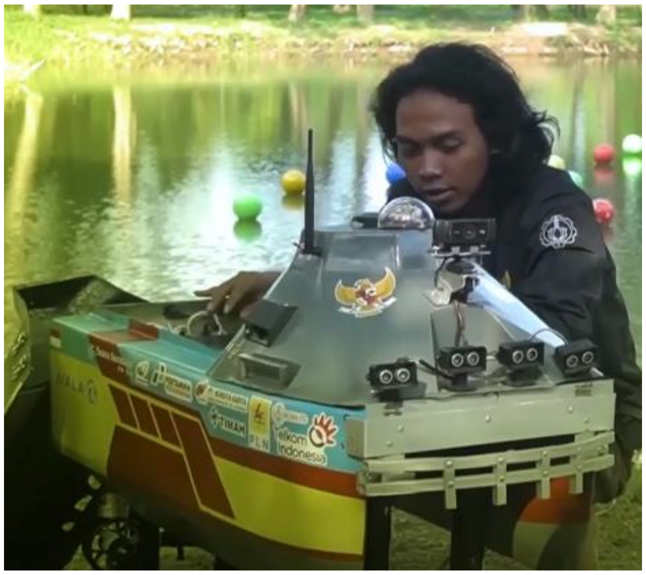


Image 14. Institut Teknologi Sepuluh Nopember's 2019 competition boat.

Since the school has many 3D printers, another goal was the ability to have spare mounts in case of damage and 3D printing offers a unique opportunity to not only reproduce the same result multiple times, but to manufacture in quick succession. These two guidelines were the driving force for the design process in CAD, aside from the physical limitations of the dimensions of the sensors themselves.

The first mount to be created was the modular fin mount shown in image 15. The fin features quick attachment and detachment from the hull in case of thruster failure or damage. This was an issue from the previous year's boat since the thrusters were hard to remove when there was a technical difficulty considering they were attached directly to the hull with epoxy. This modular fin design improves upon this by having the ability to have multiple thrusters ready to go at any time in case of failure and quickly re-set back up the boat.

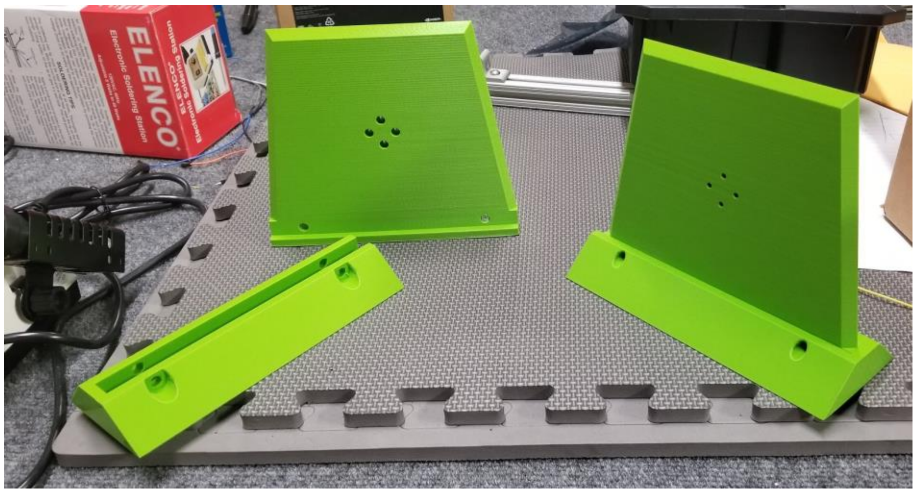


Image 15. An image depicting the modular fin mounts.

Upon closer inspection, the fins feature four countersunk holes for the M3 screws used to secure the thrusters to the fin. Once attached, two cotter pins go straight through the base mount and fin to secure each fin from falling off during operation. The base mount itself is the only portion directly mounted to the boat hull, which is to be secured with maritime epoxy.

The next mount created for the new boat was one that is both modular and adjustable. Since the highest priority sensor needed for testing was the Lidar, that was the main focus for the mount to be designed and manufactured. Image 16 shows the final rendering of the Lidar mount.

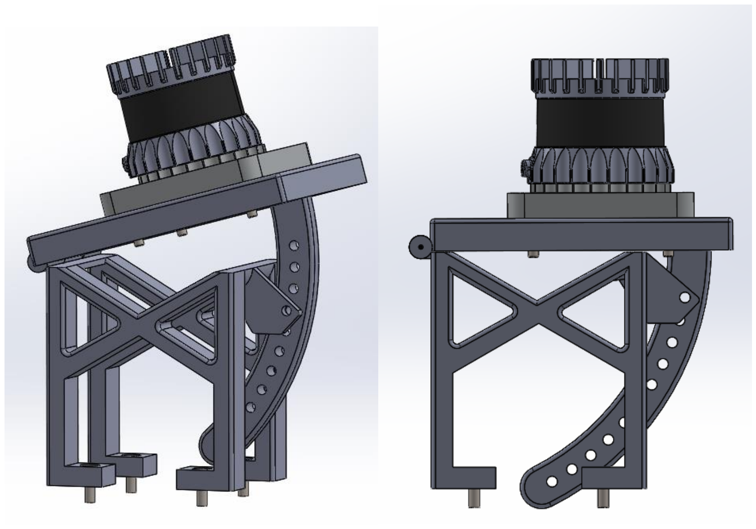


Image 16. An image depicting the Lidar mount.

The mount features an angular adjustment at which the lidar is locked in at. There are two printable pieces, the base cross beam mount and the base plate that the lidar attaches directly to. The base plate attaches to the Lidar with 4 M5 x 10 socket head cap screws with M5 hex nuts. The hinge uses a 0.047” diameter rc servo rod. The base mount uses a total of four 4-40 x ½” socket head cap screws to attach to two 8020 rails attached directly to the boat as well as a ¼” cotter pin to lock in the angle. The mount can adjust by upwards of 60 degrees from being level. This allows for adjustments of the bounds of the vertical resolution of the point cloud without having to adjust in software.

Unlike the modular fins, the Lidar mount never finished manufacturing. An Ender 3 was purchased and a Prusa Mk. 3 i3S was borrowed to speed up manufacturing time however both printers were damaged to the point of stopping the manufacturing process altogether. The Prusa was the first printer to be out of commission with a bad first layer adhesion causing a “blob” of filament to mold around the entire hot end itself which hardened and tore out the thermistor wire (the wire that regulates the temperature). This was an inexpensive fix although the time was an issue. The part itself was proprietary and unfortunately only shipped from the manufacturer itself, who is based out of the Czech Republic, and features a 3-week shipping time due to COVID 19 and international shipping constraints. The Ender 3 failed on the very next print when the extruder was clogged due to unknown reasons and eventually broke the adapter for the Bowden tube connection to the extruder driver. This was also an inexpensive part to replace but shipping took long due to COVID as well. When it was clear that the testing was primarily going to be done on the boat given to the team by the Tallahassee campus, focus was given to making sure the sensors were properly mounted to the testing boat. The Lidar fortunately only needed a ¼ - 20 screw to be mounted on an acrylic plate previously set up in the prior semester. The fins for the thrusters were also already mounted and did not need any modifications other than re-applying epoxy to ensure proper mounting for the fins. The new focus was then shifted toward the inside component mounting and Visual Feedback / VectorNav. When testing preparations were underway, it was seen that the internal component management was an issue that needed to be solved. Computers and routers were being stacked on top of each other like a tower and not secured by anything other than gravity. Components were also hard to access, which was an issue the previous boat and with the tight spaces. The issue did not arise because of lack of space but rather inefficiency in use of the space already available. This led to the solution of creating a quick slotted component holder made from foam, which can be seen in image 17.

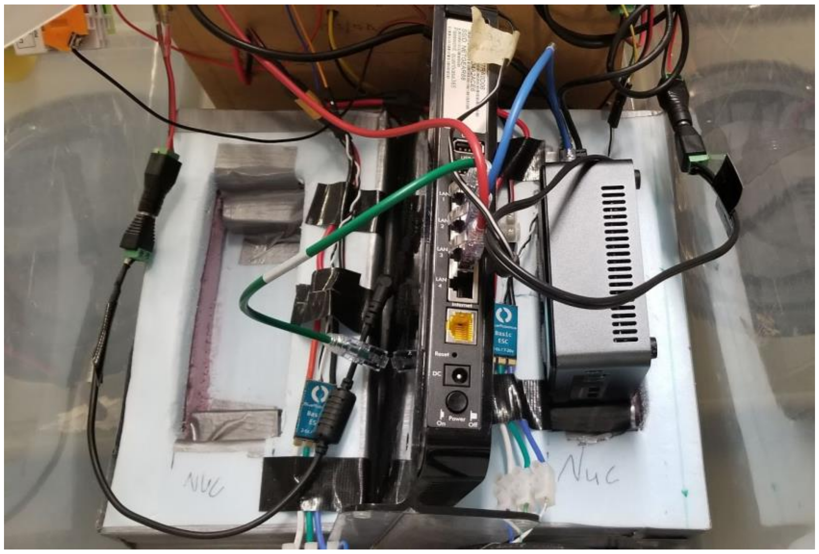


Image 17. An image depicting the component mounts.

The component mount features easy access to both power and data ports for running cables to and from without bending them at weird angles, which could cause issues electrically. This helps utilize the space that was being wasted as well as securing the components with the foam. Once the components were properly secured, focus was then directed toward setting up the Visual Feedback mount as well as the VectorNav. The result can be seen in image 18 and 19.

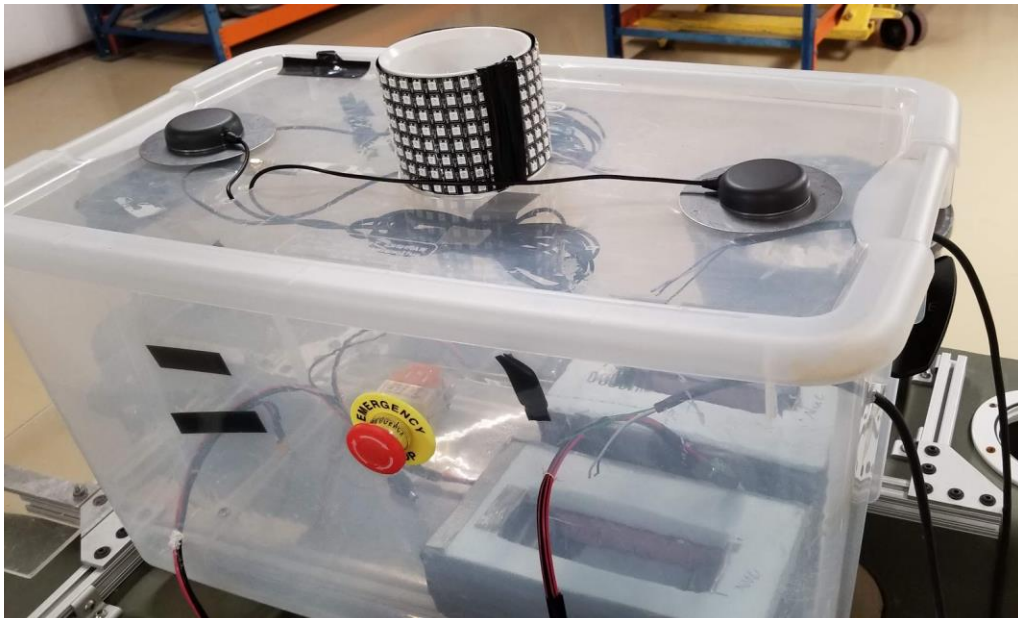


Image 18. Visual Feedback mount.

