Olin Sailbot Code Report

By Olin Sailbot Code Team 2011-2012:

Jason Curtis, Andrew Heine, Elizabeth Mahon, Luis Rayas
Oliver Gallitz, Jared Kirschner, Jaime McCandless, Steven Zhang

Table of Contents

1 Introduction

Person in charge of this section: Jason

The task of sailing is not a simple one. A sailor must use multiple sensory inputs and balance numerous priorities in order to make headway in the desired direction. The robot employs a combination of numerical optimization, fuzzy logic (in the form of "goodness functions"), and subsumption to accomplish the task of robotic sailing. An object-oriented architecture is used to split each of the events of the International Robotic Sailing Competition into smaller missions and complete them.

Our code architecture follows a standard robotic architecture, whereby the active area of the code is divided into three sections: **Sense**, which perceives the outside world, processed, and entered into a state knowledge model, **Think**, where state information and goals are combined to make decisions, and **Act**, wherein actuators are controlled to implement those decisions in the physical world. In addition, we have programmed an **Operator Control Unit** (OCU) for the robot, which allows us to remotely download mission data, monitor telemetry, and override the robot's controls.

2 LabVIEW

The software platform used on our robot is National Instruments LabVIEW. In the LabVIEW Development Environment we use G, a data-flow based, automatically parallelizable, graphical programming language that allows for efficient creation of highly accessible code. A simple example is illustrated in Figure 1 and 2.

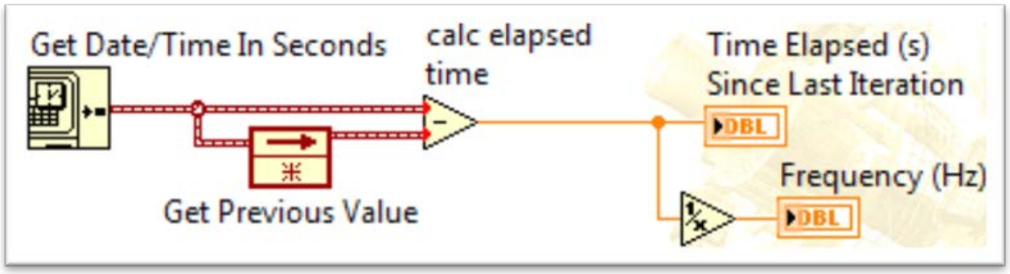
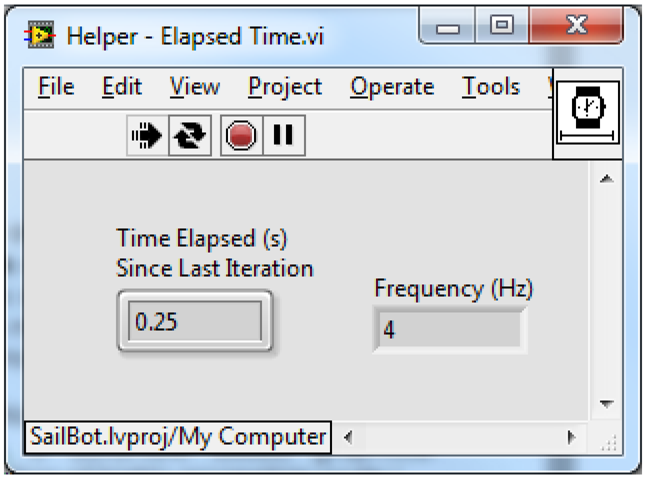


Figure 1: Code in a simple LabVIEW "block diagram"


Helper - Elapsed Time.vi
File Edit View Project Operate Tools
SailBot.lvproj/My Computer

Time Elapsed (s) Since Last Iteration: 0.25

Frequency (Hz): 4

Figure 2: the automatically created "front panel" interface for the code shown in Figure 1. As the code is run, even when it is used within other LabVIEW code, key data can be monitored here.

LabVIEW was chosen because of the ease with which it can be debugged and its integration with National Instruments computing and data collection hardware. This has facilitated the effective and seamless testing and development of the same LabVIEW code on different computer systems.

3 Platforms

Person in charge of this section: Jason

To provide development flexibility, the code team has three platforms capable of running the same Think code. The three platforms are a simulator, a small R/C boat, and the competition boat. Each platform, by necessity, has its own sensors and actuators, so the Sense and Act code is written separately for each platform. On the other hand, the more nuanced and complex Think code has the potential to be usable on any sailing platform, and we take lengths to make it so. The same Think code is therefore used on our three platforms. By running the same Think code on three separately accessible platforms, we are able to test and actively develop the code in anywhere from a basement desk to the high seas. We are not reliant on the availability of open water, wind or even a physical boat to test our code architecture. This has been extremely important in reducing the critical path for the Olin Sailbot team.

Platform A: Simulation

The first platform, built before any physical devices were available to us, is the simulator. Free of timing and location constraints and expensive equipment, the simulator lets our coders test algorithms anywhere they have their laptops.

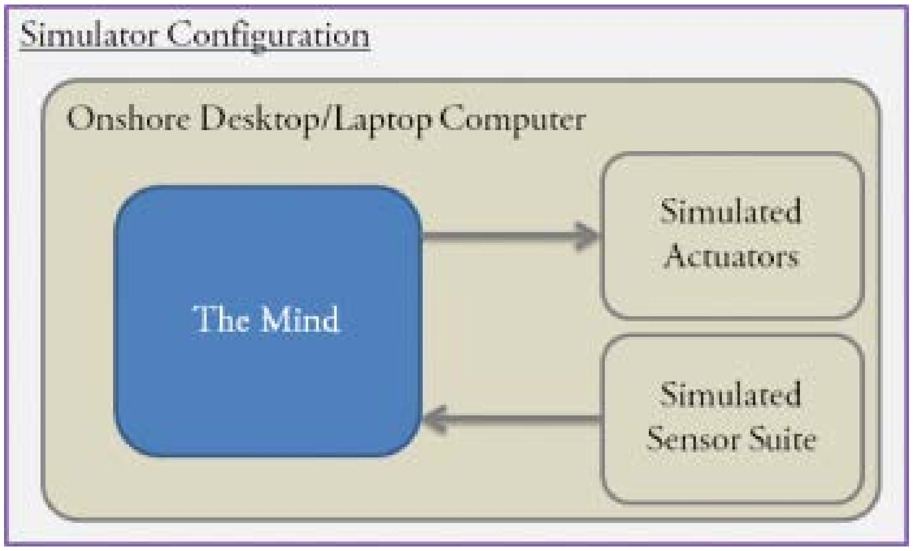


Figure 3: Simulator Sense/Think/Act configuration. For the simulator, everything occurs on the onshore host computer.

Since it doesn't involve any actual water or wind, the simulation is also largely free of realism. The simulated boat has velocity, rotation and location based on simulated wind conditions and basic physics approximations that occur on the host computer (Figure 3). The complexity of the simulation was initially kept to a minimum, but has been upgraded to include realistic currents and drag forces so that more complex behaviors can be tested. In addition, we have inserted helper functions that add randomness to movement and measurements in the simulator to simulate real-world unpredictability.

Platform B: Small R/C (Radio-Controlled) Boat

To get our algorithms running in real sailing conditions, we need something that controls a real sailboat. Thus, the Small R/C platform.

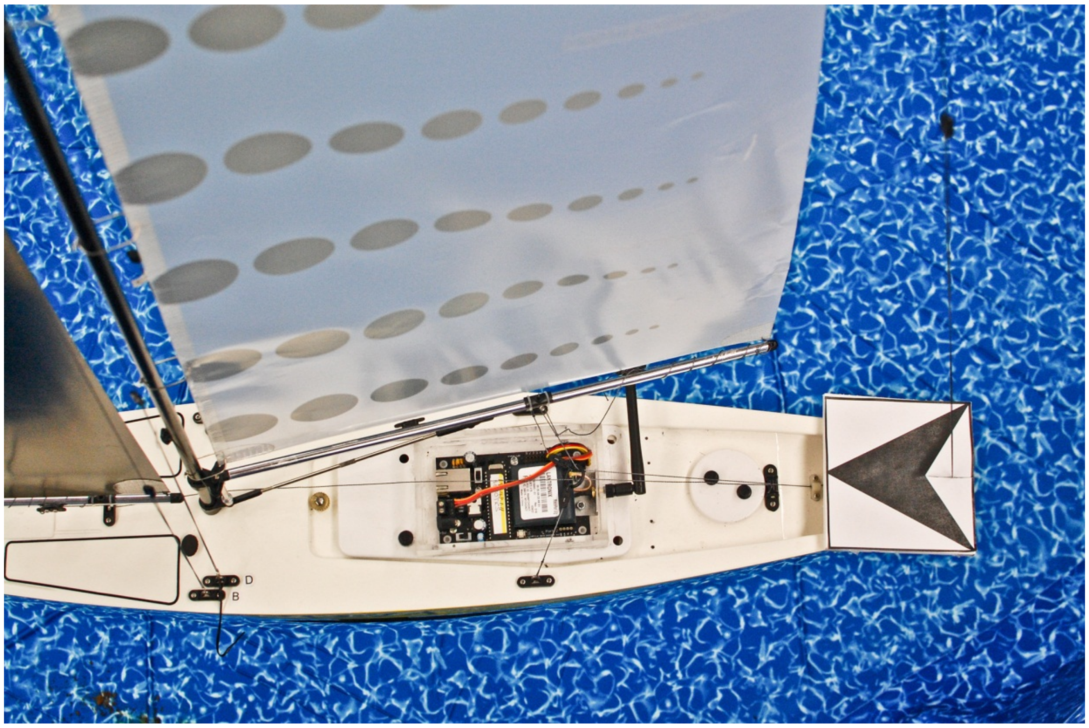


Figure 4: the Small R/C boat as seen from above, with visual tracking pattern visible astern and Wi-Fi receiver visible at the center of the image.

On the Small R/C platform, we can test algorithms that can easily be transferred over to the race boat. The Small R/C platform consists of an off-the-shelf 1m Vela remote-control sailboat modified to carry a Wi-Fi-connected receiver in place of the stock RC receiver. The boat runs in a 20ft diameter pool in Olin's Large Project Building, with a set of fans for wind.

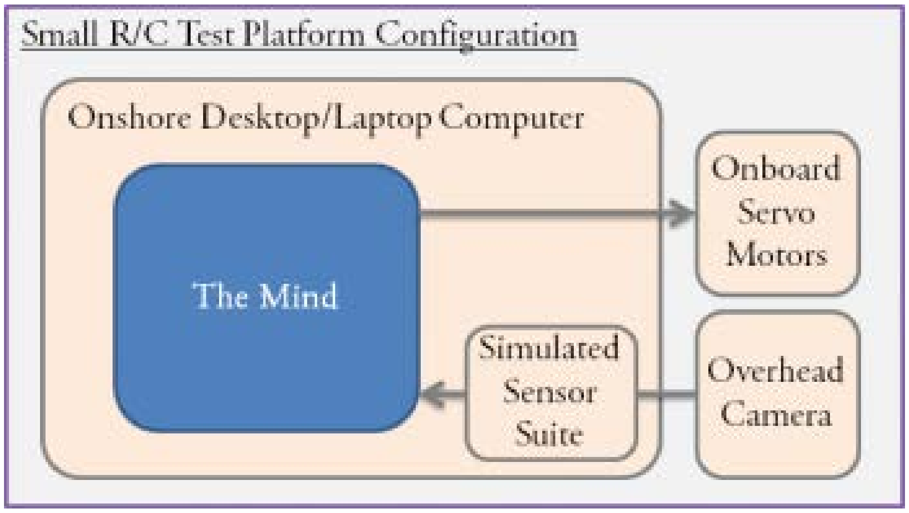

Small R/C Test Platform Configuration
Onshore Desktop/Laptop Computer
The Mind
Simulated Sensor Suite
Onboard Servo Motors
Overhead Camera

Figure 5: Small R/C Test Platform Sense/Think/Act configuration.

Since there is not enough space for sensors and computing power on the boat, the sensing and “think”ing occurs on an offboard computer (Figure 5). An overhead camera using machine-vision tracking algorithms provides simulated “sensor data” to the Think section of the code, converting the location of the boat to simulated UTM coordinates.

Tracking optimizations
With the varying lighting conditions in the pool, it proved challenging to reliably track the motion of the boat in the pool.

Two tracking patterns, one the black/white inverse of the other, are tracked independently, such that only one needs to be found in order for the boat’s location to be found. When both patterns are detected, the locations are used in conjunction to approximate the location of the boat.

To reduce false positives, the area of the camera outside the circle of the pool is masked out of the image before tracking occurs.

Small RC conclusion
When the Small R/C boat gets lost and goes adrift, it’s always within reach of the pole that we keep handy for just such an occurrence. This platform is fantastic for debugging and tuning, but it’s still in a “sandbox”, several steps away from the high seas.

Platform C: Competition Boat

This brings us to the most important platform: the competition boat. For the competition and the future we’ll need a self-contained system that can run our code and sail on the high seas. For this we’re using a National Instruments single-board RIO onboard our 2m custom-built sailboat.

