ELEC 490 Final Report
Yachtonomous – An Autonomous Sailboat

Submitted by Group 32

Liam Herbert (20202956)

Alex MacQuarrie (20213915)

Yasin Shahrami (20229169)

Faculty Supervisor

Dr. Sean Whitehall, PhD, EIT

April 3, 2025

Executive Summary

This summarizes the complete design, implementation and testing of the autonomous sailboat.

The high-level design of the boat consists of the three subsystems: physical boat model, hardware and electronics and software. All three subsystems were meticulously designed to function with each other to complete the task of navigating autonomously to the target destination. The physical boat was constructed for stability and space for housing hardware which would then communicate with the software on the microcontroller to complete navigation.

The implementation of the software was similar to the preliminary software simulations performed with the inclusion of several sensors to replace the simulation values and implement wireless communication between the different components. This was done using Bluetooth and Wi-Fi, with the ESP32 development boards communicating RSSI values to the microcontroller and the microcontroller using Wi-Fi to send this data to an external computer for navigation computations. For the algorithms, model predictive control (MPC) and an Extended Kalman Filter (EKF) was used for feedback control and localization, respectively, and a custom navigation algorithm was used.

The hardware and electronics were purchased and soldered onto four PCBs to begin testing of the PCB connections, components and their overall function inside the main chassis. The physical boat went through iterations of 3D printing to determine the best design to fit all the components and remain stable enough to perform all the necessary functions of an actual sailboat. There were several stages to testing, including the testing of the physical boat and its stability and mobility in the water, the functionality of the hardware and electronics systems inside the boat, as well as the integration of the software with the hardware inside the boat. The testing environment used was a small octagonal pool, with an outer radius of 2.79 m.

The physical boat testing began with placing the boat in the filled pool and making sure that it was able to float. Some support was added to reduce the risk of water leakage into the electronics of the boat. The mobility was testing by using the fan to push the boat through the water and making sure that it remained stable and moved through the water.

The electronics testing involved checking for the continuity of signals across test points and making sure that these test points were at their expected voltages and there were no short circuits. The components were tested individually by interfacing with the microcontroller, checking I2C bus status, GPIO response and the testing of accurate wirelessly communicated data being sent back and forth between the components and the test algorithm.

During evaluation, the boat was not successful in sailing under many wind angle conditions. During several conditions it would slip and begin travelling in the direction of the wind as opposed to using the wind to sail towards the destination. This could be largely attributed to the difficulty in creating an aerodynamically sound sailboat as well as accurately modelling the boat's dynamics. However, at a wind angle of 135 degrees relative to the boat (a broad reach), the sailboat managed to navigate relatively well and managed to reach its destination successfully. At a relative wind angle of 90 degrees (a beam reach), the sailboat slipped slightly but still performed well.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and Problem Statement

The modern global economy requires significant volumes of international trade, much of which is provided by large freight ships that travel across oceans[1]. Today, these ships are powered by combustion engines, which produce significant greenhouse gas emissions and are not energy efficient [1]. In contrast, sailboats provide an environmentally friendly and energy efficient alternative for aquatic transportation and shipping [2]. There are several reasons why sailboats are not commonly used for tasks like this. For example, they are often slower and require more skilled and involved operations as navigating and sailing these boats is much more weather dependent.

Automating the process of sailing these boats manually removes a number of these limitations by removing the need for human operation, and reducing the energy required to operate the vessel, which in turn reduces costs. In recent years, automation in ground and air-based transportation industries has grown in popularity, accelerated by advances in robotics, computing, and artificial intelligence [3]. Unfortunately, due to its unique challenges, automation in aquatic shipping and transport has not seen the same growth, despite the many unique opportunities it presents.

This project aims to demonstrate the feasibility of autonomous sailing vehicles on a reduced scope, focusing on the development of methods for navigation, localization, and control of autonomous sailboats, using onboard sensors and actuators, power, and wireless communications.

1.2 Project Overview

Yachtonomous is a small, electric, autonomous sailboat. It requires no human oversight or operation once it is turned on. Due to its small size, which was required to stay within budget, the boat is designed to operate in a small test pool, where it starts from a fixed start point and sails to a fixed destination. On-board sensors are used to gather sufficient data for navigation and state estimation, and actuators are used to provide feedback control for the rudders and sail to sail the boat to the destination. The boat is powered by an on-board lithium-ion battery.

The boat features a custom 3D printed trimaran hull and hand-made nylon mainsail. The power, sensors, actuators, and microcontrollers (MCUs) are housed on or connected to PCBs designed to fit within the main hull. A Raspberry Pi Pico W is used to drive the sensors and actuators, while ESP32 MCUs serve as Bluetooth range sensors, with one acting as a server mounted inside the boat, and others acting as clients mounted on the perimeter of the test pool. The Pico W acts as a Wi-Fi network access point, and wirelessly communicates with the ESP32 server, as well as a separate personal computer (PC), where the computationally intensive navigation, localization, and control algorithms run. Other sensors include two rotation sensors for the sail and wind angles, and an inertial measurement unit (IMU). The rudder and sails are actuated by servo motors.

2 Design

2.1 High-Level Design

The design of the boat in divided into three subsystems, which are the boat, the hardware and electronics, and the software. The boat is responsible for housing the electronics and ensuring sufficient floatation and aerodynamic propulsion using the rudder, centerboard, and sail. The electronics are responsible for power delivery and management, sensors, actuators, and connection and communication with the controllers. The electronics are housed on PCBs inside the boat and connected via wire connectors. When wired communication is impractical, Bluetooth or Wi-Fi is used to communicate between the controllers.

The software drives the electronics and implements the feedback control loop, shown below in Figure 1, along with providing a simulation environment to develop the software independently from the hardware. The navigation algorithm provides the desired trajectory, the control algorithm computes inputs to the servos based on the error between the estimated and desired position, using the mathematical model of the boat's motion. These inputs are provided to the servos, which actuates the rudders and sail. The sensors are then sampled, and the localization algorithm provides an estimated of the boat's state by fusing sensor information and the predicted motion of the boat from the model. The loop is then repeated until the boat reaches its destination.

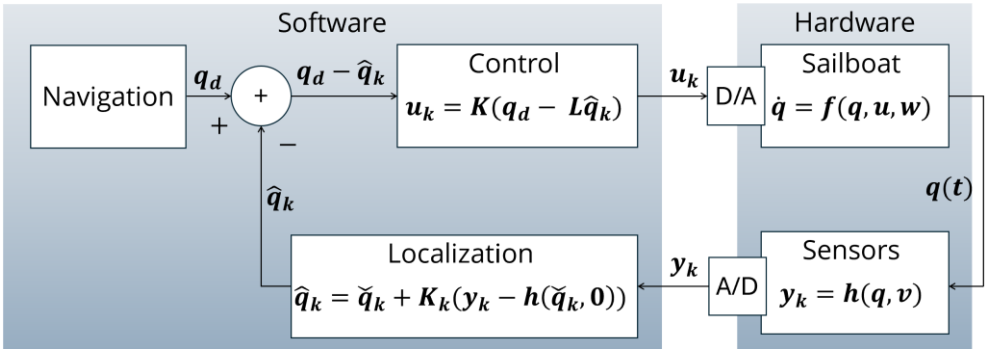

Figure 1: Control block diagram of the boat.

Additionally, a sketch of the boat in the test environment is shown below in Figure 2. Wind is provided to the boat by a fan, and the boat sails from a pre-determined start position to a pre-determined end position, following the course provided by the navigation algorithm. The Bluetooth sensors on the side of the boat are located at known positions and are used to assist in localization. The course to the destination is primarily based on the angle of the wind, which can be coming from any direction.

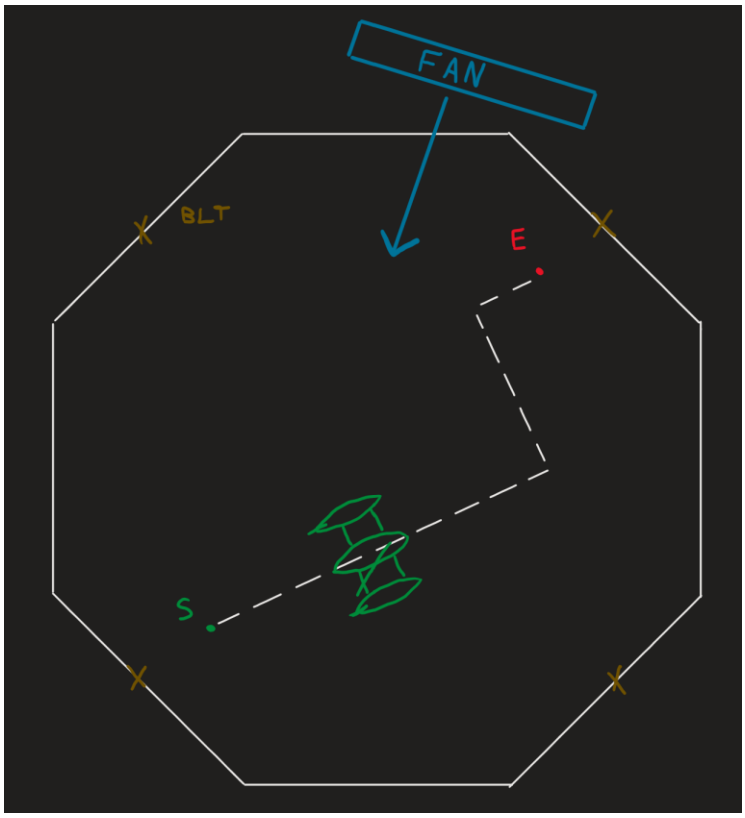

Figure 2: Sketch of the boat in the test environment.

2.2 Kinematic Model

Shown in Equation 1 below is the kinematic state transition model of the boat. This model is used by the software in the localization and control algorithms. It also must be considered by the navigation algorithm, as it generates the desired state of the boat at each point in time.

$$f(q,u) = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\gamma} \\ \dot{\phi} \\ \dot{\eta} \end{bmatrix} = \begin{bmatrix} s\cos(2\eta-\gamma)(a\gamma^4+b\gamma^2+c)\cos(\theta) \\ s\cos(2\eta-\gamma)(a\gamma^4+b\gamma^2+c)\sin(\theta) \\ -\frac{s}{l}\cos(2\eta-\gamma)(a\gamma^4+b\gamma^2+c)\tan(\phi) \\ \frac{s}{l}\cos(2\eta-\gamma)(a\gamma^4+b\gamma^2+c)\tan(\phi) \\ \omega \\ \sigma \end{bmatrix} \quad (1)$$

The states in descending order are the x and y position of the centre of the boat, the orientation of the boat relative to the x-axis, the relative wind direction, the angle of the rudders, relative to the boat's centerline, and the angle of the sail, relative to the centerline. The control inputs to the boat are ω and σ, which are the angular rates of the rudders and sail. Using Equation 2, we can observe the kinematics more clearly, shown in Equation 3.

$$speed = s\cos(2\eta-\gamma)(a\gamma^4+b\gamma^2+c) \quad (2)$$

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\gamma} \\ \dot{\phi} \\ \dot{\eta} \end{bmatrix} = speed \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ -\frac{1}{l}\tan(\phi) \\ \frac{1}{l}\tan(\phi) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \omega \\ \sigma \end{bmatrix} \tag{3}$$

In Equation 2, the maximum boat speed is s, and the factor $\cos(2\eta - \gamma)$ represents that the boat speed is maximized when the sail is trimmed to half of the relative wind angle, and smoothly decreases as it strays from the ideal trim, even allowing the boat to go backwards if trimmed to far in the wrong direction. The factor $(a\gamma^4 + b\gamma^2 + c)$ models the boat's speed as a function of it's angle relative to the wind. Boats typically move the fastest on a beam reach or broad reach and are slower sailing upwind or straight downwind. The coefficients in this factor are chosen based on a regression, which will be shown later. This factor should have a maximum around 1.

Shown in Figure 3 below is a diagram of the boat and its states, used to derive the kinematic model.

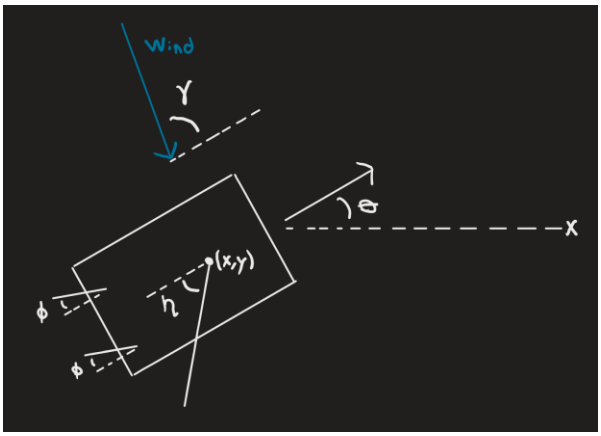

Figure 3: Rough sketch of the boat showing all states.

Equation 3 is derived from the kinematic constraints on the boat's motion. The assumption is made the boat cannot move in the direction normal to the plane of the boards (rudders and centerboard), which creates non-holonomic constraint. This is a commonly used assumption for wheeled mobile robots [4]. The constraint can be stated for the centerboard as Equation 4, shown below, where q is the state-space representation of the boat. Therefore, the boat can only move in the direction orthogonal to this constraint, leading to Equation 5.

$$\omega_b(q) = [-\sin\theta \quad \cos\theta \quad 0 \quad 0 \quad 0 \quad 0] \tag{4}$$

$$\omega_b(q)\dot{q} = 0 \tag{5}$$

Because the rudders share the same angle, we can combine their non-holonomic constraints into a single constraint. The derivation is shown below, based on the geometry of the boat.

$$x_r = x - l\cos\theta\,, \qquad y_r = y - l\sin\theta, \qquad \theta_r = \theta + \cos\phi$$

$$\dot{x_r} = \dot{x} + \dot{\theta} l\sin\theta\,, \qquad \dot{y_r} = \dot{y} - \dot{\theta} l\cos\theta$$

Then substituting into the constraint yields:

$$\begin{bmatrix} -\sin\theta_r & \cos\theta_r & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta}_r \\ 0 \\ 0 \\ 0 \end{bmatrix} = 0$$

Which eventually simplifies to:

$$-\dot{x}\sin(\theta+\phi) + \dot{y}\cos(\theta+\phi) - \dot{\theta} l \cos\phi = 0$$

Which leads to the rudder constraint given by Equation 6.

$$\omega_r(q) = \begin{bmatrix} -\sin(\theta+\phi) & \cos(\theta+\phi) & -l\cos\phi & 0 & 0 & 0 \end{bmatrix} \tag{6}$$

Then, as there are two constraints, two orthogonal vectors which are also orthogonal to both constraints are chosen to represent how the boat can move [4]. Choosing reasonable vectors that align with the control inputs to the boat gives the following equation.

$$g_1(q) = speed \begin{bmatrix} \cos\theta \\ \sin\theta \\ a \\ -a \\ 0 \\ 0 \end{bmatrix}, \qquad g_2(q) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \omega \\ \sigma \end{bmatrix} \tag{7}$$

Solving for a results in the following:

$$\omega_r(q) g_1(q) = 0 = -\cos\theta\sin(\theta+\phi) + \sin\theta\cos(\theta+\phi) - al\cos\phi$$

$$a = -\frac{1}{l}\tan\phi \tag{8}$$

Note that $\dot{\gamma}$ is simply $-\dot{\theta}$, since $\gamma = w - \theta$, where w is the absolute wind angle. This leads back to Equation 3, completing the derivation.

Overall, the kinematic model accounts for most of the desired behaviour of the boat, but there are some limitations. Without considering the dynamics, there is no notion of momentum, and therefore the model does not account for any lateral drift of the boat and makes it difficult to model how the boat passes through the wind without stopping during a tack.

The kinematic model was ultimately chosen over a dynamic model for tow reasons. The first is that the dynamic model is more mathematically complex and requires a deep knowledge of fluid dynamics to create a sufficiently accurate model. The second reason is the computational complexity. The kinematic model has 6 states, and the dynamic model would have at least had 9. This would greatly increase the time to compute matrices in the localization and control algorithms, which creates the risk of the computation being too slow to sample the sensors and provide actuator inputs frequently enough.

2.3 Boat Design

The design of the physical structure needed to be able to satisfy several key criteria relevant to the performance of the boat. Firstly, the boat needs to be able to contain all internal components. As many of these components are electrical and sensitive to water, the structure needs to keep them dry. In addition, it must be possible to actuate the angles of both the mast and rudder of the boat. Lastly, the boat must be able to float, and it must be able to sail.

2.3.1 Hull Design

Four possible options for the physical sailboat were considered. The first three were custom-designed 3D options with varying numbers of hulls: a monohull, a catamaran (2 hulls), or a trimaran (3 hulls). The last option was purchasing and modifying a remote-controlled sailboat to suit our purposes. The simple weighted evaluation matrix below in Table 1 was used to determine between the options.

Table 1: Weighted Evaluation Matrix for determining physical sailboat design.

Criteria	Monohull	Catamaran	Trimaran	RC Sailboat
Space/Protection for Components (x1.5)	4 (6)	3 (4.5)	4 (4.5)	1 (1.5)
Cost (x1)	3 (3)	3 (3)	3 (3)	1 (1)
Ability to Design/ Implement (x1.25)	3 (3.75)	3 (3.75)	3 (3.75)	4 (5)
Stability/Ability to Sail (x1.5)	1 (1.5)	2 (3)	3 (4.5)	5 (7.5)
Total	14.25	14.25	***15.75***	15.5

Ultimately, the three-hulled trimaran option was chosen largely due to its stability and ability to house and protect all essential components. An available online 3D model was chosen to act as a reference for the design.

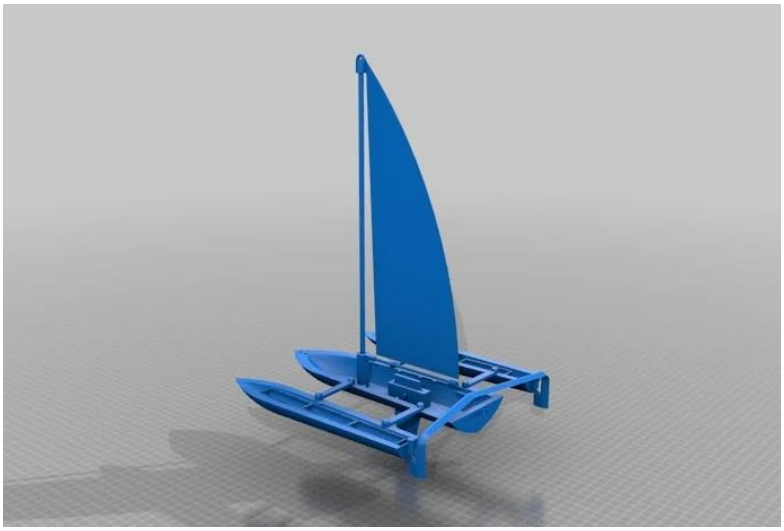

Figure 4: Livo 3 trimaran model used for design inspiration [5].

Figure 4 above shows the model that was chosen. This trimaran was chosen due to its comparable simplicity for replication and large amounts of internal hull space for housing components [5].