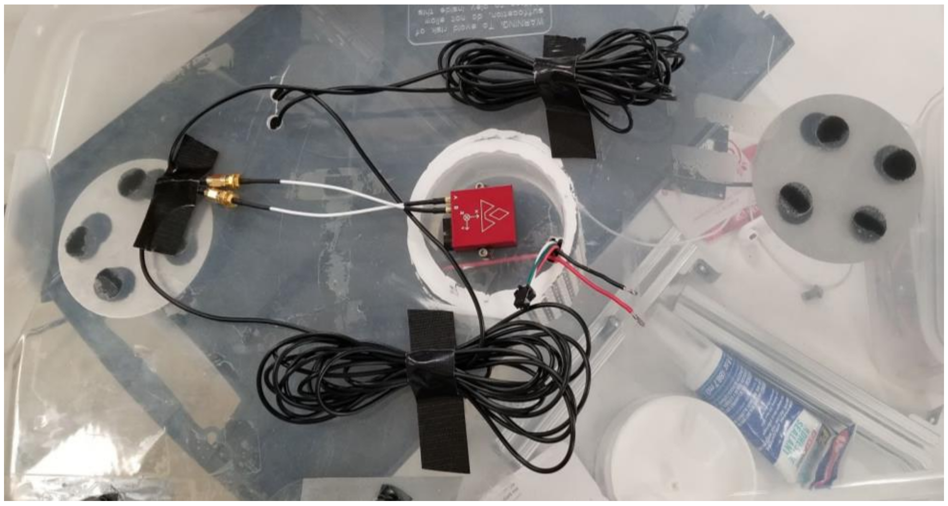


Image 19. VectorNav mount.

The VectorNav portion only consisted of two M3 screws that threaded into the top portion of the lid and two holes for wire routing. The Visual Feedback consisted of taking a PVC 3” coupler and taping the led tape to it. Holes were also drilled out for wires to be run into the center of the coupler which eventually was run through holes in the top part of the lid. The coupler itself was epoxied down to the top to ensure a strong bond to the lid. Overall, these mounts successfully worked for what was needed for testing.

D. Power Requirements

Table 6 shows the power requirements of each device used, as listed previously. Each device is listed with their rated voltage range, desired voltage, maximum rated current and maximum safety factor current. Also included is where to find the pictures of measuring the voltages for each component.

Device	Rated Voltage Range	Desired Voltage	Measured Voltage Pictures	Maximum Rated Current	Max. Safety Factor Current
Simply NUC (x2)	19 V	19 V	See Appendix C and Appendix C	3.42 A	3 A
LiDAR	22-26 V	24 V	See Appendix C	0.90 A	0.85 A
Router	12 V	12 V	See Appendix C	2.5 A	2.25 A
ESCs (x2)	7-26 V	24 V	See Appendix C and Appendix C	30 A	25 A
Arduino Mega	7-12 V	9 V	See Appendix C	1 A	0.85 A

Table 6. Power requirements for components.

IV. Software

This section will thoroughly describe the software platforms and methods used in this project.

A. Product Design Software

The team decided to use the Robotic Operating System (ROS) to implement task solving. ROS effectively creates a convenient solution to tie data together and forces all the sensors to communicate via the Transmission Control Protocol (TCP) connections. Additionally, due to the nature of ROS and its open source environment, existing algorithms written by the sensor manufacturers and other third parties can be used in conjunction with custom code and algorithms to make the vehicle exhibit desired behaviors and performances.

As stated above, for these reasons and its open source nature, ROS is the most realistic and effective solution to engineering this robot. Without it, it would be an extremely daunting task that would be very difficult to complete within the time frame of the senior design class.

The team also decided to use the Arduino library and all its functions. The Arduino is an extremely powerful and versatile microprocessor which can be harnessed with its Integrated-Development-Environment (IDE) to iterate functions that need to be called every moment during runtime. Specifically, for this team's use, it was used to fetch controlled motor commands and will transfer them to the motors via a PWM signal. This development IDE was implemented using C++ code.

B. Algorithm Design Software

ROS, as mentioned previously, is the middleware platform which connects all the data pieces. Contained within this environment are applications, or nodes. Data is published as a topic by nodes and these topics can be subscribed to by other nodes running simultaneously during the runtime of the ROS core application. It is similar to handing off data sets so everything can communicate effectively without any particular application taking too many liberties. This is known as a publisher-subscriber (Pub-Sub) architecture and is the fundamental nucleus of ROS.

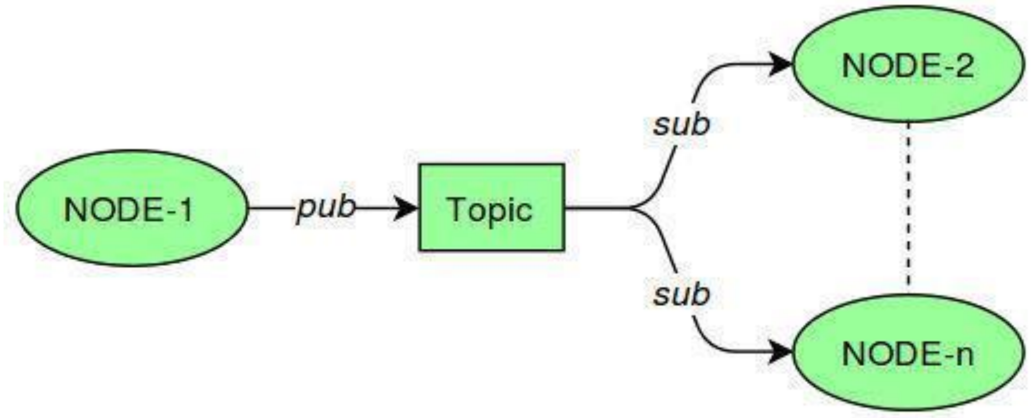


Figure 2. A block diagram of the ROS PUB-SUB system.

Specifically, in the case of the project, custom nodes containing algorithms were written and implemented in ROS. Data generated by the sensors were sent to ROS in compact data packets. Then, the data was processed by algorithms and node packages which work together in conjunction. After, the modified data sets were manipulated into commands for the motors that were modified by a controller as the last step. After the controller performed its functions, the commands were sent to the motor driver (Arduino) microcontroller. The controls were mixed on the Arduino and exported as a PWM command to the speed controllers, and in turn, the motors. Below is a diagram explaining the flow of data during runtime.

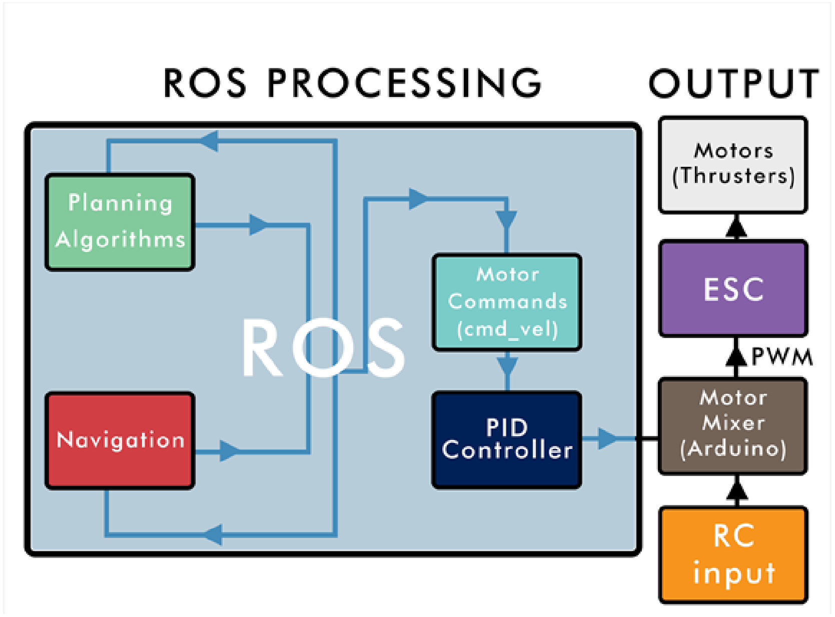


Figure 3. A Diagram depicting the flow of data in ROS.

C. Localization

Before any algorithms could be used and any autonomous locomotion achieved, the vehicle had to be localized in its own coordinate frame within ROS. This coordinate frame in the case of the boat needed two degrees of freedom. These being movement on the X and movement on the Y axis respectively. As the vehicle moves it needs to be able to understand how far it has travelled within the environment and what is located within the environment. The boat also needs to know its current speed, acceleration and orientation. These last three components make up what is known as odometry in ROS. Using frames of reference, it can be established that there will be a frame for each major component of the vehicle as well as a global frame that spans infinitely from the point of origin, or where ROS core was started. To achieve a frame of reference for odometry, and in turn achieving localization, it is imperative to first establish the boat's center of mass. This point is known as the base_link frame within ROS. Because the LiDAR will generate obstacle and environment data in real time, ROS must understand from where this data is being generated. The simplest solution that will be implemented is for another reference frame for the LiDAR, called base_laser, will be defined.

Now that there are two defined reference frames for both the LiDAR (Ouster LiDAR) and the vehicle we can relate the position of these two frames with a fixed distance because they are static with respect to one another. By doing this ROS is informed of where the LiDAR is with respect to the vehicle. In turn, this allows the vehicle to understand where obstacle data, which is generated by the LiDAR in its own frame, is with respect to itself. This effectively allows obstacle data to be interpreted all from the position of the vehicle. At this point the vehicle needs to generate the current speed, position and orientation. This is achieved by placing the IMU chip (VectorNav) onto the vehicle. After, a reference frame was defined for the IMU in a similar fashion to the LiDAR. The IMU frame was linked to the base frame which allows position speed, acceleration, and orientation position to be directly correlated to the vehicle itself. The linking of these frames sets up what is commonly known as a transform tree. After this tree of frames is established, ROS will reference the orientation and kinetic information to the global origin point. This indicates that the boat is successfully localized in the ROS coordinate frame.

D. Path Planning

The algorithms written by the team ran in conjunction to several open-source packages. One software wrapper, or collection of packages used, is called Navigation. Navigation contains several sets of packages. One of the most heavily used packages within the set is called move_base. Navigation works by taking in the generated LiDAR data which is then localized to the vehicle and constructed into what is called a costmap (refer to image 4). The costmap is constructed from an array of data values all of which are random variables called the occupancy grid. Each cell contains a value of probability. This probability being the odds of an obstacle being in the space represented by that array element. Parameters can be altered to determine how great the cost would be if the vehicle experienced a collision with an obstacle. Navigation will employ move_base to generate movement commands called cmd_vel, or command velocity. In order for Navigation to start working its powerful obstacle avoidance algorithms, it must receive a setpoint, which is simply a waypoint in the ROS coordinate frame. After the setpoint is achieved, Navigation will generate a movement command using move_base.