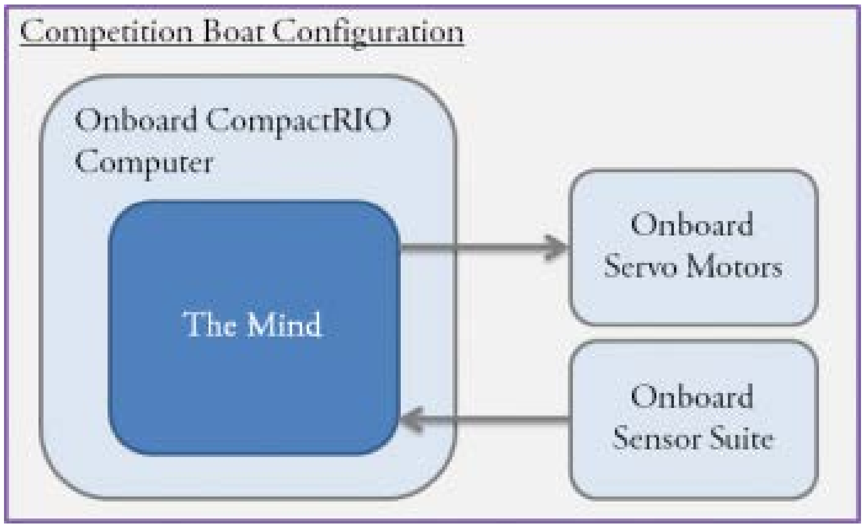


Figure 6: Competition Boat Sense/Think/Act configuration. All computation occurs on the onboard real-time computer, which interfaces with the onboard sensor suite and motors.

4 Sense

In charge of this section: Jaime

Our sensing requirements are fulfilled by a single all-in-one sensor: the Airmar WeatherStation. General specifications for the sensor can be found in Section. We wrote our own sensor driver in LabVIEW to parse the NMEA 0183 serial data output by the sensor. In addition, we have implemented a "defilter" to counteract the WeatherStation's built-in smoothing functions.

Output Selection

The WeatherStation is capable of outputting a wide range of NMEA 0183 data packages, each at variable frequency up to 10Hz. However, the sensor is limited by a maximum 38400 baud rate, so we programmed it to output only the relevant subset of its full capability:

Header	**Data type**	**Freq. (Hz)**
GPDIM	Datum Reference	0
GPGGA	GPS Fix Data	0
GPGLL	Geographic Position—Latitude and Longitude	5
GPGSA	GNSS DOP and Active Satellite	0
GPGSV	Satellites in View	0
GPRMC	Recommended Minimum GNSS	0
GPVTG	COG and SOG	5
GPZDA	Time and Date	1
HCHDG	Heading, Deviation, and Variation	0

HCHDT	True Heading	5
TIROT	Rate of Turn	10
WIMDA	Meteorological Composite	1
WIMWD	Wind Direction and Speed	0
WIMWV	Wind Speed and Angle	5
WIMWR	Relative Wind Direction and Speed	0
WIMWT	True Wind Direction and Speed	0
YXXDR	Transducer Measurements	0

Driver Structure

The general structure of the driver is:

1. Initializing the sensor
2. Reading the serial data one byte at a time and bundling it into NMEA sentences
3. Parsing each NMEA sentence to extract and store the data

The sensor must be initialized every time it is powered on. After that, the serial read loops through bytes until it has assembled a full NMEA sentence. The sentence is passed to the NMEA parser, which extracts and stores the data.

Sensor initialization routine

The sensor's default start-up setting (which cannot be changed) is to communicate on 4800 baud. We want to maximize the serial communication rate, so every time we start up we have to reprogram the sensor to communicate on 38400 baud. This initialization routine is shown in Figure 7.

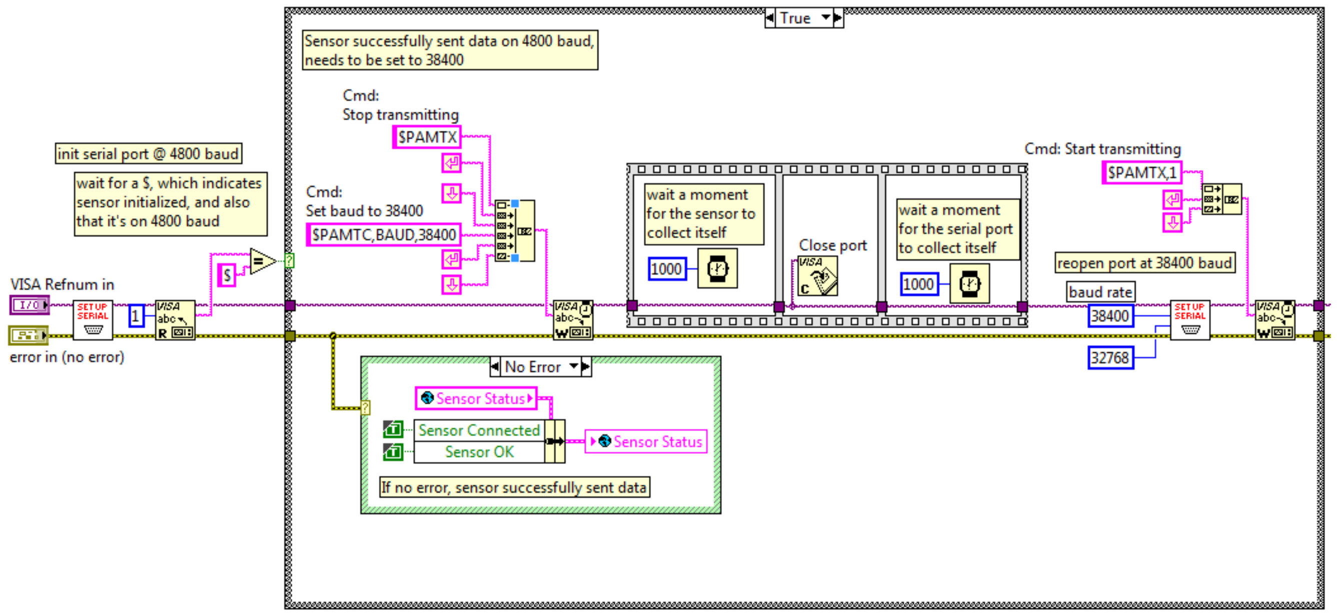

True
Sensor successfully sent data on 4800 baud, needs to be set to 38400
Cmd: Stop transmitting
$PAMTX
Cmd: Set baud to 38400
$PAMTC,BAUD,38400
init serial port @ 4800 baud
wait for a $, which indicates sensor initialized, and also that it's on 4800 baud
VISA Refnum in
error in (no error)
$
1
wait a moment for the sensor to collect itself
1000
Close port
wait a moment for the serial port to collect itself
1000
Cmd: Start transmitting
$PAMTX,1
reopen port at 38400 baud
baud rate
38400
32768
No Error
Sensor Status
Sensor Connected
Sensor OK
Sensor Status
If no error, sensor successfully sent data

Figure 7: WeatherStation initialization routine: "1A Helper – Init Sensor.vi". First, the VI opens a serial port at 4800 baud and checks to see if it can read data from the sensor. If so, it commands the sensor to set its baud rate to 38400 instead, then closes the original serial port and opens a new one at 38400 baud.

Reading serial data

During normal operation, the sensor reads and parses on NMEA sentence at a time via the "1A Read Sensor Data.vi".

NMEA 0183 sentences are structured:

$HHHHH,…*ss

In which the "$" denotes the start of a sentence, the "HHHHH" is a five-character header describing the data package to follow, the "…" is a string of comma-delimited data, and the "*ss" is the two-character checksum at the end. Thus, the VI waits for a "$", then fills a string with any characters between the "$" and the next "*", the checksum marker. This is illustrated in Figure 8.

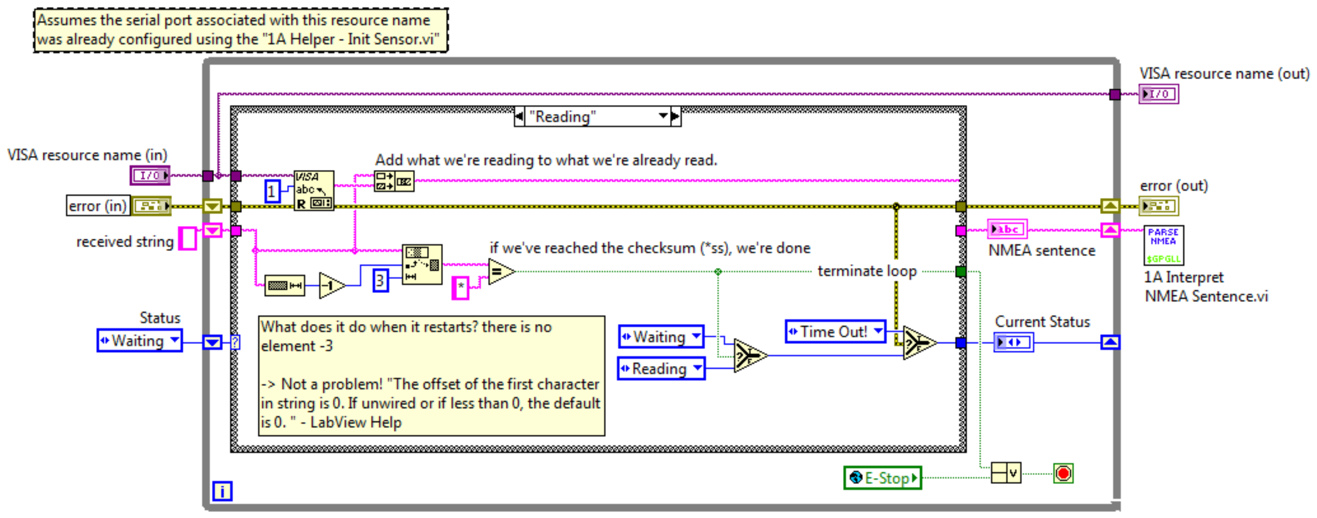

Assumes the serial port associated with this resource name was already configured using the "1A Helper - Init Sensor.vi"
VISA resource name (in)
error (in)
received string
Status
Waiting
"Reading"
Add what we're reading to what we're already read.
if we've reached the checksum (*ss), we're done
terminate loop
What does it do when it restarts? there is no element -3
-> Not a problem! "The offset of the first character in string is 0. If unwired or if less than 0, the default is 0. " - LabView Help
Waiting
Reading
Time Out!
E-Stop
VISA resource name (out)
error (out)
NMEA sentence
1A Interpret NMEA Sentence.vi
Current Status

Figure 8: Reading one sentence of NMEA 0183 data. The VI reads serial data one byte at a time, first waiting for the "$" which denotes the start of a sentence, then saving each byte in a string until the end "*" checksum marker is reached.

Parsing NMEA sentences

The "1A Interpret NMEA Sentence.vi" parses a single NMEA sentence to extract data. It also updates the relevant global variable when appropriate.

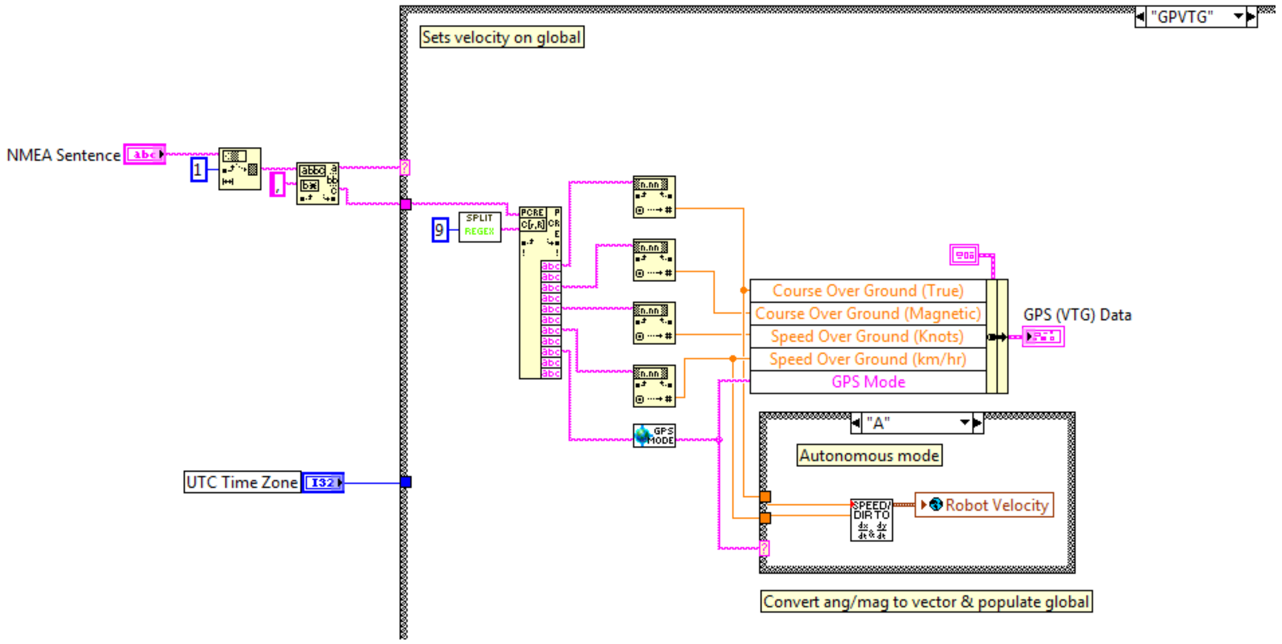

"GPVTG"
Sets velocity on global
NMEA Sentence
UTC Time Zone
Course Over Ground (True)
Course Over Ground (Magnetic)
Speed Over Ground (Knots)
Speed Over Ground (km/hr)
GPS Mode
GPS (VTG) Data
"A"
Autonomous mode
Robot Velocity
Convert ang/mag to vector & populate global

Figure 9: An example of NMEA sentence parsing.

