CUSail: Cornell Autonomous Sailboat Team
Navigation Team
Spring 2026 Report

Emith Uyanwatte, Electrical & Computer Engineering 2026, 1 credit
Nicole Luo, Computer Science 2027, 1 credit
Albert Sun, Electrical & Computer Engineering 2026, 0 credits
Sean Zhang, Computer Science 2026, 0 credits
Ludvig Fellstrom, Electrical & Computer Engineering 2027, 0 credits
Linnea Furlan, Electrical & Computer Engineering 2027, 1 credit
Wang Mak, Electrical & Computer Engineering 2027, 1 credit
Eric Cai, Computer Science 2028, 3 credits
Jonah Conolly, Electrical & Computer Engineering 2028, 3 credits
Liya Mei, Electrical & Computer Engineering 2028, 0 credits
Fiona Lin, Electrical & Computer Engineering 2029, 1 credit
Nicole Zhou, Electrical & Computer Engineering 2029, 1 credit
Kai Nizhner, Computer Science 2029, 1 credit
Angie Zhang, Computer Science 2029, 1 credit
Colin Park, Computer Science 2029, 0 credits

May 2, 2026

Abstract

TODO

Contents

1 Overall Design and Motivation

(Emith Uyanwatte)

1.1 Overview

TODO

2 Algorithm

(Kai Nizhner, Nicole Luo)

2.1 Overview

During the Spring 2026 semester, the autonomous sailing algorithm was modified in two ways. The first change was a large-scale shift from the Python heading-deciding algorithm of the 2024-2025 season, `Albo`, to the new C++ architecture, `JANK`. This effort involved a full transition to a different build system and revamping of the high-level logic of `Albo`, a long term project that will continue until competition and likely next semester. The second change involved re-addressing the logic that controls the physical state of the boat—most importantly setting the rudder, mainsail, and jib, the latter of which was an addition to the 2025-2026 season boat. These tasks were delegated almost entirely to the Teensy; therefore, this project mainly modified the Jetson code/ROS nodes used for Teensy communication, and code in the Teensy codebase.

See §2.4 Miscellaneous for information on the rest of the algorithm software not described above, as well as explanations of sailing phenomena and constraints to help contextualize the rest of this section.

2.2 The JANK Algorithm

2.2.1 Motivation

The motivation for the logic of the `JANK` heading-deciding algorithm stemmed from the logical limitations of the `Albo` algorithm. At competition at the end of the 2024-2025 season, the team noticed various undesirable behaviors that included poor tacking performance, frequent accidental jibes, and the boat getting stuck in strange loops where it would make little meaningful progress. The `JANK` algorithm attempts to solve this with an approach that is more true to real-world sailing.

The motivation for transitioning from the existing Python algorithm to a C++ architecture was the eventual goal of having the Teensy execute the algorithm independently. Currently, the Teensy sends sensor data to the Jetson, on which the various logical parts of the algorithm calculate and output rudder and sail angles for the boat; these angles are sent back to the Teensy to actually move the boat's rudder (and eventually sails). However, since the logic of the algorithm is not incredibly complex or resource-intensive, there is no real need for it to run as ROS nodes on the Jetson. It will eventually make more sense for the Teensy to handle all of this logic on its own. Turning the algorithm from a Python ROS node to a C++ ROS node thus sets up for this transition to happen soon: likely during the 2026-2027 season.

2.2.2 Conceptual Shift: Vector-Based Navigation

The fundamental difference between the sailing logic of the `JANK` heading-deciding algorithm and the `Albo` heading-deciding algorithm is `JANK`'s preference for vector-based navigation over waypoint-targeted navigation when sailing upwind. The `JANK` algorithm chooses an optimal side of the no-go zone based on where the final destination is, and, while staying on this side of the wind, dynamically changes the boat's heading to keep pointing as far upwind as possible. This approach, and some of its other new conceptual details, are summarized by the following high-level "state machine":

- **Main Upwind Sailing State:** sail as close to the wind as possible while monitoring which side of the wind is optimal for reaching the destination sooner.
- **Transition Condition:** if the boat is no longer on the optimal side of the wind, evaluate its velocity and choose a way to cross over to the other side—if sufficient speed exists, execute a tack; otherwise, default to a jibe.
- **Tack Execution:** apply maximum rudder until the boat has crossed the midpoint of the no-go zone; gradually return to center until the tack is completed (if the tack fails for any reason, reset and return to the main state).
- **Post-Tack Behavior:** attempting a tack inherently results in a loss of forward velocity—after exiting a tack, failed or successful, the velocity condition described above prevents immediate oscillations or repeated failed tacks.

Note that when the boat does not need to sail upwind to reach its destination (in other words, if the destination is not in the no-go zone), none of the above applies—it can simply point at and sail directly to the destination.

One thing missing from the above state summary is an explanation of how we decide which side of the wind is "optimal" for reaching our destination quickly. In principle, it makes sense to quantify this decision based on where the destination is exactly in the no-go zone: if the destination is heavily skewed to one side, the optimal side for the boat to be on is clearly just that side. Consider the following diagram, where the dashed gray lines represent the no-go zone, in which sailing on a *port tack* (the current orientation, where the wind passes over the port side of the boat first) is clearly favorable.

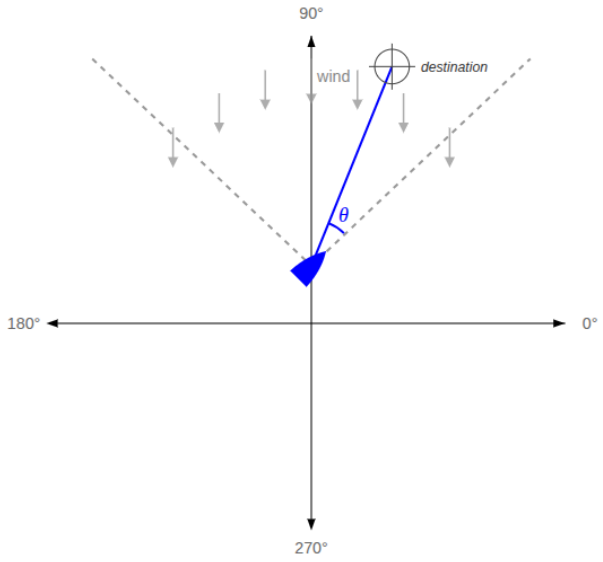

Figure 1: Clear favored side of the wind based on `heading_difference` angle θ.

However, consider an alternate scenario where the destination is directly or almost directly upwind, and we are sailing over longer distances. In this case, it will take a long time for the boat to move along a set course before the angle θ to the destination changes sufficiently to produce a clear new optimal side of the wind. Our solution is to project a bounding box over our coordinate space, as shown below, with the condition that the boat must tack once it reaches one of its sides. These forced periodic tacks ensure that a reasonable course toward the destination is maintained.

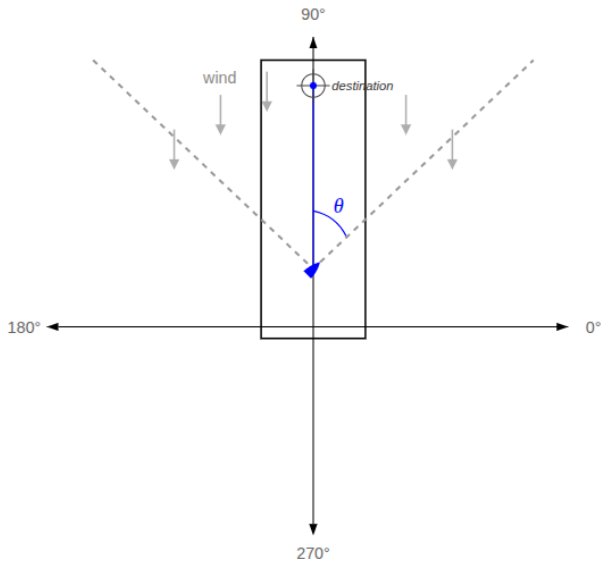

Figure 2: Unclear favored side of the wind (symmetry and large distances); default to bounding box.