As with all custom 3D components for this project, the design of the boat hull was performed in SolidWorks. Initially, both the main and side hulls were reconstructed from the dimensions of the online model, though they were both increased in size to fit required components. Figure 5 and Figure 6 below shows the replicated design of the main and side hulls.

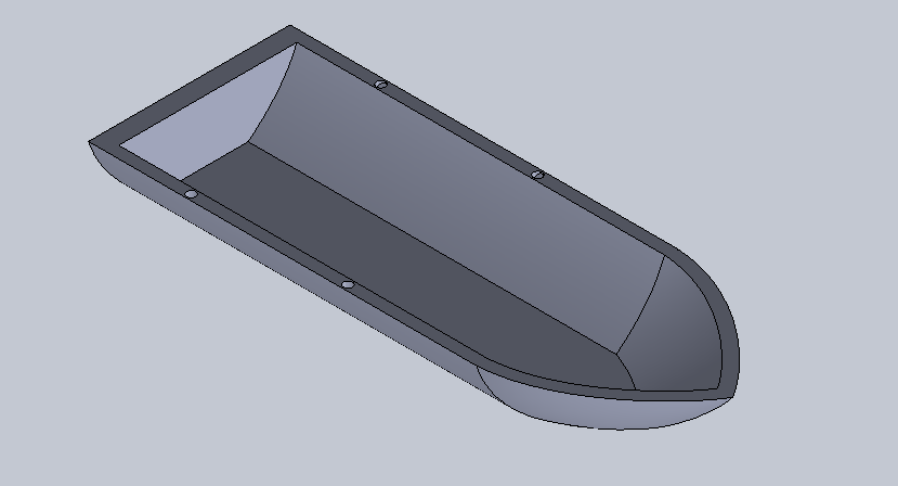

Figure 5: Main hull design replicated from Livo 3 model.

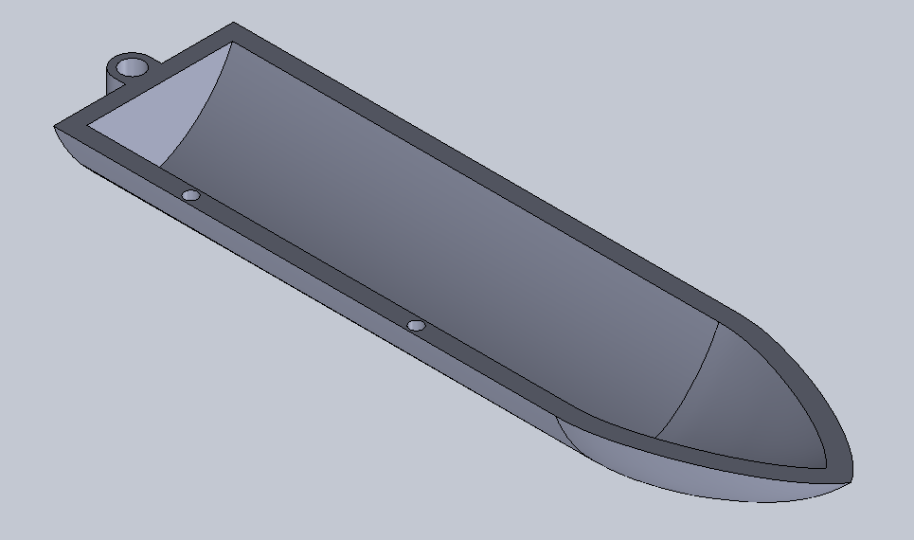

Figure 6: Side hull design replicated from Livo 3 model.

2.3.2 Actuator Design

Operation of the boat required the actuation of each the boat's rudders and sails to a specified angle. SG92R servo motors were selected for their small size, and positional rotational instead of continuous operation [6].

The rudder system involved translating rotational motion on the main hull, into rotational motion on the outer hulls. A linkage rod system was selected to achieve this. The system consists of the servo located at the center rear of the boat, connected through a servo horn to the middle of the control arm. Each end of the control arm is connected to a link rod. The center of the link arm is fixed to the stern of the boat. The opposite end of each link arm drives a rudder arm which in turn rotates the rudder. Each part needs to be precisely dimensioned in relation to each other to allow for expected rotation and avoid interference with each other and the hulls of the boat. Figure 7 below shows the full layout of the system. See Appendix B – Rudder Linkage System Components for the design of each of the individual components. All components can freely rotate about connection points.

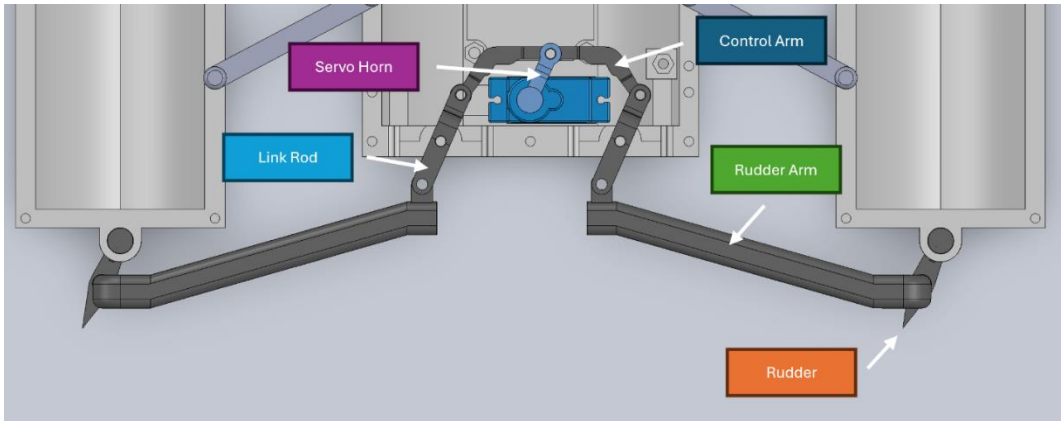

Figure 7: Labeled 3D model of linkage rod system for rudder actuation.

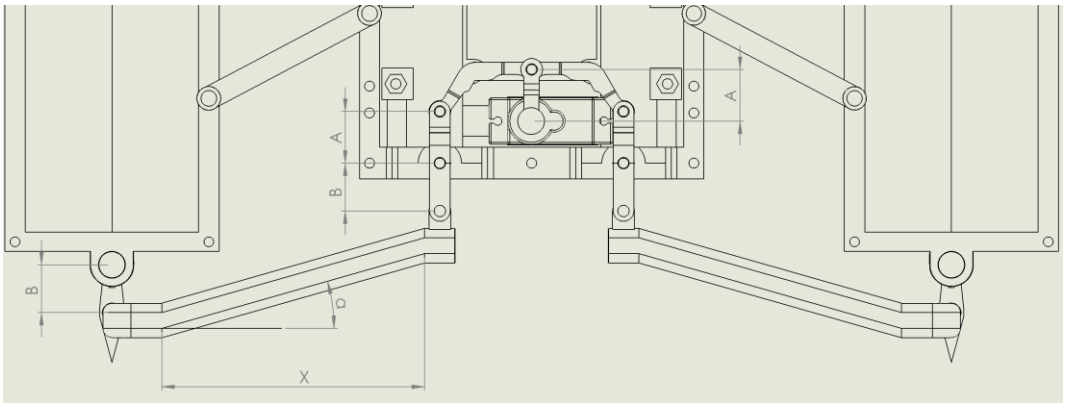

Figure 8: Drawing of rudder linkage system with a simplified demonstration of its dimensioning.

Figure 8 presents a simplified demonstration of how the linkage rod system was dimensioned. The distance between the servo and the control arm must be equal to the length of the boat side of the link rod (Length A). Similarly, the length of the link rod stretching beyond the rear of the boat must be equal to the distance between where the rudder arm connects to the rudder, and where the rudder connects the main hull (Length B). To achieve a desired horizontal distance between the main hull and side hulls, as well as account for the vertical distance between the link rods and the rudder, the dimensions of the rudder arm are adjusted (Length X, Angle a).

The rotation of the mast did not require translating motion from the main to the other hulls. It did, however, require for it to be translated a small distance within the boat since the mast could not

simply be mounted on top of the servo due to height constraints and the intent to place a rotation sensor at the bottoms of the mast.

A timing pulley system was used to accomplish this. The servo was placed at the bow of the boat and was fitted was a 3D printed timing pulley. A timing belt stretched from the servo to the mast which was mounted into a second pulley. The timing pulley mount of the mast fits into the bearing of a rotation sensor (See Rotation Sensor Design). The gear ratio between the two pulleys is equal, and thus rotation of the servo results in an equal amount of rotation for the mast. Assemblies of each the servo and mast as a part of this system can be seen below in Figure 9.

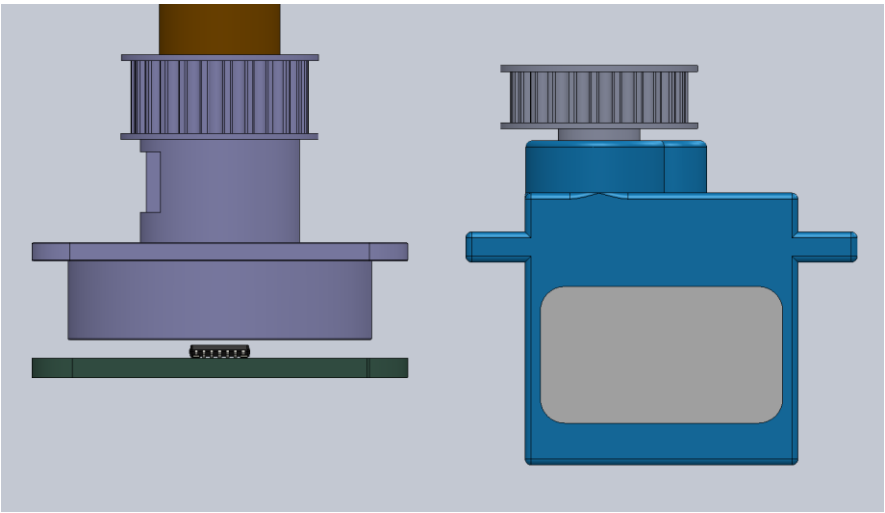

Figure 9: 3D model of timing pulley system for mast actuation.

2.3.3 Wind Vane Design

To provide the boat's control system with the real time angle of environment wind conditions, a wind vane was constructed to be placed on top of the boat's mast. Figure 10 below shows an exploded view of the wind vane design.

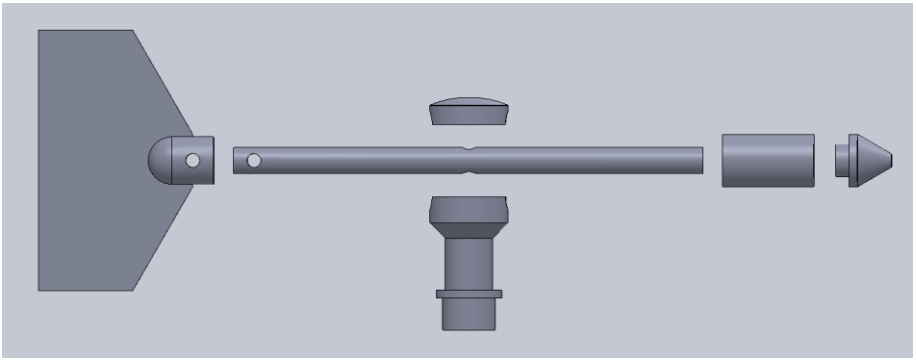

Figure 10: Wind vane exploded view.

The head of the wind vane is hollowed out to allow for additional head to be added to balance out the weight of the tail. This is important since to avoid bias toward any direction, weight should be balanced about the center of rotation. Each individual component can be seen in Appendix C – Wind Vane Components.

Similar to the mast, the base of the wind vane slots into the bearing of a rotation sensor for the wind direction to be read (See Rotation Sensor Design).

2.3.4 Assembly Design

The final design challenge of the boat was ensuring that all individual components fit securely into the boat's main hull.

To accomplish this, a SolidWorks assembly was used containing all custom and purchased parts. Holes to fit nuts, slots for the servo motors, and connection points were added to the main hull (See Figure 11) and custom holders (Appendix D – Additional 3D Components) were designed for any loose parts, such as the boat's battery and the ESP32. The assembly was also used to verify that the systems used to rotation the mast and rudder did not conflict with any other components.

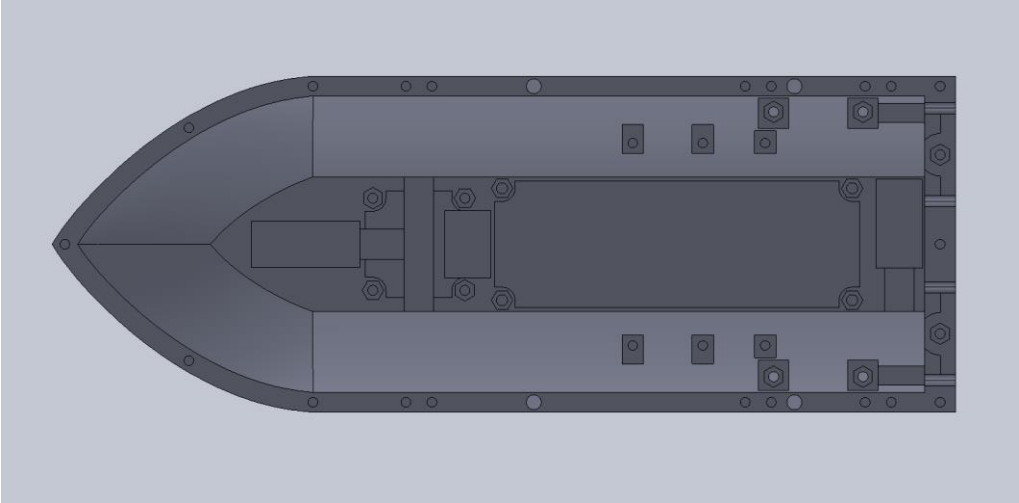

Figure 11: Main hull showing all elements for connecting internal and external components.

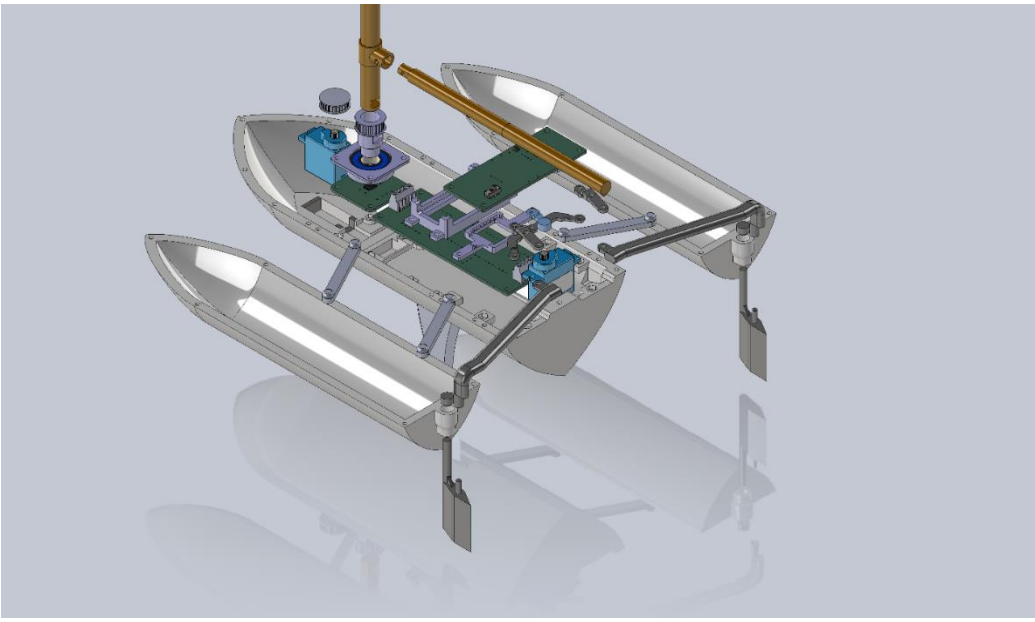

Figure 12: Exploded view of hulls, demonstrating the fit of internal components.

Figure 12 above shows an exploded view of the assembly, demonstrating how all components and systems were fitted into the main hull. For an exploded view of the entire assembly, see Appendix E – Full Assembly Exploded View.

Following the completion of the internal design, covers were designed to not only hide but also waterproof the internal components. An easy to remove cover was added on top of the power management PCB, providing easy access to the boat's switch, indicator LEDs and USB port. A separate panel was added to the center cover allowing for the antenna to be popped out when the cover is removed. The following Figure 13 shows an isometric view of the complete assembly, demonstrating both the covers and the wind vane mounted on top of the mast.

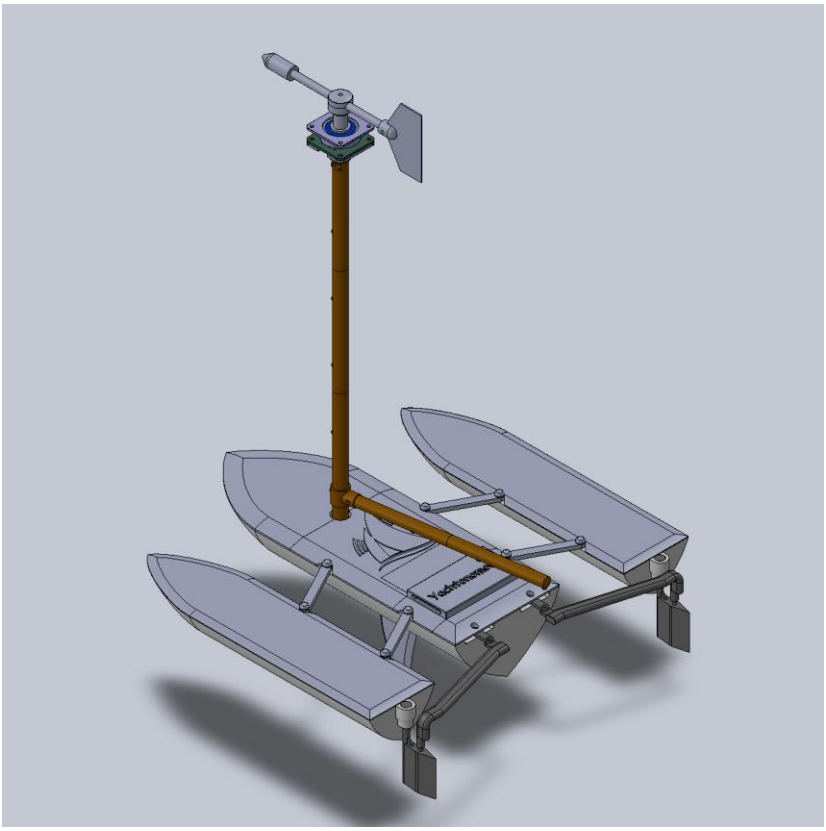

Figure 13: Full model assembly with hull covers shown

2.4 Hardware & Electronics Design

The hardware and electronics design of the boat refers to the implementation and initial design of the power management PCB, main PCB, several sensors and electronic components and how they are specifically integrated together to perform the desired function.

