The FAMU-FSU College of Engineering
FSU Panama City, Panama City FL 32405
Electrical and Mechanical Engineering Undergraduate Department

TO: Senior Design Professors for EML 4552/EEL 4915C

FROM: RoboBoat Team

Subject: Critical Design Review (CDR)

Dear Dr. Damion Dunlap and Dr. Geoffery Brooks,

Contained in this document is the current progress of the FSU PC RoboBoat group. Over the last few weeks we have begun to combine the previous work as has been completed in the last two semesters by each team. The physical boat, designed and manufactured by the Mechanical Engineers, will provide the workspace in which the Electrical Engineers will implement the sensors and software. With a functional implementation of the code, we can harness the power of the hardware which will effectively put our boat into motion. With the growing need of a larger and more accurate data set, the larger the need for more concise software and hardware integration.

As you read the document below, know we are extremely appreciative for your time.

Sincerely,

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

FAMU-FSU College of Engineering

EEL 4911C/EML 4552

Critical Design Review

RoboBoat Development Team

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

PROFESSORS: Dr. Damion Dunlap & Dr. Geoffrey Brooks
05 June 2020

TABLE OF CONTENTS

I. Summary of CDR 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 is to develop and manufacture a working boat complete with sensors and basic software that can compete in the RoboBoat competition. This goal will be achieved by completing three different subprojects. These include Software Development, Hardware Development and Boat Design and Manufacturing. Image 1 below, displays the functional decomposition of the project. This project will focus primarily on the left three branches of the image.

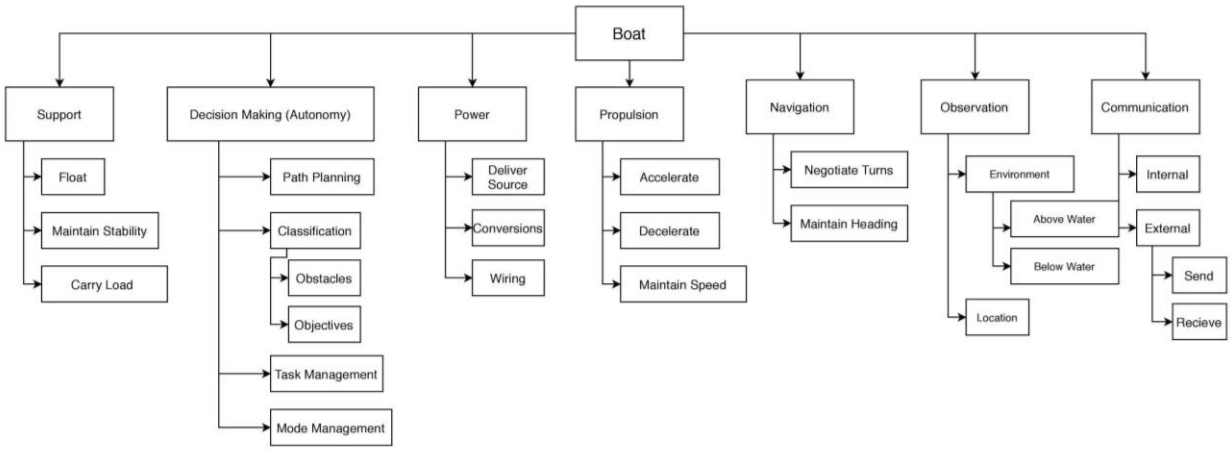

Figure 1. Functional Decomposition of the project.

i. Software Development

The software team is responsible for bringing life to the hardware components in order to make them serve a functional purpose. Using ROS, or the Robot Operating System, as our middleware platform, we can tap into pre-existing algorithms that are often tailor made for our sensors by the sensor's creators themselves. Using these algorithms and the tools in the lab we can protype our own software, written by us, 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.

ii. Hardware Development

The hardware design will essentially take each respective sensor and will wire and place it in the most optimal position of the vehicle. Because of the nature of some of the sensors, it is imperative that they are calibrated and placed in strategic locations in order to be implemented properly so that they may generate helpful data.

iii. Boat Design and Manufacturing

Utilizing the engineering design methods to meet the customer's needs of a larger, more stable boat, a new larger boat was designed for this year's competition. Therefore, a new boat will be constructed. A fiberglass and epoxy resin composite was chosen as the primary material thus, the hand lay-up method of construction will be 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 limited 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.

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 Stages

i. Spring 2020 - Previous Work

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 - Planned Work

a. The power system will be tested to ensure all voltages are outputting correctly.
b. Each necessary sensor will be connected to the power system.

c. Each sensor that is used will generate data.
d. The data collected in the first stage will be imported into ROS executable (nodes).
e. The LiDAR (Light Detection and Ranging sensor) and IMU (Inertial Measurement Unit) sensors will be tested and integrated into the ROS environment.
f. Code will be created to complete the mandatory navigation channel task.
g. Sensor data will be combined with navigation algorithms to allow the boat to perform basic obstacle avoidance.
h. The boat hull will be manufactured using hand laid fiberglass.
i. Sensor mounts will be created using CAD and manufactured using rapid prototyping.
j. The boat software, sensors and hull will be tested in water.