2.2.3 C++ Environment

As described above, the `JANK` algorithm brings a development shift: it is a ROS node written in C++. To incorporate it cleanly into the `Sailbot` monorepo, and set the stage for future code in `Sailbot` to be written in C++, we created a new module entitled `cpp_files/` directly under the `src/` directory. As of the end of the 2025-2026 season, `JANK` and relevant supporting files alone comprise the module. This module fits into our existing Docker setup simply—the CUSail Docker container is already set up with a C++ compiler and CMake tools, and includes all the required ROS C++ libraries.

2.2.4 Future Goals

At the time of the completion of this report, the theoretical foundation for the new `JANK` framework had been laid out in its entirety, but the actual code of the `JANK` algorithm remained incomplete. Work on this front will continue at least until competition in June, with the goal of it being deployed there.

As described in §2.2.1 Motivation, one of the most immediate next steps towards making the algorithm software more sustainable and lightweight in the long run is ensuring that it run entirely on the Teensy. Changing `JANK` from being a ROS node to simply a regular C++ class in the Teensy codebase, and getting sensor values locally from the Teensy rather than from ROS subscriptions, is the most pressing change that can be made for the algorithm in the 2026-2027 season.

2.3 Physical State: Teensy Codebase

The Teensy codebase serves as the link between the various parts of the navigation algorithm and the physical parameters of the boat—the Teensy is solely responsible for converting goal rudder and sail angles into PWM values that correspond to physical turns of the boat's servos. This year, beyond the regular tailoring of the servos and constants to match the physical constraints of a new boat, we added functionality to support a jib and improve sail setting.

2.3.1 Jib Support

The boat for the 2025-2026 season featured a jib as well as a mainsail, which the previous Teensy codebase did not account for. From a mechanical standpoint, this jib was set up to be controlled by two separate servos: one on the port side of the boat, and one on the starboard side of the boat. To allow the Teensy to actuate the jib servos properly, we added fields into the packets sent to the Teensy from the Jetson—on top of sending the goal rudder angle and mainsail angle, the Jetson now also sends a goal jib angle and a flag that represents what side of the boat the jib should be set on.

To prevent the two servos from mechanically conflicting and putting strain on each other or the jib sheets, we established the convention of always releasing the opposite servo entirely before actuating the servo on the side of the boat where the sail was to be set. At the architecture level, the work of actuating the two jib servos based on values extracted from packets was integrated into the refactored Teensy control loop, where monitoring and control are separated into task-like components: sensor monitoring, serial packet parsing, servo actuation, and telemetry transmission. This organization made it easier to add jib-specific behavior without collapsing the monitoring and actuation logic into a single loop.

2.3.2 Sail Angle-to-PWM Mapping

This semester, sail actuation in the Teensy codebase was reworked to better match the mechanical geometry of the boat, rather than relying solely on linear angle-to-PWM maps. This change was based on the fact that letting out the sheet a constant amount when the sail is set at a small angle will result in a greater angular displacement than if the sail was first set at a large angle. A linear map will thus overshoot small target angles and/or undershoot large target angles.

Our fix was to implement a law-of-cosines-based angle-to-PWM mapping. This function produces a much stronger estimate of the required length of the sheet needed to achieve a target sail angle; the sheet displacement calculated using the law of cosines is then converted into servo rotation and PWM output. Both the mainsail and jib use this new mapping.

2.3.3 Code Overhaul and Conventions

Beyond the immediate feature additions described above, a major goal this semester was to make the Teensy codebase maintainable for future members and future architecture changes. The firmware was better reorganized into preexisting modules (monitors, control tasks, shared state, and constants) to reduce coupling, cement the abstraction barrier between and explicit responsibilities of each module, and make the data flow easier to reason about during debugging.

Every line of the codebase was reassessed and made more explicit, from casting conventions to increased documentation: start and end flags, payload ordering, timeout behavior, and telemetry cadence are now explicitly defined in one place. Combined with packet timeout/drop handling and dropped-packet telemetry, we hope future developers on the team will now have a concrete starting point for validating communication reliability before changing higher-level control logic.

2.3.4 Corresponding Jetson-side Changes

Naturally, all the new servo-actuation logic to handle the jib in the Teensy codebase is incomplete without the sail-trim part of the algorithm on the Jetson calculating and transmitting goal jib angles. As a result, the last part of this project took part in the Jetson codebase—Jetson side serial protocols were updated to match the new format the Teensy was expecting, the `trim_sail.py` ROS node logic was redone to be clearer and map the jib similarly to the mainsail, and subscriber/publisher plumbing was established to make the new jib parameters accessible to all relevant nodes.

Like with the Teensy, we also took this as an opportunity to debug and better modularize existing code on the Jetson, and generally bring the codebase up to date (documentation, terminology conventions, style). One particularly important change was the factoring out of "magic numbers" and adding a global `constants.py` file in the same way that we have a `constants.hpp` file on the Teensy. Much of the code for the Jetson relies on implicit conventions and seemingly random harcoded values; we hope this addition will make such decisions more explicit and easier to reason about in the future.

2.4 Miscellaneous

This section describes the other notable choices and conventions in the current CUSail algorithm software.

2.4.1 Coordinate System

To simplify calculations, we utilize UTM coordinates for our algorithm. UTM coordinates are a system for pinpointing locations on Earth using a flat grid, where each location is represented by an easting (distance from the central meridian) and a northing (distance from the equator), measured in meters. This system is based on the Universal Transverse Mercator projection, which divides the world into 60 zones, each 6 degrees of longitude wide.

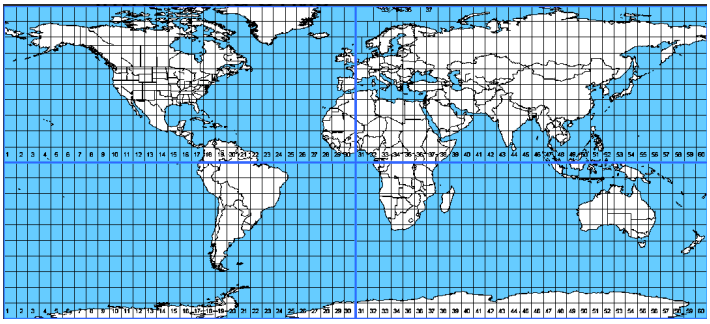

Figure 3: The world divided into UTM zones. [1]

Converting GPS coordinates into UTM allows us to assume a flat Earth model and treat all positions as Euclidean points, enabling direct computation of distances and angles. To handle rotation, we assume that East is represented by 0° and counterclockwise rotation is positive (meaning North is 90°, West is 180°, and South is 270°).

2.4.2 Black-Box Algorithm Structure

The totality of the navigation algorithm accepts four primary inputs: wind speed, wind angle, boat UTM position, and destination waypoint UTM position. The combined outputs of all the different parts of the navigation algorithm are the goal sail angles (`trim_sail` node) and goal rudder angle (`JANK` node). Operational constraints determined the feasible ranges for each of these values on the 2025-2026 season boat:

- Mainsail Angle: 0° to 90° on either side of the boat.
- Jib Angle: 0° to 80° on either side of the boat.
- Rudder Angle: -45° to 45°, where 0° represents being centered.

2.4.3 Wind-Based Mapping for Sail Angles

The angles of the mainsail and the jib are determined solely based on the angle of the wind relative to the boat. For every sailboat, there exists an angle ϕ_1 with respect to the wind such that boats can only generate lift at angles between ϕ_1 and 180° on either side of the wind. This interval $[\phi_1, 180^\circ]$ is therefore mapped linearly to sail angles (for the mainsail, for example, $\phi_1 \to 0^\circ$ and $180^\circ \to 90^\circ$). The jib and mainsail naturally move in parallel, within their respective ranges.

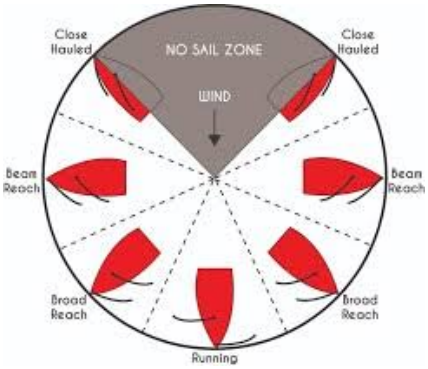

Figure 4: Points of sail diagram displaying where sailboats can generate lift and how sail angles change. [2]

Issues with accidental jibes under downwind conditions motivated discussion of shortening the domain at its upper limit (namely, a domain of $[\phi_1, 180° - \phi_2]$ where $\phi_2 > 0°$) to avoid hazardous areas, colloquially called the "danger zone".