All electronic components have connectivity with the Pico-W microcontroller since the microcontroller needs all inputs to the algorithm to generate the necessary actuation outputs for navigation [7]. There are wireless connections used with the ESP32 development boards and between the Pico-W and an external computer for computation as well as wired connections from the sensors and to the servo motors which are utilized with programmable general-purpose input/output pins on the microcontroller [8].

2.4.1 Main PCB Design

The main PCB is designed for housing the main microcontroller used which is the Raspberry Pi Pico-W, the ICM-20948 inertial measurement unit (IMU) and multiple connectors that will be used to interface with the servo motors and the wind direction sensor at the top of the mast [9].

The Pico-W footprint was custom made based on the datasheet specifications. The dimensions were made to fit the main hull of the trimaran. Likewise, the specifications for the IMU were also created based on the datasheet dimensions. The IMU has two I2C connections to specific I2C SDA and SCL pins on the Pico-W. Figure 14: Schematic of the main PCB showing connections from components to GPIO pins. and Figure 15: Image of main PCB layout, showing locations of headers, Pico-W and IMU. show the schematic and the PCB layout of the wind vane PCB.

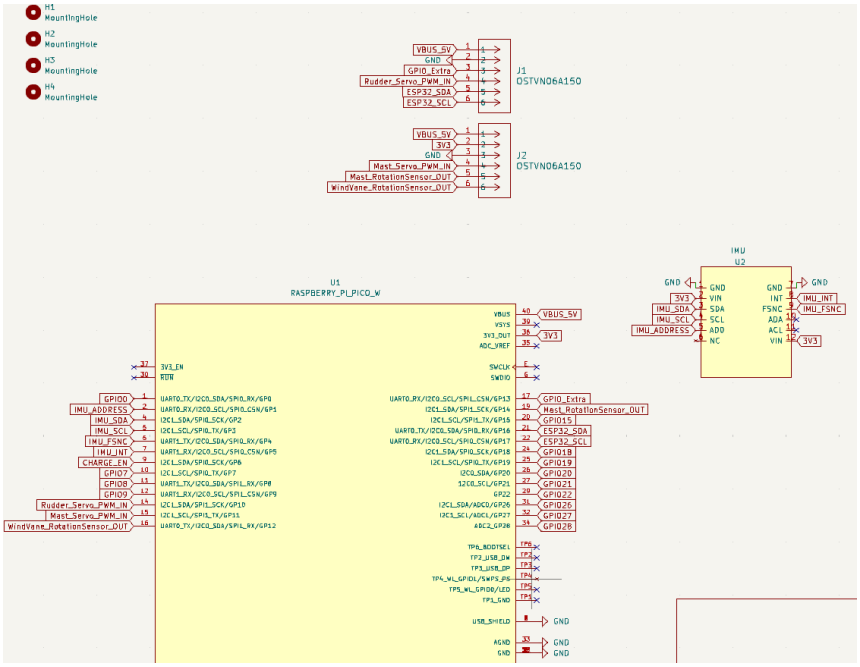

Figure 14: Schematic of the main PCB showing connections from components to GPIO pins.

The below figures show the initial design of the PCB as well as a 3D model of the design.

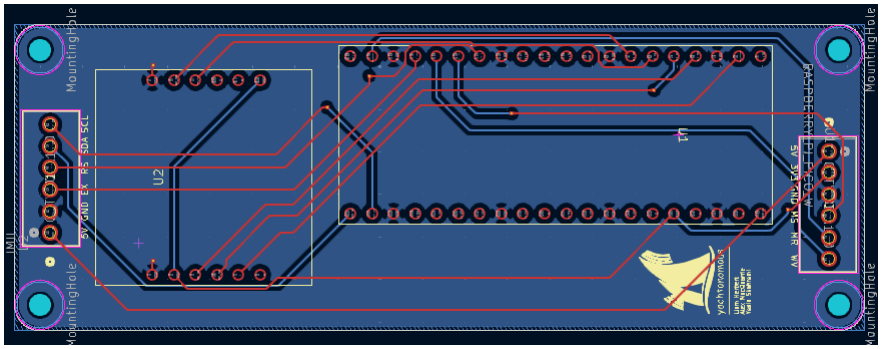

Figure 15: Image of main PCB layout, showing locations of headers, Pico-W and IMU.

2.4.2 Power Management & Control PCB Design

The power management PCB serves to power the main microcontroller PCB, a method of uploading code as well as a charger for the on-board lipo battery. The board consists of a power management chip (MCP73831), a battery, a 3.7V to 5V boost converter (TPS61023), and two micro-USB ports [10] [11] [12].

The power management chip is used to switch between charging the battery and powering the board depending on whether there is an external power source connected. If connected to the micro-USB port on the back of the board, the power management chip will direct power to the second micro-USB port to power the board and direct power towards charging the battery. When there is no external power connected to the board, the battery will be used to power the board through the second port. The circuit switches between these two as shown below in Figure 16.

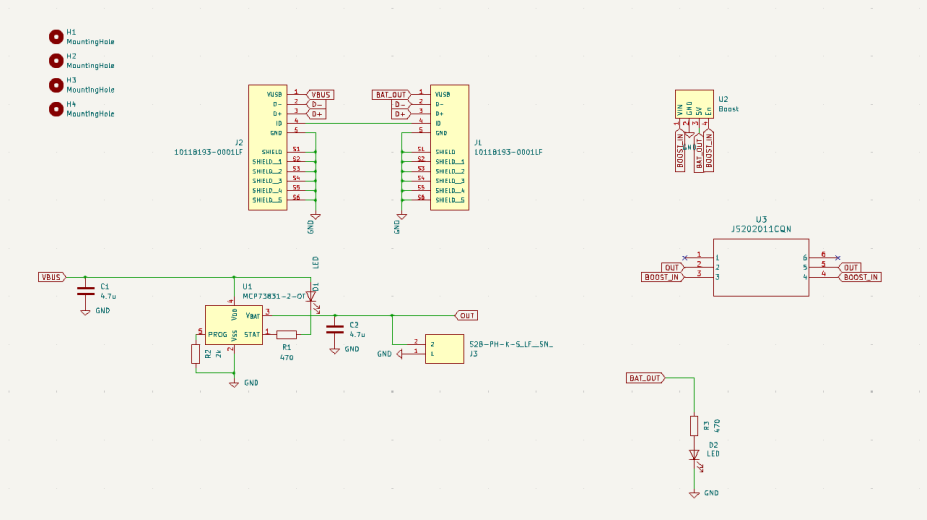

Figure 16: Schematic of the power management PCB.

The boost converter is used to convert the 3.7V power from the battery to 5V for the main PCB. Figure 17**Error! Reference source not found.** shows the layout of the PCB.

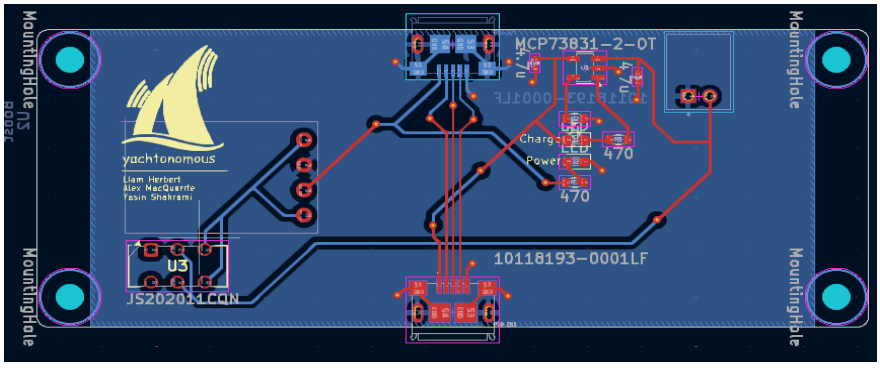

Figure 17: Image showing the power management PCB layout.

2.4.3 Rotation Sensor Design

The rotation sensor PCB was to be implemented in two separate locations on the boat, one at the top of the mast to measure the angle of a wind vane and one at the bottom to measure the angle of the mast. The resulting solution would have to work in both locations.

Since the intent was to measure the absolute angle of revolution relative to the board, commercially available relatively inexpensive incremental encoders would not be a solution. As available commercial absolute encoders were very expensive, a custom solution was implemented.

The chip selected for this purpose was the AS5048A magnetic rotary encoder [13]. Rotation is read by measuring the relative angle of a magnet suspended 0.5-2.5mm above the surface of the chip. The polarity of the magnet should be split along a plane perpendicular to the chip's surface.

This was implemented by designing a custom 3D printed flange bearing mount that was suspended over the surface of the chip. An additional custom mount to suit individual purposes, is fitted into the inner diameter of the bearing, which holds the magnet flush with the bottom of the bearing. The top of this mount connects to the apparatus of which the rotation is being measured.

While the AS5048A can communicate relative angles through both I2C and PWM, PWM was selected due to its ease of integration, and simpler code and connection. Figure 18 and Figure 19 show the schematic and the PCB layout of the wind vane PCB.

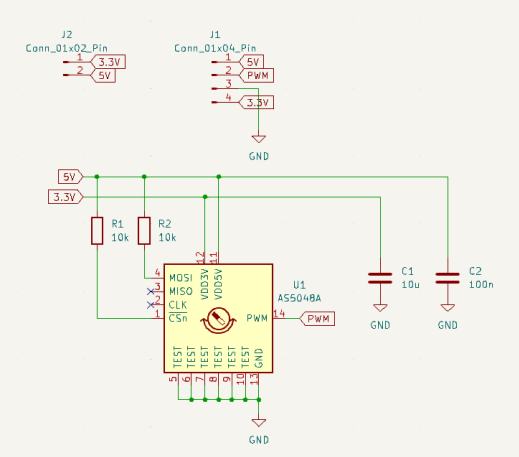

Figure 18: Circuit schematic of the wind vane PCB.

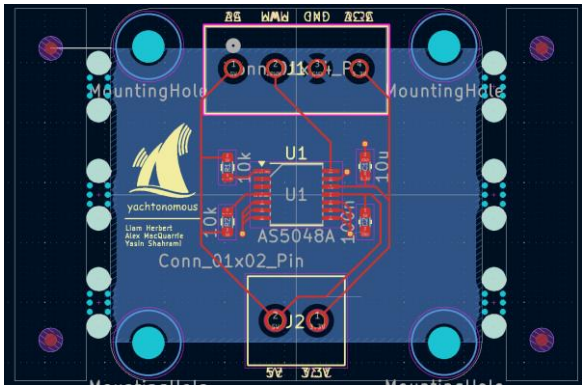

Figure 19: PCB layout of the wind vane PCB.

2.4.4 Bluetooth Localization Network

The Bluetooth localization network consists of a total of 5 XIAO-ESP32 development boards that are programmed to communicate with each other using Bluetooth and communicate to the onboard micro controller using a Wi-Fi network.

One ESP32 development board is configured as a server, this board will act as the main reference point for the boat and will receive RSSI measurements that are being sent from the 4 outer Bluetooth devices that are configured as clients. The server first advertises its MAC address for the clients to pick up and when the clients detect the specific address will attempt to connect to the server. Once a connection has been established, each client will continuously send RSSI measurements at a set time interval to be used for computing the location of the boat.

When the server receives all of the data, it will then convert the data into a JSON and send this over a Wi-Fi network created by the Raspberry Pi Pico-W. The microcontroller will then decode this data and use it for the calculations.

2.5 Software Design

To design and write the software independently from the hardware, a simulation environment was created in Python. The simulation simulates all critical components of the boat, including noise from the sensors, actuators, and model, to ensure it is as representative of the actual physical conditions as possible. The simulation generates several output plots that are used for debugging and tuning parameters.

Shown in Appendix F – Simulation Sequence Diagram is a sequence diagram showing what happens when the simulation is run. To start, the simulation parses a JSON config which contains many numerical parameters to configure the simulation. An example of a section of the simulation config is shown below in Figure 20.

```
{
    "control": {
        "max_pred_horz"    : 25,
        "state_weights"    : [10.0, 10.0, 15.0, 0.0, 2.0, 1.0],
        "input_weights"    : [1.0, 5.0],
        "input_saturation" : [6.28319, 6.28319],
        "init_inputs_deg"  : [0, 0]
    },
```

Figure 20: Control config from the simulation config JSON file.

The parser dynamically instantiates python objects for each section, whose attributes are the values of each section.

The simulation then uses an estimate of the initial state of the boat to perform navigation. If the boat believes that the destination is not upwind of it, it will sail straight to the destination, generating a series of points, each with six components for the six states of the boat, providing the desired value of each state at each point in time, which is later used by the control algorithm. Shown below in Figure 21 is how the direct trajectory is computed. Note that the course stops once it is sufficiently close to the destination, which is given by a distance in number of boat lengths in the config file.

```
### Direct trajectory
num_points = int(np.ceil(distance/point_dist))
for i in range(num_points):
    x_y = boat_pos + (i+1)*path_vector
    theta = Angle.exp(np.arctan2(path_vector[1], path_vector[0]))
    gamma = abs_wind_angle - theta

    state.append((x_y[0], x_y[1], theta.log, gamma.log, 0.0, gamma.log/2.0))
    inputs.append((0.0, 0.0))

    if np.linalg.norm(dest_pos-state[-1][:2]) < dest_thresh:
        break
```

Figure 21: Direct trajectory computation from the navigation algorithm.

It is much more complex if the boat must sail upwind to the destination, as this requires sailing as close to the wind as possible, then tacking (turning the boat through the wind) upwind to reach the destination. There are infinitely many viable paths, but the algorithmn provides the path that minimizes the number of tacks and avoid hitting the sides of the test pool, based on it's knowledge of where the Bluetooth beacons are. Shown in Figure 22 in a potion of the code that decides when to tack. Tacking is implemented in the software by a rotation matrix which can be transposed to compute it's inversed to tack back as needed.

```
if not past_layline:
    # Going past layline, tack and sail to destination
    if not _is_upwind(angle_to_dest, abs_wind_angle, crit_angle_wind):
        past_layline = True
        boat_vec     = boat_vec @ tack_matrix
        tack_matrix  = tack_matrix.transpose()
    # Going out of set bounds, tack
    elif _is_out_of_bounds(current_pos, boat_vec, x_lim_l, x_lim_h, y_lim_l, y_lim_h):
        boat_vec    = boat_vec @ tack_matrix
        tack_matrix = tack_matrix.transpose()
```

Figure 22: Tack conditions for upwind section of navigation algorithm.

Shown below in Figure 23 are outputs from the navigation algorithm for direct and upwind cases.

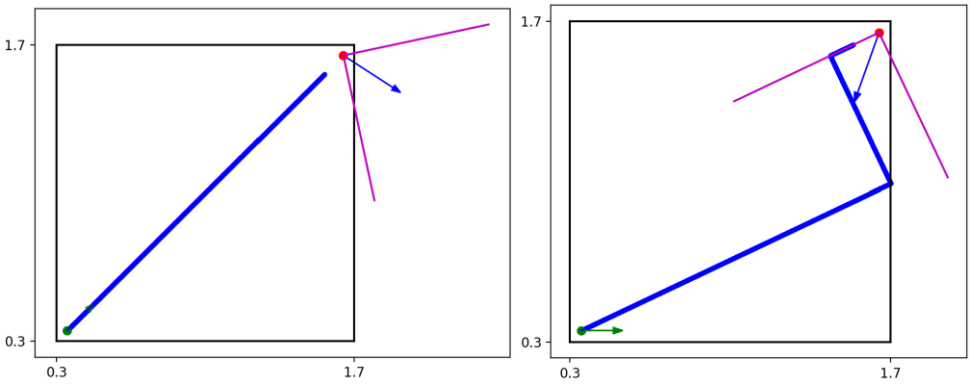

Figure 23: Sample direct and upwind trajectory outputs from the navigation algorithm.

After the navigation algorithm returns the desired state, the simulation enters the main loop. First, the measurements from the sensors are read into a vector. In the simulation, we use the boat's exact state then add random noise to the measurements. The measurements are then passed to the localization algorithm, implemented using an Extended Kalman Filter (EKF). The EKF uses linearization's of the measurement and boat model to use techniques from the Kalman filter, which estimates the boat's state while reducing the uncertainty in the state over time. The EKF follows these steps [4]:

1. Linearize the boat model using the Jacobian matrices
2. Compute the a priori estimation (state and covariance)
3. Linearize the measurement model using the Jacobian matrix
4. Compute the a posteriori estimation (state and covariance)

To linearize the boat model, we compute the Jacobian of the matrix in Equation 1 with respect to the states and inputs and evaluate at the previous state estimate. We must also convert the model to discrete time, which is accomplished via Euler integration. This is shown in the following equations.

$$A = \frac{df}{dq} = \begin{bmatrix} 0 & 0 & \frac{d\dot{x}}{d\theta} & \frac{d\dot{x}}{d\gamma} & 0 & \frac{d\dot{x}}{d\eta} \\ 0 & 0 & \frac{d\dot{y}}{d\theta} & \frac{d\dot{y}}{d\gamma} & 0 & \frac{d\dot{y}}{d\eta} \\ 0 & 0 & 0 & \frac{d\dot{\theta}}{d\gamma} & \frac{d\dot{\theta}}{d\phi} & \frac{d\dot{\theta}}{d\eta} \\ 0 & 0 & 0 & \frac{d\dot{\gamma}}{d\gamma} & \frac{d\dot{\gamma}}{d\phi} & \frac{d\dot{\gamma}}{d\eta} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (9)$$

$$B = \frac{df}{du} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \frac{d\dot{\phi}}{d\omega} \\ \frac{d\dot{\eta}}{d\sigma} & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{10}$$

Note that only the non-zero derivatives are shown, and the derivatives are not expanded and evaluated as they are very long. Euler integration of A and B leads to the following F ang G matrices:

$$F = I + TA \tag{11}$$

$$G = TA \tag{12}$$

Where I is the identity matrix, and T is the time over which the integration is performed. We can then compute the a priori estimate, shown in Figure 24 below.

```
# Compute the Jacobian matrices
F = sailboat.F(T, x_hat)
G = sailboat.G(T)

# Compute the a priori estimate (dead reckoning)
P_new = F @ P @ F.T + G @ Q @ G.T
P_new = np.tril(P_new) + np.triu(P_new.T, 1)  # Symmetry
x_new = x_hat + T*sailboat.f(x_hat, u)
```

Figure 24: Linearization and a priori estimate of EKF algorithm.