II. Software Information

A. Product Design Software

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

The team has 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 will 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 has 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 will be used to fetch controlled motor commands and will transfer them to the motors via a PWM signal. This development IDE will be implemented using C++ code.

B. Algorithm Design Software

ROS, as mentioned previously, is the middleware platform which connects all of 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.

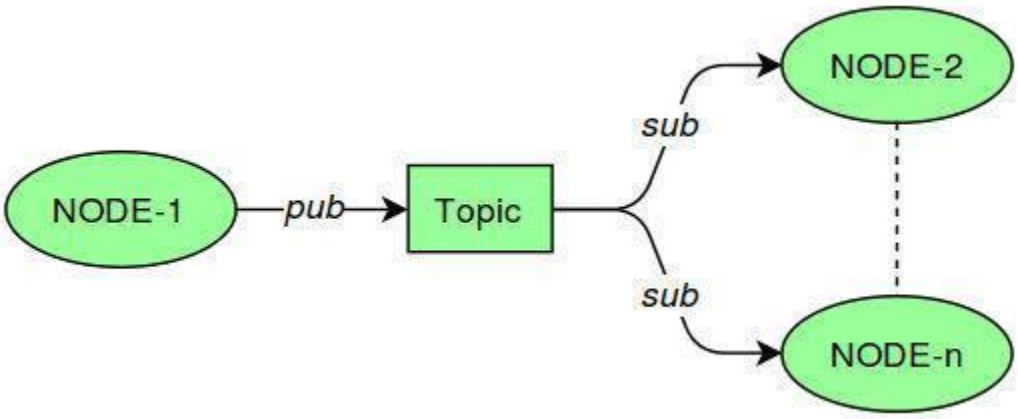

Figure 2. A block diagram of the ROS PUB-SUB system.

Specifically, in the case of the project, custom nodes containing algorithms will be written and implemented in ROS. Data generated by the sensors will be sent to ROS in compact data packets. Then, the data will be processed by algorithms and node packages which work together in conjunction. After, the modified data sets are manipulated into commands for the motors that will be modified by a controller as the last step. After the controller performs its functions, the commands will be sent to the motor driver (Arduino) microcontroller. The controls will be 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.

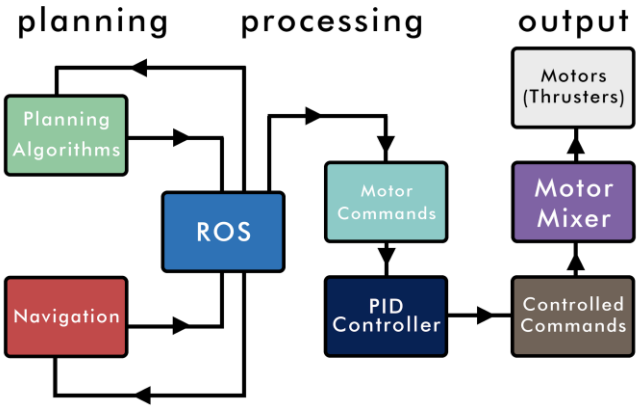

Figure 3. A Diagram depicting the flow of data in ROS.

C. Localization

Before any algorithms can be used and any autonomous locomotion can be achieved, the vehicle must be localized in its own coordinate frame within ROS. This coordinate frame in the case of the boat needs 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 will also need 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 will be achieved by placing the IMU chip (VectorNav) onto the vehicle. After, a reference frame will be defined for the IMU in a similar fashion to the LiDAR. The IMU will have its frame 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 will run in conjunction to several open-source packages. In particular, 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 located 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 will be generating the waypoints for Navigation to receive. The starting waypoint is provided to the team by the competition so the algorithms will be written from that point as the origin. The generated points of each respective obstacle will be 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 will be 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 in order 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 ϕ(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 1 below shows each vector as the arithmetic and linear algebra is applied.

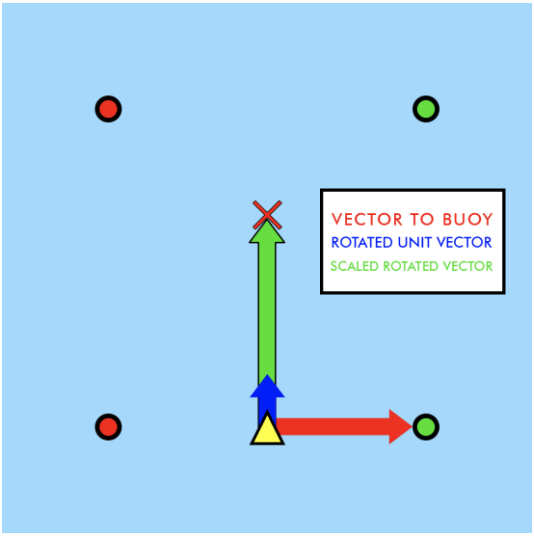

Image 1. A Visual Representation of the Vector Rotation for the Navigation Channel.