2.4.4 Rudder Logic and Tacking Behavior

Rudder decisions govern when the boat attempts to tack and how it maintains heading. A tack is considered successful if the boat maintains sufficient speed to cross the wind. Otherwise, if the boat gets temporarily "stuck" in the no-go zone with no rotational velocity, the rudder returns to neutral, allowing environmental forces to rotate the boat out of stall.

To evaluate whether a tack is progressing correctly, the `JANK` algorithm monitors the rate of change of the wind angle relative to the boat's heading, using onboard rotational data as a stability check.

2.5 For Future Teams

The navigation algorithm is relatively self-sustaining at this point, and any future CUSail boats will fare well if they are launched with the existing software. However, there are naturally still some small improvements to consider:

- See §2.2.1 Motivation and §2.2.4 Future Goals for next steps to evolve the current `JANK` architecture.
- As addendums to §2.4.4 Rudder Logic and Tacking Behavior, there is some rudder functionality that is not yet implemented, but is definitely worth considering:
  - The rudder can technically be used to influence the outcome of a failed tack. For example, if the boat is in the no-go zone with no rotational velocity, it is feasible to use the rudder as a paddle to help push the boat out on the desirable side of the no-go zone, by pushing it aggressively in one direction and slowly bringing it back.
  - When sailing upwind in high wind speeds, the sails generate large amounts of sideways lift that cause the boat to tilt over, a phenomenon called "heeling". On the boat, this heeling should be feasible to measure using *roll* data. The sideways lift generated tends to push the boat sideways, rather than the desirable outcome of the boat moving straight forward. This can be counteracted by having the rudder offset at some angle, rather than being perfectly centered, in order to add a balancing lift force that results in the boat moving straight forward.
- As addendums to §2.4.3 Wind-Based Mapping for Sail Angles, there is some sail functionality that is not yet implemented, but is definitely worth considering:
  - Our current convention, which begins in the `AnemometerMonitor` in the Teensy codebase but affects all ROS nodes that subscribe to wind direction data, is that a value of 180° represents the boat pointing directly into the wind (directly in the no-go zone), and a value of 0° represents the boat pointing fully downwind. This is an outdated and counterintuitive convention; it would be worth changing this so 0° represents being directly upwind and 180° represents being directly downwind. When this change is implemented, it would also be worth changing all ROS nodes that are hard-coding for this old convention to simply use the `constants.py` file, so that future convention changes like this are very simple to make, limited to flipping some centralized values.
  - As mentioned above, the boat is not currently allowed to sail directly downwind, due to the danger zone implementation. However, if the danger zone is ever minimized or removed, it would make sense to implement *wing-on-wing* sailing functionality for when the boat is going straight downwind, on a "run". This refers to setting the jib all the way out on the *opposite* side of the main so that it can catch far more wind; such a feat is possible at the software level due to the `jib_side_flag` parameter that is passed from the Jetson to the Teensy.

  - In the `JANK` heading-deciding algorithm, wind speed data is not used for anything besides the tacking speed condition. On heavy-wind days, it may be worth using wind speed as a factor to influence sail trim. When the wind is above a certain threshold, having "proper" sail trim and generating too much sideways lift can cause the boat to *heel over* impractically far, making steering erratic and harder to control. A simple way to minimize this is to let the sail out slightly more then necessary, sacrificing extra power for stability and balancing out the boat — at the software level, add a linear sail displacement once the wind speed exceeds a certain threshold.

- As addendums to to §2.3.4 Corresponding Jetson-side Changes, there are some non-essential changes to conventions that could be established on the Jetson side to greatly improve clarity:
  - ROS subscribers and publishers for mainsail values follow an outdated naming convention, dating back to when we had only one sail: topics refer to simply "sail" instead of "mainsail". For the most part, local variables have been updated to reflect that a sail can either can be a "mainsail" or a "jib"; this movement should also be standardized to all the ROS plumbing.
  - We have noticed that debugging logs for ROS nodes are overwhelming due to the sheer number of messages being logged. This phenomenon makes it quite hard to make sense of a timeline through any logging. It would make sense to have some timer that controls how often a certain message gets logged—if every message only gets logged once every five seconds, for example, it might make analyzing the logs much more manageable.
  - On the Jetson side of communications between the Jetson and Teensy—the `teensy.py` file, for example—there are many defensive checks before any values are sent. While a good practice on the surface, many such checks are redundant and just make code harder to read through. Currently, the final line of defense in the Teensy is strong on its own—servos will not be set unless parameters sent from the Jetson are properly defined. As a result, it may be worth getting rid of unnecessary checks on the Jetson side (perhaps replacing them with more explicit method preconditions), as long as everything essential is checked once at the right time.

As mentioned in §2.3 Physical State: Teensy Codebase, it is worth noting that some tuning to a new boat is inevitable every year. With regard to the Teensy codebase, all important physical constants are contained in one `constants.hpp` file; the `constants.py` file does the same thing on the Jetson side. With regard to the heading-deciding algorithm `JANK`, notable values that must be tuned include (but are not limited to) the no-go zone angle ϕ_1, danger zone angle ϕ_2, and the wind-speed based transition condition. Both communication with the Mechanical subteam and testing in the field are required to finalize these parameters.

Finally, the most long-term algorithm project is an integration with Sailbench (see §10 Sailbench for more information). In theory, a well trained RL model could handle autonomous navigation far better than any hard-coded algorithm, taking far more variables into account and weighing them much more precisely. This will naturally be much more resource-intensive and would therefore run on the Jetson. Having the option to run either the hard-coded algorithm on the Teensy or an RL algorithm on the Jetson would be the best possible version of the CUSail autonomous navigation software.

3 Computer Vision

(Eric Cai)

3.1 Overview

Development continued on the YOLO-based object detection system. A major improvement of this system is that it has nearly zero false positives at close range. However, the previous search algorithm did not take full advantage of this because when it does not direct the boat to return to its search pattern if a buoy is no longer detected. Therefore, a goal for this semester was to engineer a search algorithm that could correct itself.

3.2 Search Behavior

The boat begins the search event by following a zig-zag search pattern perpendicular to the direction of wind. This avoids the need for upwind sailing. If a buoy is detected, the boat will set a waypoint at the detected location and sail closer until it reaches the waypoint, and thus, the buoy. This behavior is similar to the previous particle filtering algorithm, where the boat keeps track of a collection of particles that hold the probability that a buoy is at the detected location. The new search algorithm adds an additional feature: if the detected buoy location is in the camera's field of view but the buoy is no longer detected, the algorithm will lower the probability of the buoy being at that location. If the probability drops below the detection threshold, the boat will return to its search pattern. The boat will remember the decreased probabilities (i.e., the boat remembers where the buoy isn't, instead of just trying to determine where the buoy is).

3.3 Future Goals

The design of the new search algorithm has been determined. However, significant testing is still required to determine how well the YOLO object detection system works in the wild. These tests will have major implications on the exact implementation of the new search algorithm. First, the YOLO object detection requires the setting of a confidence threshold hyperparameter that determines when to count a single buoy detection. In preliminary testing, a value of about 0.6 was found to balance precision and recall, but this may change on water. The exact rules for scaling probabilities must also be tuned to the new object detection system. Currently, the proposed approach is to weigh detections (or lack of detections) close to the boat very heavily, but the exact relationship between distance and confidence will be best determined empirically.

4 Webserver Changes

(Colin Park, Nicole Luo)

4.1 Overview

There have been two related tracks of webserver work this semester: a dedicated *read-only* telemetry and map viewer for observers and judges, and a longer-term migration from the legacy HTML/JavaScript ground station to a MERN (MongoDB, Express, React, Node.js) architecture.