To linearize the measurement model, we compute the Jacobian with respect to the boat's state, and evaluate it at the current state estimate, which is shown below in the following equations. Note than in the non-linear measurement model, h, there are four measurements from the Bluetooth range sensors, which are non-linear functions of x and y. The remaining three measurements are direct linear measurements of other states. Also note that normally distributed white noise, v, is added to each measurement. The subscripts b_i denote coordinates of the Bluetooth beacons.

$$h(q, v) = \begin{bmatrix} a + b\ln\left(\sqrt{(x - x_{b1})^2 + (y - y_{b1})^2}\right) \\ a + b\ln\left(\sqrt{(x - x_{b2})^2 + (y - y_{b2})^2}\right) \\ a + b\ln\left(\sqrt{(x - x_{b3})^2 + (y - y_{b3})^2}\right) \\ a + b\ln\left(\sqrt{(x - x_{b4})^2 + (y - y_{b4})^2}\right) \\ \theta \\ \gamma \\ \eta \end{bmatrix} + v \tag{13}$$

$$H_k = \frac{dh}{dq} = \begin{bmatrix} \frac{b(x-x_{b1})}{(x-x_{b1})^2+(y-y_{b1})^2} & \frac{b(y-y_{b1})}{(x-x_{b1})^2+(y-y_{b1})^2} & 0 & 0 & 0 & 0 \\ \frac{b(x-x_{b2})}{(x-x_{b2})^2+(y-y_{b2})^2} & \frac{b(y-y_{b2})}{(x-x_{b2})^2+(y-y_{b2})^2} & 0 & 0 & 0 & 0 \\ \frac{b(x-x_{b3})}{(x-x_{b3})^2+(y-y_{b3})^2} & \frac{b(y-y_{b3})}{(x-x_{b3})^2+(y-y_{b3})^2} & 0 & 0 & 0 & 0 \\ \frac{b(x-x_{b4})}{(x-x_{b4})^2+(y-y_{b4})^2} & \frac{b(y-y_{b4})}{(x-x_{b4})^2+(y-y_{b4})^2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (14)$$

The code for this is shown below in Figure 25.

```
# Linearize measurement model, compute the Jacobian
num_features = f_map.shape[1]
log_measures = range_sensor(x_hat, log_parms, 0, f_map)
H = np.empty((num_features+3, sailboat.num_states))
for j in range(0, num_features):
    sq_distance = get_distance(x_hat, f_map, j)**2
    H[j, :] = np.array(
        [
            log_parms[1]*(x_hat[0] - f_map[0, j])/sq_distance,
            log_parms[1]*(x_hat[1] - f_map[1, j])/sq_distance,
            0,
            0,
            0,
            0,
        ]
    )
# Add a measurement for theta, gamma, eta (IMU, wind sensor, rotation sensor)
H[num_features  , :] = np.array([0, 0, 1, 0, 0, 0])
H[num_features+1, :] = np.array([0, 0, 0, 1, 0, 0])
H[num_features+2, :] = np.array([0, 0, 0, 0, 0, 1])
```

Figure 25: Measurement linearization for EKF algorithm.

We can then compute the Kalman gain and a posteriori estimate, which is implemented in Figure 26 below. Note that this code uses some performance optimizations to compute the Kalman gain.

```
# Compute the Kalman gain
K = scipy.linalg.solve(H @ P_new @ H.T + R, H @ P_new).T

# Compute a posteriori state estimate
z_hat = np.zeros(num_features+3)
z_hat[0:num_features] = log_measures  # x, y
z_hat[num_features]   = x_new[2]      # theta
z_hat[num_features+1] = x_new[3]      # gamma
z_hat[num_features+2] = x_new[5]      # eta
x_new += K @ (z - z_hat)

# Compute a posteriori covariance
I_KH  = np.eye(sailboat.num_states) - K @ H
P_new = I_KH @ P_new @ I_KH.T + K @ R @ K.T
P_new = np.tril(P_new) + np.triu(P_new.T, 1)  # Symmetry

# Return the estimated state and covariance
return x_new, P_new
```

Figure 26: Posteriori estimate for the EKF algorithm.

Once localization is completed, model predictive control (MPC) can be used to implement the feedback control [4]. We use a prediction horizon to predict how the boat will behave over the next p iterations and minimize the following cost function, $J(u)$, with respect to the boat's inputs to compute the most optimal inputs [4]. The matrices Q, R are positive, diagonal weight penalties on the state error and input effort, respectively. These are tuned to provide more desirable performance.

$$J(u) = (q_d - q)^T Q(q_d - q) + u^T R u \tag{15}$$

$J(u)$ can be expanded into the following. Note that q_d is the desired state of the boat, and q_k is the discrete-time state of the boat, both are concatenated vectors over the entire prediction horizon, p. L and M are matrices that compute the future state over the prediction horizon.

$$L = \begin{bmatrix} F \\ F^2 \\ \vdots \\ F^p \end{bmatrix}, \qquad M = \begin{bmatrix} G & 0 & 0 & \cdots & 0 \\ FG & G & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ F^{p-1}G & F^{p-2}G & \cdots & \cdots & G \end{bmatrix}$$

$$q = Lq_k + Mu$$

$$J(u) = (q_d - Lq_k - Mu)^T Q(q_d - Lq_k - Mu) + u^T R u \tag{16}$$

Minimizing $J(u)$ leads to the following expression for the optimal control inputs, u^*:

$$u^* = (M^T Q M + R\ )^{-1} M^t Q(q_d - Lq_k) \tag{17}$$

MPC is the most computationally expensive part of the code, and therefore multiple performance optimizations, mostly using numpy and scipy functions, have been implemented to improve performance. Shown below in Figure 27 in the complete MPC implementation in the simulation.

```python
def mpc(sailboat:boat, control_config:settings, T:float, x_d:arr, x_hat:arr) -> float:
    ''' Model Predictive Control for feedback control '''
    # Prediction horizon
    p    = min(control_config.max_pred_horz, x_d.shape[1])
    n, m = sailboat.num_states, sailboat.num_inputs

    # Initialize state error and control effort weight matrices
    Q = scipy.linalg.block_diag(*[np.diag(control_config.state_weights)]*p)
    R = scipy.linalg.block_diag(*[np.diag(control_config.input_weights)]*p)

    # Compute all discrete time approximate linearizations
    F_stack = [sailboat.F(T, x_d[:, i]) for i in range(p)]
    G       =  sailboat.G(T)

    # Compute L
    L = np.vstack([np.linalg.matrix_power(F_stack[i], i+1) for i in range(p)])

    # Compute powers of F for M sequentially
    F_powers = np.array([reduce(np.matmul, F_stack[:i], np.eye(n)) for i in range(p)])

    # Compute M
    M = np.zeros((n*p, m*p))
    for i, j in zip(*np.tril_indices(p)):
        M[n*i:n*(i+1), m*j:m*(j+1)] = F_powers[p-i-1-j] @ G

    # Compute optimal control inputs
    K = scipy.linalg.solve(M.T @ Q @ M + R, M.T @ Q, assume_a='pos')
    u = K @ (x_d[:, :p].ravel(order='F') - L @ x_hat)

    # Take first inputs & clip
    return np.clip(u[:m], -control_config.input_saturation, control_config.input_saturation)
```

Figure 27: Full model predictive control (MPC) algorithm.

The control inputs computed by the MPC are rate inputs, not angular inputs, which the servos require, so they must be integrated to compute the actual angular inputs, shown in Figure 28 below.

```python
def integrate_inputs(prev:arr, rate:arr, T:float,
                     max_eta:float, max_phi:float,
                     sigma_w:list) -> arr:
    ''' Integrate rate inputs to actual servo angle inputs, including noise '''
    u_act = prev + rate*T + sigma_w*np.random.randn()
    u_act[0] = np.clip(u_act[0], -max_eta, max_eta)
    u_act[1] = np.clip(u_act[1], -max_phi, max_phi)
    return u_act
```

Figure 28: Input integration function from rate to angle.

The main loop of the simulation is shown below in Figure 29.

```python
for k in range(1, N):
    # Simulate the robot's motion
    x[:, k] = rk_four(sailboat.f, x[:, k-1], u[:, k-1],
                      test_config.T,
                      boat_config.max_phi,
                      boat_config.max_eta)

    # Take measurements
    z = get_measurements(x[:, k], x_hat[:, k-1], u[:, k-1], sailboat.f, test_config.log_parms,
                         noise_config.sensor_noise, f_map, test_config.T)

    # Use the measurements to estimate the robot's state
    x_hat[:, k], P_hat[:, :, k] = ekf(sailboat, test_config.log_parms, test_config.T,
                                      x_hat[:, k-1], P_hat[:, :, k-1], u[:, k-1], z, Q, R, f_map)

    # Feedback control (servo rates)
    u[:, k] = mpc(sailboat, control_config, test_config.T, x_d[:, k:], x_hat[:, k])

    # Integrate to get actual servo inputs
    u_act[:, k] = integrate_inputs(u_act[:, k-1], u[:, k],
                                   test_config.T,
                                   boat_config.max_eta,
                                   boat_config.max_phi,
                                   noise_config.input_noise)
```

Figure 29: Simulation main loop.

The main loop then repeats until it runs out of points, at which point in generates several output plots, examples of which are shown below in Figure 30. An animation is also generated but not shown below.

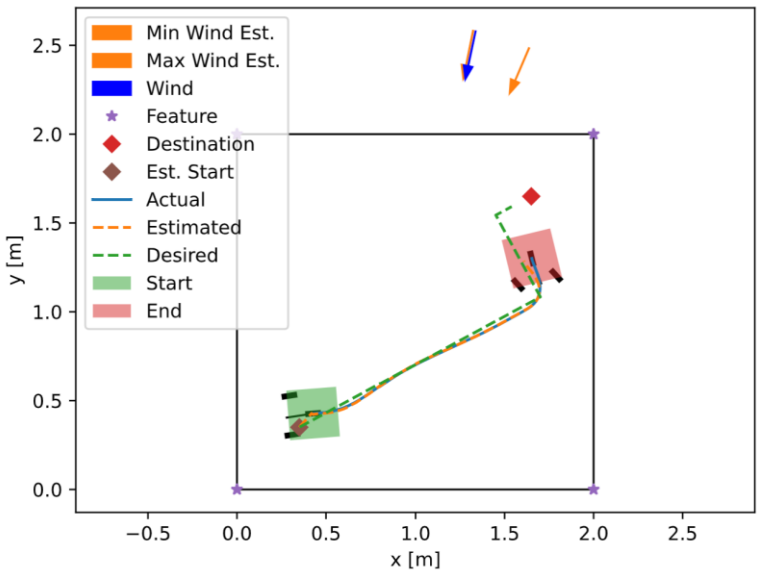

Figure 30: Sample simulation output drawing.

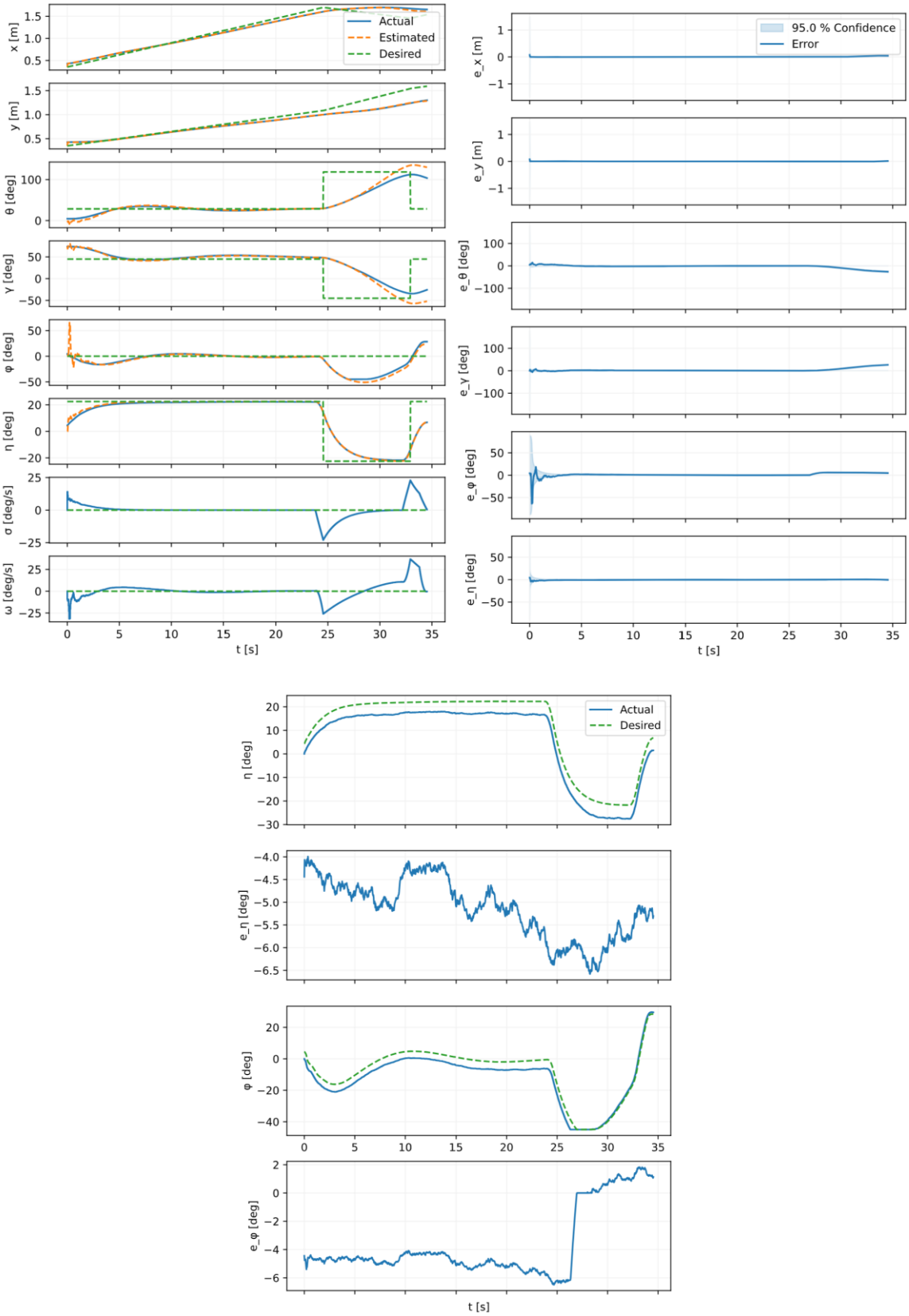

Figure 31: Output plots from the simulation for states, errors, and integrations.

3 Implementation

3.1 High-Level Implementation

The modelled boat was implemented by 3D printing each individual component and ensuring that they were able to fit with each other as expected. This involved re-printing each component, typically several times, to test tolerances. Once finished, the boat was constructed and wired as intended as in the SolidWorks assembly.

High-level implementation of the electronics is similar to the design. The sensor components were connected to their respective housing PCB by being soldered either directly or with wires. The sensors that used I2C were connected to I2C supported GPIO pins on the microcontroller and checked to make sure their I2C address was discoverable. The power management and main PCB were connected using a micro-USB cable connecting from one port on the power management PCB to the port on the Raspberry Pi. The wind direction sensor PCB, the servo output signals, and output power were all connected to designated headers allocated on the main PCB and programmed using the Raspberry Pi to send actuation commands from the algorithm for navigation.

At a high level, the implementation of the software is very similar to the simulation. The main differences are that instead of simulation the boat's motion and generating noisy sensor readings, the software uses real sensors reading and uses the actuators inputs to control the boat's motion. Also, the drivers for the rotation sensors, Bluetooth RSSI range sensors, IMU, and servos must be implemented on the ESP32 and Raspberry Pi Pico W microcontrollers in C++ and MicroPython, respectively. Communication between these boards must also be implemented. For communicating between the ESP32 clients (sensors on the side of the test pool) and sever (on the boat) is accomplished using Bluetooth, while the Pico communicates with the ESP32 server and the PC using Wi-Fi with a UDP protocol [8]. Furthermore, the software must be used to calibrate the sensors and actuators, including measuring IMU drift bias, rotation sensor offsets, zero positions for servo motors, and calibrating the RSSI measurements to compute distance. The software must also be performant enough to sample the sensors and actuate the servos frequency enough to accurately capture and control the boat's motion.

The hardware and electronics used in this project were purchased from a variety of locations. Electronics were purchased from AliExpress, Digikey and Amazon. Hardware was purchased from Digikey, and JLCMC. The printed circuit boards were purchased from JLCPCB. Lastly, the materials for the test environment were purchased from Amazon and Canadian Tire.

This project's initial budget of $600 was significantly exceeded, reaching $824.91. All costs incurred have been displayed below in Appendix G – Cost Breakdown Table. It is important to note that much of this additional cost came from having to reorder non-functional PCBs and replacing electronic components damaged during construction and testing. Between the purchase cost, shipping and duties the cost spent reordering items was $82.19 for the PCBs and $66.08 for the electronics. Neglecting these costs, a total of $676.94 was spent on the project. Additionally, $128.11 was spent on test pool, and items require to inflate and fill it.

3.2 Tools

Shown in Table 2 below is a summary of the tools used throughout the project, including software development and version control tools, various programming languages, 3D modelling software, and PCB development tools.

For the programming languages used, Python was used for the simulation environment, and the PC-side code for the boat where the navigation, localization, and control algorithms run. MicroPython was used on the Raspberry Pi to write drivers for the sensors and actuators that would interface well with the PC code, and finally C++ was used for the ESP32 MCUs that served as Bluetooth range sensors.

SOLIDWORKS and KiCAD were used for 3D modeling and PCB design, respectively, free licenses were available, and they include all the necessary functionalities. For the physical tools, 3D printers were used to print the design, and components were soldered to the PCBs.

Table 2: Tools used in the project.

Tool	Purpose
Git [14]	Project software version control
GitHub [15]	Shared online platform to host project software
Thonny [16]	IDE for writing MicroPython and uploading to Pico W
VSCode [17]	Editor for writing Python code
Arduino IDE [18]	IDE for writing C++ and uploading to ESP32
Mermaid [19]	Diagrams, documentation, and visualizations
KiCAD [20]	Circuit & PCB design
SOLIDWORKS [21]	3D modelling of the boat
Python [22]	PC-side code and simulation code
MicroPython [23]	Programming the Pico W
ulab [24]	Numpy-like library for MicroPython
Soldering	Soldering components to the PCBs
3D Printing	Printing parts for the boat

3.3 Boat Implementation

The 3D design of the boat in SolidWorks was converted part by part to step files, and then 3D printed. Once printed, each part was tested for it's fit with the parts it would interact with. Each component was re-printed, usually several times, to achieve a proper fit. This was significantly complicated by tolerances varying depending on sizes of holes, the amount of printed material around the hole, and the materials that were intended to fit together. In addition, some parts were reprinted to test different print settings, such as layer height and infill, or because the print failed.

Each side and main hulls were sprayed with FlexSeal to ensure that water did not seep into the plastic. Floatation testing was then performed to ensure that each could easily float by themselves. Once 3D printing had been completed, the implemented boat was assembled according to the specifications and locations that had been determined in the SolidWorks assembly. All the PCBs, servo motors, sensors and devices were wired to each other. Figure 32 shows the inside of the main hull with all internal components mounted in place and correctly wired.