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 will use a modified version of Visual Studio Code, it can be configured to understand ROS syntax. Using C++, most of the functions will be 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 can be written using vector algebra to calculate target waypoints for the boat to travel to. These waypoints will 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 in the past few weeks.

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.

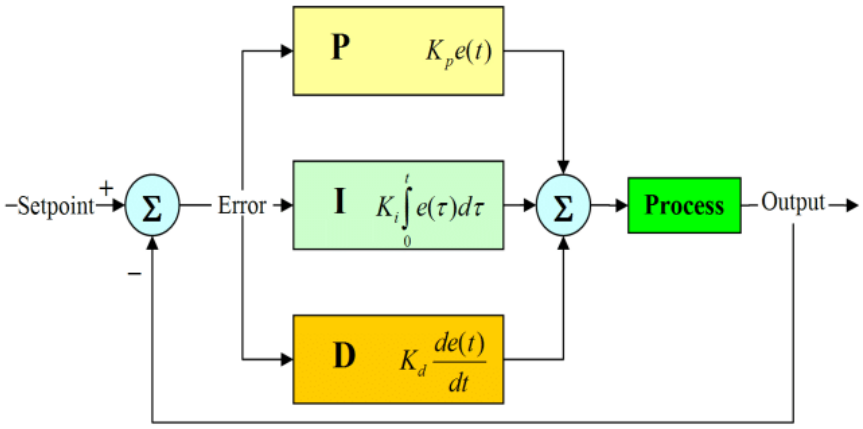

Figure 4. A block diagram of the PID controller.

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 the figure below.

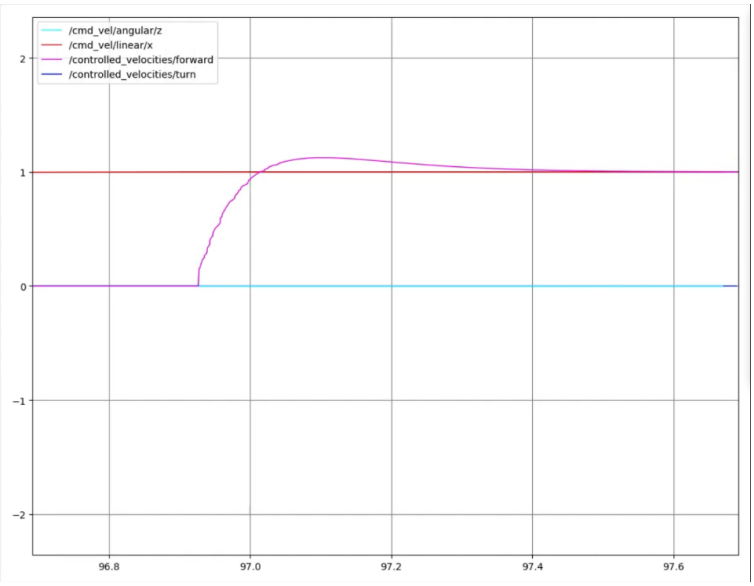

Image 2. An image of the PID in action.

With any controller, the gains of each respective channel must be tuned to achieve the most efficient curve for the tasks. The goal this semester is to tune it so that it handles very well on the hardware as the autonomous algorithms are running.

F. Simulation and Real-Life Results

The boat was simulated in the ROS environment with the controller to view the response of the system. The results turned out to perform as expected. Below is an image of the simulated water vehicle in ROS.