Perhaps the most difficult task is generating the waypoints for Navigation to receive. The starting waypoint is provided to the team by the competition, so the algorithms are written from that point as the origin. The generated points of each respective obstacle are averaged and only one point will represent an entire obstacle. Because the navigation channel starts with two buoys, two single points using this method can be found with respect to the global frame which is linked to the vehicle. Using the midpoint formula:

$$m = \frac{p_1+p_2}{2} \tag{1}$$

Aside from the given starting waypoint, the midpoint is the first waypoint passed into Navigation. After the boat is at the midpoint location it will need to continue forward with its motion. ROS navigation relies heavily on the use of waypoints, so for the boat to continue path planning, a waypoint will need to be generated that is some distance forward from the vehicle's position. This is because the boat may not sense the next set of buoys. Therefore, a vector from the midpoint to the rightmost buoy will be determined. This vector will then be normalized to unity and rotated 90 degrees. Firstly, the vector to the buoy will be divided by its magnitude. Then using linear algebra, the rotational matrix can be applied to force the 90-degree rotation. This rotational equation is below,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2}$$

If (phi) is evaluated at 90 degrees, the new x coordinate is the negative of the original y coordinate and the new y coordinate is the original x coordinate. From this point, the vector assumed to be unity, can be multiplied by any value. The higher the value is, the further it will move the new waypoint from the midpoint. This can be dynamically changed depending on the situation. To start 25 meters will be used. Image 20 below shows each vector as the arithmetic and linear algebra is applied.

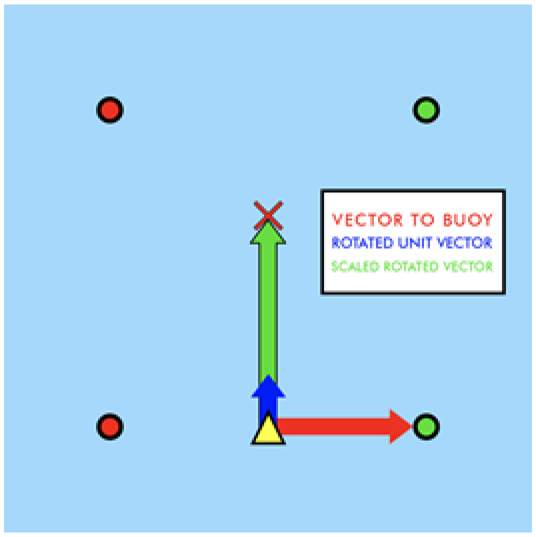


Image 20. A Visual Representation of the Vector Rotation for the Navigation Channel.

Theoretically, this new waypoint should be enough to get the vehicle to the next set of buoys. After the new buoys are detected, an interrupt will occur. This interrupt will run the first subroutine using the next midpoint as the new waypoint. This effectively will repeat the same process as before and place the vehicle between the next set of buoys. After, using the vector rotation process twice more, a waypoint will be placed to the right of the last set of buoys. From here, the vehicle will be given the origin of the ROS coordinate frame as a waypoint and the boat will return to the start of the obstacle.

In terms of development environments for syntax checks, the software created within this project used a modified version of Visual Studio Code, which can be configured to understand ROS syntax. Using C++, most of the functions are defined in custom classes. These classes include *detection*, *task* and *buoys.* These classes serve to facilitate their own variables which will be manipulated within each executable. The executable will import the data from the ROS topic and, depending on the task, logic was written using vector algebra to calculate target waypoints for the boat to travel to. These waypoints generate intended speeds which then are ferried to the controller and then the motors. Each task changes and therefore each method of logical analysis and task solving will change as well. All this functionality is defined in the Visual Studio Code IDE and is compiled similarly.

Much of the code written for this project can be viewed in the appendix section of this document. This code is just a sample of what has been developed over the project period.

E. Controller Design

The controller for this project follows the proportion-integral-derivative control theory concept. This controller theory takes the idea that the desired output can be reached by finding the current error and correcting it by driving it to zero. The way the error is driven to zero is by multiplying it by a proportionate amount, integrating it to prevent saturation errors and then taking its derivative to prevent an exceptional overshoot. Below is a block diagram of the system to further illustrate its concept.

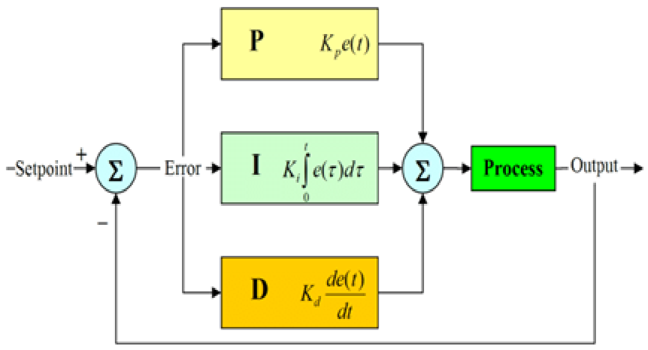


Figure 4. A block diagram of the PID controller.

This controller was written in C++ and is implemented in the ROS environment. It should be noted that the blue and red values are uncontrolled command velocities from the move_base package. After the PID controller is implemented, the controlled outputs, or controlled commands velocities will be outputted as a ramped function. This ramped function will ensure a smooth output and will directly correlate to smooth accelerations. The controller was lightly tested on hardware last semester, and its data was plotted and is described in image 21 below.

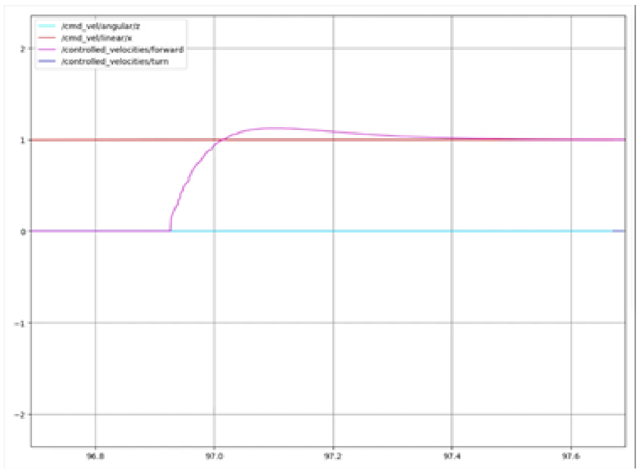


Image 21. A plot of the PID in action before tuning it.

With any controller, the gains of each respective channel must be tuned to achieve the most efficient curve for the tasks. Over the semester, the PID controller was tuned so that the controller was more responsive and able to handle commands better. Image 22 below, shows the updated output of the PID controller.

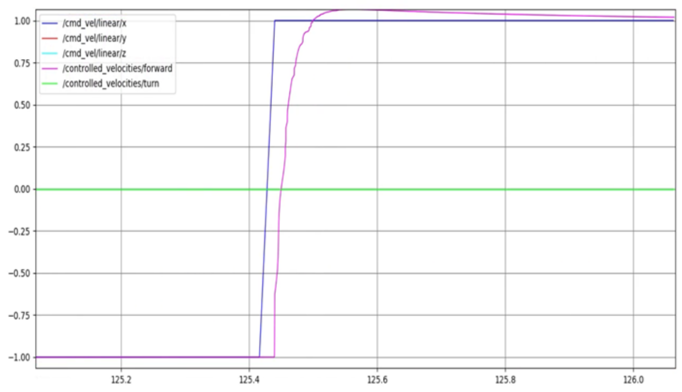


Image 22. A diagram of the updated PID in action.

G. Motor Mixer

After the PID modifies the output from move_base the command velocities are ferried to the motor mixer microcontroller. The motor mixer intakes both information from ROS and the RC receiver and then remaps those values into PWM values to be sent to the ESC controlling the thrusters. Since the boat is using differential drive, the code takes in a linear x and angular z velocity and by using equations 3 and 4, this generates an individual motor signal.

In the equations, v is the linear velocity, ω is the angular velocity, C1 and C2 are constants that will be changed experimentally to trim the motors, and ***φ***1 and ***φ***2 are each thruster.

$$v = C_1 * \varphi_1 + C_2 * \varphi_2 \tag{3}$$

$$\omega = C_1 * \varphi_1 - C_2 * \varphi_2 \tag{4}$$

V. Testing

The following section will recount the process executed to test the hull, mounts, power and software for the project. Also included is an updated testable requirements table with results on if each requirement passed or not.

A. Testing Process and Procedure

i. Boat Hull

Testing for the boat hull took place in Courtney's, the Manufacturing Lead, hot tub which simulated the actual competition closely because it was fresh water with no tidal current and minimal wave action. The boat was placed in the hot tub and eight dive weights each weighing three pounds were arranged in the hull. The components weigh ~22 lbs., so 24 lbs. was a close estimate. The timer was started, and the boat was observed for 30 minutes to monitor any water encroaching the interior of the hull. After 30 minutes it was seen that there was no water in the interior portion. This can be seen in images 23 and 24. Also, the waterline was measured to be three inches on each side and a level was placed on the higher central section and it was level throughout the test. For the deflection test, the hull was placed on a table and measured from the table to the central portion that would not be in the water. The dive weights weighing 24 lbs. were placed in the stern section and measured again. The before and after adding weights can be seen in images 25 and 26. Image 27 shows the weights placed in the hull during this testing. The boat hull is to have minimal weight, so the hull was placed on a scale and weighed. The total weight of the hull and the front lid was 18.4 lbs. The hull weighed 14.4 lbs. and the lid weighed 4 lbs. The front lid and hull being weighed are seen in images 28 and 29.

Image 23. The boat hull floating while carrying 24 pounds for 30 minutes with no leakage

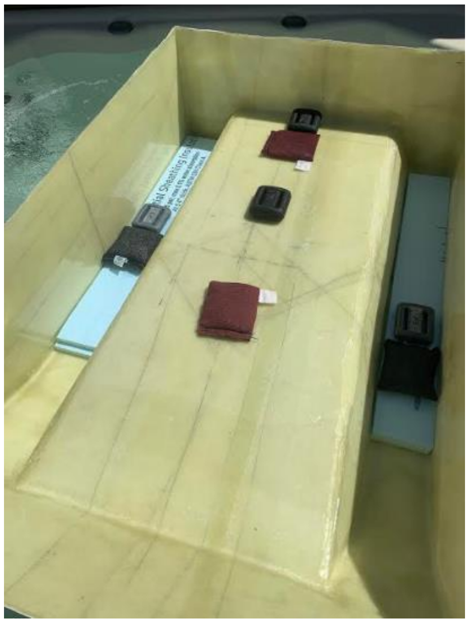


Image 24. The floatation test with the dive weights distributed correctly.