The read-only viewer shipped as its own entry point (`viewer.html` and supporting modules) that connects through rosbridge for subscriptions and read-only service calls, but omits command publishers and other write paths. It is meant to be usable *today* alongside the existing operator-facing dashboard, without giving monitors a way to change boat behavior.

The MERN migration replaces that legacy stack incrementally: a Node.js backend receives ROS 2 traffic (typically via rosbridge on the host), exposes APIs, and persists mission-relevant data in MongoDB while the React frontend is still rolling out. During this transition, the static read-only pages and the newer backend may both appear in the deployment workflow depending on the milestone; what does not change is that *live* autonomy, sensing, and control authority remain on the ROS 2 stack running on the vehicle. The subsections below describe each track in more detail.

4.2 Read-Only Webserver Viewer

During the Spring 2026 semester, we developed a dedicated read-only webserver viewer to separate *observation* from *control*. The primary dashboard is intended for operators and includes write-capable actions (command publishes and runtime parameter updates), but many testing and competition scenarios require telemetry access without command authority. In particular, judges, on-shore observers, and teammates inspecting algorithm behavior should be able to monitor state without any path to changing boat behavior. The read-only viewer addresses this requirement by exposing live status and map context while removing write paths from the interface.

4.2.1 Architectural Approach

Instead of introducing a runtime toggle inside the existing control page, we implemented a distinct frontend entry point (`viewer.html` + `viewerMain.js`). This was done to keep the read-only path isolated from control logic and to reduce the chance of accidental coupling with write-capable handlers. We also introduced a dedicated ROS bridge module (`rosConnectionViewer.js`) that initializes topic subscriptions and read-only service calls only. In contrast to the full control module, the viewer path does not initialize publishers for sail/rudder command topics and does not register UI handlers that mutate mode or tuning parameters.

4.2.2 Telemetry and Map Instrumentation

The viewer subscribes to core runtime telemetry used by both operators and algorithm developers: control/event mode, commanded sail and rudder values, radio overrides, wind angle, heading, GPS position, dropped packet count, and algorithm debug fields (e.g., tacking state, heading direction, destination distance, and no-go/neutral zone values). These streams are throttled at the rosbridge subscription layer and rendered directly into collapsible dashboard panels.

Map instrumentation was designed to provide fast situational awareness. The page renders:

- the live boat position marker from GPS,
- heading orientation derived from IMU yaw,
- the current algorithm destination marker, and

- a short rolling trajectory trail (approximately one minute) to visualize recent motion.

This combination makes it easier to diagnose behaviors such as oscillation near no-go boundaries, failed tacks, or delayed destination updates without needing log parsing.

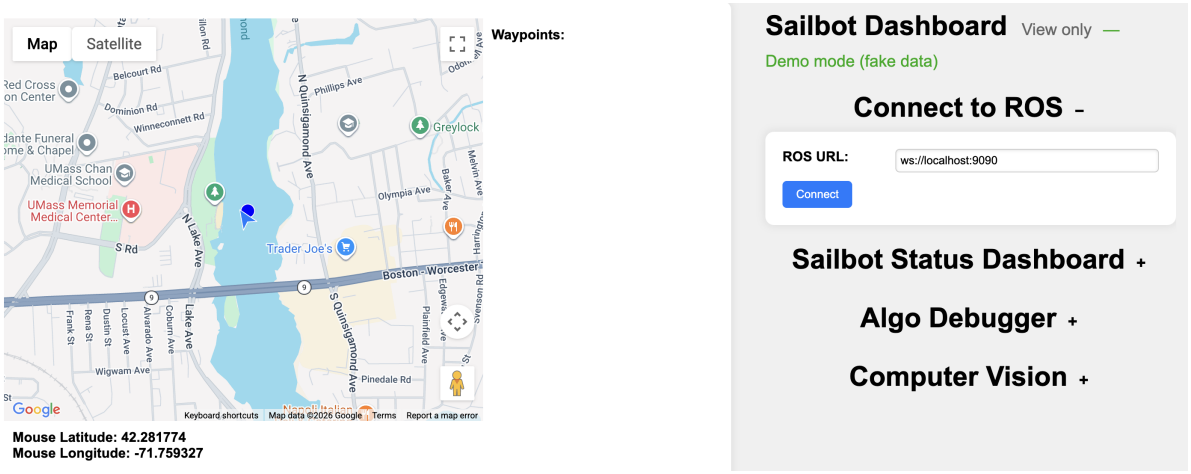

Figure 5: Read-only webserver viewer showing live telemetry panels and map-based situational context.

4.2.3 Read-Only Waypoint Visibility

A key requirement was preserving waypoint context while preventing edits. To do this, we reused the existing waypoint manager with a `readOnly` option. In this mode, the frontend still synchronizes queue state from backend waypoint services ("get" semantics) and renders waypoints/paths, but disables mutation interactions (adding, deleting, and drag-reordering). Reusing this module avoided duplicating map/queue logic while preserving a clear safety boundary at the UI interaction layer.

4.2.4 Demo Mode and Validation Workflow

We added a standalone demo path (`viewer.html?demo=1`) that injects synthetic telemetry at a fixed interval. This supports repeatable validation of layout, rendering, and map updates without requiring rosbridge availability, hardware-in-the-loop setup, or lake conditions. In practice, this reduced iteration time when tuning panel organization, validating marker/trail behavior, and debugging conditional rendering between algorithm and RC-related status rows.

4.2.5 Impact and Next Steps

The read-only viewer makes lake tests more efficient and more useful by giving team members a stable telemetry interface that can be shared broadly without exposing control actions. During on-water runs, this improves situational awareness by centralizing live boat status (position, heading, control mode, actuator values, and algorithm debug outputs) in a single monitoring surface, making it easier to detect anomalous behavior quickly and coordinate test observations across the team.

The next steps are to thoroughly test the viewer across representative lake-test scenarios and integrate it cleanly with the rest of the codebase and deployment workflow. This includes validating behavior under real rosbridge connectivity conditions, confirming compatibility with current topic/service definitions, and ensuring the read-only path remains consistent as the main webserver and navigation stack continue to evolve.

4.3 MERN Webserver Migration

We migrated the Sailbot webserver from a legacy HTML/JavaScript implementation to a full MERN architecture. The goal of this transition is to replace our current "stateless" system, which relies on direct DOM manipulation and global variables, with a scalable, persistent framework. By introducing a Node.js backend and a MongoDB database, we have established a robust data flow (ROS → Node.js → MongoDB → React) that ensures telemetry and mission waypoints are stored and maintained across sessions, providing a more reliable and maintainable ground station for both developers and operators.

4.3.1 Backend Architecture

The backend was redesigned to serve as the "source of truth" for the boat's state. We implemented a Node.js server using Express to handle API requests and Mongoose to interface with a MongoDB instance.

- Persistent State: Unlike previous versions where boat data lived only in frontend memory, the current system stores telemetry (GPS, heading, wind), buoy locations, and mission waypoints in MongoDB. This ensures that if the browser is refreshed or the connection is briefly canceled, the operator immediately regains the full context of the mission.
- ROS2 Integration: To facilitate communication between the host-based webserver and the containerized ROS 2 Humble navigation stack, we implemented a WebSocket bridge utilizing the rosbridge_suite. This integration maps the container's internal DDS traffic to the host on port 9090, allowing the Node.js backend to subscribe to high-frequency topics such as /fix and /imu.

4.3.2 Future Goals in Frontend Architecture

With the backend and persistence layers currently operational, we are now focusing on the deployment of the React-based frontend dashboard. We are already partially through the implementation of the real-time mapping system using Leaflet.js, which is designed to visualize the boat's rolling trajectory alongside the target buoys stored in MongoDB. The transition from static DOM manipulation to a modular, component-based architecture is currently in progress, allowing for more responsive telemetry gauges and wind-vane displays that pull live data directly from the backend API.

We expect to finalize the full integration of the write-capable control interface by the end of the semester. This final phase focuses on the command-injection logic, enabling operators to modify the waypoint queue and toggle autonomous modes through the established ROS service bridge. Once the map instrumentation and status panels are fully synced with the database, the system will undergo final validation to ensure the read-only and command interfaces remain consistent and synchronized during lake-test scenarios.