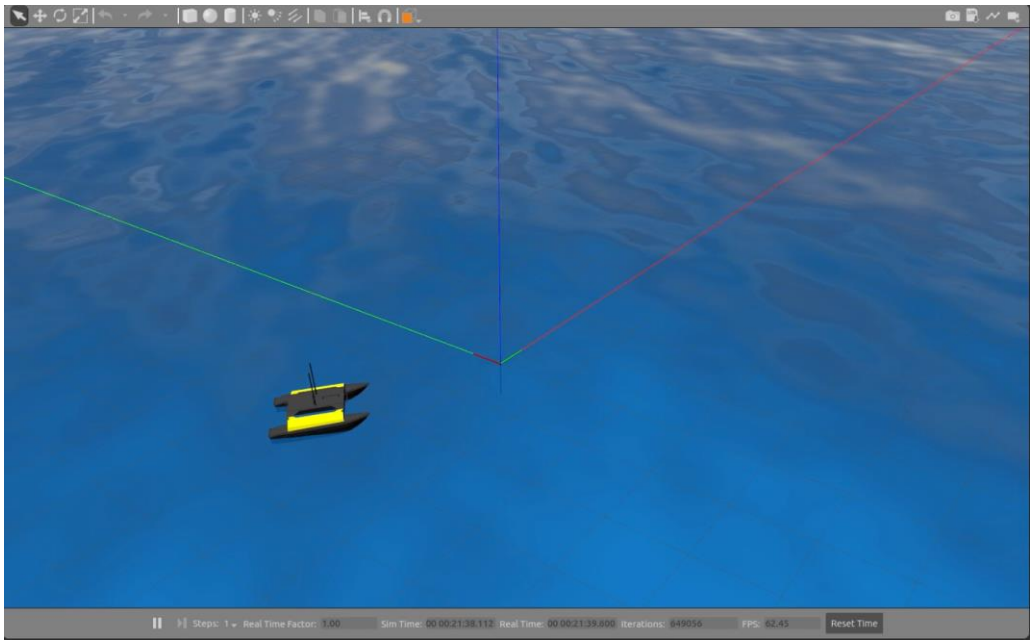

Image 3. An image of the simulation using the PID to drive the vehicle.

Additionally, the LiDAR data being generated in the local coordinate frame was attached to a prototype vehicle and the environment was mapped according to the coordinate frame given. The environment being mapped is the first floor of the FSU PC Holley building in real life. The image 3 below demonstrates the environment mapped and the obstacle detection algorithm, created using a 2d RP LiDAR.

The creation of this costmap using the RP LiDAR is a first step in understanding how LiDARs work. Using the experience gained through creating this map with the RP LiDAR will then be used to implement similar functions with the Ouster LiDAR which creates a 3d point cloud map. This point cloud will be flattened into a 2d plane called laser_scan and fed into ROS Navigation software. This will allow the boat to better understand its environment and better discern between the obstacles and the objectives located within that environment.

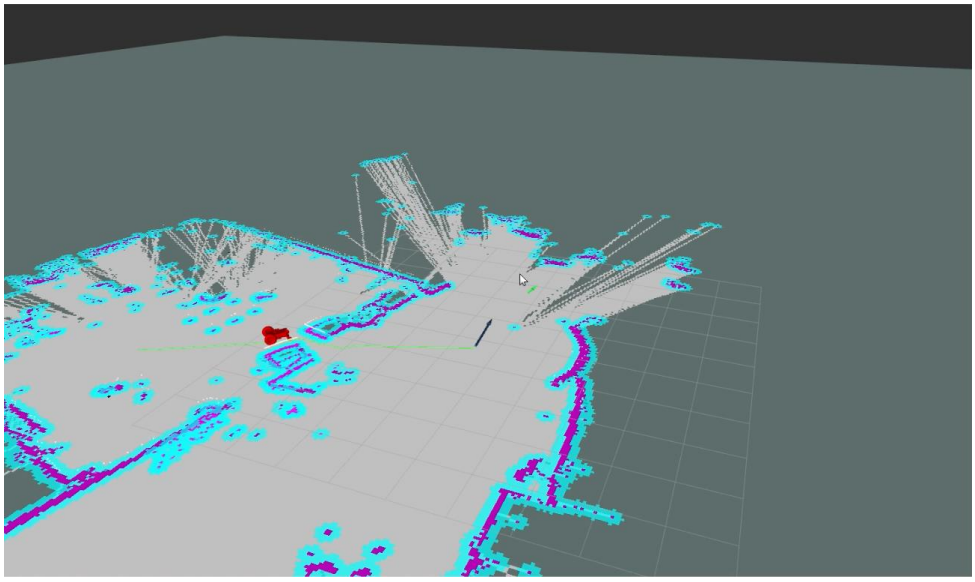

Image 4. A physical map of the Holley building mapped by the LiDAR in real time.

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.

$$v = C_1 * \varphi_1 + C_2 * \varphi_2 \quad (3)$$

$$\omega = C_1 * \varphi_1 - C_2 * \varphi_2 \quad (4)$$

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 $\boldsymbol{\varphi}1$ and $\boldsymbol{\varphi}2$ are each thrusters.

III. 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 in order 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 assembly sketches can be seen in the images below.