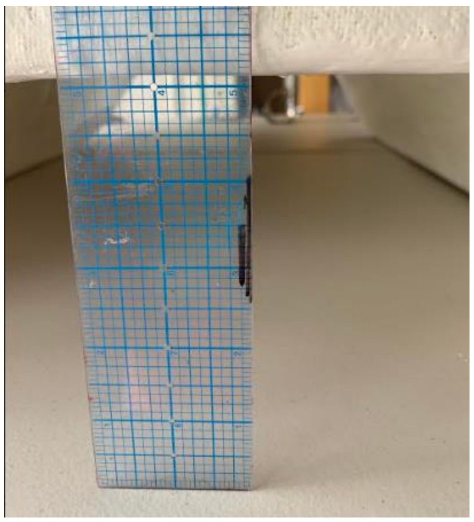


Image 25. The deflection test before weight

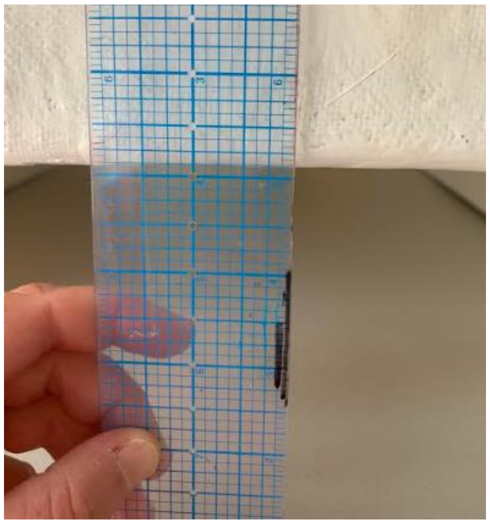


Image 26. The deflection test after weight

Image 27. The dive weights placed in the hull during the deflection test

Image 28. Front Lid Weight

Image 29. Hull Weight

ii. Mounts, Power, & Software.

The power, mounts and software system were tested using a prototype boat nicknamed "the tank," as shown in image 30, below. This boat is constructed of two large pontoons and a plastic bin attached by steel 80/20 rails. The two pontoons are held in position by rebar. This heavy frame is extremely stable and buoyant. This allows for safe testing of equipment and power without risk of damaging or flipping the boat, should a test fail.

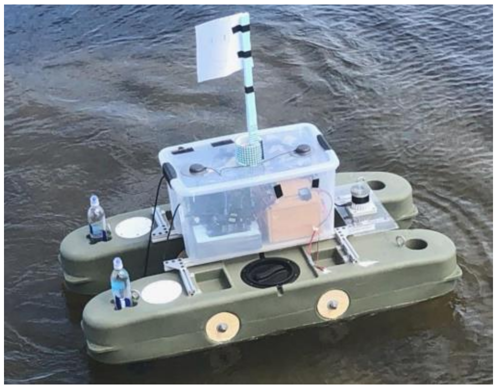


Image 30. Tank, the test boat.

First, the output voltage of the wires that were used for the components were tested according to Table 6. Before any test, the batteries were checked to ensure they were within a range of 14.8 V to 16.7 V (optimally above 15.5 V). After the components were plugged in, the voltages could then be checked to make sure the voltages were still the same.

To begin with, the power and software, specifically the remote-control portion of the code, were tested in a pool to ensure that the safety features of the system were working in a controlled environment.

This initial test allowed the power team to validate that their wiring system was set up correctly and functioning as expected. The power team was also able to double check the recorded voltages and currents were being sent to the sensors correctly. Image 31 shows the tank being tested in the pool.

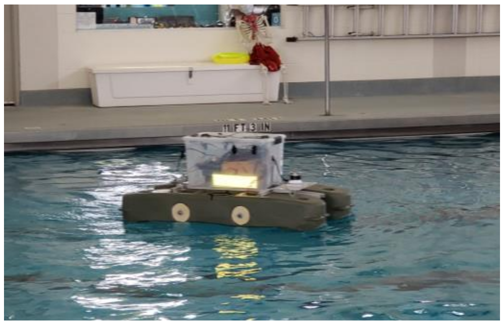


Image 31. An image demonstrating the manual control capabilities.

This test also proved the abilities of the remote control. Additionally, it demonstrated that the remote control is not only able to manually, but also allowed for three different modes. These modes make up the team’s software safety system. The boat can operate in autonomous mode, which is denoted by a blue led light, manual, a yellow led light, and red, which signifies that the boat has been put into safety mode.

The safety switch is triggered using the remote control which is triggered by a toggle switch on the controller. When triggered, the microcontroller will not send PWM signals to the electronic speed controllers which effectively halts any commands being sent to the thrusters. This will ensure that with the safety switch being triggered, there will be no possibility of the boat moving allowing for safe handling of the boat in the case of an emergency.

Once this test passed, the next step of the testing procedure was to test the boat in the bay to have space to test the autonomous software. During this test, the cost map created by combining our sensors with the ROS navigation stack and move base was tested. This cost map is in image 32, below.

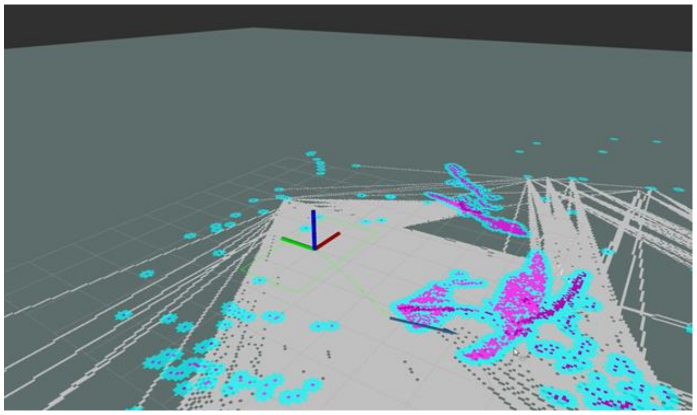


Image 32. An image of the costmap generated during testing on the bay using ROS navigation stack and move base.

This test proved the boats ability to follow waypoint commands while identifying objects and calculating the best path forward.

The final test completed while on the water was the straight-line test. This test proved the validity of the code when driving the vehicle in a straight line. This test was essential to test the boat's ability to complete the mandatory navigation path for the competition. This task requires the boat to enter a straight channel between two buoys that extends between 50 to 100 ft to a second set of buoys that mark the end of the channel. A mockup of the navigation channel is displayed in image 33 below.

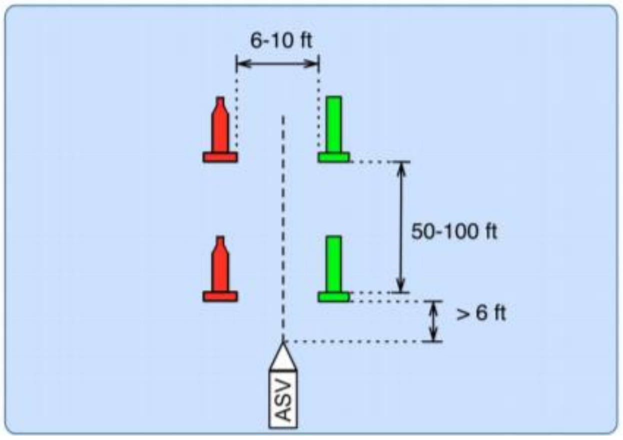


Image 33. The above image is an illustration of the mandatory navigation channel.

B. Testable Requirements & Results

Below is a tabulated list of the testable requirements that the team tested (Table 7). It has been updated since the original proposed requirements and these changes are shown in the table below. Please see Appendix B for the original testable requirements table.

Requirement	**Testing Method**	**What is Success?**	**Passed (Y/N)**
	Hull		
Hull Floats	Place completed hull in a swimming pool.	The hull does not sink, it floats.	**Y**
Hull Carries 15 lbs	While in the swimming pool, dive weights will be added incrementally until 15 lbs is reached (dive weights are 3 lbs each).	The hull will carry 15 lbs with the pontoons only be submerged less than 4 inches.	**Y**
Hull weighs <25 lbs	Place hull on scale and read weight.	Weight is < 25 lbs.	**Y**
Hull doesn’t leak	Place hull in pool carrying 15 lbs for a minimum of 30 minutes.	Hull has no water in the interior.	**Y**
Minimal Deflection	Place 9 lbs on the center section and measure deflection with a ruler.	The measured deflection will be less than ⅛”.	**Y**

Hardware/Wiring (Components Not Connected)			
Power output for the Ouster OS1-16 LiDAR (not connected)	Using a multimeter, measure the voltage output from the power source to the Ouster OS1-16.	The voltage is within the range of 22-26 V, optimally at 24 V.	**Y**
Power output for the two ESCs (not connected)	Using a multimeter, measure the voltage output from the power source to the two ESCs.	The voltage, for each ESC, is within the range of 7-26 V, optimally at 16 V.	**Y**
Power output for the Arduino Mega (not connected)	Using a multimeter, measure the voltage output from the power source to the Arduino Mega.	The voltage is within the range of 7-12 V, optimally at 9 V.	**Y**
Power output for the NETGEAR N900 Wireless Router (not connected)	Using a multimeter, measure the voltage output from the power source to the NETGEAR N900 Wireless Router.	The voltage is within the range of 12-19 V.	**Y**
Power output for the two Simply NUC computers (not connected)	Using a multimeter, measure the voltage output from the power source to the Simply NUCs.	The voltage is within the range of 12-19 V.	**Y**
Hardware/Wiring (Components Connected and ON)			
Power output connection to the Ouster OS1-16 LiDAR (connected)	Using a multimeter, measure the voltage output and current draw to the Ouster OS1-16. After measuring the voltage, divide the maximum allowed power by this measured voltage to calculate the maximum allowed current.	The voltage is within the range of 22-26 V, optimally at 24 V. The power is within the range of 14-20 W (peak 22 W at startup).	**Y**
Power output connection to the two ESCs (connected)	Using a multimeter, measure the voltage output and current draw to the two ESCs.	The voltage, for each ESC, is within the range of 7-26 V, optimally at 16 V. The max current (constant), for each ESC, is 30 A.	**Y**
Power output connection to the Arduino Mega (connected)	Using a multimeter, measure the voltage output and current draw to the Arduino Mega.	The voltage is within the range of 7-12 V, optimally at 9 V.	**Y**
Power output connection to the NETGEAR N900 Wireless Router (connected)	Using a multimeter, measure the voltage output and current draw to the NETGEAR N900 Wireless Router.	The voltage is within the range of 12-19 V. The current does not exceed 2.5 A.	**Y**
Power output connection to the two Simply NUC computers (connected)	Using a multimeter, measure the voltage output and current draw to the Simply NUCs.	The voltage is within the range of 12-19 V. The current must not exceed 3 A	**Y**