5 Jetson Orin Nano Integration

(Emith Uyanwatte, Eric Cai)

5.1 Overview

TODO

5.2 CV and CUDA Core Integration

PyTorch is the primary dependency of the computer vision system. A standard PyTorch installation (e.g., via `pip install torch` from the official Python Package Index) will not properly leverage CUDA on a Jetson Orin Nano because those prebuilt binaries are compiled for `x86_64` architectures and discrete NVIDIA GPUs, rather than the ARM-based architecture and integrated GPU found in Jetson devices. Precompiled PyTorch builds were instead sourced from the Jetson AI Lab PyPI mirror, which provides wheels specifically built for Jetson platforms with CUDA support enabled. This ensures compatibility with the device's GPU acceleration capabilities and allows PyTorch operations to fully utilize CUDA cores for improved performance in computer vision workloads.

Other changes were made for the computer vision system to work inside the Docker container. When running the container, the `/dev`, `/cuda`, and `/aarch64-linux-gnu` directories must be mounted so that the camera and GPU dependencies can be accessed. Additional installations for GPU dependencies were added to the Docker image. The current system is not portable due to the hardware requirements of the current system, so it is worth reconsidering its integration with Docker in future work.

6 Primary Microcontroller Changes

(Wang Mak, Emith Uyanwatte)

6.1 Overview

The primary microcontroller architecture was redesigned this semester to transition the vessel from a competition-specific platform to a robust system capable of long-range autonomous traversal. To improve endurance, we prioritized reducing power consumption and system complexity by offloading core navigation logic from the power-intensive Mini-PC to a microcontroller environment. This new architecture move towards a dual-microcontroller failsafe system to ensure hardware redundancy and operational reliability during extended deployments. Furthermore, since the removal of the Mini-PC eliminated the previous WiFi-based control link, we implemented a point-to-point communication system using XBee radio modules. By interfacing these modules directly with the microcontroller's serial pins, the boat can now receive low-latency command packets from a laptop or mobile app, providing a reliable manual override and telemetry link that operates independently of a local network.

6.2 New Functions

The Teensy firmware was refactored to include tasks that manage expanded communication overhead through a dual-monitor serial architecture. The system now operates two independent monitoring tasks: the ROS Monitor, which handles high-level commands from the Jetson, and the Radio Monitor, which polls the XBee hardware serial buffer for manual overrides. Each task validates incoming data packets for integrity, ensuring the precise parsing of servo positions. Crucially, a Control Mode flag is included specifically within the radio data packet; when toggled by the operator, this flag allows the Teensy to decide whether to follow the autonomous instructions sent by ROS or to relinquish authority to the direct radio stream. This modularity simplifies the upcoming transition of porting the navigation algorithm directly onto the Teensy, which will eventually allow the microcontroller to function as a fully independent primary navigator.

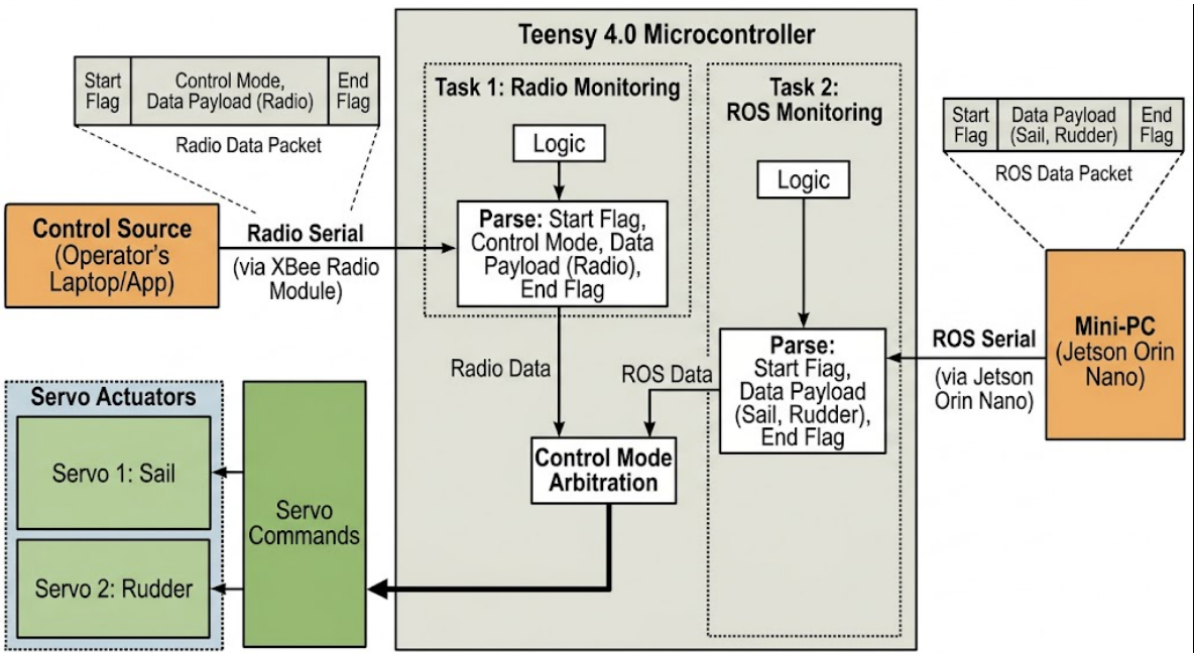

Figure 6: Functional block diagram of the microcontroller's command arbitration system.

Basically summarize what was added as of now. Wang write about the new RC stuff, I can talk about introduction of wind speed data. Wang, it'd be good to put a diagram detailing what the Teensy is sending to the Mini PC (and vice versa) somewhere.

6.3 Future Goals

I (Emith) can write this. The goal is really to let the boat move autonomously with just a microcontroller.

7 PCB Architecture Redesign

(Jonah Conolly, Liya Mei, Emith Uyanwatte)

7.1 Overview

Motivate why we're doing this, and give a brief overview of the project itself. Within past years, the team have utilized various purchased power step downs alongside team designed controls boards, with attempts made to unify this process. This results in unoptimized, expensive items that don't directly match our needs in terms of space and efficiency. This semester our controls PCB was completed from last semester, alongside a brand new implementation of a power stepdown, distribution, and control board which together can connect and power all the primary electronics on the board, with future-proofing and robustness incorporated into the system.

7.2 Controls PCB

This board seeks to take all of our various power sources and data signals and connect them to various locations in the boat. Primarily, this is the home of our main Teensy microcontroller, alongside its connections to our entire suite of motors and sensors. Important consideration was made to ensure PC connection, while planning for a future when this is omitted. Special consideration was given to ensure safe communication was enabled between VectorNav and Teensy due to their different data voltage levels.

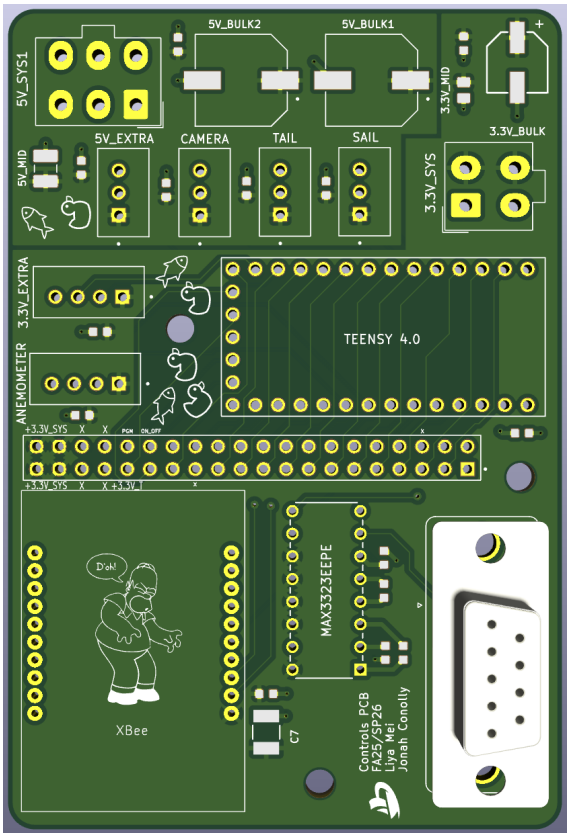

Figure 7: Full Control PCB with Teensy microcontroller and important connections across aspects of the boat

7.2.1 External Connections

This board features various important external connections which allow communication to the various devices on the boat, alongside power input. For power, the board features a 5V and 3.3V input port. The 5V port powers the servos on the boat and the teensy, with the 3.3V powering the anemometer, and XBee. Additionally, decoupling capacitors were implemented on the board, allowing for noise-free, clean, and consistent voltage input onto this board. The board also features ports to connect directly to these sensors and servos, alongside future-thinking additional connections through dedicated 5V and 3.3V control output ports and general access to Teensy pins through headers.