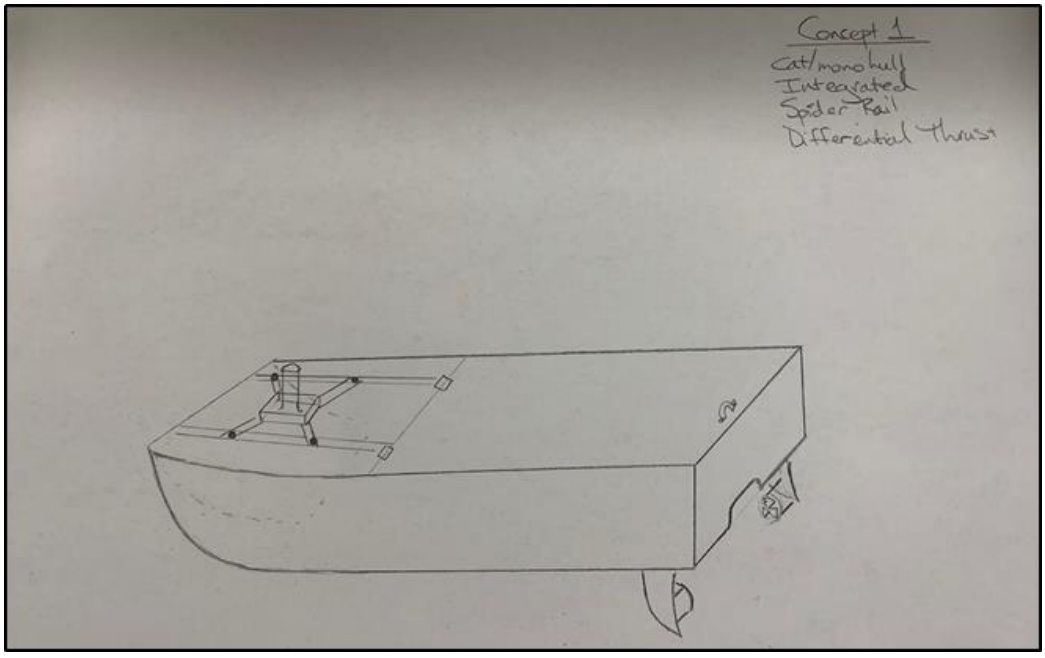

Image 5. Concept 1: A catamaran monohull hybrid with an integrated super structure, spider rail sensor mount, and differential thrust.

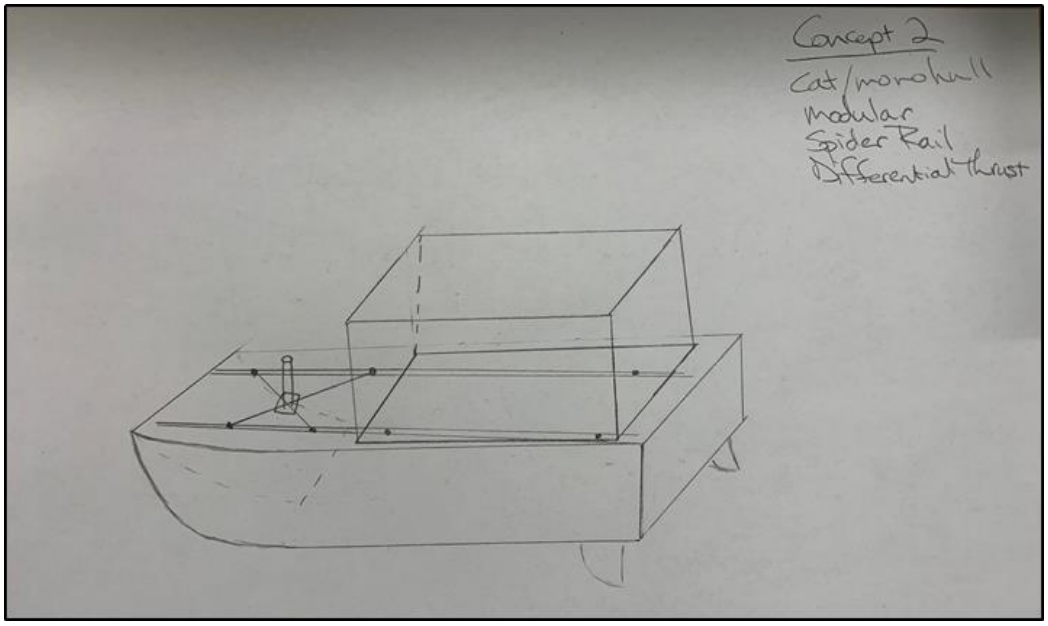

Image 6. Concept 2: A catamaran monohull hybrid with a modular superstructure, spider rail sensor mount, and differential thrust.