GPS Uncertainty

In charge of this section: Elizabeth

We rely on the GPS for a significant portion of our information. In addition to our position, we use it to calculate our velocity, and, from there, the apparent wind. Furthermore, knowing our position is essential to knowing if we have accomplished any of our missions. As a result, knowing how accurate it is at any point is extremely important. According to Airmar, under optimal conditions our sensor should be accurate to within 3 meters. However, these optimal conditions are very specific. We need a clear view of at least four satellites, as well as being able to contact an additional wide area augmentation system (WAAS), which provides adjustment data to the sensor.

In testing, we determined that under most circumstances, we tend to have drift of about 5 to 10 meters over a period of 10 minutes. As a result, while we can trust our GPS to have a fairly accurate difference between two points in close succession, over time it will not be nearly as reliable. This is in part because we do not consistently communicate with the WAAS satellite.

To improve and manage the error within our GPS calculations, we took a few steps. First, when the GPS only has a view of 3 satellites or has a bad view of more than 3, it uses a “2D” fix, in which it cannot determine altitude. In this mode, it is less accurate. However, it is possible to give it a fixed altitude for it to use when it is in this mode, which increases its accuracy, so we did. It is also possible to get a reading of the estimated precision from the sensor, which is called the dilution of precision. To help the human in the loop understand the potential inaccuracy and change their commands accordingly, we acquire this information and display it on the OCU.

Defiltering

In charge of this section: Jason and Steven Z

In testing, we found that the wind speed and direction outputs from the Airmar PB200 sensor are filtered with some kind of moving average filter such that any sudden change in wind speed or direction input results in an output with a lag on the order of a few seconds of time lag.

For example, Figure 10 shows a trial we did of measuring the PB200’s wind direction sensitivity by suddenly rotating the PB200 while facing a fan. The response time of the PB200 is on the order of ten seconds.

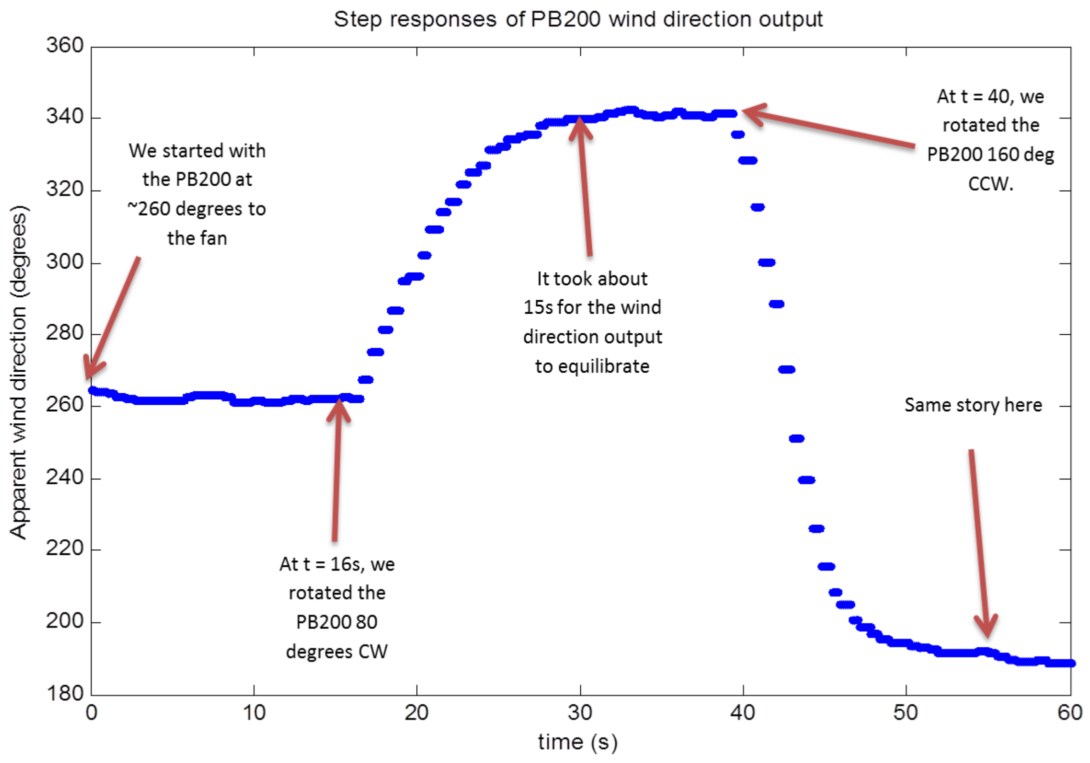


Figure 10: Airmar PB200 wind direction response

The PB200's wind speed output has similar sensitivity issues:

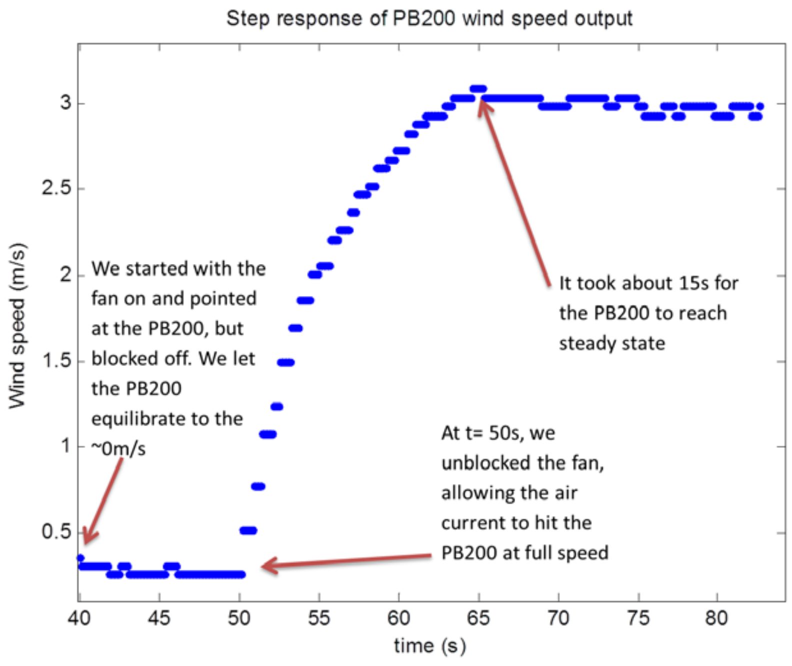


Figure 11: Airmar PB200 wind speed response

Obviously, this kind of response is not ideal, since our boat is making decisions about how to trim the sails and rudder on the order of seconds, not tens of seconds. Deviation of measured wind direction from actual wind direction could cause huge miscalculations. Wind speed sensitivity is not as important as wind direction, but could still affect performance.

Methodology

We took video (http://youtu.be/XG9UZ7fsSPE) showing us doing an impulse and step response of wind speed. Ideally we'd use a wind tunnel, but we hacked something together with a fan and a big wooden board.

Contacting Airmar

We called Airmar and their sales rep. informed us that indeed, the output of the wind sensor is filtered "over many seconds" since that's what the majority of users (presumably owners of non-robotic boats) demanded.

Evidently, some customer segments had similar needs as us and wanted a more sensitive wind sensor, so Airmar plans to release a sensor with unfiltered output later this year. The sales rep. told us that they may be able to get us a sample of this sensor early. The sales rep. also promised to ask the engineering department and find the filtering specifications for us, but Airmar never responded to our repeated email requests. Thus, we decided characterize the filter ourselves and create an "unfiltering filter," or "defilter", in software.

The solution: Theory

The theory behind unfiltering the PB200 is rather simple. The PB200 applies a filter h(t) to the input signal (in this case, the actual wind speed or direction). We want to create a filter in software that negates it with impulse response h^(-1)(t). Ideally, the system diagram would look like this:

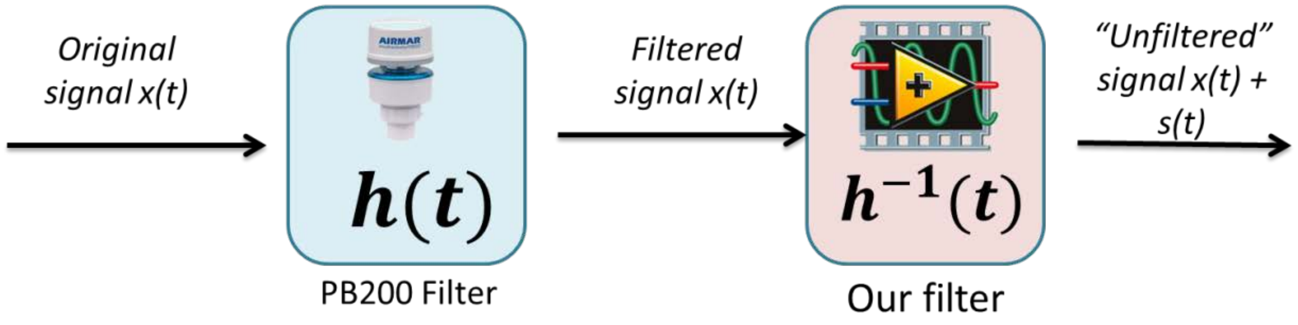


Figure 12: "defiltering" concept diagram

The output signal from our filter will inevitably have noise or other source-induced error (denoted by s(t) in the diagram above). We want to minimize s(t) when we construct h^(-1)(t)

Finding h(t)

The first step to constructing h^(-1)(t) is finding the impulse response h(t). As the name suggests, this is as simple as giving the PB200 a sudden, but short, gust of wind for wind speed, and a quick rotation back and forth for wind direction. We used the fan and wooden board technique described above for generating the short and sudden gust#. We smoothed the data a bit to get the following impulse response for wind speed:

It was a bit tricky to collect data for the impulse response of wind direction. We ended up taking the time derivative of the step response to get the following plot:

Calculating h-1(t)

A simple way to calculate h^(-1) (t) is just to take the inverse Fourier transform of the reciprocal of the frequency response H(ω). To show why this works, we have the condition for h^(-1) (t) such that

$$h(t)*h^{(-1)}(t)=\delta(t)$$

$$h(t)*h^{(-1)}(t)*x(t)=x(t)$$

where * is the convolution operator. Taking the Fourier transform of both sides, we get:

$$H(\omega)\cdot H^{(-1)}(\omega)=1$$

So the Fourier transform of h^(-1) (t) is just the reciprocal of the Fourier transform of h(t). However, the problem with this direct approach is that any small noise in H(ω) becomes really big as one takes the reciprocal. So we used a more sophisticated approach that compensated for this effect for small values of H(ω). We used a threshold gamma inverse filter adapted from here.

The result

Our “unfiltering” filter works quite well. Here’s a video of the wind speed filter working. Pay attention to the laptop screen on the bottom left. The white line shows our defiltered output, while the red fill shows the direct, filtered output from the PB200. You should be able to see how much more responsive the white line is than the red filled line:

Limitations

We encountered some problems applying this technique to the wind direction data. The main problem was the data wrapping that is inherent with angular measurements (when wind is blowing at 350 degrees, a slight movement can cause the data to jump to 0 degrees). We resolved this by playing around with some angle unwrapping functions. In the end, we used the filter coefficients generated for wind speed for angle. It seemed to work rather well.

Internally, the PB200 apparently measures wind speed by measuring wind speed in x and y axes independently and pythagorating the vectors. Apparently, the individual x and y-direction speed measurements are filtered before being summed. Evidence of this comes from our experiments when the PB200 is at steady state wind speed, any rotation causes the speed to quickly drop down to 0 before slowly coming back to steady state. This is something we can't really address, so we'll have to hope that this effect doesn't affect things too much during competition.

5 Think

In charge of this section: Luis

What is Think

The think loop is responsible for synthesizing behaviors which can be performed by the act loop using as input the data collected by the sense loop. The majority of the control logic is contained within the think loop. We have designed the system such that the think loop can be utilized modularly between different boat platforms. This section will detail the processes which take place in the think loop.

Different Components, Roadmap

The think loop contains the code which determines decision priorities (the arbiter), organizes the structure used to define missions and objectives (mission class), and makes navigational decisions (waypoint navigation). These overarching processes are discussed in order in this section.