7.2.2 Sensor and Servo Connections

Within this compact package, connections are optimized and unified for ease of use and simple implementation. The Teensy 4.0 microcontroller has 40 input/output pins and is connected to the Mini PC via USB Serial. This controls

the servos using PWM signals, with the specific pins for each servo defined in our codebase. The Teensy also handles communication with the anemometer, which measures wind speed and direction through its analog input pins (A0-A9). The anemometer features four wires: green for wind direction output, yellow for power, red for ground, and black for wind speed, although the black wire is unused in the current configuration. The RC Xbee is also located on the control board, and allows for remote operation of the boat over radio.

7.2.3 VectorNav Communication

The data streams from the VectorNav and Teensy are at different voltage levels, so this board implements the MAX3323EEPE+ chip to allow for safe communication between these devices through the VectorNav's DE-9 port. The MAX3323EEPE+ chip was chosen based off of its capability to perform stepdown for low power but high data rates. This system allows for future control held just by the Teensy as it can directly communicate with the VectorNav without PC as in past boats.

7.3 Power PCB

Overall, this board seeks to maintain the power distribution across the boat, primarily converting 22V from our input battery down to 3.3V, 5V, and 12V for use across multiple applications. Alongside this, the board holds relays that are remotely controllable with an XBee and Arduino Nano that allows for power to be selectively "turned off" for specific parts of the boat.

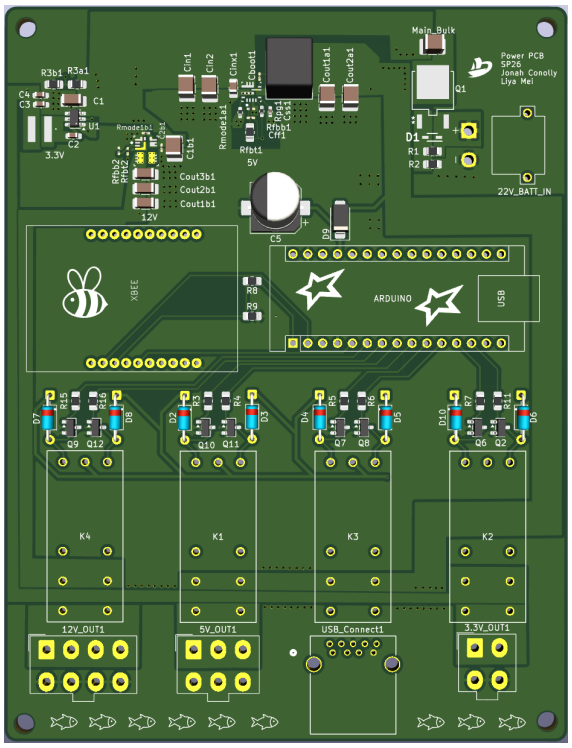

Figure 8: Full Power PCB with each buck converter and relay shown, alongside control devices in the center

7.3.1 Power Stepdown and Distribution

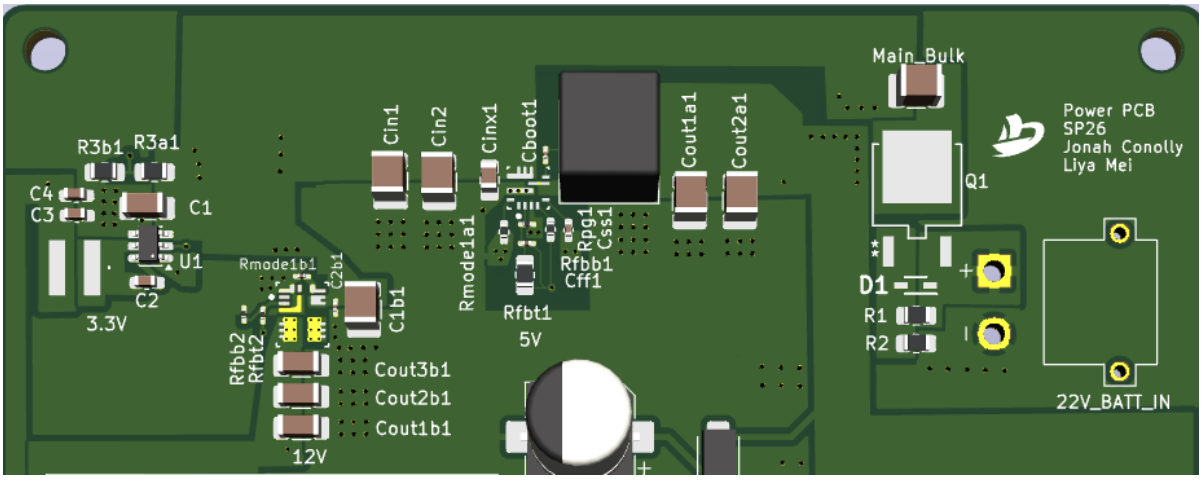

Figure 9: Power stepdown section of the power board displaying each buck converter system

Within this board, the input 22V from our LiPO battery is reduced to 3.3V, 5V, and 12V for various aspects of our boat. Immediately following battery input, the board holds a simple reverse polarity protection circuit and zener diode to protect electronics from incorrect battery connection, and unforseen battery variances. Then, to complete the transfer to lower voltages, each output voltage requires a unique buck converter chip to step down to the correct value with the correct tolerances. These work by rapidly connecting and disconnecting the input voltage to a system of capacitors and inductors, resulting in an output voltage as designed. These were chosen due to their overall efficiency and consistency in output. Perhaps most important to these implementations was a close understanding of the chip layout and requirements given on the data sheet, alongside a careful monitoring of heat management, fixed through numerous vias and large copper fill planes. The circuitry and surrounding board behavior for each buck was closely designed with special attention given to the datasheet to ensure all the precise components were arranged in a manner that ensures the stability of each chip. Additionally, each component was chosen for these chips by utilizing software such as TI's own WEBENCH design software to ensure correct functionality. The most complicated implementation of this comes within the 5V stepdown, where recent motor updates alongside an eye for future boats demands an output of 10A. To complete this task, the TI TPS56A37 is utilized and implemented at the center of the power section of the PCB. An important consideration for the tracks connecting to this output voltage, and to a lesser extent the output of the other bucks, was their thickness and ability to effectively transfer the required current without overheating or bottle-necking. This can be seen by the wide fills across multiple layers connecting these important voltages around the board, which are comparably larger than small data traces found elsewhere. Additionally, this board holds a 12V stepdown utilizing the TI TPSM84338, chosen for stability and consistent access to 3A. Finally, we implemented the AP63203 to similarly allow for 3.3V access on our board.