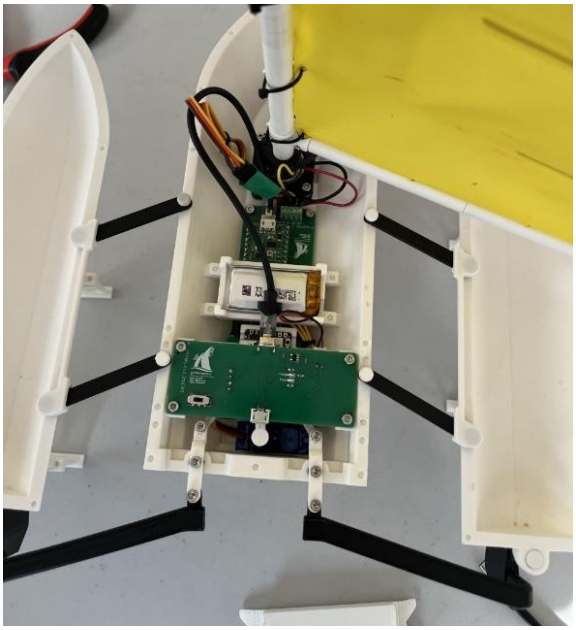

Figure 32: Inside of main hull showing wiring of electrical system.

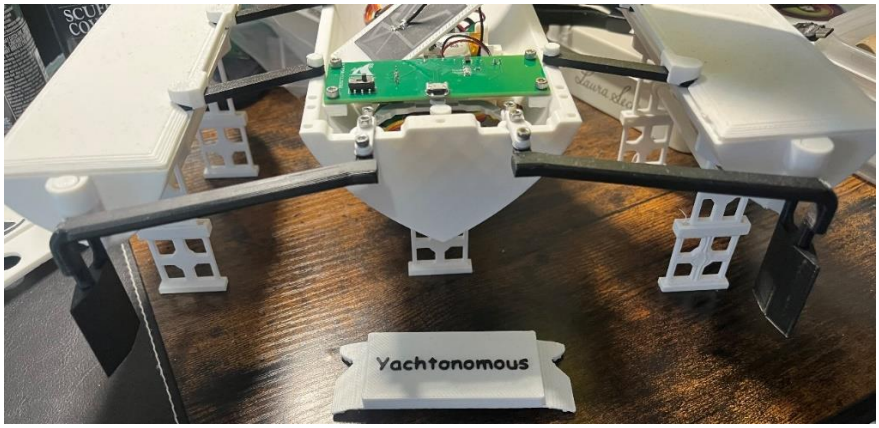

Figure 33: Rudder linkage system in boat.

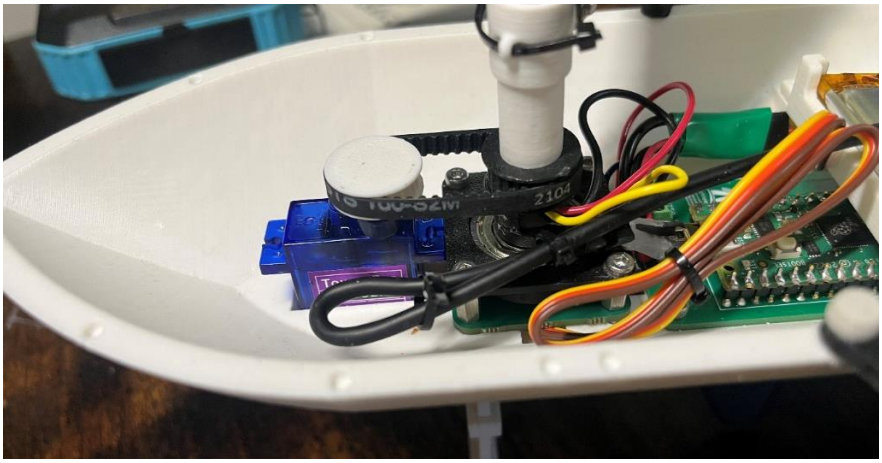

Figure 34: Mast timing pulley system in boat

The systems for actuating the rudders (Figure 33) and the mast (Figure 34) can be seen above.

Finally, the completed boat with hull covers in place is below in Figure 35. The raised cover containing the boat's name, "Yachtonomous", houses the Power Management and Control PCB, which allows for turning on and charging the boat. Located directly in front of it is the antenna insert, which allows for the removal of the covers without detaching the antenna from the ESP32. The mainsail was constructed from nylon using toothpicks as battens. It was modelled after a Laser mainsail [25].

Figure 35: Fully assembled model sailboat.

3.4 Hardware & Electronics Implementation

The hardware and electronics implementation involves integrating the PCBs into the boat interfacing them with each other. The power management PCB has a micro-USB port that can be plugged into from the back of the boat. This will be used to upload code to the microcontroller and charge the battery.

The power management PCB is connected to the main PCB via a micro-USB cable that runs from a second port on the power management board to the port that exists on the microcontroller itself. The main PCB needs connections to the servo motors for actuation and to the wind direction sensor. The main PCB has screw terminal connectors that have pins for power, ground and input/output signals for the wind direction sensors and the servo motors.

Ultimately, the power management PCB, the wind direction sensor and the ESP32 server are all integrated with the main microcontroller PCB for data inputs and outputs.

Figure 36 below shows each designed PCB without components soldered on. Eventual placement of each PCB within the hull of the boat is in Figure 37. This picture does include the second rotation sensor PCB, as it is used for the wind vane located on top of the mast.

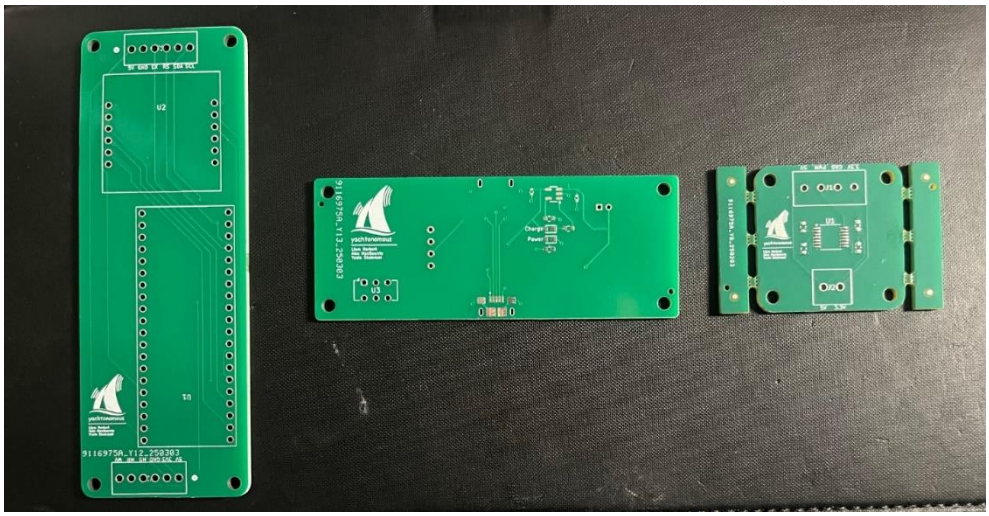

Figure 36: All unsoldered PCBs (Main PCB, Power Management PCB, Rotation Sensor PCB).

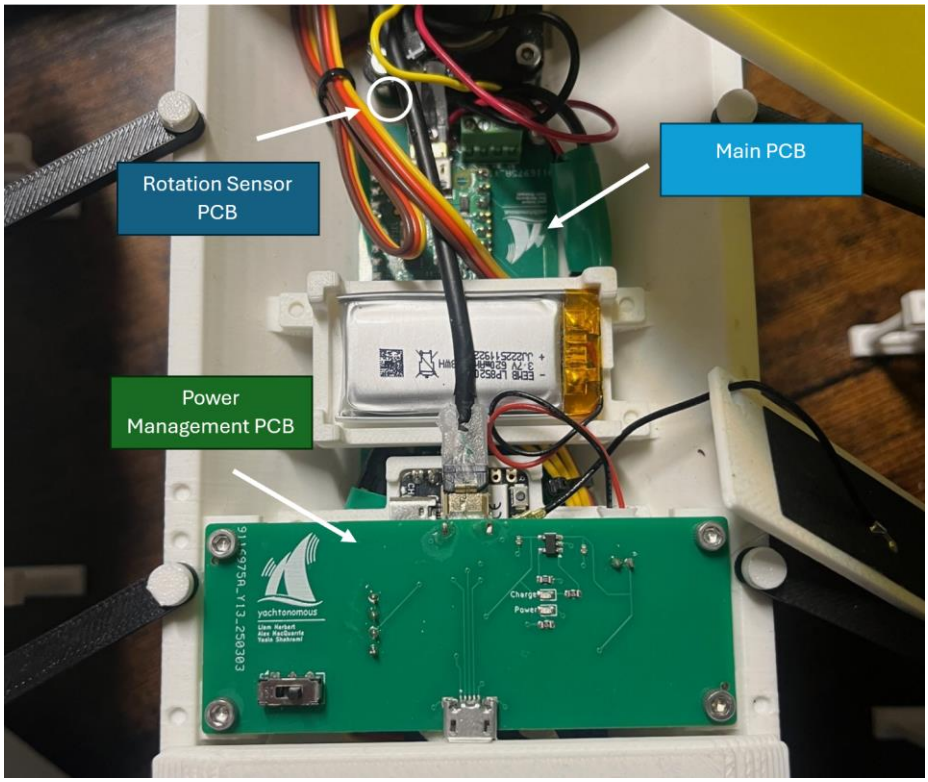

Figure 37: Labeled location of each PCB located with main hull.

3.5 Software Implementation

As previously mentioned, the software is responsible for implementing the main control loop. This consists of the custom navigation algorithm, the feedback control using MPC, and localization using an EKF. These algorithms rely on the mathematical model of the boat and its linearization's. In total, the project is around 3000 lines of code, of which approximately 80% is Python or MicroPython, with the rest being C++.

These three main algorithms are written in Python and run on a PC. The implementation is the same as descried in the software design section about the simulation. Originally, the plan was to run these on the Pico W in MicroPython, but the Pico W does not have sufficient RAM to allocate some of the large matrices used in the MPC algorithm. Instead, the code the on the Pico W still uses MicroPython, but now simply drives the sensors and actuators, and serves as the main communication point for the wireless communications on the boat. For the ESP32 Bluetooth sensors, the four clients on the sides of the test pool all run very similar C++ code to transmit their RSSI values to the ESP32 sever location on the boat over Bluetooth. The ESP32 server then sends the list of four RSSI values to the Pico W over Wi-Fi using a UDP connection, with the Pico W acting as an access point, hosting it's on Wi-Fi network. The PC also communications with the Pico W using a UDP connection to receives sensor reading and send actuator inputs.

Show below in Figure 38 is the main loop for the Pico W. The file is named main.py, which causes the Pico to run this file each time it is powered on. The loop effectively just allows the Pico to receive and respond to requests on the socket from the PC and the ESP32 server. To maintain a reliable, flexible, and consistent format, all requests use a JSON format.

```
# Set up UDP socket
pico_socket = udp_socket(LOG_EN)

while True:
    # Receive request
    command, request_json = pico_socket.get_request()

    # Send data back or use actuator values
    if   command == 'SEND_INIT_WIND':
        initial_gamma = estimate_initial_gamma(num_samples = NUM_WIND_SAMPLES,
                                               delay_us    = WIND_AVG_DELAY_US)
        pico_socket.send_init_gamma(initial_gamma)
    elif command == 'SEND_DATA':
        pico_socket.send_sensor_readings(get_measurements())
    elif command == 'RECV_DATA':
        sail_and_rudder_servos.actuate_servos(request_json['eta'], request_json['phi'])
    elif command == 'ESP32':
        rssi_manager.update(request_json['number'])
    elif command == 'END_COMMS':
        if LOG_EN:
            print('Ending program')
        break
    else:
        raise Exception(f'Unkown command from client: {command}')

    # Blink to show loop working
    led.toggle()
```

Figure 38: Main MicroPython loop for the Pico W

Shown below in Figure 38 is the constructor for the Pico UDP socket, which creates the Wi-Fi access point. On the PC, there is a similar UDP socket as shown below in Figure 40.

```python
class udp_socket:
    def __init__(self, log_en:bool, timeout:float):
        ''' Set up UDP socket and default parms'''
        self.__client_socket  = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.__server_address = ('192.168.4.1', 12345)  # Pico W AP default IP and port
        self.__buf_size       = 1024
        self.__log_en         = log_en
        self.__client_socket.settimeout(timeout)
```

Figure 39: Constructor method for the UDP socket for the PC.

```python
class udp_socket:
    def __init__(self, log_en:bool):
        ''' Set up access point and UDP server '''
        # Set up Pico W as access point
        ap = network.WLAN(network.AP_IF)
        ap.config(essid=_SSID, password=_PASSWORD)
        ap.active(True)
        while not ap.active():
            time.sleep(_ACT_DELAY_S)

        # Set up UDP server
        self.log_en         = log_en
        self.buf_size       = _BUF_SIZE
        self.client_address = None
        self.server_socket  = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.server_socket.bind((ap.ifconfig()[0], _UPD_PORT))
        if self.log_en:
            print('Server setup complete')
```

Figure 40: Constructor method for the UDP socket for the Pico W.

The PC makes requests on the socket in the main loop, as shown below. These are requests for sensor measurements provided by the Pico, or requests to actuate the servos. This is very similar to the main loop of the simulation, with one notable exception being that the sample period is updated dynamically, as with noise, actuation, sensor reading, and communication times, this value is not exactly the same for each iteration, but it must be accurate to be used for integrations. This is shown below in Figure 41.

```python
# Run simulation
for k in range(1, x_d.shape[1]):
    # Get start time for dynamic T
    start_time = time.time()

    # Get measurements from Pico
    z = pico_socket.get_measurements(x_hat[2], num_features, T)

    # Use the measurements to estimate the boat's state
    x_hat, P_hat = ekf(sailboat, test_config.log_parms, T, x_hat, P_hat, u, z, Q, R, f_map)

    # Feedback control (servo rates)
    u = mpc(sailboat, control_config, T, x_d[:, k:], x_hat)

    # Integrate to get actual servo inputs (angles)
    u_act = integrate_inputs(u_act, u, T, boat_config.max_eta, boat_config.max_phi)

    # Send inputs to actuators via Pico
    pico_socket.send_actuator_inputs(u_act)

    # Log results for debug
    runtime += T
    logger.log_results(x_hat, x_d[:, k], u, u_act, runtime)

    # Wait to not overwhelm server
    time.sleep(test_config.wait_time_s)

    # Measure T
    T = time.time() - start_time
    if test_config.measure_T:
        times[k] = T
```

Figure 41: Main loop for the PC-side Python code.

Shown below in Figure 42 is some of the code on the ESP32 clients to send the RSSI value it measures to the ESP32 server over Bluetooth. First, we remove the negative sign, as it not needed, then we append zeros to the left based on the number of digits in the RSSI value, to ensure the format is consistent when the client ID for the sensor is appended as the most significant digit. The RSSI values seen by the server would look something like “3124”, which would be decoded into an RSSI value of “124” from client “3”. The four clients are enumerated 0-3, and the RSSI values (after taking the absolute value) never exceed three digits, and rarely even exceed two digits, to due to logarithmic relationship between distance and RSSI.

```
// Write the (positive) integer value to the server in  the correct format
String value = String(-1*currentRSSI);
while(value.length() < 3)
{
    value = "0" + value;
}
value = cliID + value;
Serial.println(value);

String dataToSend = value;
pRemoteCharacteristic->writeValue(dataToSend.c_str(), dataToSend.length());
```

Figure 42: Client-side RSSI serialization.

On the ESP32 server, a buffer of four integer values is maintained to keep track of the RSSI measurements. When the server reads an RSSI value from a client on callback, the value is inserted into the correct position in the buffer as shown below in Figure 43.

```
void updateRSSIList(int newRSSI)
{
    // Deserialize the RSSI value from the client and insert into buffer
    int cli_id   = newRSSI / 1000;
    int rssi_val = newRSSI % 1000;
    arr[cli_id]  = rssi_val;
}

// Custom callback class to handle characteristic writes
class MyCharacteristicCallbacks : public BLECharacteristicCallbacks
{
    void onWrite(BLECharacteristic* pCharacteristic)
    {
        // When the client writes to the characteristic, we read the value
        String value = pCharacteristic->getValue();
        int receivedInteger = value.toInt();
        Serial.print("Received Integer from client: ");
        Serial.println(receivedInteger);

        // Update buffer of RSSI values
        updateRSSIList(receivedInteger);
    }
};
```

Figure 43: Server-side RSSI parsing.

This buffer is then periodically sent to the Pico W over Wi-Fi in a JSON format, shown in Figure 44.

```
void loop()
{
    // Add key-value pairs to the JSON object
    StaticJsonDocument<200> doc;
    doc["command"] = "ESP32";
    doc["number"] = arr;

    // Serialize the JSON object into a string
    String jsonMessage;
    serializeJson(doc, jsonMessage);

    // Send the JSON message over UDP to the Pico W
    udp.beginPacket(udpAddress, udpPort);
    udp.write(reinterpret_cast<const uint8_t*>(jsonMessage.c_str()), jsonMessage.length());
    udp.endPacket();

    Serial.print("Sent JSON message: ");
    Serial.println(jsonMessage);

    delay(delayMs);
}
```

Figure 44: Server-side loop to send buffer of RSSI values.

Shown below in Figure 45 is the driver for the IMU. Because the IMU is only used to measure the yaw of the boat, we only need to use the Z-axis gyroscope. The magnetometers could also be used but given that there are relatively strong magnets nearby for the rotation sensors, the gyroscope was chosen. Communication with the IMU breakout board is achieved via I^2C using MicroPython libraries. For the roation sensors, measuring the angle is achieved by timing the duty cycle of the rotation sensor PWM pin, which corresponds to a given angle.

```
class imu_i2c:
    def __init__(self) -> None:
        self.__i2c = I2C(_I2C_BUS_ID, sda=Pin(_SDA_PIN), scl=Pin(_SCL_PIN), freq=_IMU_FREQ)
        self.__scan_i2c_bus()
        self.__initialize_imu()

    def __scan_i2c_bus(self) -> None:
        ''' Scan for I2C devices '''
        if not self.__i2c.scan():
            raise Exception('No I2C devices found')

    def __imu_write_register(self, reg, data) -> None:
        ''' Write a byte to a register '''
        self.__i2c.writeto(_IMU_ADDR, bytes([reg, data]))

    def __imu_read_register(self, reg, length) -> int:
        ''' Read bytes from a register '''
        self.__i2c.writeto(_IMU_ADDR, bytes([reg]))
        return self.__i2c.readfrom(_IMU_ADDR, length)

    def __initialize_imu(self) -> None:
        ''' Wake up and configure IMU '''
        self.__imu_write_register(_PWR_MGMT_1, _HIGH)

    def read_gyro_z_rps(self) -> float:
        ''' Read Z-axis gyroscope data and convert to rad/s '''
        raw_data = self.__imu_read_register(_GYRO_XOUT_H, _SIX_BYTES)
        _, _, gz = unpack('>hhh', raw_data)
        return math.radians(gz/_SENSITIVITY_DPS) - _GZ_OFFSET_RAD
```

Figure 45: Object to control IMU via I2C.