5.1 Arbiter

In charge of this section: Luis

Why/What it does

The purpose of the Arbiter is to determine which high level priorities should be considered. We run 4 algorithms in parallel that output sail and rudder setpoints. Each algorithm considers one of four primary goals of the boat:

-Manual Override: do what the controller is telling it to do

-Don't Flip: that would be bad

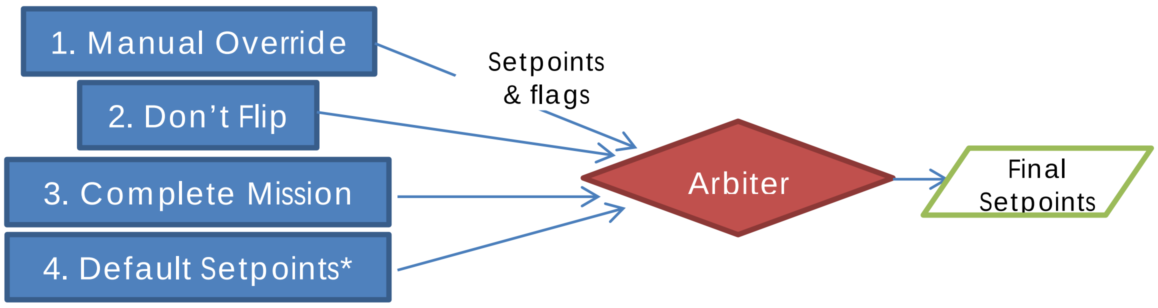
-Mission Objective: so that we may win

The arbiter decides which of these algorithms has priority to ultimately decide what the 'Act' block will act upon.

Each algorithm must be constantly running, which means there needs to be a system to continuously monitor each algorithm and output a pair of setpoint values.

Priority Balancing

In order to organize these priorities, we introduced the idea of "soft" and "hard" priorities. The soft priority of an input to the arbiter is hardcoded into the arbiter. The purpose of the soft priority is to define which functions are generally more important. The algorithms are ordered in "soft" priority like so:



"Soft" priority is used to distinguish from "hard" priority where "hard" priority will be used to denote which algorithm is setting the setpoints for the boat at any given moment in time. Since we do not always want to control the boat, and the boat is not always in danger of flipping, there must be a system to determine when to consider these priorities and when to pass control onward. "Hard" priority is determined by a pair of Boolean flags that each objective outputs. When a Boolean flag is set to true, the algorithm is telling the arbiter "listen to my output now". The arbiter will consider this, and if no other algorithm with a higher "soft" priority is outputting true for that flag, the arbiter will give "hard" priority to this algorithm. The sail and rudder setpoints are set independently (i.e. each algorithm outputs 2 flags), both in this fashion.

*Default Setpoints are used only when for any reason no input is giving instructions to the arbiter. In this case we would like a default behavior so that our code has something to do.

Priority Details

Each priority type has a method which determines when it takes priority and what it does with the priority. These are as follows:

Priority 1: Manual Override
In charge of this section: Jason

This priority takes effect when a manual override signal is received from the OCU over the network stream. The operator at the OCU can take individual control of the sails or rudder as needed. For more details of OCU operation, see section 8 (page 40).

Priority 2: Don't flip
In charge of this section: Olli

Priority 3: Mission Objective
In charge of this section: Luis

The mission objective priority is active so long as we are not at risk of flipping and the operator has not taken manual control. The bulk of the intelligence of the boat exists in the mission objective. A detailed explanation of the mission objective follows in section 5.2.

5.2 **Missions: Determining Waypoints**

In charge of this section: Jared

We have divided the various challenges of the competition into a series of smaller tasks which we refer to as *missions*. Any possible sailing challenge can be fully described as a serial combination of missions. A competition event is programmed as a series of one or more of these missions, which are executed in successive order. The current mission will have full control over the path-planning of the boat until the mission is complete, at which point the next missions will be started.

This approach has two primary difficulties:

- Determining a complete basis set of missions from which larger challenges can be accomplished
- Providing a given mission with full control over the path-planning of the boat

Rather than determining a complete basis set for missions, we created a modular design which allows us to implement new types of missions as necessary. If the set of implemented missions is not sufficient to accomplish some future goal, we can simply modify the set of missions. Through the abstraction of mission interfaces (the actions that this class can perform), the updated set of missions will integrate seamlessly with the existing code.

We implement this abstraction in LabVIEW by creating a GenericMission class which defines the public interfaces common to every possible type of mission. The four interfaces are:

1. A method which runs a start-up sequence for the mission
2. A method to update the current waypoints based on mission state and sensor data
3. A method which indicates whether or not the mission is done
4. A method which indicates the general physical location of the mission, so that the previous mission can set the boat on an appropriate course to begin the new mission.

All actual missions are child classes of this generic, abstract class. The currently implemented class hierarchy is shown in Figure 13. Each mission only needs to override these four methods and it will integrate seamlessly with the existing architecture. Due to the object-oriented structure, the proper version of each of these methods is called based on the identity of the caller child class.

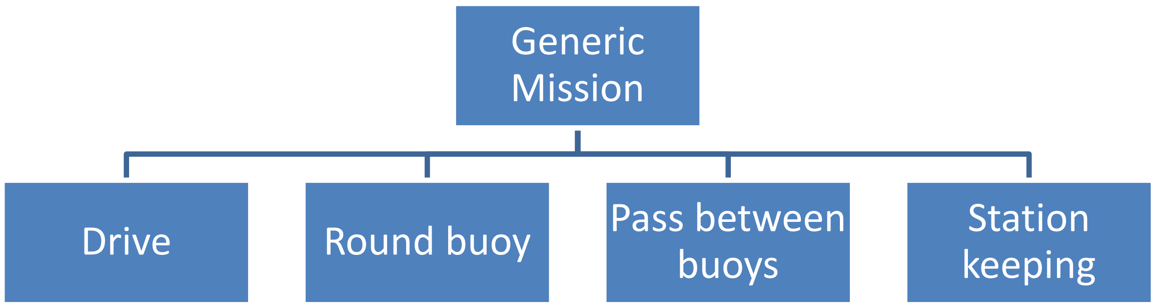


Figure 13: Object-oriented mission structure diagram. Generic Mission defines all of the necessary interfaces for any child class to be successfully executed by the Think code. Currently, we have defined four child classes—one for each mission type. Adding another mission type is as simple as creating another descendent class of Generic Mission.

A competition event is programmed as a series of one or more of these missions, which are executed in successive order. At any point in time, the mission that is currently being executed keeps a list of next waypoints for the sailbot to get to, along with a list of no-sail zones, which the sailbot avoids entering.

The current mission is given full control over the path-planning ability of the sailbot by being provided with the ability to control the set of waypoints that the boat tries to reach and the list of no-sail zones which the boat tries to avoid. We have found that these two abilities are sufficient for each of the currently implemented missions to guide the sailbot towards their completion.

The user is able to programmatically add missions to the sailbot upon start-up or while running through the OCU. When the user requests that a new mission be added from the OCU, this new mission is placed into a queue of GenericMission objects—the parent class of all missions. During each cycle, the "Think" loop checks to see if the current mission is complete. If so, it removes the first mission from the queue, makes this mission the new current mission, and then starts the mission. The method which updates waypoints is then called by the current mission. This method will modify the waypoints that the boat is constantly trying to satisfy. In this way, each mission is able to guide the actions of the boat until the mission determines that it has been completed. By stringing together a series of these missions, a large, complex task can be completed systematically.

Diagram key

In charge of this section: Jason

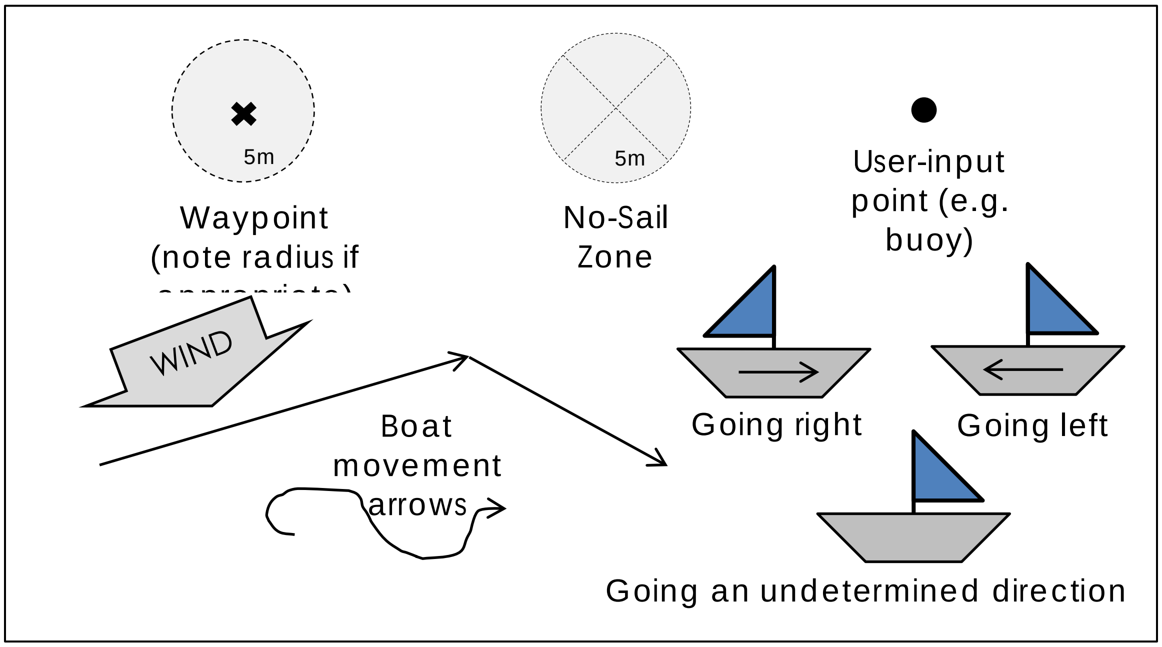


Figure 14: Diagram example showing the symbols we use.

Drive

In charge of this section: Jason

The Drive mission is the most basic, and instructs the sailboat to sail to a single waypoint. It is used internally by our team to test the robot's navigational capabilities.

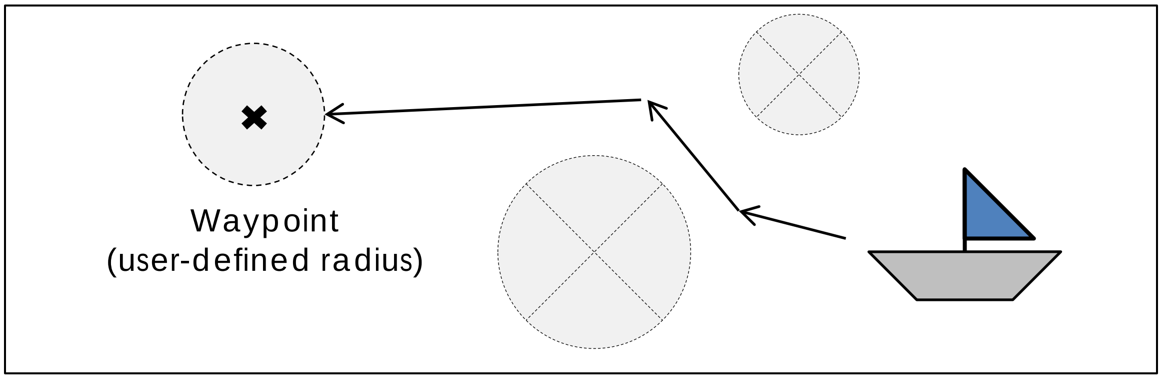


Figure 15: the sailbot completing a Drive mission.

Definition
The drive mission is defined by a single waypoint (with radius) with any appropriate no-sail zones (with radii).

Action
When a Drive mission becomes active, it places a single waypoint with its no-sail zones into the waypoint list.

End Condition
A Drive mission is complete when the waypoint has been reached.

Round Buoy

In charge of this section: Olli

The algorithm to round buoys has to ensure a certain behavior of the boat:

- The boat has to round the buoy on the desired side
- The official rules for rounding buoys have to be regarded
- The boat must not turn into irons while rounding the buoy

Definition
The drive mission is defined by a single UTM location and a direction of rounding.

Action
Different wind directions make determining a fixed path around buoys almost impossible. During sailing competitions, many paths can be chosen depending on opponents, wind speed & direction. To address that, we developed a pattern that determines the ideal path around a buoy. The path is calculated by working backwards. The following steps determine the way around the buoy:

1. Create a start and endpoint in a distance determined by the waypoint tolerance and the safety distance

2. Determine the heading from the endpoint to the next planned waypoint, based on wind speed
3. Set points on this line in the other direction (behind the rounded buoy)
4. Determine the headings from the starting point to each of the points set in 3.
5. Choose the best one of them (fastest and closest to buoy)

Rounding a buoy is one of the missions in the sailboat competition. To successfully round buoys in actual sailing competitions, the teams have to regard not only the wind direction, but also the opponents and their position relative to the buoy and their own boat. Therefore a wide range of different rounding tactics exist, and most of them include staying close to the buoy.

We use a basic way of rounding the buoy, in part because the GPS accuracy does not allow us to get too close to the buoys (danger of rounding them on the wrong side or sailing into them), and because we cannot react to changing wind conditions as quickly as humans can through watching the water and weather.

At the sailboat competition, the rules for rounding a buoy imply a line which is perpendicular towards the “leeward leg”. In the navigation and the long distance race, the buoys are placed in a way that the boat will always go with or against the wind to round it. To make this rule more applicable to other challenges and races, we choose the line from the next waypoint of the mission to the buoy and define a perpendicular line on this one for the start and endpoints of the buoy rounding.

This makes the Round Buoy code work in the two races where rounding buoy is an official task, but also in other races if wanted.

Regarding these rules, our tactics for rounding buoys is to build up the ideal path backwards. Three points will be created to determine the way around the buoy:

Step 1: Depending on the rounding direction, the start and the end point of the rounding path is created. They will be placed at a predefined distance, which is the point tolerance plus an additional value, to avoid either checking the wrong waypoint or running into the buoy. The points are placed on a line that has been calculated previously.