7.3.2 Relay Implementation

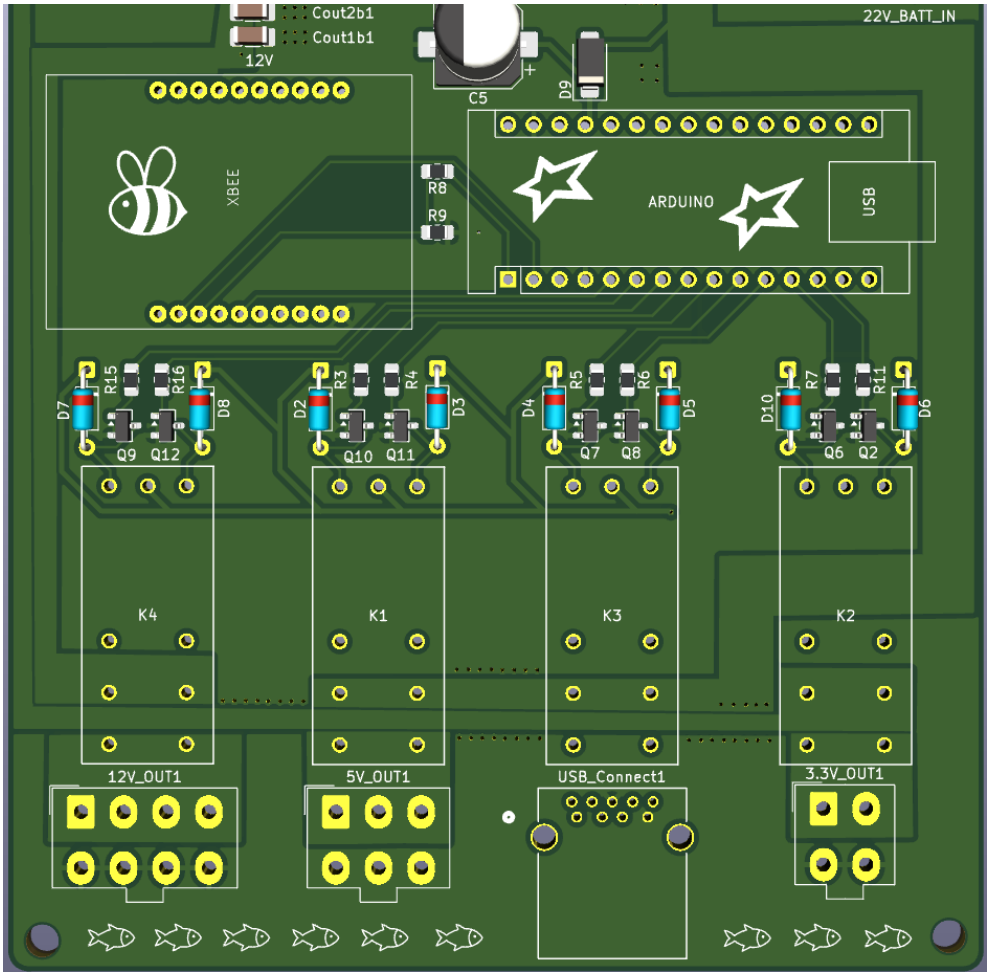

Figure 10: Relay and relay control section of the power board displaying each relay implementation and the control system above

This board holds four RT304F05 relays which are designed to allow for selective shut down of specific aspects of the boat. This setup was intended to allow for quick restarts or debugging of the boat whilst on the water, reducing the amount of times the boat is brought out of the water. The relays control 3.3V, 5V, and 12V outputs from this board, alongside one specifically designed to power the VectorNav. With these, servos and sensors can be individually reset while keeping PC functionality for example. Thus, remote resets can be targeted and only as severe as necessary. The entire relay setup is controlled by an Arduino Nano which receives data from an XBee radio module, which talks to controls on land. This system is always powered, ensuring the boat is always available to communicate with even when power is being reset. Additionally, the Arduino has its own input voltage capacitor to ensure it stays online even in the event of unforeseen voltage drops. The Arduino utilizes GPIO pins to toggle the relays, with an important consideration found within the transport of this signal to the relay itself. While the Arduino cannot directly toggle the relay due to low output from its own pins, this board implements a diode setup placed above the relay which allows our 5V power to be toggled by the Arduino to switch the relay. So, the Arduino still sends a digital signal to the diodes to toggle a 5V signal which actually toggles the relay.

7.3.3 Relay Control

I (Emith) will do this, it's just explaining the software on the Arduino on the Power PCB.

8 Manual Controls Overhaul

(Linnea Furlan, Ludvig Fellstrom)

8.1 Overview

During the last Sailbot competition, CUSail's competition team observed a noticeable delay between the team on shore and the instructions being received by the boat. As such, a new objective for the Spring 2026 semester was to reduce latency in our design. A decent portion of the observed delay can be attributed to boat-satellite communication since the team routed real time steering instructions through a satellite hosted web server making every command travel several kilometres to be actuated. To fix this, the system was redesigned to use a mobile controller that communicates directly with a Raspberry Pi Pico W over Bluetooth Low Energy, eliminating the satellite link from the control path. Here, the Pico W acts as a bridge, receiving sail and rudder data via GATT writes and immediately forwarding them as 6-byte frames through serial to an XBee radio module that handles the radio link to onboard servos.

8.2 Mobile Phone App

The manual controls overhaul features an Android application that provides remote operation of the sailboat over Bluetooth Low Energy (BLE). It involves a front-end activity that handles input and display and a BLE manager that controls radio logic. The phone acts as a BLE central device while the Pico W is acting as a BLE peripheral. This structure is standard in BLE and defines how devices are connected. Peripherals are typically smaller, low-power devices that advertise their presence while centrals are capable of higher-level computation and scan for and connect to peripheral devices.

The interface presents a connection status indicator, a connect button and two sliders corresponding to the rudder and sail angles. The rudder slider is mapped from its raw 0 to 90 degree range (Pico W firmware expects unsigned values) to a signed -45 degree to +45 degree range. This provides a more intuitive user interface by imitating the motion of the rudder onboard.

When the user initiates a connection, the phone's radio starts listening on the three BLE advertising channels for packets. Since the Pico W periodically sends advertising packets containing its device name, the phone's OS filters incoming packets for the expected device name. On the Pixel 7a, this filter is offloaded to the Bluetooth controller chip so the application processor only wakes when a match is found. This is why the scanning process remains battery-efficient even though the radio is constantly listening.

The BLE manager itself uses a state machine with four states (scanning, connecting, connected and disconnected) which tracks the lifecycle of the radio connection and GATT layering. State transitions are triggered by Bluetooth callbacks and protected to prevent invalid actions like starting a scan twice or writing to a closed connection. As such, any observed failure closes the GATT client, clears cached references and returns the system to its disconnected state.

8.3 Raspberry Pi Pico W and Xbee

The Pico W is used to translate between two means of communication, the BLE connection from the mobile remote control application and serial communication with an XBee radio module to reach the boat on the water. Two forms of communication are necessary since BLE is convenient for short range communication between a phone and a nearby device but does not provide the range needed to reach the sailboat on the water. Meanwhile, the XBee modules offer significantly greater range but are not natively supported by mobile devices (including the team's Pixel 7a) so the Pico W is needed to bridge the gap.

The Pico W relay operates on two distinct packet formats, one for each device connection. Both encode sail and rudder values as 16-bit little endian integers but differ in framing. This is because the UART link is a continuous byte stream so framing is needed to define packet alignment and reject corrupted data. Specifically, the GATT write carries just 4 bytes (two 16-bit little endian values for sail and rudder that are stored back to back) which the XBee then wraps into a 6-byte frame. The receiver (both the boat and Pixel 7a) know what every byte means without parsing beyond checking for frame markers.

The XBee module itself is driven over UART at 9600 baud in transparent mode, where bytes written to serial are sent directly to the paired remote module without additional overhead. Outbound commands are framed and transmitted immediately when received while inbound sensor data is handled by a heartbeat timer that fires every 1000 ms, draining the UART receive buffer and validating frames using expected start and end bytes. These valid frames update the cached sail and rudder angle values.

8.4 Future Goals

Some future goals involve the inclusion of additional sensor data (such as anemometer data) or controls (such as the incorporation of a gaming controller) which would require changes to the packet formats on both links alongside updates to the boat's firmware. In addition, the current version of the mobile app focuses on the BLE integration and interaction with

hardware and therefore displays angles as numerical values. In the future, the incorporation of a top-down illustration of the boat with the rudder and sail animated to imitate off-shore behavior would help the user read the boat's configuration at a glance without having to rely on a mental image. This would also distinguish the intended state from actual state of the sail and rudder onboard.