Power output connection to the Ouster OS1-16 LiDAR (connected)	The LiDAR will be turned on and observed for 3 minutes.	The LiDAR runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	**Y**
Power output connection to the two ESCs (connected)	The two ESCs will be turned on and observed for 3 minutes.	The two ESCs run smoothly without any brownouts, shutting off, malfunctioning, or overheating.	**Y**
Power output connection to the Arduino Mega (connected)	The Arduino Mega will be turned on and observed for 3 minutes.	The Arduino Mega runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	**Y**
Power output connection to the NETGEAR N900 Wireless Router (connected)	The NETGEAR N900 will be turned on and observed for 3 minutes.	The NETGEAR N900 runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	**Y**
Power output connection to the two Simply NUC computers (connected)	The Simply NUCs will be turned on and observed for 3 minutes.	The Simply NUCs run smoothly without any brownouts, shutting off, malfunctioning, or overheating.	**Y**
ESCs and Thrusters	Run the thrusters, which are connected to the ESCs, to max power. Measure the voltage and the current.	The voltage does not exceed 26 V, and the current does not exceed 30 amps.	**Y**
Turnigy High Capacity 10000mAh 4S LiPo Batteries	During testing, check the voltage output from the batteries.	The voltage range is maintained at 14.8-16.3 V.	**Y**
Sensor Design			
Sensor mounts articulate	Sensors will be placed on the mount and the angle will be adjusted by raising and lowering the mount.	Mount is able to adjust to different angles.	**Y**
Sensor mount will be adaptable	Mounts created will be modular to fit onto two 80/20 rails.	Mount will fit on the 80/20 rail showing that the sizing is correct and other mounts can be made using these sizings.	**Y**
Mounts are easily replaceable	The mounts will be 3D printed and spares will be made.	Print can be made on most 3D printer beds with common filament (PLA or	**Y**

		PETG).	
Software			
Boat detects obstacles	Obstacles will be introduced in a controlled manner and the data will be logged.	Software accurately and repeatedly identifies obstacles.	**Y**
A PID controller is capable of creating smooth continuous motion.	System will be driven using a PID controller.	System moves in a smooth and continuous manner.	**Y**
Boat Localized	System will be traveled around a specific path several times and the data logged.	The data points gathered at each point will agree with each other (within a 10% margin of error).	**Y**
Basic Waypoint Navigation Completed	System will be tasked with a waypoint within ROS.	System arrives at the waypoint within a reasonable amount of time.	**Y**

Table 7. Testable Requirements Checklist.

These compiled requirements served as a checklist for everything needed to stay efficient, productive, and successful during the test of the vehicle.

VI. Conclusion

This project was the culmination of three semesters of work from two different teams. From the design of the hull, to the creation of the boat and testing of the system, this project created a basis from which future teams can start from and build to compete in future Roboboat competitions. This paper is an attempt to demonstrate the work done on this project.

Appendix A

The PID Node Executable

```
/***********************************************
 *   Mark Hartzog  <markthartzog@gmail.com>     *
 ***********************************************/

#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "controller/Drive.h"
#include "stdio.h"
#include "pid.h"

// Define Global Variables

float linear_vel;
float angular_vel;
float process_var_x = 0.0;
float process_var_z = 0.0;
float previous_error_x = 0.0;
float previous_error_z = 0.0;

// Define callback to unfiltered cmd_vel
void cmdvelCallback(const geometry_msgs::Twist vel){

// Set the x and z equal to data published by an unfiltered cmd_vel
linear_vel = vel.linear.x;
angular_vel = vel.angular.z;

}


int main(int argc, char **argv) {

// Initialize ROS node
   ros::init(argc, argv, "pid");

   ros::NodeHandle nh;

// Subsrcribe to unfiltered cmd_vel
   ros::Subscriber sub = nh.subscribe("/cmd_vel", 1, cmdvelCallback);

// Define publisher for filtered cmd_vel
```

```
    ros::Publisher controlled = nh.advertise<controller::Drive>("/controlled_velocities", 1);

// Define a loop rate to prevent overflow of data to the thread
    ros::Rate loop_rate(25);

// Define a handler for the PID class
    PID pid;

    // Define the PID gains and feed them into the Class
    // In Order: Kd, Ki, Kd, dt

    float pgain_x = 0.025;
    float igain_x = 0.0028;
    float dgain_x = 0.0066;
    float dt_x = 0.052;
    float max_x = 2.0;
    float min_x = -2.0;

    // Define the PID gains and feed them into the Class
    // In Order: Kd, Ki, Kd, dt

    float pgain_z = 0.025;
    float igain_z = 0.0033;
    float dgain_z = 0.0062;
    float dt_z = 0.052;
    float max_z = 1.0;
    float min_z = -1.0;


    // Define an object Twist
    geometry_msgs::Twist vel;

    // Define an object Twist
    controller::Drive drive;

    // Take in the X the goals
    pid.valueslinear(pgain_x, igain_x, dgain_x, dt_x, max_x, min_x);
    // Take in the Z the goals
    pid.valuesangular(pgain_z, igain_z, dgain_z, dt_z, max_z, min_z);

    // Define the increment variables
    float increment_x = 0.0;
    float increment_z = 0.0;
    ROS_INFO("The PID controller is on...");

while (ros::ok()) {

    // Checks the linear input to create limiter

    if (linear_vel > max_x){
        ROS_WARN("\nThe incoming linear cmd_vel exceeds limits. Setpoint being set to ([%f]):", max_x);
        linear_vel = max_x;
    } else if (linear_vel < min_x){
        linear_vel = min_x;
        }
    // Checks the angular input to create limiter
    if (angular_vel > max_z){
    ROS_WARN("\nThe incoming angular cmd_vel exceeds limits. Setpoint being set to ([%f]):", min_x);
        angular_vel = max_z;
    } else if (angular_vel < min_z){
        angular_vel = min_z;
        }

    // Call the control function of the linear x
    increment_x = pid.controllinear(linear_vel, process_var_x, previous_error_x);
    // Call the control function of the angular z
    increment_z = pid.controlangular(angular_vel, process_var_z, previous_error_z);

    // Feed in previous error
    previous_error_x = linear_vel - process_var_x;
    previous_error_z = angular_vel - process_var_z;

    // Add new increment contribution to the previous process variable
    process_var_x += increment_x;
    process_var_z += increment_z;

    // Set the velocities equal to the publisher data
    drive.forward = process_var_x;
    drive.turn = process_var_z;

    // Publish
    controlled.publish(drive);

    // Spin and sleep
    ros::spinOnce();
    loop_rate.sleep();
}

    return 0;

}
```

The PID Header File

```
/*
Mark Hartzog <markthartzog@gmail.com>
Special thanks to Bradley J. Snyder <snyder.bradleyj@gmail.com>
*/

#include "ros/ros.h"
#include "cmath"

class PID {

public:

    // Define all varibles for linear
    float KP_X;
    float KD_X;
    float KI_X;
    float dt_X;
    float max_X;
    float min_X;
    float error_X;
    float integral_X;
    float derivative_X;
    float previous_error_X;

    // Define all varibles for angular
    float KP_Z;
    float KD_Z;
    float KI_Z;
    float dt_Z;
    float max_Z;
    float min_Z;
    float error_Z;
    float integral_Z;
    float derivative_Z;
    float previous_error_Z;
    // The PID function prototype which allows the transfer of values from the main exe for the linear control
    void valueslinear(float KP_X, float KI_X, float KD_X, float dt_X, float max_X, float min_X);

    // The PID function prototype which allows the transfer of values from the main exe for the angular
control
    void valuesangular(float KP_Z, float KI_Z, float KD_Z, float dt_Z, float max_Z, float min_Z);

    // Defines the control loop function feeds in setpoint variable and process variable
    float controllinear(float SP_X, float PV_X, float prev_err_x);

     // Defines the control loop function feeds in setpoint variable and process variable
    float controlangular(float SP_Z, float PV_Z, float prev_err_z);

    // Define an error return for the derivative path
    float feedbackerror(float pre_x_er);

};


// The PID function definition
void PID::valueslinear(float gk, float gi, float gd, float delt, float h, float l){

// Delete records of the past and clear old errors

previous_error_X = 0.0;
integral_X = 0.0;

// Set variables equal to variables fed in from the main.
KP_X = gk;
KI_X = gi;
KD_X = gd;
dt_X = delt;
max_X = h;
min_X = l;

}

// The PID function definition
void PID::valuesangular(float gk, float gi, float gd, float delt, float h, float l){

// Delete records of the past and clear old errors

previous_error_Z = 0.0;
integral_Z = 0.0;
```

```
// Set variables equal to to variables fed in from the main.
KP_Z = gk;
KI_Z = gi;
KD_Z = gd;
dt_Z = delt;
max_Z = h;
min_Z = l;

}

float feedbackerror(float pre_x_er){

}

float PID::controllinear(float SP_X, float PV_X, float prev_err_x){

   // Define the loop variables used for processing
   float proportional_output;
   float integral_output;
   float derivative_output;
   //Redefine total output at 0
   float total_output;

   //ROS_INFO("The derivative gain: ([%f])", KD_X);
   // Define the error between the setpoint and the process variable
   error_X = (SP_X - PV_X);

   // Multiply by the proportion amount and define the output
   proportional_output = (KP_X * error_X);

   // Define the integrator summer
   integral_X += (error_X * dt_X);

   // Define the integrator output
   integral_output = (KI_X * integral_X);

   // Define the differentiator
   derivative_X = (error_X - prev_err_x) / dt_X;

   // Define the differential output
   derivative_output = (KD_X * derivative_X);

   // Define the total output
```

```
    total_output = proportional_output + integral_output + derivative_output;

    return total_output;

}

float PID::controlangular(float SP_Z, float PV_Z, float prev_err_z){

    // Define the loop variables used for processing
    float proportional_output = 0.0;
    float integral_output = 0.0;
    float derivative_output = 0.0;
    float integral_Z = 0.0;
    float derivative_Z =0.0;

    // Redefine output as 0
    float total_output = 0.0;

    // Define the error between the setpoint and the process variable
    error_Z = (SP_Z - PV_Z);

    // Multiply by the proportion amount and define the output
    proportional_output = (KP_Z * error_Z);

    // Define the integrator summer
    integral_Z += (error_Z * dt_Z);

    // Define the integrator output
    integral_output = (KI_Z * integral_Z);

    // Define the differentiator
    derivative_Z = (error_Z - prev_err_z) / dt_Z;

    // Define the differential output
    derivative_output = (KD_Z * derivative_Z);

    // Define the total output
    total_output = proportional_output + integral_output + derivative_output;

    //Save error record
    previous_error_Z = error_Z;

    return total_output;
```

```
}
```