Shown below in Figure 46 is the MicroPython object representing a servo motor. It is connected to a PWM pin with a given frequency based on the servo motor spec. To set the servo to a given angle, the duty cycle is used. There are two linear interpolations used, one between 0 and 90 degrees, and the other between 90 to 180 degrees, to improve the accuracy of the servo actuation. The constants used in the duty cycle calculations were determined during testing.

```
class servo:
    def __init__(self, pin:int, pwm_freq:int):
        ''' Instantiate PWM pin '''
        self.__pwm_pin = PWM(Pin(pin), freq=pwm_freq)

    def set_angle(self, angle_radians:float) -> None:
        ''' Set the angle for a servo, input should be [-pi/2, pi/2] '''
        # Map [-pi/2, pi/2] -> [0, 180]
        mapped_angle = math.degrees(angle_radians) + _90_DEG

        # Linearly interpolate duty cycle over 2 windows for greater accuracy (0-90 and 90-180)
        if mapped_angle <= _90_DEG:
            duty = int(_MIN_DUTY+(mapped_angle/_90_DEG)*(_MID_DUTY-_MIN_DUTY))
        else:
            duty = int(_MID_DUTY+((mapped_angle-_90_DEG)/_90_DEG)*(_MAX_DUTY-_MID_DUTY))

        # Set duty cycle
        self.__pwm_pin.duty_u16(duty)
```

Figure 46: Object to control servo via PWM.

4 Testing

4.1 Physical Boat & PCB Testing

Floatation and press fit testing were performed on the physical boat structure. The wind vane was constructed independently from the rest of the boat and was tested by itself to verify its performance. This required iteratively adjusting the size and shape of the tail, to ensure that it was large enough to aim in the direction of wind, but light enough to avoid an imbalance of weight. Each the rudder and mast rotation systems were successfully tested.

PCB testing was done by making sure all the individual components on the PCB's functioned as expected prior to soldering as well as checking connections on the PCB for continuity to make sure that there were no shorted signals.

After soldering and checking for continuities, the boards had to be tested after integration. First the power management PCB was tested by attempting to power the Pico-W using the battery and an external power source connected to the micro-USB port. The charging of the battery was also tested by using a multimeter to check the voltage of the battery after use and after an hour of charging. Its connection to the microcontroller also allowed for code to be uploaded to test the IMU with I2C connections.

Lastly, the servo motors and the wind direction sensor had physical wires soldered into terminals for incoming and outgoing signals. The integrity of the soldering had to be tested after placing the PCBs into the boat to make sure that the connections were behaving as expected.

4.2 Sensor & Actuator Testing

Shown below in Table 3: Summary of sensor and actuator noises.Table 3 are quantitative results from testing the noise of the sensors and actuators. These noise measurements are useful to evaluate the accuracy of the sensors and actuators, and they are also required in the EKF to help the state estimation converge and reduce uncertainty over time. Overall, the sensors and actuators performed well, however the rotation sensor noise was higher than desirable, especially the wind sensor, which has effectively double the noise of the mast rotation sensors since the mast angle must be subtracted from the measured wind angle to calculate the relative wind angle, causing the noise to compound.

Table 3: Summary of sensor and actuator noises.

Sensor/Actuator	Input/Output Noise [σ]	Source
Servo (Mast & Rudder)	0.05 rad	Observation
Bluetooth RSSI sensors	0.08 m	Sampling & Regression
IMU	0.01 rad	Sampling
Wind rotation sensor	0.30 rad	Sampling
Mast rotation sensor	0.15 rad	Sampling

For the servos, noise was calculated by placing a small servo horn on the servos and repeatedly actuating the servos between 0 and 90 degrees. Using a protractor under the servo horn, the actual angle was measured. This was done 25 times, after which the standard deviation was calculated.

The IMU was first calibrated by reading the gyroscope Z-axis rotation for 10,000 samples and taking the mean to measure the drift and correct for it. After this, the yaw angle of the IMU was calculated via integration, and sampled 10,000 times, after which the standard deviation of the angle was taken. The IMU bias is shown below in Table 4. Also shown in this table is the fixed offset of the wind and mast rotation sensors to ensure they read an angle of 0 at the correct angle.

Table 4: Summary of sensor offsets and biases.

Name	Value
IMU Bias	-0.01082 rad
Wind Sensor Offset	129.48 deg
Mast Sensor Offset	16.27 deg

Shown below in Figure 47 is the MicroPython script used to perform the calibration.

```
# External
from time import time
from ulab import numpy as np
# Internal
from sensor import get_measurements
from servo import sail_and_rudder_servos

N, T, angle_rad = 10000, 0, 0
vals_rad = np.zeros((4, N))

for i in range(N):
    start = time()

    sail_and_rudder_servos.actuate_servos(0, 0)

    z = get_measurements()
    gyro_z_rps = z[4]
    angle_rad += gyro_z_rps*T
    wind_angle = z[5]
    sail_angle = z[6]

    vals_rad[:, i] = np.array([gyro_z_rps, angle_rad, wind_angle, sail_angle])

    T = time()-start

means, stds = np.mean(vals_rad, 1), np.std(vals_rad, 1)
print(f'gyro_mean_rad={means[0]}            \n theta_std_deg={stds[1]}           \n '
      f'gamma_mean_deg={np.degrees(means[2])}\n gamma_std_deg={np.degrees(stds[2])}\n '
      f'eta_mean_deg={np.degrees(means[3])}  \n eta_std_deg={np.degrees(stds[3])}  \n ')
```

Figure 47: MicroPython code to calibrate the boat.

For the Bluetooth RSSI sensors, the regression shown in Figure 48 below was used.

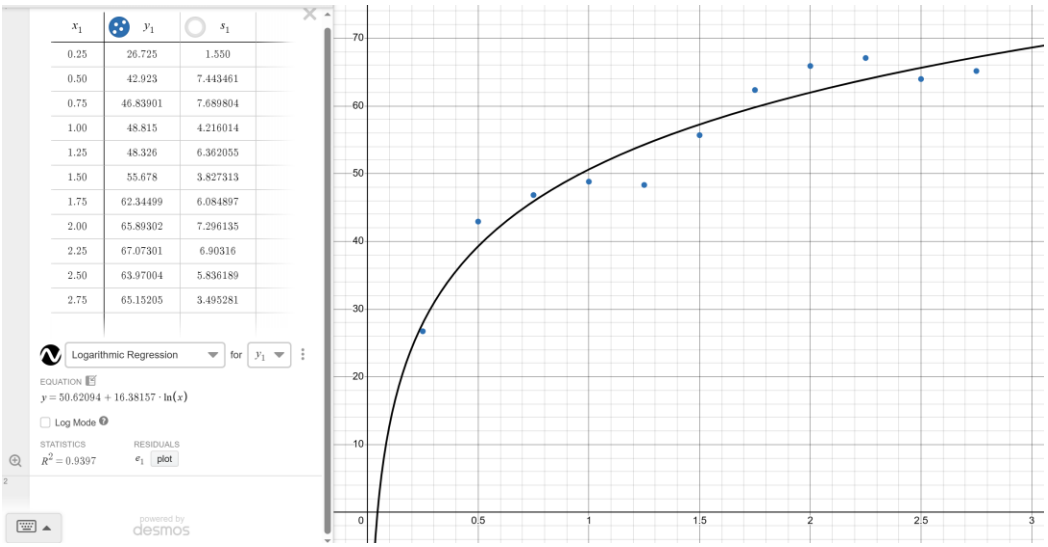

Figure 48: RSSI to distance regression in Desmos.

Samples were taken every 25 centimetres over the relevant range of the sensors given the size of the test pool and fit to a logarithmic regression. For each distance, 1000 samples were read, and the mean and standard deviation were calculated. A logarithmic regression is used because it fits well and makes sense physically as signal strength would likely reduce exponentially as a function of distance. The maximum standard deviation occurred at 75 cm, which was used to calculate the noise in metres by passing that value through the inverse of the regression.

4.3 Software Simulation & Testing

Several tests were conducted on the software to ensure it was fully functional. Due to having limited time to test the boat fully integrated in the test environment, much of the software testing used the simulation to test independently from the hardware.

Shown below in Table 5 are the outcomes of two regressions to calibrate the software. The Bluetooth RSSI regression was previously shown above. The RSSI regression fit a logarithmic curve very well, and the coefficients were used by the EKF to determine the boat's distance from each sensor, allowing it to determine the boat's position.

The second regression describes how the boat's speed changes depending on its orientation relative to the angle of the wind. Typically, sailboats cannot move when pointing directly into the wind (a point of sail commonly referred to as "irons") and reach maximum speed on a broad or beam reach. Sailing upwind is slower than sailing on a reach, and the boat's speed declines rapidly as the relative wind angle approaches zero. The data points represent the boat's speed at each angle (normalized to its maximum speed). This was determined in the test pool environment by placing the boat on the correct point of the sail in the test environment and timing how long it took to cover a given distance.

Table 5: Summary of regression parameters.

Purpose	Regression Type	Coefficients	R^2
Bluetooth RSSI	Logarithmic	50.621, 16.382	0.9397
Boat speed relative to wind	Quartic	-0.02427, 0, 0.2600, 0, 0.31282	0.7264

Shown below in Figure 49 is the quadratic boat speed regression. As expected, only the even terms are non-zero, ensuring that the function is even. Overall, the regression did not fit as well as the Bluetooth RSSI regression fit, but the quartic regression still fit the best out of any other model.

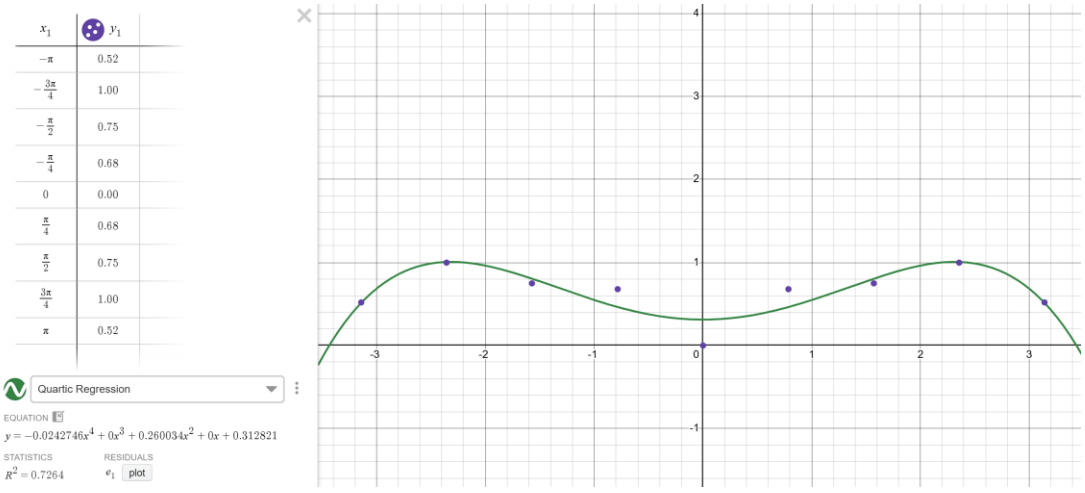

Figure 49: Boat speed relative to wind angle regression in Desmos.

Using the simulation, the state error and input effort weights for the MPC algorithm were tuned. The tuning was done qualitatively based on observing how well the boat followed its desired trajectory

to the destination. The highest weight was given to the angle of the boat, as this was critical to ensuring it did not point too high into the wind and slow down, or tack to slowly. The x and y weights are also high to ensure the boat stays close to its desired path in space and does not overshoot tacks. The weights on the sail and rudder angle were low, as these will need to change to allow the boat to adjust is direction and speed. The weight on the relative wind direction had to be set to zero to avoid a bug when sailing downwind, where if the desired relative wind angle was 175 degrees, for example, but the sensor read -181 degrees, this gives an inflated error of 356 degrees, when the actual error is only 6 degrees. This weight is not very important as the orientation state conveys the same information, given a constant wind angle. All weights are shown below in Table 6.

Table 6: Final MPC weights for states and inputs.

State/Input	x	y	θ	γ	φ	η	σ	ω
Value	10	10	15	0	2	1	1	5

Another value tested in the simulation was the prediction horizon for the MPC algorithm. Ultimately, 25 provided the best performance, as not looking ahead far enough did not allow the boat to predict its future behaviour sufficiently, and at higher values, the algorithm is much slower, and sensor and actuator noise cause the accuracy of the prediction to decay over time.

Testing the navigation algorithm was needed to ensure that a valid course was generated for the boat to sail each time. In addition to qualitatively checking most reasonable combinations of inputs to the navigation algorithm, a unit test was written, shown below in Figure 50, that can be run to check a very large number of combinations (~400,000 in the example below). A failure is raised if the algorithm encounters an error, or if the path length gets too long, which occurs when the navigation has an error and fails to reach the destination.

```python
def _test_navigation() -> None:
    ''' Full Navigation Test Suite '''
    # Create permutations
    crits       = [Angle.exp(i, deg=True) for i in range(30, 70, 10)]
    winds       = [Angle.exp(i, deg=True) for i in range(-180, 180, 20)]
    boats       = [np.array((0.25+i/10, 0.25+j/10)) for i in range(-2, 3) for j in range(-2, 3)]
    dests       = [np.array((1.75+i/10, 1.75+j/10)) for i in range(-2, 3) for j in range(-2, 3)]
    boat_angles = [Angle.exp(i, deg=True) for i in range(10, 90, 10)]

    # Get number of tests
    num_tests = len(crits)*len(winds)*len(boats)*len(dests)*len(boat_angles)-len(boats)
    print(f'Running {num_tests} tests...')

    # Run all tests
    start_time = time()
    for boat_angle in boat_angles:
        for boat in boats:
            for dest in dests:
                for crit in crits:
                    for wind in winds:
                        # Skip when boat == dest
                        if np.array_equal(boat, dest):
                            continue
                        # Run the test
                        try:
                            plot_course(boat, dest, wind, boat_angle, crit, plot_ctrl=PlotCtrl.ON_FAIL)
                        except NavigationError as err:
                            print('FAILURE!!!')
                            raise NavigationError(f'Nav error for Boat={boat}, '
                                                  f'Dest={dest}, Wind={wind}, Crit={crit}, '
                                                  f'BoatAngle={boat_angle} -> {err}')
    print(f'All tests passed in {(time()-start_time)/60.0:.2f} minutes')
```

Figure 50: Navigation unit test in Python.

Overall, the simulation was critical to ensure the algorithm could be developed in parallel with the hardware, and it also allowed for extensive testing and tuning of several parameters.

4.4 Performance Testing Environment

The testing environment selected was a Stella & Finn Octagonal Wading pool, shown below in Figure 51. It has an outer radius of 2.79m.

Figure 51: View of fully setup test environment.

4 of 8 sides of the pool contained an antenna adhered to the surface attached an ESP32, aiding in the localization of the boat. The locations of the pool marked with “X” demonstrate the boat’s intended start and finish locations.

The pool was inflated by hand and filled by running a hose out of a second-floor window, attached to a bathroom sink. Once the setup process was complete, floatation testing was performed with the fully constructed sailboat. Figure 52 shows this testing in progress.

Figure 52: Floatation testing of boat within test pool

5 Results

5.1 Summary of Critical Values

Shown below in Table 7: Various critical values used by the software.Table 7 is a summary of many of the critical values needed by the boat software. Values where obtain from datasheets, measuring the boat or test environment, profiling the software, or determined to be effective by the software simulation.

Table 7: Various critical values used by the software.

Name	Value	Source
Max servo rotation rate [deg/s]	600	Datasheet [6]
Max mast servo angle [+/- deg]	90	Measurement
Max rudder servo angle [+/- deg]	45	Measurement
Rudder to centerboard length [m]	0.127	Measurement
Width between rudder [m]	0.219	Measurement
Destination threshold [boat lengths]	1	Simulation
Mean sample period [s]	0.03	Code Profiling
Point speed factor	0.9	Simulation
Effective pool size [m, m]	1.6, 1.6	Measurement
Start position [m, m]	0.25, 0.25	Measurement
Destination position [m, m]	1.35, 1.35	Measurement
Border padding [m]	0.05	Simulation
Maximum boat speed [m/s]	0.1	Measurement
Minimum angle to wind [deg]	45	Measurement

The mean sample period is the time taken to complete an iteration of the main program loop. It must be sufficiently high to ensure the sensors are sampled frequently enough and the actuators are actuated often. This value is updated dynamically to ensure linearization and integrations are performed accurately, but the mean is used to determine the distance between points in the navigation, as this and the boat's estimated speed are required to determine how far apart the points should be generated. The point speed factor is used to slightly reduce this speed to ensure the desired course does not surpass the boat's position.

The minimum angle to the wind is the closest the boat can sail to directly upwind without a significant reduction in boat speed. The destination threshold is how far away from the destination, in boat lengths, the navigation algorithm should stop. The maximum servo rate and angles are used in the software to ensure we do not try to actuate the servos faster or further than they can move. Unfortunately, the control and localization algorithms cannot efficiently incorporate these constraints, so they can cause the estimated state to deviation from the actual state, which slightly reduces performance. It would be ideal to get more range out of the rudder servo, as in even in simulation it was shown that it would be beneficial to have a range of approximately ±60 degrees, however this was limited by space constrains with the servo, the servo horn, and the rudder linkage system.

5.2 Performance Results

Table 8 below shows observations relating to the performance of the boat in the test environment under several different wind directions. Relative wind angles were tested every 45 degrees.

Table 8: Summary of boat performance in different wind angles.

Relative wind direction [deg]	Point of Sail	Observation
0	Irons	Did not sail at all, as expected
45	Close hauled (upwind)	Could not sail upwind, slipped sideways when pushed by wind
90	Beam Reach	Worked alright, still pushed
135	Broad Reach	Worked well. Boat sailed to destination
180	Run (downwind)	The boat turned away from the sail instead of sailing straight

When the relative wind angle was 0 degrees the boat was not able to sail effectively. This is expected since boats cannot sail in irons.

Upwind sailing was unsuccessful, the boat would slip and begin to travel in the direction of the wind as opposed to sailing against it. This is again due to the design of the boat and the significant effect of the force of the wind. Similarly, when sailing straight downwind, the sail being on one side of the boat caused the boat to rotate instead of sailing in a straight line.

The best test conditions were at wind angles 90 degrees and 135 degrees relative to the boat, which is a beam and broad reach. Under these conditions, the boat was reliably able to sail to the destination, with more errors on a beam reach due to the wind pushing the boat more.

Ultimately, designing a sailboat that is mechanically and aerodynamically sound in water is a difficult task and contributed greatly to its difficulty to move effectively in the test environment. However, there were multiple instances where the navigation was successful despite the shortcomings of the design. The testing and performance of the boat was relatively successful although it could be improved with further research.

Figure 53: Fully constructed boat floating in test pool.

5.3 Comparison to Specifications & Requirements

Shown below in Table 9 is the hardware requirements table from the project blueprint. 7/11 specifications were met, 3/11 were partially met, and 1/11 were not met.

All the test pool and boat size specifications were met, as well as the mast rotation range. The rudder rotation range fell short due to physical space constraints within the hull of the boat and the linkage system, which did harm the performance of the boat.