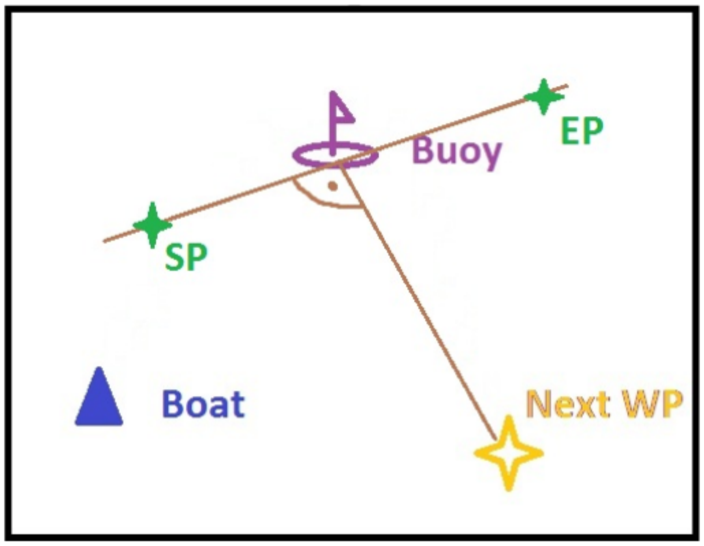


Figure 16: Round Buoy Step 1

Step 2,3: Starting at the endpoint, a line towards the rounding side of the buoy is created. This line has the length of the distance between start and endpoint, to avoid running away from the buoy during the turn. It is at an angle of 45 degrees towards the line from the endpoint to the starting point.

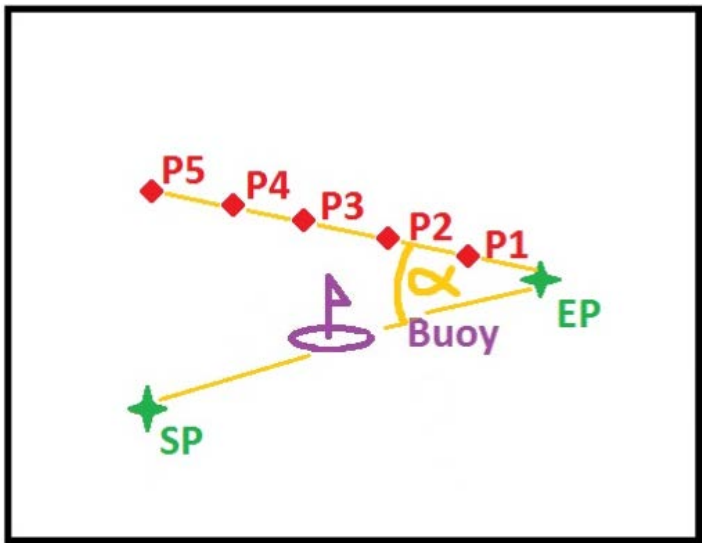


Figure 17: Round Buoy Step 2

There are five points created on this line, all of them are potential waypoints. An algorithm calculates the heading towards each waypoint. At the same time, the best angle VI calculates the fastest heading towards each of those points. The point with the smallest difference between actual direction and fastest direction will be set as waypoint.

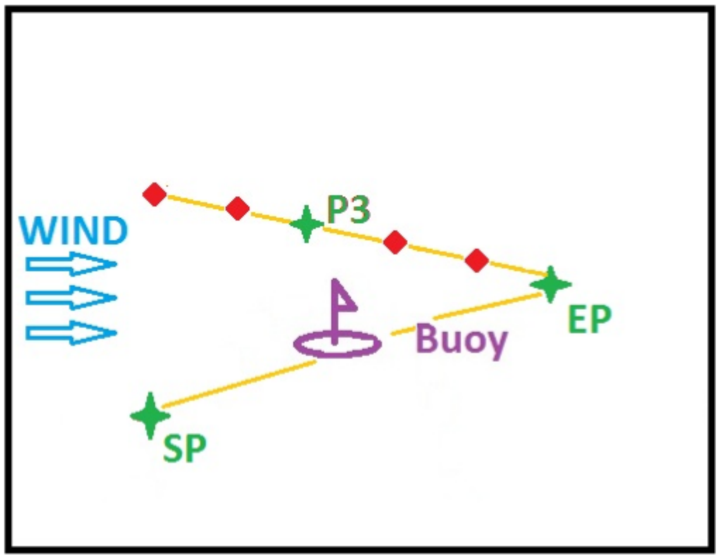


Figure 18: Round Buoy Step 3

While building up this system, we also had the idea to include the heading towards the next waypoint as a factor for choosing the angle of the line. This was not efficient though, as this heading was often far too steep, causing the boat to go a longer path than necessary. Even though the boat speed would be optimized in this scenario, the overall time for rounding the buoy was unpredictable and often times longer than in the running version.

This VI continuously recalculates the waypoint until it reaches the starting point. This ensures that the waypoints change if the wind changes before the rounding started. After the starting point has been reached, the CI stops recalculating its waypoints, to avoid changing the points while trying to cross it (might end up in a loop or getting further away from the waypoint).

End Condition

The Round Buoy VI creates an array with three waypoints: The Startpoint, the calculated crossing point and the Endpoint. This array is inserted into the waypoint array. After this array is created, the VI will be terminated and waits for its next call.

Pass Between Buoys

In charge of this section: Andrew

Passing between buoys is a mission in which the boat passes between two buoys in a specified direction. This is a subtask of the Navigation Test and the Long Distance Course. On the Long Distance Course, the distance between buoys is 40 meters, making the task easier. The Navigation Test employs buoys that are spaced 3 meters apart. With the limited accuracy of our GPS, which can drift up to seven meters, this will be difficult. Additionally, we have to ensure that, no matter the conditions, we follow a path between the buoys. We use a set of three waypoints to more clearly define the safe path through these buoys. The path is shown in Figure 28. While a simple, straight path, we have to consider the distances between the waypoints, and the radii of the tolerances. This avoids the boat trying to travel around the wrong side of the buoys, because of a strange, but optimal, path. We choose relatively tight tolerances for the first two waypoints, to ensure a straight line. However, the third waypoint is easy to check off, since it is positioned after the passing. We also incorporate small No-Sail Zones in to the waypoints, to avoid a collision with the buoys themselves, and to further encourage the boat to aim for the center.

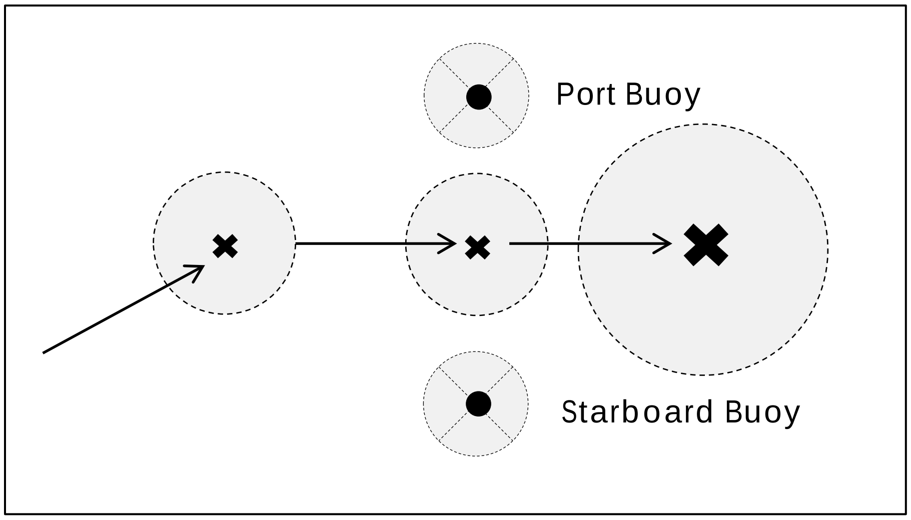


Figure 19: Path used for the boat to pass between two specified buoys. The waypoints are approximately eight meters from each other.

Station Keeping

In charge of this section: Andrew

Station Keeping is an entire event in the competition, and is also structured as a single, integrated mission in our code. The event involves sailing inside a 40 meter square boundary for five minutes, and leaving the boundary as soon afterwards as possible. The coded mission is divided into three phases. First, we simply try to stay in the station. Second, as the end nears, we continue this goal, but put ourselves in a position more conducive to quickly leaving. Third, with seconds to go, we leave the station as quickly as possible. Each of these phases has its own challenges and considerations, described below.

Phase 1. Stay in the Station
The first step of station keeping is simply to stay inside the box. To achieve this, we continue to give the boat waypoints that keep it in control and away from the edges of the box. We experimented with many patterns, but chose a single waypoint as our pattern, because our boat was successful at inventing its own pattern to continue to track the same waypoint. The pattern is shown in Figure 29. Experimentally, this pattern worked better than using a complicated set of waypoints with theoretical benefits, such as minimizing tacks. The waypoints simply the midpoint shifted upwind by five meters, so that if the boat does lose control, it has more time to recover before it is pushed downwind out of the box. Each of our waypoints also incorporates No-Sail Zones (section 1.6.1), which keep the boat from taking paths outside of the box to the next waypoint. This cycle continues, until there are 20 seconds remaining, at which point, the boat begins Phase 2.

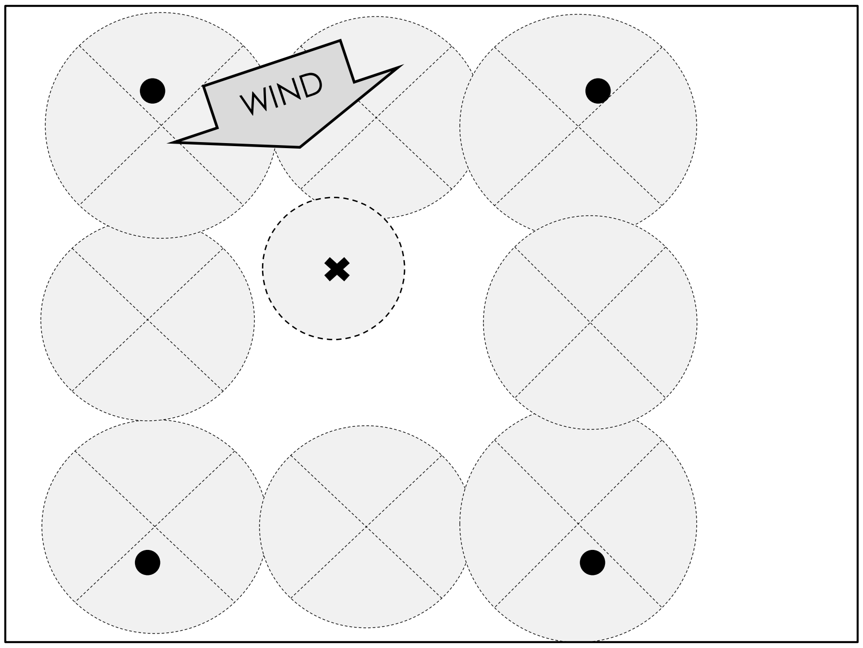


Figure 20. A map of Phase 1 of the Station Keeping Mission, including four boundaries, two waypoints, and No-Sail Zones near the boundaries.

Phase 2. Prepare to Leave
Our strategy for preparing to leave the station is to sail along one of the boundaries of the box, so that we do not have far to travel when we want to leave. The challenge here is deciding which edge to follow, and which direction to follow it. We start with eliminating the two edges that are more downwind, again so that we do not accidentally drift out. In Figure 29, this would be the bottom and right edges. To decide between the remaining two, we look ahead to which direction we would travel along them, and compare those paths. We will choose the direction that allows us to move to the next preferred edge if we happen to reach the end of the first edge

during this phase. Knowing this, we can pick the edge that would put the wind more behind us, so that there is no danger of sailing into irons and losing speed and control. With this method, we are able to sail upwind, but at least 45 degree off of irons (since the two options are 90 degrees apart). We also are able to continue sailing on an upwind edge if we need to continue the phase. This phase can go on indefinitely, sailing back and forth along this edge. Since our average speed is 3 m/s, we expect getting to the edge (20 meters) and traversing the edge (30 meters) to take 16 seconds. When only three seconds remain on the five minute timer, the boat begins Phase 3.

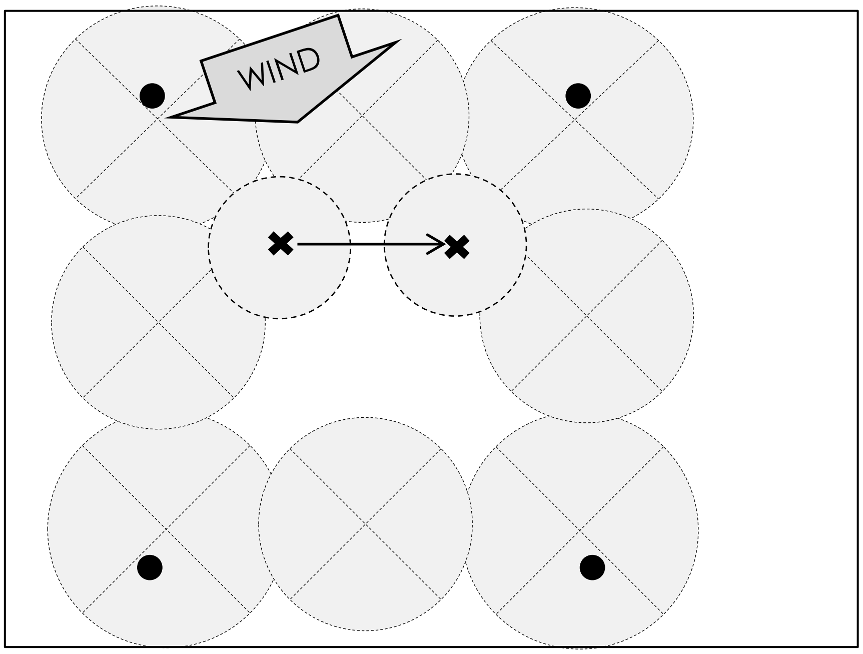


Figure 21. Path of the boat when following the station's edge.