The Waypoint Solver Algorithm

```
/*****************************
 *   2020 by Mark Hartzog            *
 *   and Michael Kirke                  *
 *   markthartzog@gmail.com        *
 *   kirkeml1997@gmail.com          *
 *                                              *
 *****************************/

#include "ros/ros.h"
#include "ros/time.h"
#include "std_msgs/String.h"
#include "std_msgs/String.h"
#include "geometry_msgs/Pose.h"
#include "geometry_msgs/Twist.h"
#include <costmap_converter/ObstacleArrayMsg.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <iostream>
#include <array>
#include <cmath>
#include <math.h>

// Define global variables

bool first_bouy_reached = false;
bool second_waypoint_reached = false;
float PI = 3.14159265;
bool detection = false;

class Task{

   public:

   Task get;

   void vectormath(float ly, float ry, float lx, float rx, float scale, float &wpx, float &wpy){

      float u_y = ly - ry;
      float u_x = lx - rx;
      ROS_INFO("Vector component for x: ([%lf])", u_x);
```

```
    ROS_INFO("Vector component for y: ([%lf])", u_y);
    // Call the normalize method
    float normalized_u_x = (u_x / (sqrt((pow(u_x, 2.0)) + (pow(u_y, 2.0)))));
    float normalized_u_y = (u_y / (sqrt((pow(u_x, 2.0)) + (pow(u_y, 2.0)))));
    ROS_INFO("The normalized vector component for x: ([%lf])", normalized_u_x);
    ROS_INFO("The normalized vector component for y: ([%lf])", normalized_u_y);

    float angle_between_buoys = (atan2(u_y, u_x));
    float magnitude_u = sqrt(pow(u_x, 2.0) + pow(u_y, 2.0));

    // Perform the 90 deg rotation
    float sx = 0.0;
    float sy = 0.0;
    sx = normalized_u_x;
    sy = normalized_u_y;
    normalized_u_x = sy;
    normalized_u_y = -1 * sx;

    //Scale up the vector

    wpx = normalized_u_x * scale;
    wpy = normalized_u_y * scale;

    ROS_INFO("The scaled rotated vector component for x: ([%lf])", wpx);
    ROS_INFO("The scaled rotated vector component for y: ([%lf])", wpy);

}

bool navgoal(float x, float y){

    bool flag = false;

    typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

        // Tell the action client that we want to spin a thread by default
        MoveBaseClient ac("move_base", true);

        // Wait for the action server to come up
        while(!ac.waitForServer(ros::Duration(5.0))){
            ROS_INFO("Waiting for the move_base action server to come up");
        }
            move_base_msgs::MoveBaseGoal goal;

            ROS_INFO("Setting x Waypoint to: ([%lf])", x);
            ROS_INFO("Setting y Waypoint to: ([%lf])", y);

            // Send a goal to the robot to move towards the first set of buoys
            goal.target_pose.header.frame_id = "map";
            goal.target_pose.header.stamp = ros::Time::now();

            goal.target_pose.pose.position.x = x;
            goal.target_pose.pose.position.y = y;

            //Need to fix this to be a dynamic quaternion. Not hardcoded to 1.0.
            //geometry_msgs::Pose orient;

            goal.target_pose.pose.orientation.w = 1.0;

            ROS_INFO("Sending goal");
            ac.sendGoal(goal);

            ac.waitForResult();

            if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
              ROS_INFO("The first set of bouys were reached");
              flag = true;
              //ros::shutdown();
            }

            else{
              ROS_INFO("The rover failed to move for some reason");
              ros::shutdown();
            }



  }
}

class Buoy{

  Buoy buoyLeft;
  Buoy buoyRight;

  public:

  float point1_x;
  float point2_x;
```

```
    float point3_x;
    float point1_y;
    float point2_y;
    float point3_y;
    float angle;


    float average_x(){
        float calcX = (point1_x + point2_x + point3_x) / 3;
        //ROS_INFO("The x position: [%lf]", calcX);
        return calcX;
    }

    float average_y(){
        float calcY = (point1_y + point2_y + point3_y) / 3;
        //ROS_INFO("The y position: [%lf]", calcY);
        return calcY;
    }

    float anglefinder(float y, float x){

        float angle = (atan2(y, x));
        return angle;
    }

    float midpoint_locator(float p1, float p2){

        float midpoint = ((p1 + p2) / 2);
        return midpoint;
    }
};
// Defines the position callpack function

void positionCallback(const costmap_converter::ObstacleArrayMsg pos){

    buoyLeft.point1_x = pos.obstacles[0].polygon.points[0].x;
    //ROS_INFO("The x points: [%lf]", buoyLeft.point1_x);
    buoyLeft.point2_x = pos.obstacles[0].polygon.points[1].x;
    buoyLeft.point3_x = pos.obstacles[0].polygon.points[2].x;

    buoyLeft.point1_y = pos.obstacles[0].polygon.points[0].y;
    buoyLeft.point2_y = pos.obstacles[0].polygon.points[1].y;
    buoyLeft.point3_y = pos.obstacles[0].polygon.points[2].y;
    buoyRight.point1_x = pos.obstacles[1].polygon.points[0].x;
    //ROS_INFO("The x points: [%lf]", buoyRight.point1_x);
    buoyRight.point2_x = pos.obstacles[1].polygon.points[1].x;
    buoyRight.point3_x = pos.obstacles[1].polygon.points[2].x;

    buoyRight.point1_y = pos.obstacles[1].polygon.points[0].y;
    //ROS_INFO("The y points: [%lf]", buoyRight.point1_y);
    buoyRight.point2_y = pos.obstacles[1].polygon.points[1].y;
    buoyRight.point3_y = pos.obstacles[1].polygon.points[2].y;

    if ((buoyRight.point1_x != 0) || (buoyRight.point2_x != 0) || (buoyRight.point3_x != 0) ||
(buoyRight.point1_y != 0) || (buoyRight.point2_y != 0) || (buoyRight.point3_y != 0)){
        detection = true;
    }
    if ((buoyLeft.point1_x != 0) || (buoyLeft.point2_x != 0) || (buoyLeft.point3_x != 0) || (buoyLeft.point1_y
!= 0) || (buoyLeft.point2_y != 0) || (buoyLeft.point3_y != 0)){
        detection = true;
    }
}

int main(int argc, char **argv){
    ros::init(argc, argv, "straight_line_task");

    // Declares and defines node object
    ros::NodeHandle nh;

    // Subscribes to the the obstacle detection package to gather position data
    ros::Subscriber sub = nh.subscribe("/costmap_converter/costmap_obstacles", 10000, positionCallback);

    while (ros::ok()) {

        ros::spinOnce();

        // Calculates midpoint between the two.

        if (detection == true){

            if(first_bouy_reached == false){
                float midpoint_x = buoyRight.midpoint_locator(buoyLeft.average_x(), buoyRight.average_x());
                float midpoint_y = buoyRight.midpoint_locator(buoyLeft.average_y(), buoyRight.average_y());
                first_bouy_reached = get.navgoal(midpoint_x, midpoint_y);
            }
        }
        // Define the straight line action
        /*if(first_bouy_reached == true){
           typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;
           //tell the action client that we want to spin a thread by default
           MoveBaseClient ac("move_base", true);
           //wait for the action server to come up
           while(!ac.waitForServer(ros::Duration(5.0))){
              ROS_INFO("Waiting for the move_base action server to come up");
           }
              move_base_msgs::MoveBaseGoal goal;
              float px = key.pointpublisher_x();
              float py = key.pointpublisher_y();
              ROS_INFO("Setting the next x Waypoint to: ([%lf])", key.average_x());
              ROS_INFO("Setting the next y Waypoint to: ([%lf])", py);
              //we'll send a goal to the robot to move towards the first set of buoys
              goal.target_pose.header.frame_id = "map";
              goal.target_pose.header.stamp = ros::Time::now();
              goal.target_pose.pose.position.x = px;
              goal.target_pose.pose.position.y = py;
              //Need to fix this to be a dynamic quaternion. Not hardcoded to 1.0.
              //geometry_msgs::Pose orient;
              goal.target_pose.pose.orientation.w = 1.0;
              ROS_INFO("Sending goal");
              ac.sendGoal(goal);
              ac.waitForResult();

              if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
                 ROS_INFO("The first set of buoys were reached");
                 second_waypoint_reached = true;
                 //ros::shutdown();
              }
              else{
                 ROS_INFO("The rover failed to move for some reason");
                 //ros::shutdown();
              }

        } */

     }
           ROS_INFO("I REACHED THE END OF THE NODE");
  return 0;

  }
```

Arduino Motor Mixing Code and Visual Feedback

```
//**********************************//
// Brandon Bascetta <brandonbascetta@gmail.com>
// Toni Weaver <tfs32413@gmail.com>
//**********************************//

//Include Libraries
#include "ros.h"
#include "std_msgs/Int16.h"
#include "Servo.h"
#include "FastLED.h"

//Function Prototypes

//Autonomous control
void cmd_control(int duty_l, int duty_r);

//Manual RC
void esc_control_manual();

//Read in rc input
void rc_read_in();

//Light Control
void lightboi(int light_mode);

//Define pins and such
#define CH1 3
#define CH2 4
#define CH5 5
#define CH6 6
#define CH8 7
#define ESCL 10
#define ESCR 11
#define LED_PIN 8
#define NUM_LEDS 256

//Led panel control object
CRGB leds[NUM_LEDS];

//Servo objects for esc writing
```

```
Servo escl;
Servo escr;

//ros node handler
ros::NodeHandle nh;

//Some global variables
int left_duty = 0, right_duty = 0;
unsigned long ch1 = 0;
unsigned long ch2 = 0;
unsigned long ch5 = 0;
unsigned long ch6 = 0;
unsigned long ch8 = 0;

//mode for lights
int mode = 1;
//1 = manual
//2 = autonomous
//3 = kill

//Toni's variables
//Variables for the code
long thrusterL = 0;
long thrusterR = 0;

//linear value x
int linx = 0;

//angular value w
int omega = 0;

//Velocity map values
int minV = -10;
int maxV = 10;

//angular
int minA = -10;
int maxA = 10;

//These values represent the output velocities of the thrusters
int escMin = 1100;
int escMed = 1500;
int escMax = 1900;
long rcescL = 0;
long rcescR = 0;

//
bool manual = false;
bool lockEngaged = true;
bool horn = false;


//These values reflect general values of the rc transmitter may not be exact numbers
int rcMed = 1500;
int rcLow = 980;
int rcHigh = 2000;

//Calback Functions
void duty_input_left( const std_msgs::Int16& vall)
{
  left_duty = vall.data;
}

void duty_input_right( const std_msgs::Int16& valr)
{
  right_duty = valr.data;
}
//Setting up ros subscribers
ros::Subscriber<std_msgs::Int16> sub1("drive_cmd_left" , duty_input_left);
ros::Subscriber<std_msgs::Int16> sub2("drive_cmd_right" , duty_input_right);

void setup() {

  //Initialize Pin I/O's
  FastLED.addLeds<WS2812B, LED_PIN, GRB>(leds, NUM_LEDS);
  pinMode(CH1, INPUT);
  pinMode(CH2, INPUT);
  pinMode(CH5, INPUT);
  pinMode(CH6, INPUT);
  pinMode(CH8, INPUT);

  //initialize node and topic subscriptions
  nh.initNode();
  nh.subscribe(sub1);
  nh.subscribe(sub2);

  //default esc signal to 1500 ms
  left_duty = 1500;
  right_duty = 1500;

  //Startup for lights
  for (int i = 0; i < NUM_LEDS; i++) {
    leds[i] = CRGB(0, 10, 10);
    FastLED.show();
  }

  for (int i = 0; i < NUM_LEDS; i++) {
    leds[i] = CRGB(10, 10, 0);
    FastLED.show();
  }

  for (int i = 0; i < NUM_LEDS; i++) {
    leds[i] = CRGB(10, 0, 0);
    FastLED.show();
  }

  //Attach onject to esc pin and set min and max output
  escl.attach(ESCL, 1100, 1900);
  escr.attach(ESCR, 1100, 1900);
}

void loop() {
  //check callbacks
  nh.spinOnce();

  //check rc
  rc_read_in();

  if (lockEngaged == false) {

    //for autonomous control
    if (manual == false) {
      nh.loginfo("Autonomous!");
      cmd_control(left_duty, right_duty);

    }
    //for manual mode
    if (manual == true) {
      nh.loginfo("Manual!");
      esc_control_manual();
    }
  }
  lightboi(mode);

}

void cmd_control(int duty_l, int duty_r) {

  //write esc commands from node
  escl.writeMicroseconds(duty_l);
  escr.writeMicroseconds(duty_r);
}

void rc_read_in() {
  ch1 = pulseIn(CH1, HIGH);
  ch2 = pulseIn(CH2, HIGH);
  ch5 = pulseIn(CH5, HIGH);
  ch6 = pulseIn(CH6, HIGH);
  ch8 = pulseIn(CH8, HIGH);

  if (ch8 < 1500) {
    horn = true;
  }
  else {
    horn = false;
  }

  if (ch5 > 1500 || ch5 < 900)
  {
    lockEngaged = true;
    //nh.loginfo("Lock Engaged!");
    mode = 3;
    escl.writeMicroseconds(1500);
    escr.writeMicroseconds(1500);
    nh.loginfo("Killed!!");

  }

  else
  {
    lockEngaged = false;
    nh.loginfo("Lock Disbaled!");

    //Manual/auto switch
    if (ch6 > 1500)
    {
      manual = true;
      mode = 1;
    }
    else
    {
      manual = false;
      mode = 2;
    }
  }
}
void esc_control_manual()
{

  //input from ch1 for linear velocity and ch2 for angular velocity
  linx = map(ch1, rcLow, rcHigh, minV, maxV);
  omega = map(ch2, rcLow, rcHigh, minA, maxA);


  //convert to driving each motor
  thrusterL = linx - omega;
  thrusterR = (linx + omega) * 0.75;


  //convert to pwm for esc
  rcescL = map(thrusterL, minV, maxV, escMin, escMax);
  rcescR = map(thrusterR, minV, maxV, escMin, escMax);

  //send command to esc
  escl.writeMicroseconds(rcescL);
  escr.writeMicroseconds(rcescR);


}


void lightboi(int light_mode) {

  //Manual
  if (light_mode == 1) {
    for (int i = 0; i < NUM_LEDS; i++) {
      leds[i] = CRGB(10, 10, 0);
    }
  }

  //Autonomous
  if (light_mode == 2) {

    for (int i = 0; i < NUM_LEDS; i++) {
      leds[i] = CRGB(0, 10, 10);
    }

  }

  //Killed
  if (light_mode == 3) {

    for (int i = 0; i < NUM_LEDS; i++) {
      leds[i] = CRGB(10, 0, 0);
    }

  }

  FastLED.show();
}
```