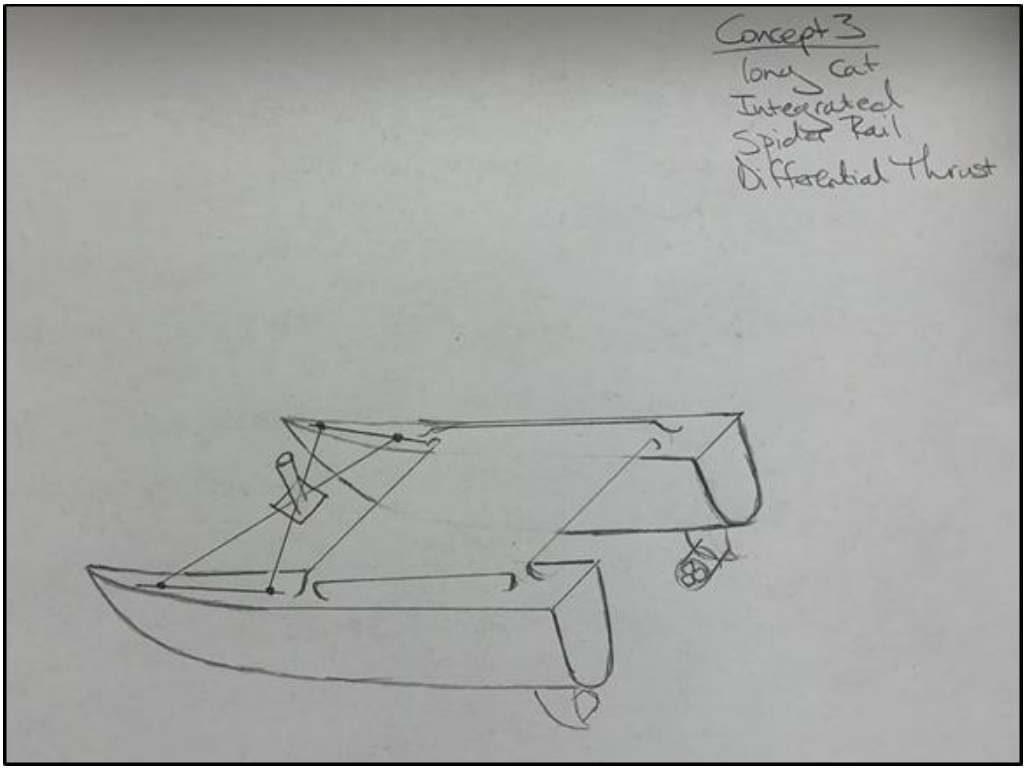

Image 7. Concept 3: A long catamaran hull with an integrated super structure, spider rail sensor mount, and differential thrust.

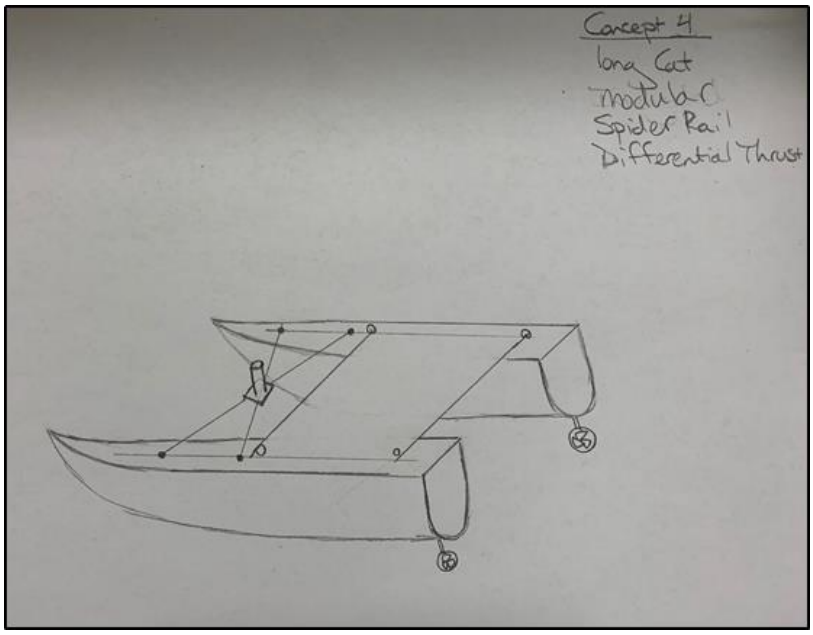

Image 8. Concept 4: A long catamaran hull with a modular super structure, spider rail sensor mount, and differential thrust.

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 9.

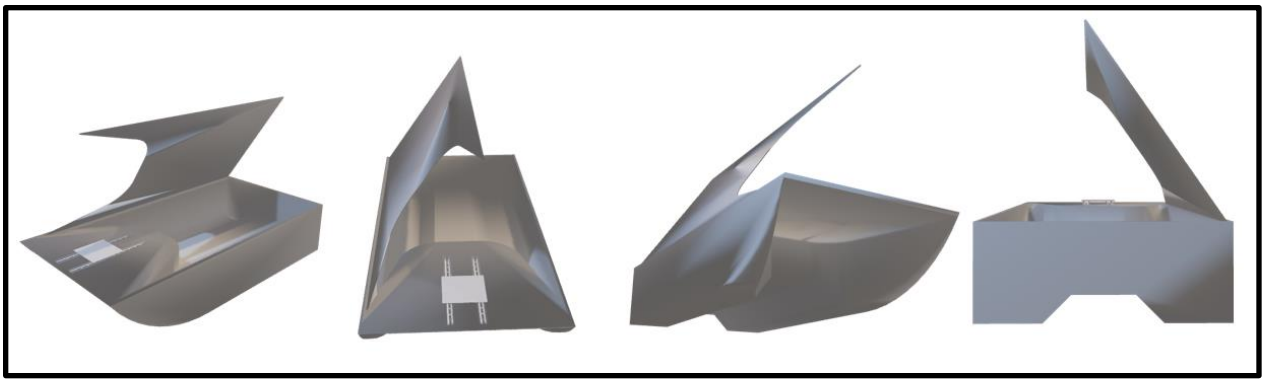

Image 9. High fidelity CAD drawings of the boat that are based upon concept 1's attributes.