The specification for the speed of the wind generation was partially met, as while the wind speed was not actually measured, it generated a sufficient wind speed for the boat to sail in. For the Bluetooth and wind sensors, the ranges were met, however measuring a specific tolerance differs in the final product, as while the measured noise values are greater than the tolerance specification, the EKF algorithm decreases the uncertainty in the state that these sensors measure over time to below the specified tolerance value. The team was not considering an algorithm like the EKF when the specification was written. The hardware specifications are below in Table 9.

Table 9: Hardware spec table from the blueprint.

Specification	Value Type	Target Value	Tolerance (+/-)	Achieved Value?
BT Localization Network	Object Coordinates	5 - 200 cm	3 cm	Partially
Rudder Angle System	Angle of Rudder	135° Range	0.001°	No (90°)
Mast Rotation System	Angle of Mast	135° Range	0.001°	Yes (180°)
Electronic Wind Vane	Angle of Wind Vane	360° Range	1°	Partially
3D Printed Boat	Boat Length	25 – 35 cm	0.5mm	Yes (28 cm)
3D Printed Boat	Boat Width	20 – 30 cm	0.5mm	Yes (28 cm)
3D Printed Boat	Boat Height	30 – 40 cm	0.5mm	Yes (36 cm)
Test Environment	Test Pool Length	1.0 – 1.5 m	1 cm	Yes (1.6 m)
Test Environment	Test Pool Width	1.0 – 1.5 m	1 cm	Yes (1.6 m)
Test Environment	Test Pool Depth	10 – 15 cm	1 cm	Yes (15 cm)
Wind Generation System	Wind Speed	3 – 5 knots	0.5 knots	Partially

Shown below in Table 10 is the software requirements table from the project blueprint. 9/15 were met, 3/15 were partially met, and 3 were not met.

For the functional requirements, the navigation algorithm works as specified, and driving the servo motors worked well. Controlling the boat along is trajectory was partially met, as while this worked in simulation and on some points of sail, issues with the boat sailing well which are unrelated to the software make it impossible to determine if the software was capable of fully controlling the boat as well as it did in the simulation. The software fit on the Pico W and worked in MicroPython, but could not ultimately run on the Pico W, as it did not have enough memory to allocate some of the large matrices used in the MPC algorithm. This was worked around well by having the Pico W communicate wirelessly with a PC that could easily run the code in Python.

All the interface requirements were met, and most of the performance requirements were met. The power consumed by the boat was low enough to complete all testing (10+ runs) on a single charge, and with a sampling period of approximately 30 milliseconds, the software was fast enough to control the boat accurately. The noise on the servos was too high to always actuate within ±1 degree, which is also true of calculating the wind angle, but using the EKF, the wind angle converges to below this tolerance, hence partially meeting the spec. Similarly, with to boat position, without the EKF the noise is too high, but the EKF allows this range to converge as the boat moves towards the destination.

Table 10: Software spec table from the blueprint.

Functional Requirements	Specification Met?
Software must fit and run on desired microcontroller	No
Software will be written primarily in MicroPython	No
Calculate a sailable trajectory from the boat to the destination	Yes
Control the boat to follow the calculated trajectory to the destination	Partially
Drive 2 servo motors to actuate the sail and rudder angles	Yes
Read input from Bluetooth transceivers and use to perform localization	Yes

Read input from wind direction sensor and translate into wind direction	Yes
Interface Requirements	
Read switch inputs to turn on and off the boat	Yes
Drive LEDs to display basic information about the state of the boat	Yes
Provide debug logs with detailed information on boat state and operation	Yes
Performance Requirements	
Operate at sufficiently high frequency to achieve control of the boat	Yes
Draw low enough power to allow the boat to perform 5+ tests without recharging	Yes
Calculate boat and designation position to ±2 cm	Partially
Actuate servo angles to ±1 degree	No
Calculate wind angle to ±2 degrees	Partially

6 Conclusion

6.1 Conclusions

The design, implementation, and testing of the "Yachtonomous" demonstrated both the potential and the challenges associated with the development of a fully functional autonomous sailboat. The project accomplished the integration of the three distinct software, electrical, and physical systems into a working prototype.

Each of the distinct subsystems proved their functionalities in isolation. The software system through simulations demonstrated the developed kinematic model, and in practice was able to process input from the boat's sensors and transmit its output to the electrical system. The electrical system provided power to each of the boat's electrical components. During operation it collected sensor readings, transmitted them to the software system, and actuated the relayed commands to the rudders and mast. The physical boat provided space for all internal components and optimized the process of connecting them. It was able to float by itself and keep each of its internal components dry. Each the mast and rudder provided a method for actuation and ability to rotate within their required ranges.

The iterative process for the design of each subsystem ensured not only the functionality of each, but crucially a seamless integration of all three connected together. Sensors located on the physical boat routed to a microcontroller, transmitted information that was processed and transmitted back, corresponding to on board adjustments depending on current environmental conditions and the intended direction of motion.

The completed boat when tested was able to successfully sail between two defined points under certain wind conditions. However, for a significant proportion of possible angles, the boat was unable to sail in its intended direction. This was not due to short comings in the boat's systems, but due to aerodynamic and weight balance discrepancies between the 3D printed model, and that of a fully functional sailboat. This was in many ways, beyond the scope of this project.

6.2 Recommendations

There are several improvements that could be made to improve the function and performance of the boat. Most of the solution were not implemented due to either cost, time, or knowledge constraints.

First, using higher quality sensors with reduced noise would improve the performance of the localization and navigation algorithms. Using sensors like GPS for position could provide greatly improved accuracy over Bluetooth RSSI, at an increased cost. Higher quality IMUs can also be used with reduced drift over time. If similar rotation sensors were to be used, reading the angle via I^2C instead of the PWM signal would likely improve their performance.

Another addition that would improve state estimation would be to use higher quality servo motors for the mast and rudder that also have encoders to measure their angle. This would allow direct observation of these two states, improving localization, and the encoders could be used to improve the actuation accuracy of the servos.

Further, instead of trying to approximately model the kinematics of the boat, it would be better to attempt to fully model the boat's dynamics. This would likely increase the number of states in the model from six to ten, which would also increase the time and space complexity of the control algorithms, especially the MPC. Performance degradation might be an issue in this case, necessitating further optimization. The added states would be the rate of change of the position, angle of the boat, and relative wind angle. Shown below is a sample of what the state transition model of the dynamics could look like.

$$f(q,u) = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\gamma} \\ \ddot{x} \\ \ddot{y} \\ \ddot{\theta} \\ \dot{\gamma} \\ \dot{\phi} \\ \dot{\eta} \end{bmatrix} = \begin{bmatrix} \int \ddot{x}dt \\ \int \ddot{y}dt \\ \int \ddot{\theta}dt \\ -\dot{\theta} \\ \frac{1}{m}\left(F_{thrust} - F_{drag}\right)\cos(\theta) \\ \frac{1}{m}\left(F_{thrust} - F_{drag}\right)\sin(\theta) \\ \frac{1}{I}\left(\tau_{sail} + \tau_{rudder} - \tau_{drag}\right) \\ -\ddot{\theta} \\ \omega \\ \sigma \end{bmatrix} \tag{18}$$

Finally, perhaps the most important improvement would be to improve the aerodynamics of the boat and sail. Making a boat and sail that are aerodynamically sound and able to sail properly is very challenging and involves understanding and modelling the fluid dynamics of the air and water, the forces on the boat as it moves, and how these impacts sailing performance. This was the biggest challenge faced when constructing the boat and would likely still be the greatest challenge moving forward.

References

[1] Y. C. C. Team, “Maritime shipping causes more greenhouse gases than airlines,” Yale Climate Connections. Accessed: Apr. 03, 2025. [Online]. Available: yaleclimateconnections.org/2021/08/maritime-shipping-causes-more-greenhouse-gases-than-airlines/

[2] W. Steavenson, “‘It’s a little bit of utopia’: the dream of replacing container ships with sailing boats,” *The Guardian*, Jul. 14, 2022. Accessed: Apr. 03, 2025. [Online]. Available: https://www.theguardian.com/world/2022/jul/14/replacing-container-ships-with-sailing-boats-cargo-shipping-wind-power

[3] “The AI revolution in transportation,” ITS International. Accessed: Apr. 03, 2025. [Online]. Available: https://www.itsinternational.com/feature/ai-revolution-transportation

[4] J. Marshall, *Fundamentals of Autonomous Ground Vehicle Navigation & Control with Examples in Python, v0.5.5*. Kingston, Ontario, Canada, 2024.

[5] Thingiverse.com, “Livo 3 by lucaeale,” Thingiverse. Accessed: Apr. 03, 2025. [Online]. Available: https://www.thingiverse.com/thing:918471

[6] “Servo Motor Micro SG92R,” ProtoSupplies. Accessed: Apr. 03, 2025. [Online]. Available: https://protosupplies.com/product/servo-motor-micro-sg92r/

[7] “Pico-series Microcontrollers.” Accessed: Apr. 03, 2025. [Online]. Available: https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html

[8] “Seeed Studio XIAO ESP32C3 - RISC-V tiny MCU board.” Accessed: Apr. 03, 2025. [Online]. Available: https://www.seeedstudio.com/Seeed-XIAO-ESP32C3-p-5431.html

[9] “DS-000189-ICM-20948-v1.3.pdf.” Accessed: Apr. 03, 2025. [Online]. Available: https://cdn.sparkfun.com/assets/7/f/e/c/d/DS-000189-ICM-20948-v1.3.pdf

[10] “4654,” DigiKey Electronics. Accessed: Apr. 03, 2025. [Online]. Available: https://www.digikey.ca/en/products/detail/adafruit-industries-llc/4654/12697636

[11] “EEMB 3.7V Lipo Lithium Polymer Ion Battery.” Accessed: Apr. 03, 2025. [Online]. Available: https://www.amazon.ca/dp/B095VVWTSH?ref=ppx_yo2ov_dt_b_fed_asin_title

[12] “MCP73831T,” DigiKey. Accessed: Apr. 03, 2025. [Online]. Available: digikey.ca/en/products/detail/microchip-technology/MCP73831T-2ACI-OT/964301

[13] “AS5048-DS000298.pdf.” Accessed: Apr. 03, 2025. [Online]. Available: https://look.ams-osram.com/m/287d7ad97d1ca22e/original/AS5048-DS000298.pdf

[14] “Git.” Accessed: Apr. 03, 2025. [Online]. Available: https://git-scm.com/

[15] AlexMacQuarrie, *AlexMacQuarrie/yachtonomous*. (Mar. 25, 2025). Python. Accessed: Apr. 03, 2025. [Online]. Available: https://github.com/AlexMacQuarrie/yachtonomous

[16] “Thonny.” Accessed: Apr. 03, 2025. [Online]. Available: https://thonny.org/

[17] “VS Code.” Accessed: Apr. 03, 2025. [Online]. Available: https://code.visualstudio.com/

[18] “Software.” Accessed: Apr. 03, 2025. [Online]. Available: https://www.arduino.cc/en/software

[19] “Mermaid.” Accessed: Apr. 03, 2025. [Online]. Available: https://mermaid.js.org/

[20] “KiCad EDA.” Accessed: Apr. 03, 2025. [Online]. Available: https://www.kicad.org/

[21] “SOLIDWORKS.” Accessed: Apr. 03, 2025. [Online]. Available: https://www.solidworks.com/

[22] “Python,” Python.org. Accessed: Apr. 03, 2025. [Online]. Available: https://www.python.org/

[23] “MicroPython.” Accessed: Apr. 03, 2025. [Online]. Available: http://micropython.org/

[24] Z. Vörös, *v923z/micropython-ulab*. (Apr. 02, 2025). C. Accessed: Apr. 03, 2025. [Online]. Available: https://github.com/v923z/micropython-ulab

[25] “Holt ILCA 6 / Laser Radial Replica Mainsail.” Accessed: Apr. 03, 2025. [Online]. Available: https://www.coastwatersports.com/holt-ilca-laser-radial-replica-mainsail-p-8289.html

Appendix A – Contribution Table

Shown below in Table 11 is the contribution table for the project. All team members put in a maximal and equal effort over the course of the project.

Table 11: Project Contribution Table.

Name	Overall Effort Expended [%]
Liam Herbert	100%
Alex MacQuarrie	100%
Yasin Shahrami	100%