Appendix B

Requirement	Testing Method	What is Success?	Passed (Y/N)
Hull			
Hull Floats	Place completed hull in a swimming pool.	The hull does not sink, it floats.	
Hull Carries 15 lbs	While in the swimming pool, dive weights will be added incrementally until 15 lbs is reached (dive weights are 3 lbs each).	The hull will carry 15 lbs with the pontoons only be submerged less than 4 inches.	
Hull weighs <25 lbs	Place hull on scale and read weight.	Weight is < 25 lbs.	
Hull doesn't leak	Place hull in pool carrying 15 lbs for a minimum of 30 minutes.	Hull has no water in the interior.	
Minimal Deflection	Place 9 lbs on the center section and measure deflection with a ruler.	The measured deflection will be less than ⅛".	
Hardware/Wiring (Components Not Connected)			
Power output for the Ouster OS1-16 LiDAR (not connected)	Using a multimeter, measure the voltage output from the power source to the Ouster OS1-16.	The voltage is within the range of 22-26 V, optimally at 24 V.	
Power output for the two ESCs (not connected)	Using a multimeter, measure the voltage output from the power source to the two ESCs.	The voltage, for each ESC, is within the range of 7-26 V, optimally at 16 V.	
Power output for the kill switch Arduino Mega (not connected)	Using a multimeter, measure the voltage output from the power source to the kill switch Arduino Mega.	The voltage is within the range of 7-12 V, optimally at 9 V.	
Power output for the PID Arduino Mega (not connected)	Using a multimeter, measure the voltage output from the power source to the PID Arduino Mega (from the Simply NUC).	The voltage is 5 V.	
Power output for the USB Hub (not connected)	Using a multimeter, measure the voltage output from the power source to the USB Hub.	The voltage is within the range of 5-12 V.	
Power output for the NETGEAR N900 Wireless Router (not connected)	Using a multimeter, measure the voltage output from the power source to the NETGEAR N900 Wireless Router.	The voltage is within the range of 12-19 V. Should be closer to 19 V due to how the power source was	

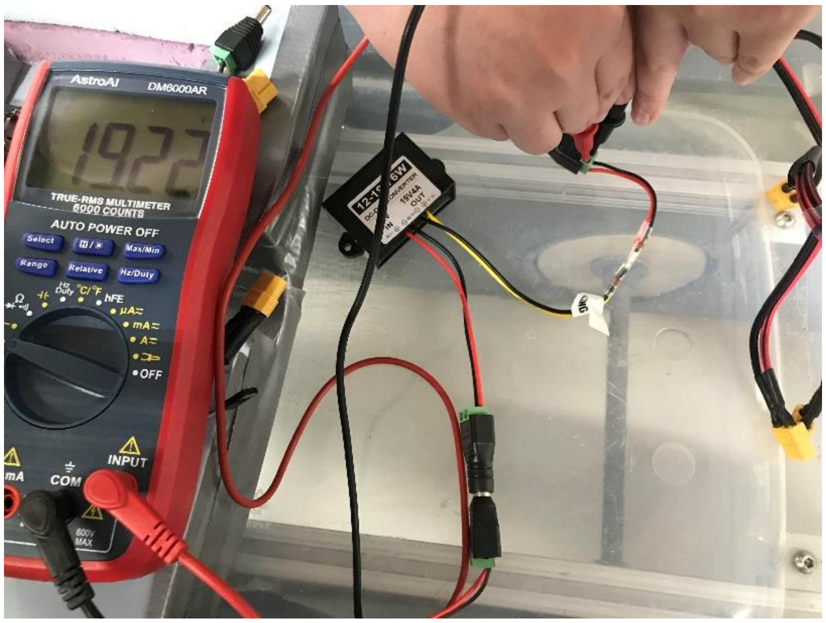
		made.	
Power output for the Jetson Xavier (not connected)	Using a multimeter, measure the voltage output from the power source to the Jetson Xavier.	The voltage is within the range of 9-20 V.	
Power output for the Simply NUC (not connected)	Using a multimeter, measure the voltage output from the power source to the Simply NUC.	The voltage is within the range of 12-19 V.	
Hardware/Wiring (Components Connected and ON)			
Power output connection to the Ouster OS1-16 LiDAR (connected)	Using a multimeter, measure the voltage output and current draw to the Ouster OS1-16. After measuring the voltage, divide the maximum allowed power by this measured voltage to calculate the maximum allowed current.	The voltage is within the range of 22-26 V, optimally at 24 V. The power is within the range of 14-20 W (peak 22 W at startup).	
Power output connection to the two ESCs (connected)	Using a multimeter, measure the voltage output and current draw to the two ESCs.	The voltage, for each ESC, is within the range of 7-26 V, optimally at 16 V. The max current (constant), for each ESC, is 30 A.	
Power output connection to the kill switch Arduino Mega (connected)	Using a multimeter, measure the voltage output and current draw to the kill switch Arduino Mega.	The voltage is within the range of 7-12 V, optimally at 9 V.	
Power output for the PID Arduino Mega (connected)	Using a multimeter, measure the voltage output and current draw to the PID Arduino Mega (from the Simply NUC).	The voltage is 5 V.	
Power output connection to the USB Hub (connected)	Using a multimeter, measure the voltage output and current draw to the USB Hub.	The voltage is within the range of 5-12 V. The current does not exceed 4 A.	
Power output connection to the NETGEAR N900 Wireless Router (connected)	Using a multimeter, measure the voltage output and current draw to the NETGEAR N900 Wireless Router.	The voltage is within the range of 12-19 V, will likely be closer to 19 V. The current does not exceed 2.5 A.	
Power output connection to the Jetson Xavier (connected)	Using a multimeter, measure the voltage output to the Jetson Xavier.	The voltage is within the range of 9-20 V.	
Power output connection to the Simply NUC (connected)	Using a multimeter, measure the voltage output and current draw to the Simply NUC.	The voltage is within the range of 12-19 V. The current must not exceed 3 A	

Power output connection to the Ouster OS1-16 LiDAR (connected)	The LiDAR will be turned on and observed for 3 minutes.	The LiDAR runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	
Power output connection to the two ESCs (connected)	The two ESCs will be turned on and observed for 3 minutes.	The two ESCs run smoothly without any brownouts, shutting off, malfunctioning, or overheating.	
Power output connection to the kill switch Arduino Mega (connected)	The kill switch Arduino Mega will be turned on and observed for 3 minutes.	The kill switch Arduino Mega runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	
Power output connection to the PID Arduino Mega (connected)	The PID Arduino Mega will be turned on and observed for 3 minutes.	The PID Arduino Mega runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	
Power output connection to the NETGEAR N900 Wireless Router (connected)	The NETGEAR N900 will be turned on and observed for 3 minutes.	The NETGEAR N900 runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	
Power output connection to the Jetson Xavier (connected)	The Jetson Xavier will be turned on and observed for 3 minutes.	The Jetson Xavier runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	
Power output connection to the Simply NUC (connected)	The Simply NUC will be turned on and observed for 3 minutes.	The Simply NUC runs smoothly without any brownouts, shutting off, malfunctioning, or overheating.	
ESCs and Thrusters	Run the thrusters, which are connected to the ESCs, to max power. Measure the voltage and the current.	The voltage does not exceed 26 V, and the current does not exceed 30 amps.	
Turnigy High Capacity 10000mAh 4S LiPo Batteries	During testing, check the voltage output from the batteries.	The voltage range is maintained at 14.8-16.3 V.	
Sensor Design			
Sensor mounts articulate	Sensors will be placed on the	Mount can adjust to	