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 10.

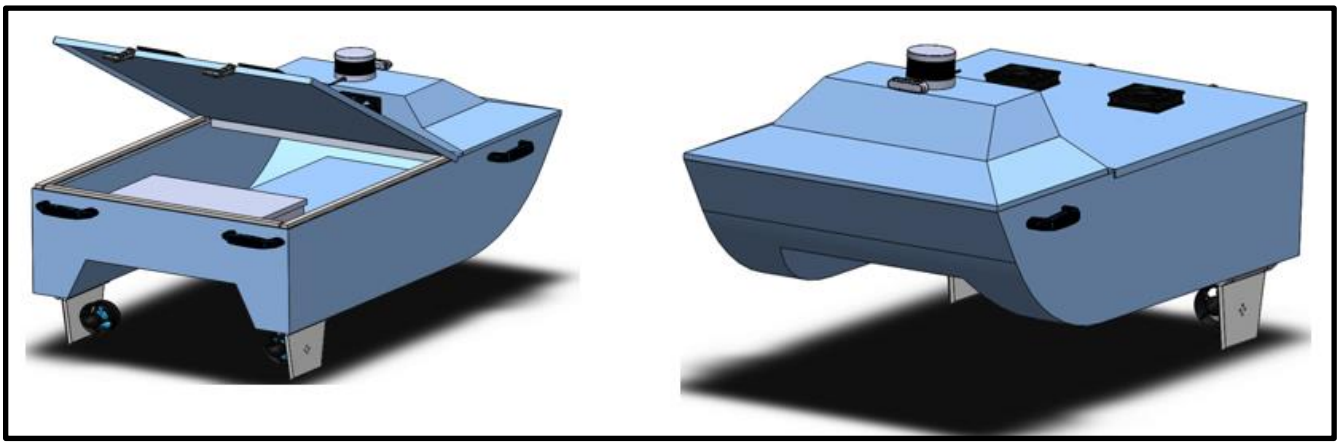

Image 10. Final CAD drawing of the boat hull design.

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. In order 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, 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. Since a fiberglass composite was chosen, the construction method will consist 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 11. The fiberglass layers will cover the mold and the epoxy resin will be applied thus saturating the cloth, and the excess resin will be squeegeed out. Once the resin has cured, the mold will be removed and the fiberglass will be sanded to smooth finish. Then, the fiberglass will be painted with a marine grade paint. Once all three main pieces are complete they will be assembled together. The handles and the customized modular fins will be attached as well.

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 will lead to a superior final boat hull.

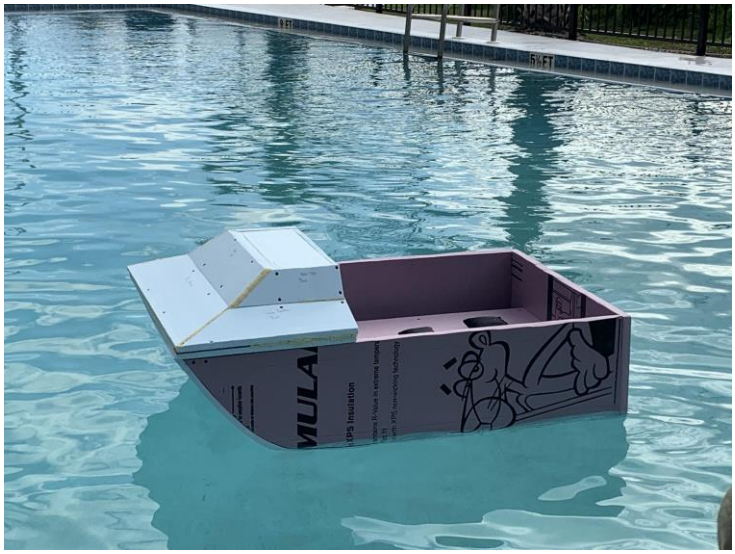

Image 11-The boat mold floating with 12 lbs of weight applied

IV. Hardware

The following section will provide information on the plan for the wiring and integration of the hardware, along with the power requirements for the devices used.