Appendix B – Rudder Linkage System Components

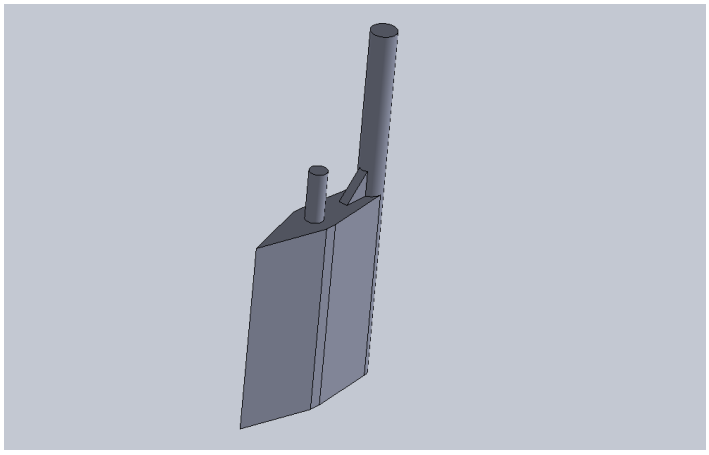

Figure 54: Rudder

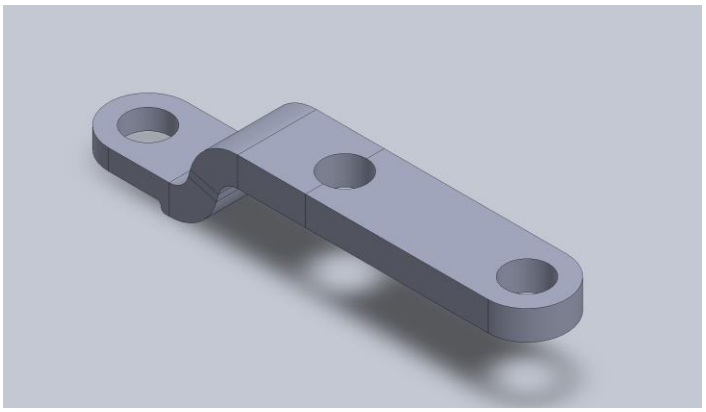

Figure 55: Link Rod

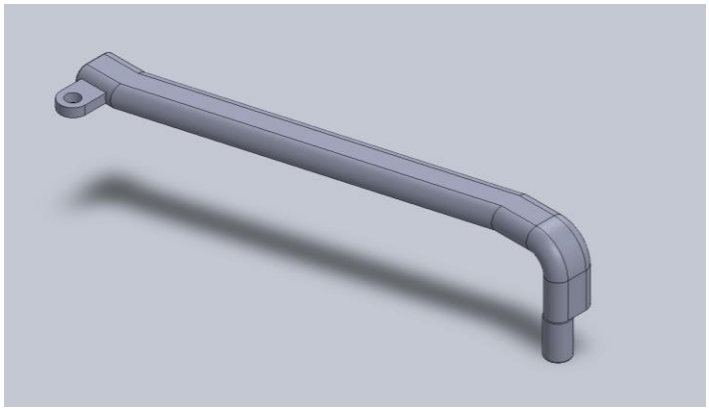

Figure 56: Rudder Arm

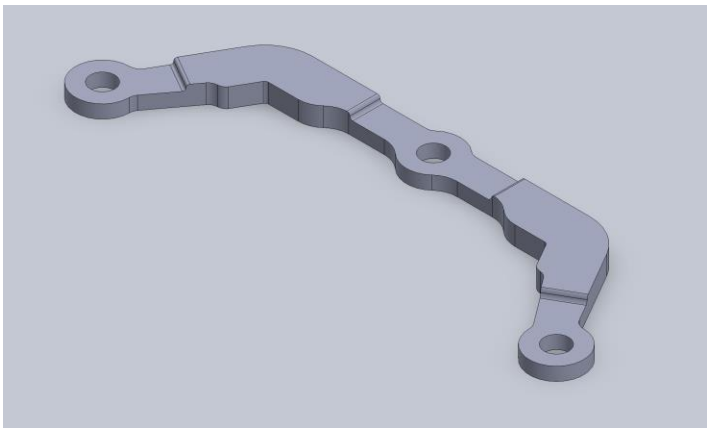

Figure 57: Control Arm

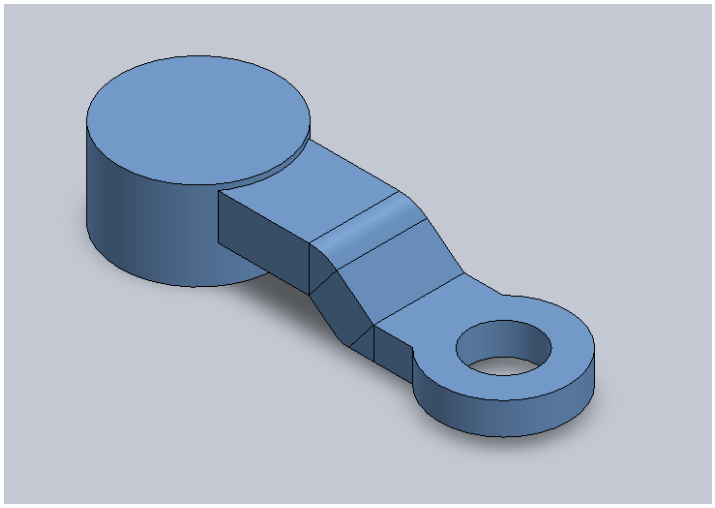

Figure 58: Rudder Servo Horn

Appendix C – Wind Vane Components

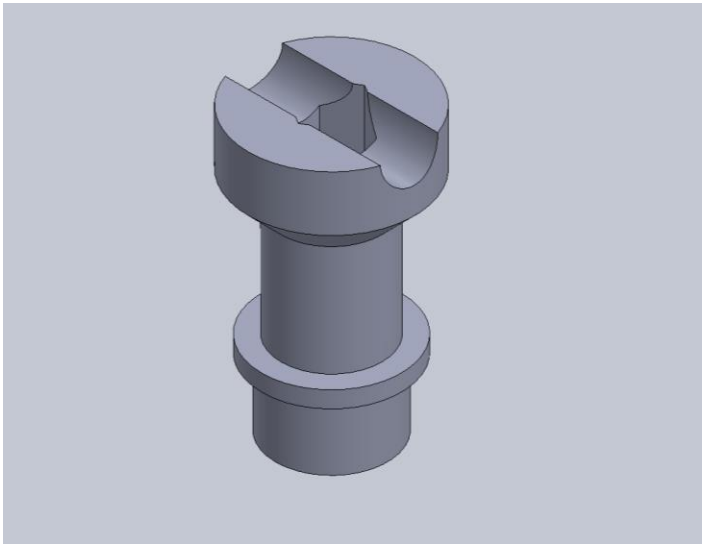

Figure 59: Wind Vane Mount

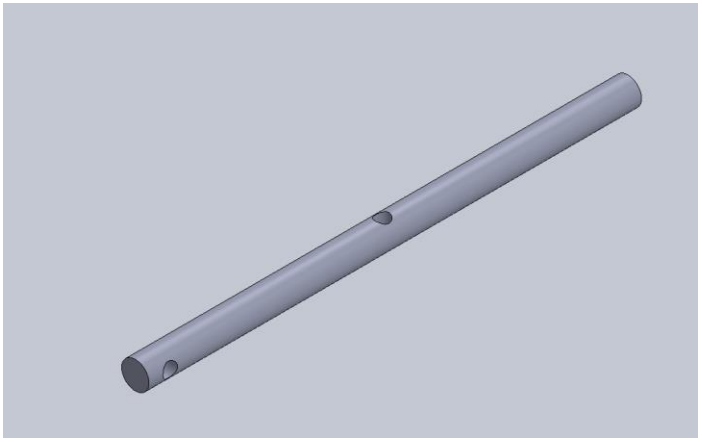

Figure 60: Wind Vane Shaft

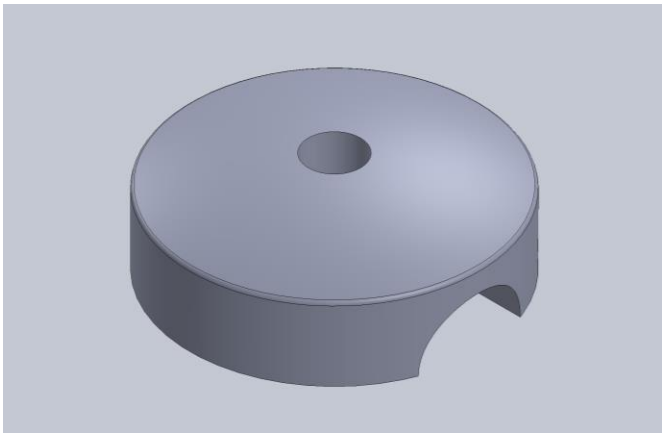

Figure 61: Wind Vane Mount Cap

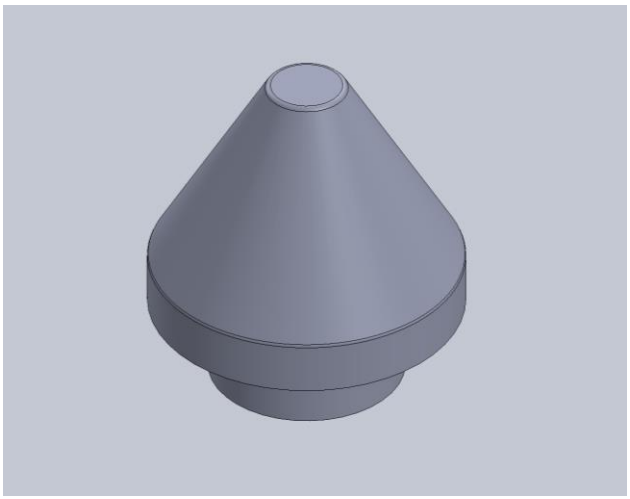

Figure 62: Wind Vane Head Cap

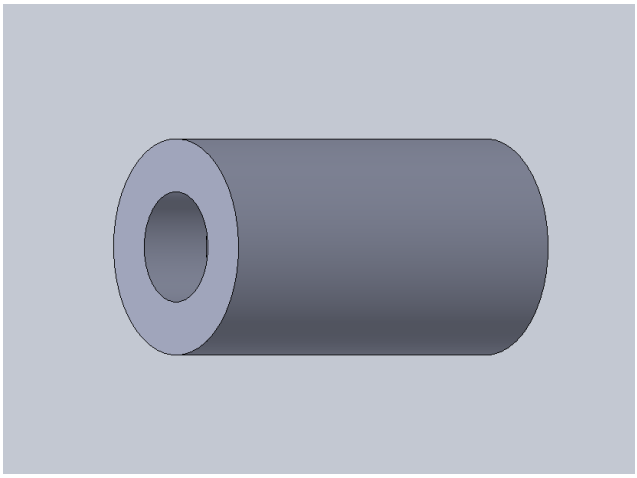

Figure 63: Wind Vane Head

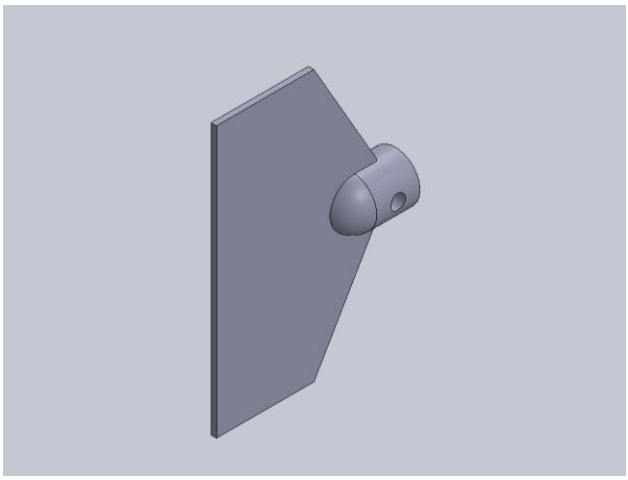

Figure 64: Wind Vane Tail

Appendix D – Additional 3D Components

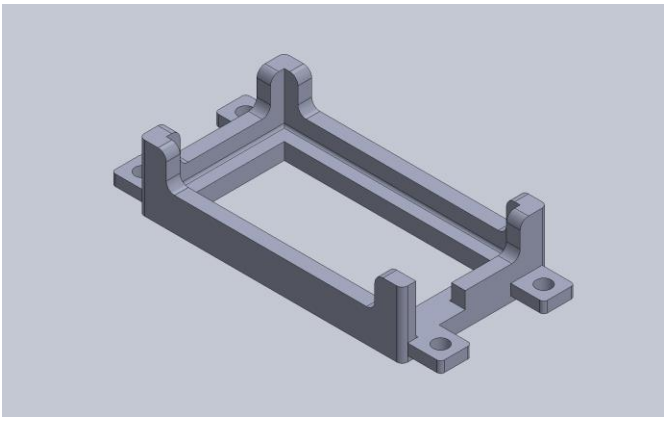

Figure 65: Battery Holder

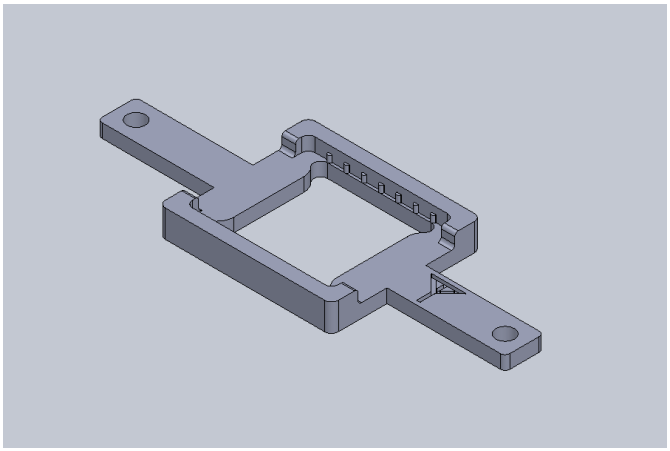

Figure 66: ESP32 Holder

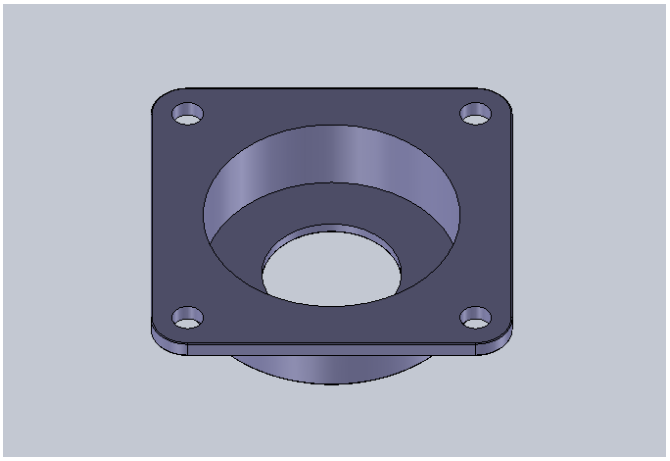

Figure 67: Bearing Flange Mount

Appendix E – Full Assembly Exploded View

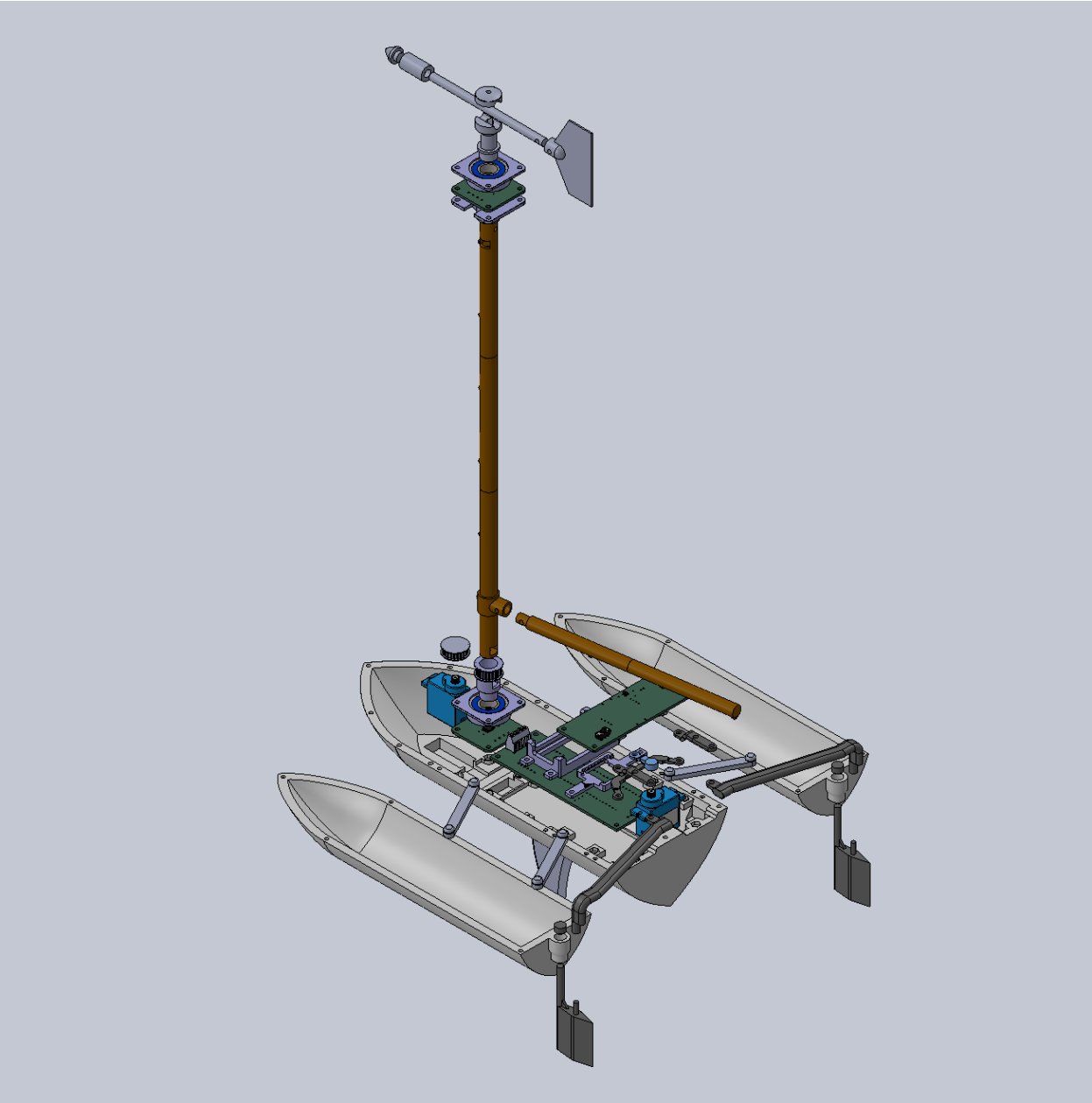

Figure 68: Exploded View of Entire Assembly

Appendix F – Simulation Sequence Diagram

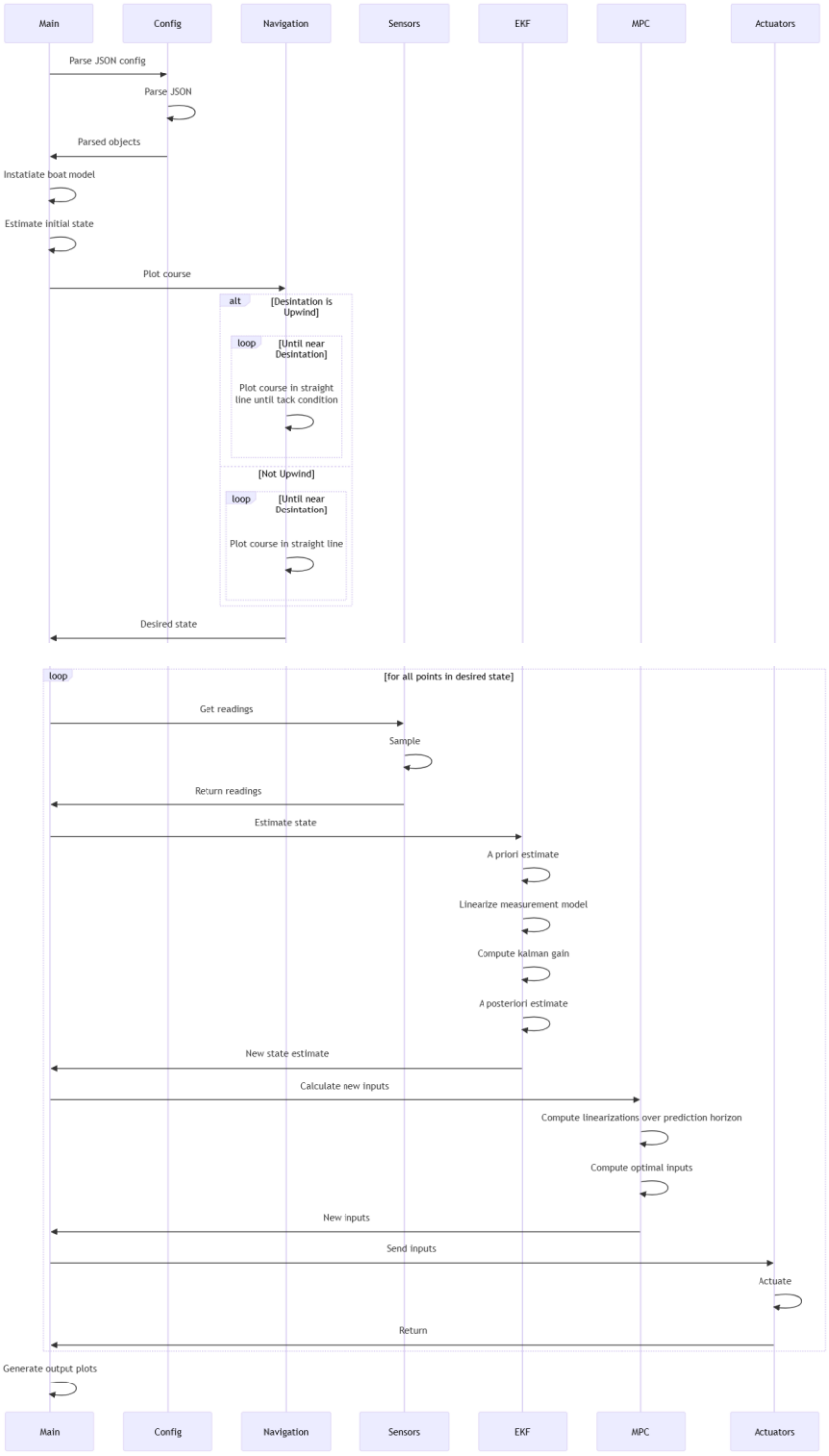

Figure 69: Simulation sequence diagram

Appendix G – Cost Breakdown Table

Table 12: Cost breakdown of all purchases made.

	Quantity	Cost	
AliExpress			
Micro USB Ribbon Cable	2	$ 1.84	
USB Horizontal Connector	2	$ 3.50	
USB Right Angle Connector	2	$ 3.50	
Shipping		$ 10.13	
Tax (13%)		$ 2.47	
Total			$ 21.44
Amazon			
3.7V 700mAh Lipo Battery	1	$ 19.99	
Garden Hose	1	$ 22.46	
Garden Hose Sink Attachment	1	$ 12.99	
Micro USB to Micro USB Cable	1	$ 9.99	
Intex Double Quick III Hand Pump	1	$ 14.99	
Tax (13%)		$ 10.45	
Total			$ 90.87
Canadian Tire			
Flexseal	1	$ 11.99	
Sandpaper	1	$ 5.99	
Inflatable Test Pool	1	$ 62.93	
Tax (13%)		$ 10.52	
Total			$ 91.43
Digikey			
Cylindrical Magnet	3	$ 1.47	
PCB Connectors (2 Pos, Top Entry)	2	$ 3.34	
PCB Connectors (2 Pos, Side Entry)	2	$ 2.42	
PCB Connectors (4 Pos, Side Entry)	1	$ 4.64	
PCB Connectors (5 Pos, Side Entry)	2	$ 13.94	
PCB Connectors (6 Pos, Side Entry)	1	$ 3.57	
Seeed Studio XIAO ESP32C3	6	$ 48.48	
SG92R Servo Motor	2	$ 18.08	
JST-02 Battery Connection Header	3	$ 0.45	
Adafruit Miniboost - 5V Boost Converter Board	2	$ 12.78	
Sparkfun ICM-20948 IMU Breakout Board	2	$ 56.21	
Raspberry Pi Pico W Microcontroller	2	$ 9.11	
Micro USB Port	4	$ 4.36	
M2 Hex Key	1	$ 1.67	
Zip Ties	20	$ 2.58	
Header Pins	1	$ 8.86	
Various Wire		$ 10.35	

Miscellaneous Hardware		$ 12.20	
Shipping		$ 24.00	
Tax (13%)		$ 31.01	
Total			$ 269.52
JLCPCB			
Main PCB (Version 1) Construction	5	$ 3.42	
Main PCB (Version 1) Assembly	2	$ 23.08	
Wind Vane PCB (Version 1) Construction	5	$ 1.71	
Wind Vane PCB (Version 1) Assembly	2	$ 31.80	
Power Management PCB (Version 1) Construction	5	$ 3.42	
Power Management PCB (Version 1) Assembly	2	$ 36.53	
Main PCB (Version 2) Construction	5	$ 5.71	
Power Management PCB (Version 2) Construction	5	$ 1.71	
Power Management PCB (Version 2) Assembly	2	$ 37.97	
Shipping		$ 71.11	
Duties		$ 49.31	
Tax (13%)		$ 28.14	
Total			$ 293.91
JLCMC			
Timing Belt	1	$ 0.80	
Various Sizes of Ball Bearings		$ 5.57	
Miscellaneous Hardware		$ 2.26	
Shipping		$ 42.47	
Tax (13%)		$ 6.64	
Total			$ 57.74
Total			**$ 824.91**