Phase 3. Leave the station

The goal of Phase 3 is to leave the station as soon after the timer as possible. The beginning of Phase 3 is timed well enough so that its goal is to leave as quickly as possible altogether. To do this, we make a waypoint outside of the box, this time without No-Sail Zones. The question of where to put the waypoint is resolved by picking one which results from a heading with the greatest velocity towards the edge. This takes both angle and speed into account. We also include a preference for not tacking. Once we are past the boundary line, we are done with the mission.

5.3 Addition of Non-ideal Elements

In charge of this Section: Andrew

The formulas in Section 4.4 give us the fastest ideal path. However, that path is difficult to stay on, since currents and wind push the boat around, in a non-forward direction. This is called leeway. Additionally, currents affect the max speeds given by the idealized polar plot. Lastly, we have to consider weather helm, which is the tendency of the boat to turn upwind, due to heeling. Where we consider these factors is shown in Figure 32. The details of how we resolve them are discussed below.

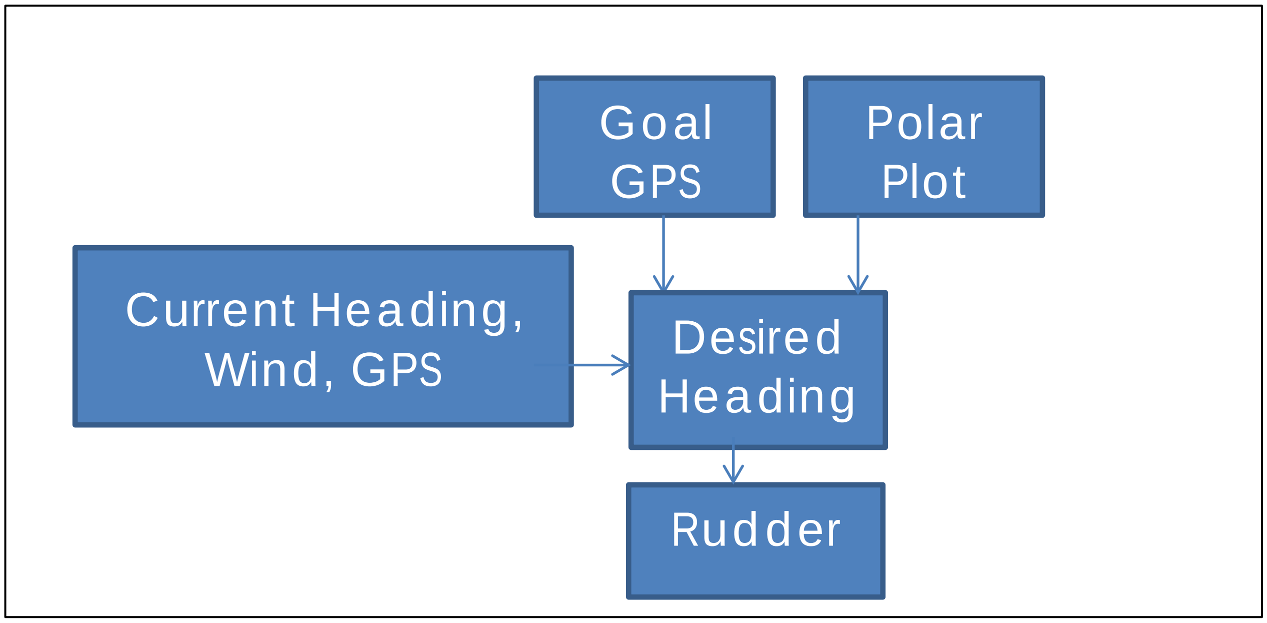


Figure 31. Block diagram showing the elements that determine steering, before non-ideal conditions: leeway, currents, and weatherhelm.

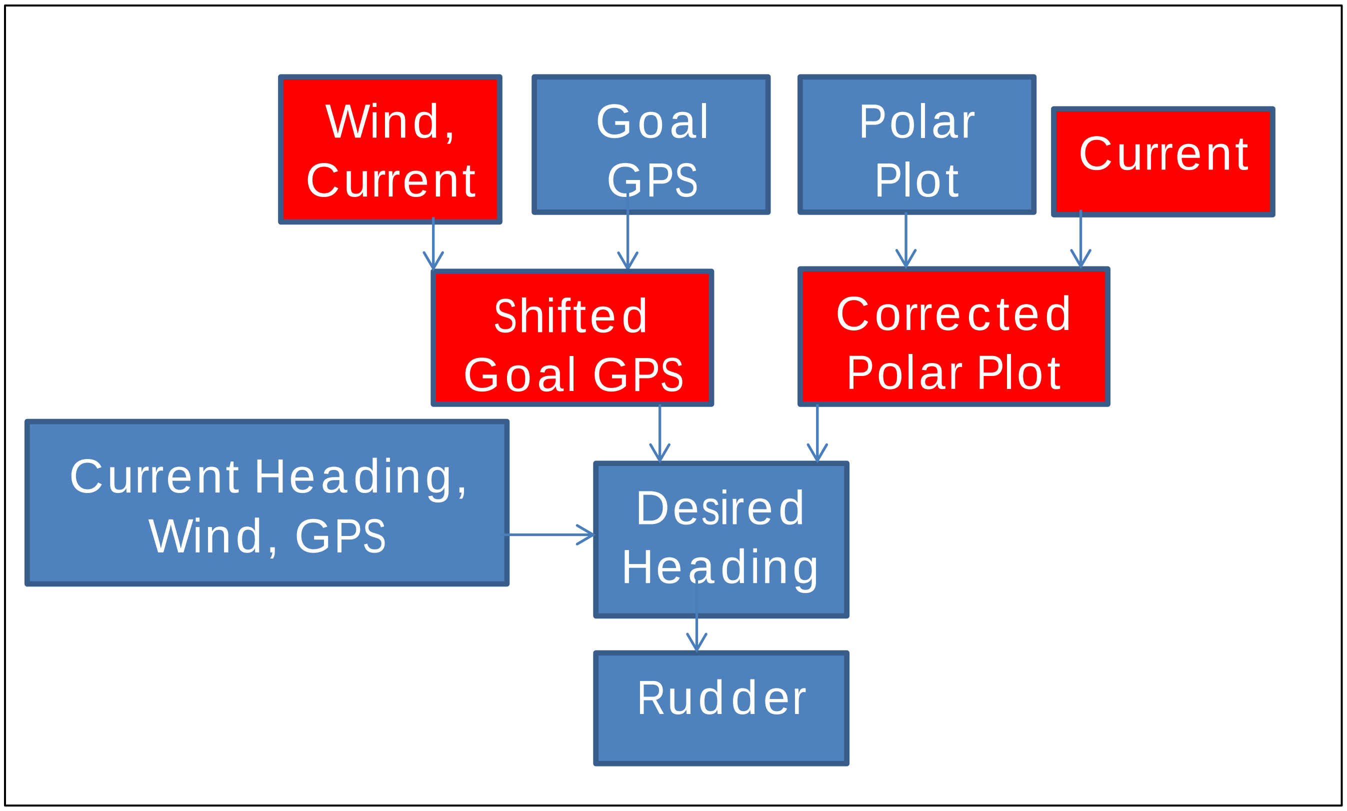


Figure 32. Block diagram showing the elements that determine steering, after the consideration of currents and drift from wind.

Currents affect max speed in any direction.

Because polar plots are affected by currents alone, and not leeway, we cannot change the polar plot, since we do not separate these terms. However, a theoretical solution for the shifted polar plots follows. Water currents affect the speed that the boat can travel by exerting a force on the boat. The speed that the boat can travel in any direction is affected by the current, and so we must consider this before we determine the fastest route to a waypoint. We do this by altering the polar plot that is used in our algorithm. Instead of using absolute wind, in the polar plot, for each point, we add the projection of the current onto that direction with the speed given by wind. However, instead of using absolute wind, we use the wind speed relative to the current. To illustrate this, think of the case where the wind vector is equal to the water current vector. The max speed is not the sum of these, because the wind does not add any force beyond the current.

Currents and wind affect the path you'll end up on.

Not only does the current affect the max speed in the direction that the boat faces, it pushes the boat laterally, so that the course that we map out for the boat is not the real course that the boat takes. Wind has the same effect, as it is able to push the boat sideways in the water, against the drag force of the keel and hull. We can calculate the combined effects of these by measuring our movement angle and comparing it to our heading, or expected movement angle. This will give us only the current that is perpendicular to our heading, but we will see that this is all we require. We can then use this feedback to adjust our plan.

Shift the goal.

The solution to this problem is not to turn the boat to stay on the originally-planned straight-line path. This would change wind angle to the boat, and void the calculations done for path-planning. Instead, we can predict the total amount of boat translation that the wind and currents will cause, by integrating the effect over the time that the calculated path should take. Then, we recalculate the optimal path, using a target that is shifted by this amount. That way, we use the actual angles to the wind in calculating the path.

Error in this approach

The problem with this approach is that the predicted amount of wind and current translation will change when we alter the target and the path. We can repeat this method, using the recalculated path to predict the total translation, until the amounts converge. However, at this point, we only implement the estimation once, as the error is small.

Weatherhelm

Due to heeling and the relationship between center of effort on the sails and the center of lateral resistance, our boat has a tendency to feel a torque which points it more upwind. To combat this, the rudder must be set at a different angle than expected. This is dealt with using a PID controller, described in section 5.

5.4 Navigation (getting to waypoints)

In charge of this Section: Luis

Choosing a path isn’t simple. Up until now, we’ve described our goals, and where we need to go to accomplish them. Were we driving a car, we would be nearly done. However, sailing is a different kind of beast; in this section we describe the logic necessary to cope with the less straight-forward environment on the water.

Elements in Decision Process.

Navigation on water becomes a bit more complicated than something like driving on a road when dealing with wind and sail angles because the shortest path is not always the fastest path. There are 3 primary elements considered in navigating to a waypoint. These are developed into “goodness functions” which are the main vehicle through which the boat heading is chosen. The goodness function paradigm is rooted in fuzzy logic. A goodness function in essence gives the desirability of a given heading with respect to a certain goal, taking a value between 0 and 1 where 1 is the most desirable and 0 is absolutely unacceptable. The benefit of this approach is that multiple unrelated considerations may be taken into account with different goodness functions. Furthermore, because many headings have a calculated goodness, the algorithm is able to select a heading that is the most agreeable to many considerations where a strict logic system may have to deal with conflicting goals in a variety of situations. In particular, we know that the fuzzy logic approach will always give a somewhat desirable heading, even if suboptimal, when faced with conditions that are unexpected or not accounted for. We then give each individual goodness function an importance weight and take the geometric mean to determine which heading is the most agreeable.

The 3 elements our navigation system incorporates are getting to the waypoint as quickly as possible, avoiding no-sail zones, and changing the heading as little as possible. The following sections will discuss each of these elements in depth and show how we develop goodness functions that represent these objectives.

Fastest way to the waypoint.

We start with the observation that there is a pair of optimal angles for the boat to travel upon to reach a waypoint if we assume that there will be no change in the wind conditions. Since we do not have any method of predicting these changes, we will work under the assumption that there will be none. We will correct a bit for this assumption when we consider the heading change goodness function. Once we agree that there is a single pair of optimal angles, these can be found given the polar plot for the boat given the current conditions.