A. Component Setup

Image 12 below shows the wiring diagram of the hardware to be assembled, specifically sensor and networking devices. 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 will be connected to the power box, that was developed by a previous Senior Design Team, with 12 AWG wire and this in turn will supply the appropriate power requirements to each component through 20 AWG wire. These components are 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 will provide 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 will be connected to the power box through 12 AWG wire and will also be connected to the thrusters while receiving pulse width modulation (PWM) signal from the Arduino in order to control the thrusters.

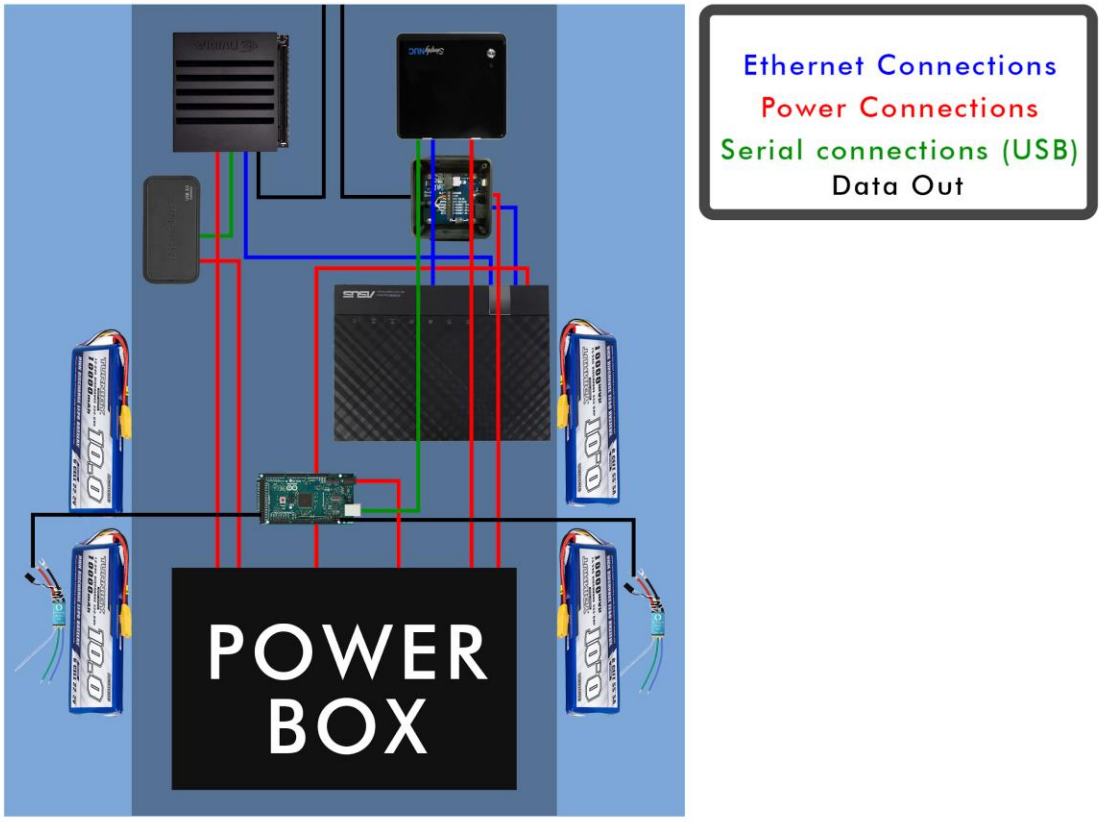

Image 12. A wiring schematic and layout of the components and sensors.

B. Power Requirements

Table 6 shows the power requirements of each device, as listed previously, while also including the DC fan. Each device is listed with their required voltage and current, with the exception of the two computers which don't have a current listed. This is due to the fact that the computers have regulating hardware and will draw what current is needed after being connected to a 19 V source. The amount of current will depend on the devices attached; as more devices are attached, the load and amperage required is increased.

Device	Voltage	Current
Computer #1	19 V	
Computer #2	19 V	

LiDAR	24 V	3 A
Router	19 V	5 A
ESCs (x2)	16 V	24 A
Arduino Mega (x2)	9 V	2 A
DC Fan	12 V	5 A
USB Hub	12 V	5 A

Table 6. Power requirements for components.

V. Conclusion

A comprehensive and well-defined plan is necessary in any major project. This project is a collaborative effort between manufacturing, hardware, and software teams. With the current projected plan, the project goal of creating a working boat capable of competing within the RoboBoat competition should be attainable. This preliminary design review thoroughly examines the focus of the Senior Design team and its contribution to the project as a whole. The team's role is the synthesis of every subject material focused on during the FSU PC electrical and mechanical engineering program. The team is extremely confident in its ability to produce the final product at the end of the Summer 2020 semester.

VI. Appendix

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 bouys
            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 bouys
      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 bouys 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 Prototpyes

//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 onjecy 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();
}
```

VII. 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.