9 RC Power Controls

(Nicole Zhou, Fiona Lin)

9.1 Overview

The RC power control system was developed to address a limitation of the boat: being unable to power cycle the vessel remotely. During testing, if the boat's software system crashes in the field, the fastest recovery method is a hard reset. Without remote power control, this requires manually bringing the boat back to shore and redeploying, wasting significant time and effort. Additionally, some scenarios only require power cycling a single component rather than the entire system. This project solves this problem of labor, providing a handheld RC Power Control Panel that is able to selectively activate power to individual subsystems on the boat wirelessly. This panel communicates with the boat with an XBee radio module, leveraging existing infrastructure on the boat's side that was already prepared to receive these signals. Our team was responsible for designing the physical enclosure, a microcontroller to read switch states and send signals for XBee transmission, and a custom PCB to house all the electronics.

9.2 Physical Design

The control panel is housed in an enclosure designed for handheld use in the field. This design was created in collaboration with the mechanical subteam, who assisted with CAD modeling. The enclosure was designed to be endurable and easy to grip, similar to a Xbox controller. The front panel contains the following controls:

- **State Switches:** These SPDT switches are able to toggle between on and off states, allowing the operator to independently enable or disable each subsystem. Only two pins were used on these switches.
- **State LEDs (4 States, XBee connection, Box state):** The LEDs turn on when their corresponding switches are activated. The XBee connection indicator shows whether a valid connection to the boat has been established, and the box state indicates whether the RC power control box is powered on.
- **Emergency Stop (Big Red button):** Pressing this button will immediately cut off power to all subsystems simultaneously.

A custom circuit board was also designed for this project in order to integrate all electronics cleanly and compactly, and ordered through JCLPCB. The PCB was mounted with an Arduino Nano, XBEE module, a 5v to 3.3v buck converter, connector heads for switches and LEDs, and a USB-C connector.

For this PCB, we have the Nano powering the LED, reading the switch states, determining the connection to the XBees, and powering the button. The power travels from the plugged in USB-C to the buck converter, which steps down the voltage from 5V to 3.3V. This ensures that the Nano, and more importantly, the XBee (which only takes 3.3v) is safely powered.

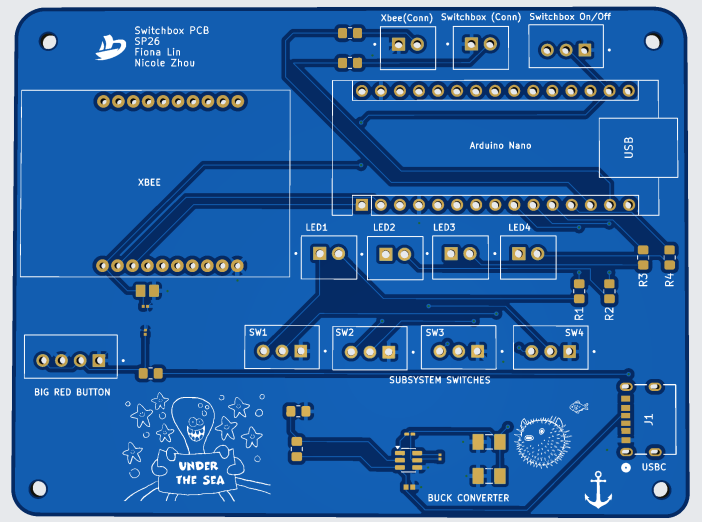

Figure 11: RC Control PCB with all traces components labeled

9.3 Nano Software Logic

The Arduino Nano acts as the central controller for communication with the boat via Xbee. Four digital input pins corresponding to switches are compared against their previous states to detect changes. If a change is detected, a serial command is sent to the XBee, which instructs the boat to shut down a certain subsystem. Sequential transmissions and short delays between them reduce risk of communication overlap. The Nano also checks for a periodic signal from the boat for status. If the shutdown button is pressed, then all systems are set to a latched off state which persists until the system is reset.

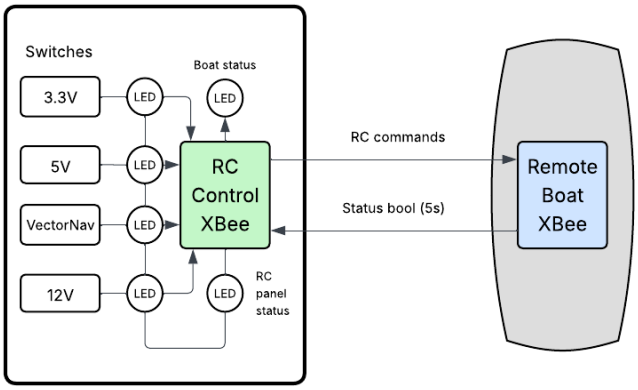

Figure 12: Diagram showing inputs and outputs from RC and boat XBees

10 Sailbench

(Angie Zhang, Sean Zhang, Albert Sun)

10.1 Overview

Sailbench is a simulation and training platform for exploring autonomous sailing. It provides a controlled environment where users can test boat designs and control strategies without needing lake-testing.

The platform makes it easy to iterate by adjusting boat setups and environmental conditions, which is useful for prototyping, comparing different approaches, and benchmarking performance. It also supports reinforcement learning, allowing agents to be trained on tasks like waypoint navigation.

10.2 Physics Simulation

The sailbench physics engine is centered on a simulation hub responsible for evolving the boat state over discrete timesteps. At each step, the system computes and aggregates forces and moments generated by individual components and applies numerical integration to update the boats's motion.

The simulation operates in 3DOF, with state variables including position, linear velocity, heading, and angular velocity. Wind speed is currently the only environmental input incorporated into the force calculation.

Although the current implementation provides a functional approximation of sailing dynamics, it relies on simplified physical models and assumptions. These simplifications may lead to inaccuracies under certain conditions, particularly at extreme parameter values or high nonlinearities.

10.2.1 Interchangeable Components / Models

Sailbench also features a modular component architecture. Boat configurations are constructed from independent components, each responsible for modeling a specific physical subsystem (sail, hull, rudder, keel).

Each component encapsulates its own parameters and force-generation logic, operating within its local reference frame. Components are configured via YAML files, enabling flexible composition and rapid experimentation.

This design supports systematic exploration of design variations, substitution of alternative physical models, and separation of concerns between simulation infrastructure and model definition.We explored a multitude of different ways to model these subsystems and found that X-Foil was an efficient way to model the sail, keel, and rudder.

10.2.2 Transform Tree

The transform tree provides a structured framework for managing coordinate frames across the simulation. Because individual components operate in local reference frames, transformations are required to express forces, positions, and orientations in a consistent global frame.

The system maintains a hierarchical relationship between frames, typically rooted at the boat's global reference frame, with a local frame of the boat and further child frames corresponding to individual components (sail, rudder). Transformations include both rotational components and are applied during force aggregation and state updates.The transform tree is especially helpful and can be used further in other projects, such as clarifying and standardizing angles in the boat's algorithm.

10.3 Reinforcement Learning

Sailbench incorporates a reinforcement learning framework for training autonomous control policies. The environment conforms to the Gymnasium interface, enabling compatibility with standard RL libraries. The primary task implemented is waypoint navigation, in which an agent must control the rudder and sail angle to reach a sequence of waypoints. Training is supported through Proximal Policy Optimization (PPO) via Stable-Baselines3.

The reward function is shaped to encourage progress toward waypoints while penalizing inefficient or unstable behavior. Additional curriculum strategies are employed to improve training stability and convergence.During experimentation, agents demonstrated the ability to learn effective navigation strategies. However, instances of "reward collapse" were observed, wherein performance degrades after reaching a high-performing policy. Potential contributing factors include overfitting, sensitivity to reward design, and inaccurate physics simulation.

Training outputs include model checkpoints, configuration files, and performance metrics, with optional visualization through TensorBoard and web-based interfaces.

10.4 Future Goals

While sailbench is in a functional state, several areas remain for further development and refinement, such as improving the accuracy of the physics simulation in alignment with real sailing. Establishing collaboration with ROS will enable testing of new algorithms without requiring immediate lake testing.Future work in reinforcement learning may focus on improved reward design and curriculum strategies, as well as testing across varying vessel configurations and environmental parameters.

References

[1] Kananaskis Improvement District. Don't get lost: Utms explained. Digital Image: UTM World Zone Map. [Online]. Available: https://kananaskis.org/dont-get-lost-utms-explained/

[2] Sailing Passion. Post about points of sail. Facebook Group Post. [Online]. Available: https://www.facebook.com/groups/371368279065178/posts/472822615586410/