In Figure 22 we draw and label the general representation of this problem. The goal will then be to minimize $\boldsymbol{t = t_\varphi + t_\theta}$, or equivalently, maximize $\boldsymbol{v_{net} = \frac{d}{t_\theta + t_\varphi}}$. We begin by writing the known relationships:

$$\boldsymbol{v_\theta t_\theta \sin(\theta) - v_\varphi t_\varphi \sin(\varphi) = 0}$$

$$\boldsymbol{v_\theta t_\theta \cos(\theta) + v_\varphi t_\varphi \cos(\varphi) = d}$$

Putting the system into matrix form yields

$$\begin{bmatrix} v_\theta sin\theta & -v_\varphi sin\varphi \\ v_\theta cos\theta & v_\varphi cos\varphi \end{bmatrix} \begin{bmatrix} \boldsymbol{t_\theta} \\ \boldsymbol{t_\varphi} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{d} \end{bmatrix}$$

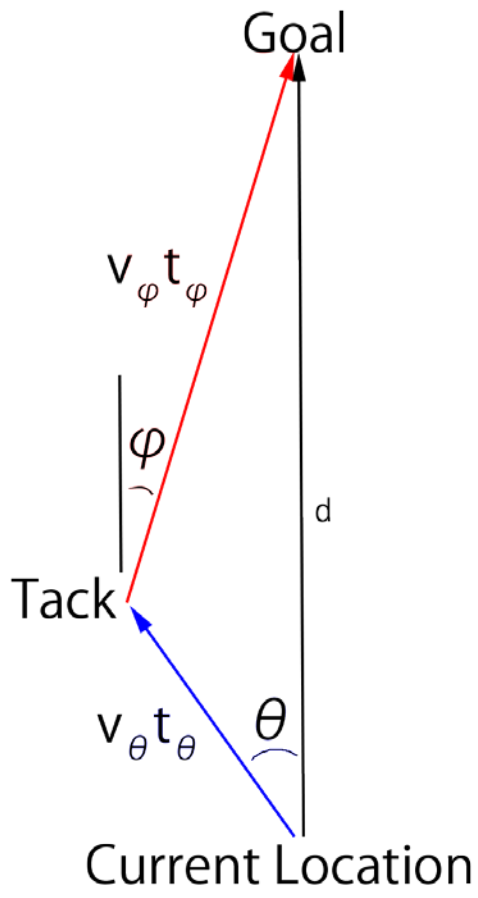


Figure 22: Pictorial representation of waypoint navigation problem

Solving this system is easy using Cramer's Rule and gives us:

$$t_\theta = d * \frac{sin\varphi}{v_\theta * \sin(\theta+\varphi)}$$

$$t_\varphi = d * \frac{sin\theta}{v_\varphi * \sin(\theta+\varphi)}$$

Solving for v_{net} gives:

$$v_{net} = \frac{v_\theta v_\varphi \sin(\theta+\varphi)}{v_\theta sin\theta + v_\varphi sin\varphi}$$

In order to define a goodness function that encompasses this equation and will attempt to maximize v_{net}, we use the softmax function, defined for a series of values x_1 to x_n as: $softmax(x_1, x_2, \ldots, x_n) = \log_b \sum_{i=1}^{n} b^{x_i}$. For any given angle θ, we will take the softmax of the net velocity over every φ, 0<φ<90 in combination with the given θ. The intention here is that an angle with a high goodness will be part of a pair of angles with a high net velocity and will have a high net velocity when paired with many angles. Finally, we turn this value into a goodness from 0 to 1 by taking 1-exp(-softmax/k) where k is a tuning constant in units of velocity to normalize our calculated value. This goodness function should then output a high goodness when the tested heading angle has complimenting pairs with a short theoretical time to the waypoint.

Avoiding No-Sail Zones

In order to avoid no sail zones and encompass this task in a goodness function, we begin by considering each no sail zone separately. We will then combine the goodness functions given by each no sail zone by simply taking the product. Now considering a single no sail zone, we define the goodness function to be 1 at all angles that are more than 90 degrees away from the direction of the no sail zone. In other words, any angle that does not decrease the distance to the no sail zone is perfectly good. From 0 to 90 degrees (0 being the exact direction of the no sail zone), we define the goodness function like so:

$$goodness(z) = 1 - e^{-\frac{1}{k} * \frac{d_z - r_z}{90 - \theta}}$$

where d_z is the distance to the center of the no sail zone 'z', r_z is the radius of the no sail zone ($\therefore d_z - r_z$ is the minimum distance to the no sail zone), and θ is the angle between the tested heading and the center of the no sail zone with respect to the current position. Finally, k is a tuning constant in units of length to normalize our formula.

Qualitatively we can see that as the distance from the no sail zone increases, the goodness function increases for all potential headings and that as the heading moves toward directly at the no sail zone (θ becoming smaller) the goodness function decreases. Furthermore, as soon as the boat is right up to the radius of the no sail zone, the goodness function becomes 0 for all headings that are not away from the no sail zone; this is desirable so that if the boat does come near a no sail zone, it will inherently be top priority to move away from the no sail zone. Given this, it is simply a matter of tuning the constant k such that we are not too sensitive nor not sensitive enough to the no sail zones.

Keeping the Current Heading.

The final element in navigating to a waypoint is simply that we would like to turn the boat as little as possible. This serves two purposes: the first is to desensitize the targeted heading to small changes in the goodness functions. This is important so that the boat keeps from overreacting in short periods of time and will generally cause the boat to change headings more gradually than all of a sudden.

Secondly, when heading upwind, this goodness function determines the frequency the boat can tack at. Presumably, heading upwind will yield two peaks in the goodness function for the fastest way to the waypoint. If we did not consider the importance of maintaining the current tack whatsoever, the boat would tack infinitely frequently, essentially approximating a straight line to the waypoint. However, if we weight headings that are closer to the current heading of the boat, the boat will hold a tack until the second heading is more desirable than the current heading by a ratio of at least the goodness difference defined by the current heading goodness function. This function is simple to define given the current heading 'h':

$$goodness = e^{-\left(\frac{\theta - h}{k}\right)^2}$$

where again k is a tuning constant with units of radians. This gives us a goodness of 1 at the current heading and decreasing goodness as we move away from the current heading.

Combining Goodness Functions.

The last step in determining the desired heading is the combination of the goodness functions described above and selection given this combined curve. The combination is actually quite simple, it is a weighted geometric average of the goodness functions. We use the geometric average as opposed to the arithmetic average primarily because if a given goodness function determines that a certain heading has a goodness of 0, we want to avoid this heading at all costs, and this is not guaranteed with the arithmetic average.

Formally, this combination looks like:

$$combined\ goodness = ({g_1}^{w_1} * {g_2}^{w_2} * {g_3}^{w_3})^{\frac{1}{w_1+w_2+w_3}}$$

where g_1, g_2 and g_3 are the three goodness functions and $w_1, w_2\ and\ w_3$ are the respective importance weights. Once we have calculated this combined goodness curve, we select the heading with the highest combined goodness at which point it becomes the desired heading which the boat will try to achieve.

Getting wind data while the boat is running

In charge of this section: Olli

The wind sensor is mounted on the bow. As a result, while the boat is on a run the sensor is obscured by the sails and cannot get proper wind data. After reading out the logfiles from testing, we saw that the apparent wind directions appears to come from around +-45 degrees in those cases (wind blows around the sails). The wind speed also varies under these conditions. This wind data could not be used for sailing, and the boat would start to become unstable as soon as it receives the proper wind data.

The solution for this issue is to recognize when the boat is on a run. Before the wind data becomes wrong, it would be replaced by a calculated approximation of the current wind data. The boat would switch back to sailing with the sensed wind data as soon as the boat changes its heading by a predefined range. An additional feature is, that a “convincing range” would terminate this wind data override if the wind is coming from the completely different direction. This prevents the boat from sailing into irons in case the wind direction changes completely.

The functionality of checking the wind range values, calculating the override values for the wind data and activating the override is split up into three Vis.

The first VI is run with every Think Loop iteration. It first compares the wind direction with the robot’s heading to determine if the robot is getting into the run range. If this is not the case, the current wind data values get written into an array. This array saves the wind data of the last five seconds. As soon as the boat is getting into the run range, a Boolean is set and the array is not filled further. Instead, the second VI of the so-called “Runprotect” function is run.

This second VI calculates an approximation of the wind data. It reads the previously described array and generates this value. Two global variables for wind direction and wind speed are set.

The third VI reads out those global variables. It replaces all previous consumers of the Apparent Wind and Absolute Wind data. Thus, while the boat is on a run, any VI that uses wind data will instead receive “runprotected”, simulated data.

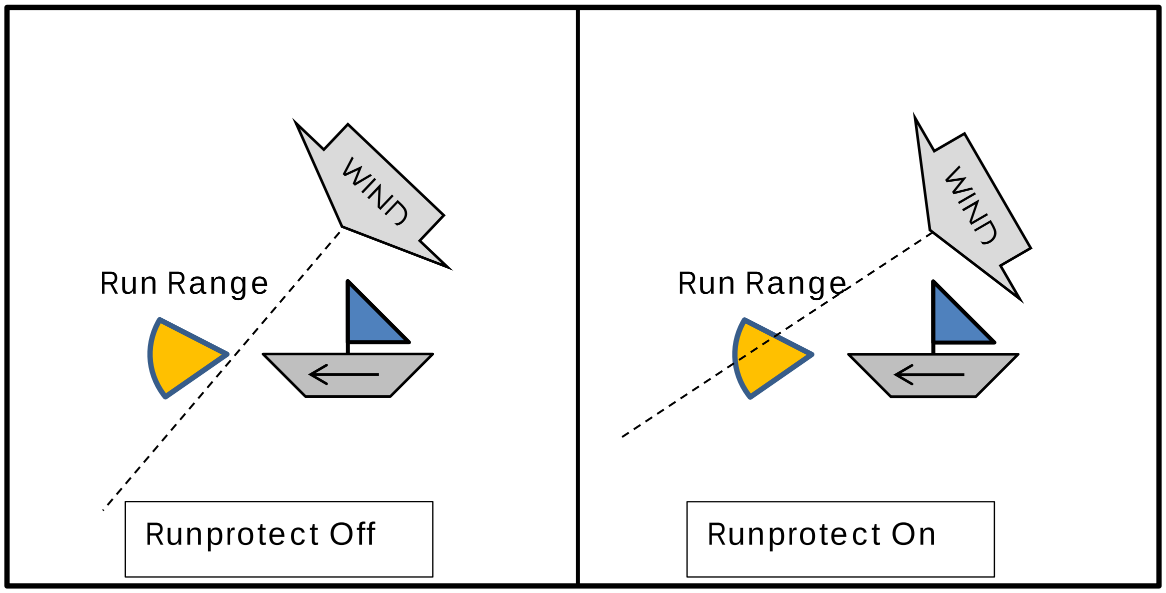


Figure 23: The Runprotect Status

6 Act

In charge of this section: Elizabeth and Olli

There are two main actuators on the competition boat: The Sailwinch and the rudder servo. Both are controlled via PWM signals from our sbRIO. We had to realize the following functions:

- Rudder control with predefined max range
- No acceptance of out-of-range values for the motors
- Fine-adjustable neutral rudder position
- Maximum Sailwinch range

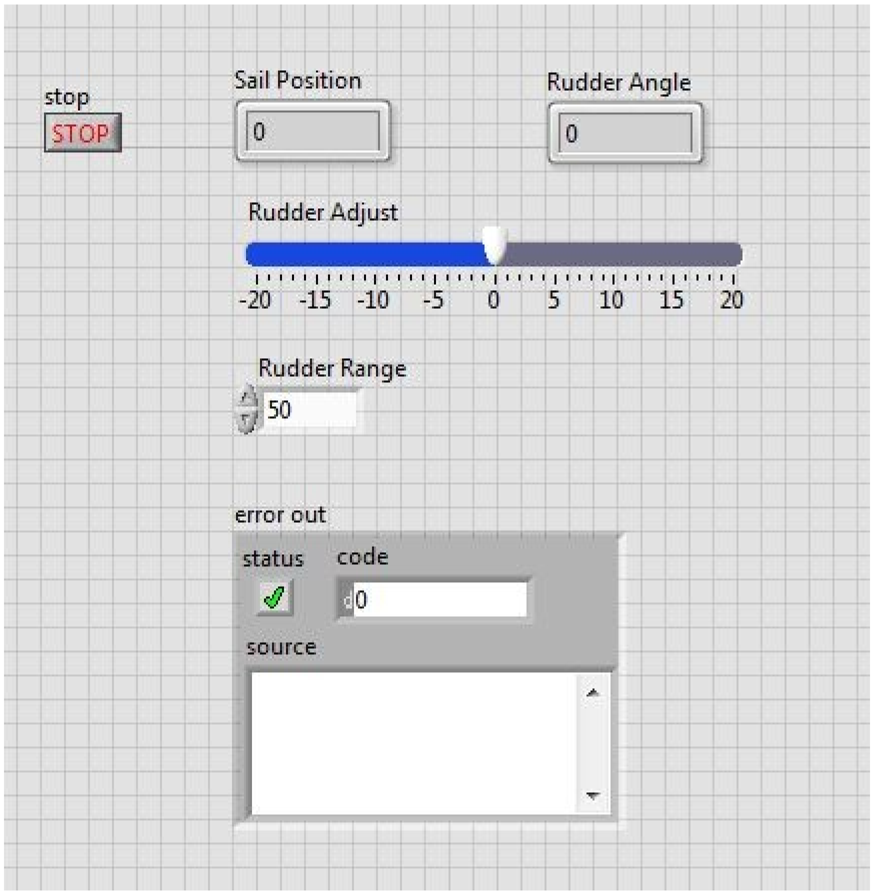


Figure 24: The Motor PWM Control VI

Rudder Control

In charge of this section: Olli

The rudder control function is a PID controller that creates a rudder angle out of the current heading and the desired heading. The PID is using hardcoded PID constants. The Rudder Control is communicating with our PWM Rudder Control, which allows us to adjust a rudder trim in case the rudder is off centered. It is important to mention, that the Don’t Flip VI does not override the Rudder control. Even if we are in the Don’t Flip state, the PID controller is steering to its desired position.

The PWM Rudder Control VI also allows us to set a maximum range. All values that are out of range will be coerced, or replaced with the closest value in range.