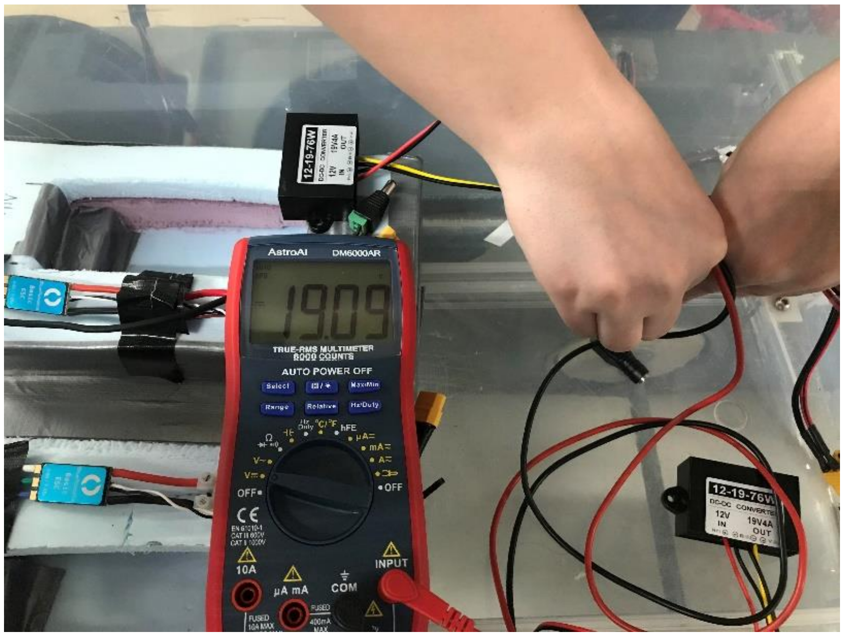
	mount and the angle will be adjusted by raising and lowering the mount.	different angles.	
Sensor mount will be adaptable	Mounts created will be modular to fit onto two 80/20 rails.	Mount will fit on the 80/20 rail showing that the sizing is correct and other mounts can be made using these sizes.	
Mounts are easily replaceable	The mounts will be 3D printed and spares will be made.	Print can be made on most 3D printer beds with common filament (PLA or PETG).	
Software			
Boat detects obstacles	Obstacles will be introduced in a controlled manner and the data will be logged.	Software accurately and repeatedly identifies obstacles.	
PID controller is capable of creating smooth continuous motion.	System will be driven using PID controller.	System moves in a smooth and continuous manner.	
Boat Localized	System will be traveled around a specific path several times and the data logged.	The data points gathered at each point will agree with each other (within a 10% margin of error).	
Basic Waypoint Navigation Completed	System will be tasked with a waypoint within ROS.	System arrives at the waypoint within a reasonable amount of time.	

Appendix C

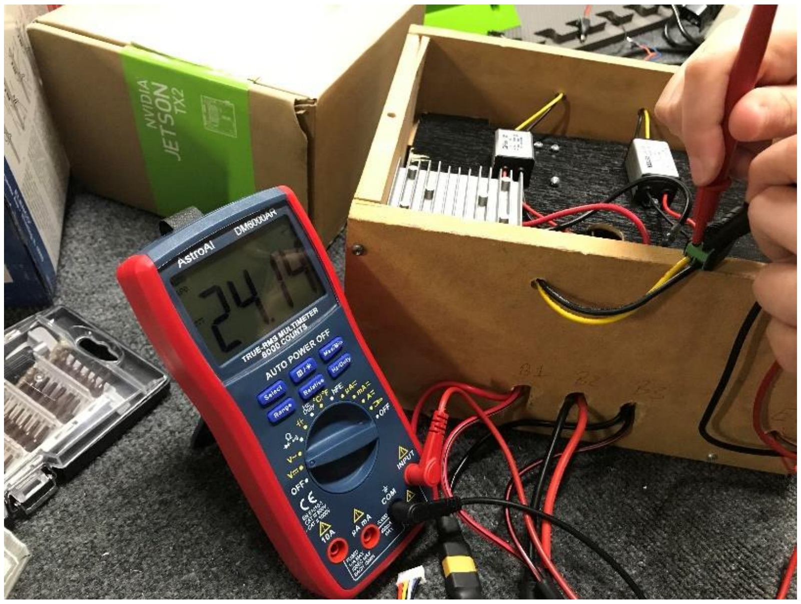
Measured voltage for one of the Simply NUC computers.



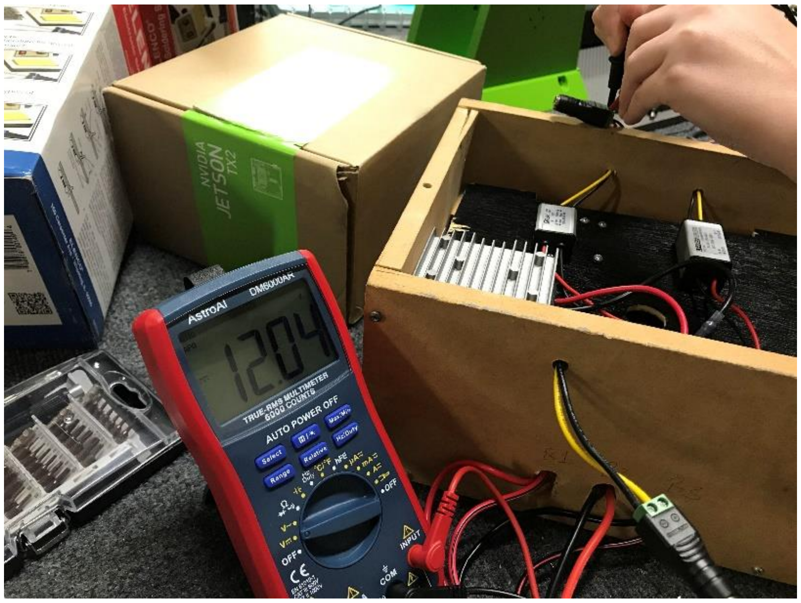
Measured voltage for one of the Simply NUC computers.



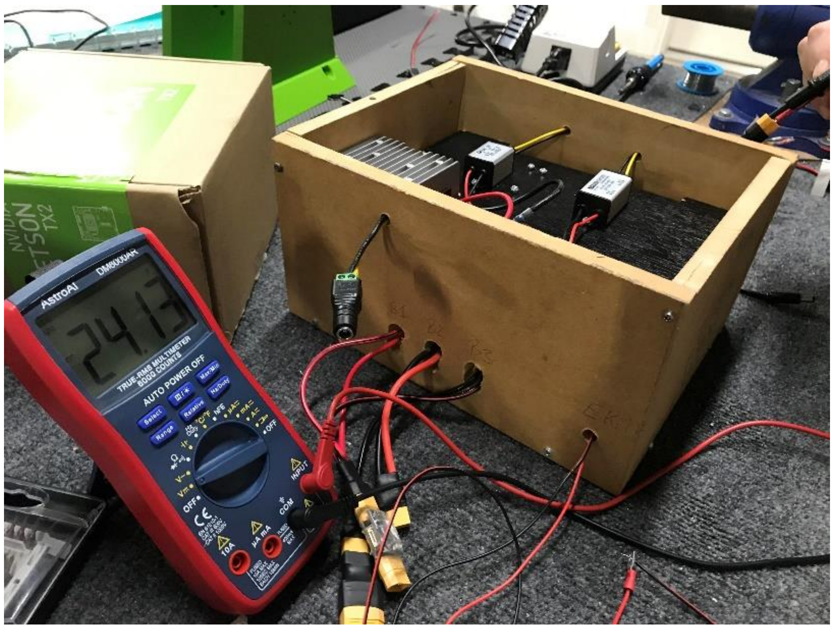
Measured voltage for the LiDAR.



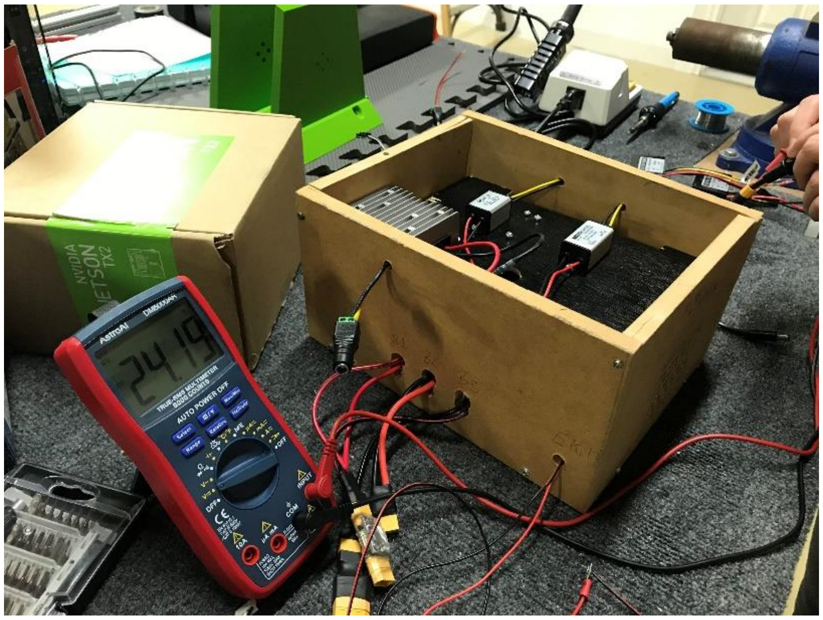
Measured voltage for the router.



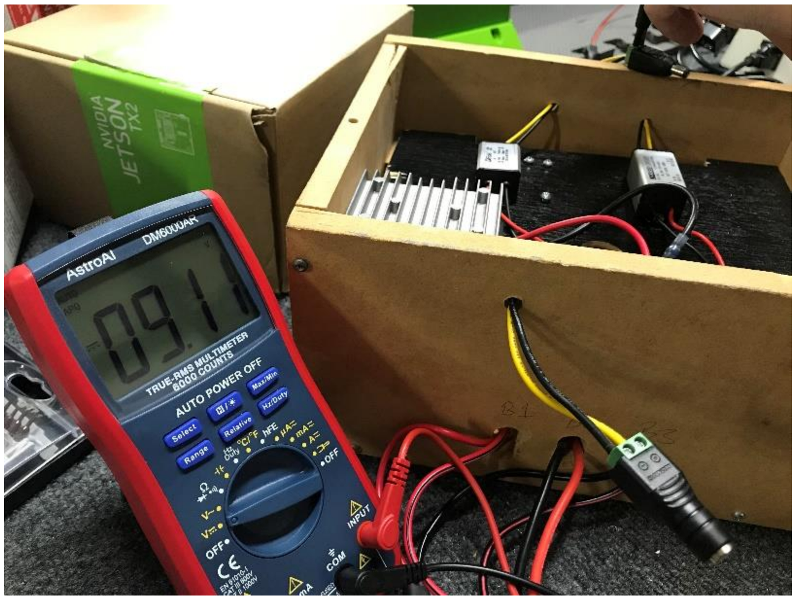
Measured voltage for one of the ESCs.



Measured voltage for one of the ESCs.



Measured voltage for the Arduino Mega.



References

Figures provided by the references below:

Mistry, Siddharth, et al. “Design of HMI Based on PID Control of Temperature.” *Research Gate*, May 2017, www.researchgate.net/publication/316709017_Design_of_HMI_Based_on_PID_Control_of_Temperature.

Pebrianti, Dwi. “Exploration of Unknown Environment with Ackerman Mobile Robot Using Robot Operating System (ROS).” *Research Gate*, Dec. 2015, www.researchgate.net/publication/289882516_Exploration_of_unknown_environment_with_Ackerman_mobile_robot_using_robot_operating_system_ROS/figures?lo=1.