Sail Control

In charge of this section: Elizabeth and Olli

At the heart of our sail control function is a lookup table which has a hardcoded value for how much to let out the sail depending on what angle we are to the wind. This was done because a lookup table is easier to manipulate than a function to calibrate to many varying situations (different sail sizes, different mass distributions of the boat, different wind speeds), and because few functions would fit the qualitative shape that we need. At the moment, that shape is to keep the sail in from 0 degrees to 45 degrees, then to let it out in a linear fashion from 45 degrees to 90 degrees, at which point we would have the sail halfway out. From there, it again linearly increases until it is fully out when the wind is at our backs, at 180 degrees. In addition to this lookup table, we test whether the boat is crossing the wind, and if it is, we let the sail out, so we are less likely to slow down or be blown off course when changing our tack through the wind.

All this is only true when the boat is making its way toward the goal we set for it. If it determines that tipping over is an imminent problem, it will check whether we are heading into or away from the wind. If we are heading into the wind, it will pull the sail in, and if we are heading away from the wind, it will let the sail out.

7 Logging

In charge of this section: Elizabeth

When the boat is running autonomously on the water, we cannot always see what is going on inside it or the code. This is especially true when we lose connection to it, which happens on occasion. However, in order to find out what happened if we have a problem, we need to see the data. To solve this problem, we have implemented a data logging system that allows us to see the boat's status at a given time after the fact. In addition, finding out how the boat responds to various circumstances allows us to improve the response if we are unsatisfied with its performance.

On the implementation level, the logger is implemented as a class, which prevents non-logging code from interfering with its operation. This helps standardize the way things are written to the log files and prevents the log files from being opened multiple times or other such memory issues.

In addition, instead of writing directly to the file every time we want to log data, we have a subroutine run in the background to write data to the files. More specifically, this subroutine stores log messages in an internal queue (memory buffer) before writing to the disk. By writing multiple log messages to file at once rather than each time the logger is called, the number of computationally expensive disk writes is limited. This allows logging functionality to be called within time-critical loops, such as our mission execution code.

We have two different logs, both of which operate on the same code but write different information to different files.

Status Logging

The first log we have is a data logger, which logs the status of the boat once every second. Examples of the types of information we log are the wind direction and speed, how fast the code is moving, our network status, and what commands the code is outputting to the boat. Logging these allows us to determine whether the boat is responding correctly to external stimuli and the limits of our connectivity to our boat. In addition, we can later look over how the boat responded to various stimuli and change how we want to respond in order to improve our performance. It logs by pulling various pieces of data off the global, formatting them into a text string, and saving it to the file.

Message Logging

The other log we have is a message logger, which logs messages about the status of the boat when we tell it to, as well as the time at which that message occurred. For instance, the operator of the boat can type in their own message – perhaps “It started raining”, and the log would note that fact and when it happened. This could help us understand if we have drastic changes in our sensor data by noting things that happen that the boat is incapable of knowing about. It also has proved to be useful for taking notes while testing for everyone to see and remember what problems showed themselves. In addition, the boat itself notes certain pieces of information in the message log. Currently, it logs when it starts a mission, what type of mission it is, when it checks off a waypoint, and when it completes a mission.

Status Log Processing

We log far too much information far too often for the logs to be easily read by humans (though it can be done), so we have a separate piece of log reading code. What this code does is it pulls the file and looks for the beginning of each time we wrote to the file. From there, it reverses the encoding we used to put the data into text to get the data back out, and then iterates through every time we wrote to the file and graphs the pieces of data.

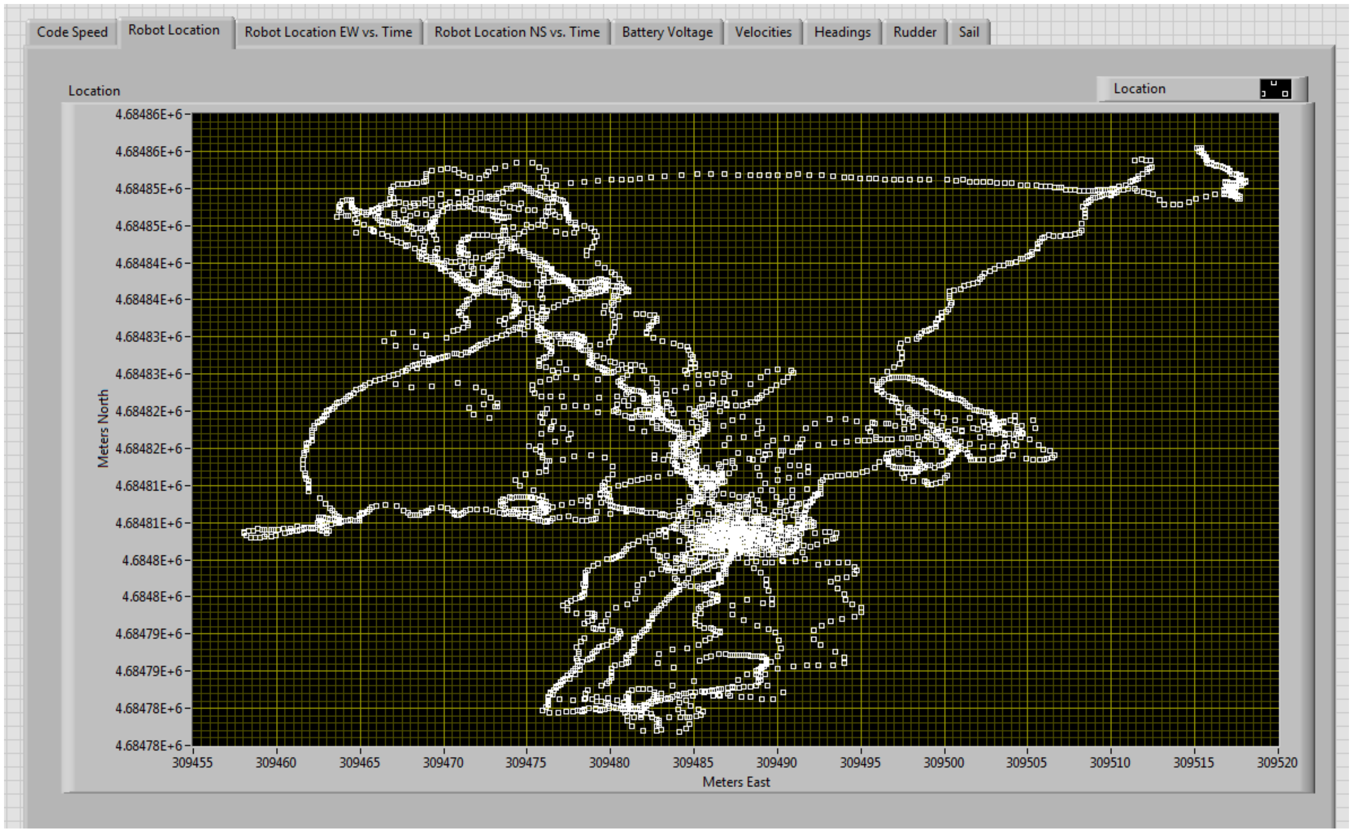


Figure 25: Data logger graphing position data.

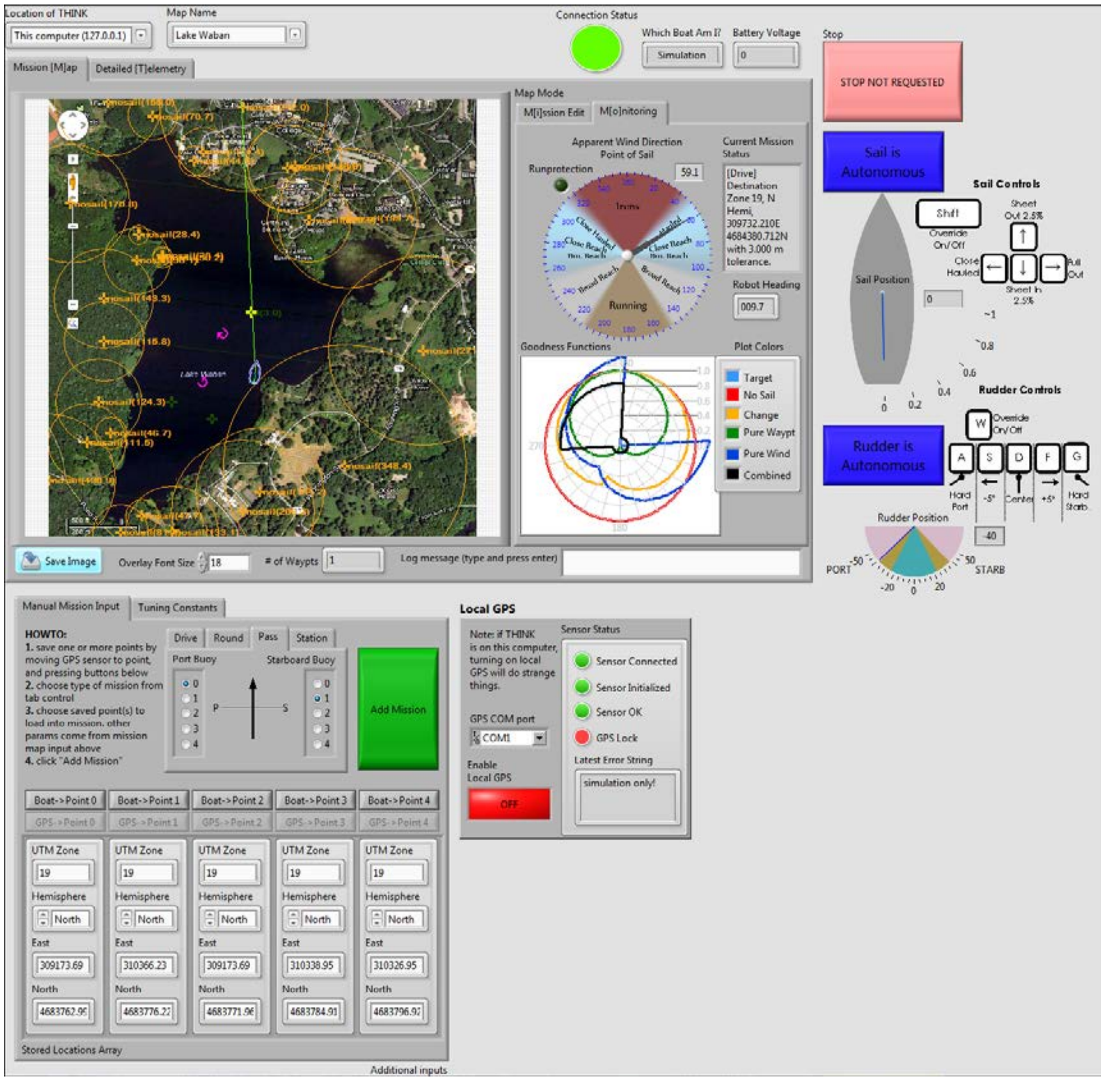
Message Log Processing

The message log, while smaller, can also generate a large amount of information when run for long periods of time, and intersperses mission and user messages indiscriminately. To make it easier to read, we also have code that processes the message log. It goes through the log line-by-line and checks what type of message each message is. It then sorts the messages into either user messages or mission-related messages, and if it is a mission-related message determines how long the mission ran and whether or not it was aborted by the user partway through.

8 Operator Control Unit (OCU)

In charge of this section: Jason

In order to facilitate control, debugging, and tuning of the sailbot code, the team has built a multi-featured OCU that provides easy access to several useful functions.



Location of THINK: This computer (127.0.0.1)

Map Name: Lake Waban

Which Boat Am I?: Simulation

Battery Voltage: 0

STOP NOT REQUESTED

Sail is Autonomous

Rudder is Autonomous

Apparent Wind Direction: 59.1

Current Mission Status: [Drive] Destination Zone 19, N Hemi, 309732.210E 4684380.712N with 3.000 m tolerance.

Robot Heading: 009.7

Rudder Position: -40

Overlay Font Size: 18

# of Waypts: 1

HOWTO:
1. save one or more points by moving GPS sensor to point, and pressing buttons below
2. choose type of mission from tab control
3. choose saved point(s) to load into mission. other params come from mission map input above
4. click "Add Mission"

Note: if THINK is on this computer, turning on local GPS will do strange things.

GPS COM port: COM1

Enable Local GPS: OFF

Latest Error String: simulation only!

	Point 0	Point 1	Point 2	Point 3	Point 4
UTM Zone	19	19	19	19	19
Hemisphere	North	North	North	North	North
East	309173.69	310366.23	309173.69	310338.95	310326.95
North	4683762.96	4683776.22	4683771.96	4683784.91	4683796.92

Figure 26: Full OCU interface.

Monitoring

In charge of this section: Jared

The ability to monitor the information which affects the decision-making of the boat is critical for debugging during system development and for effective operation during deployment. For example, if the boat is behaving erratically, the ability to monitor the sensor data acquired by the boat could indicate the source of the problem. It is also possible that the source of the erratic behavior is not the data, but the response of the boat, in which case the operator can use the sensor data to effectively control the sailbot with manual override.

Monitoring is accomplished by packing up all of the information relevant to the user on the sailbot and passing it to the OCU using our network stream communication protocol. The OCU acquires the most recent monitoring data sent by the sailbot, performs any necessary processes, and then displays the information to the operator.

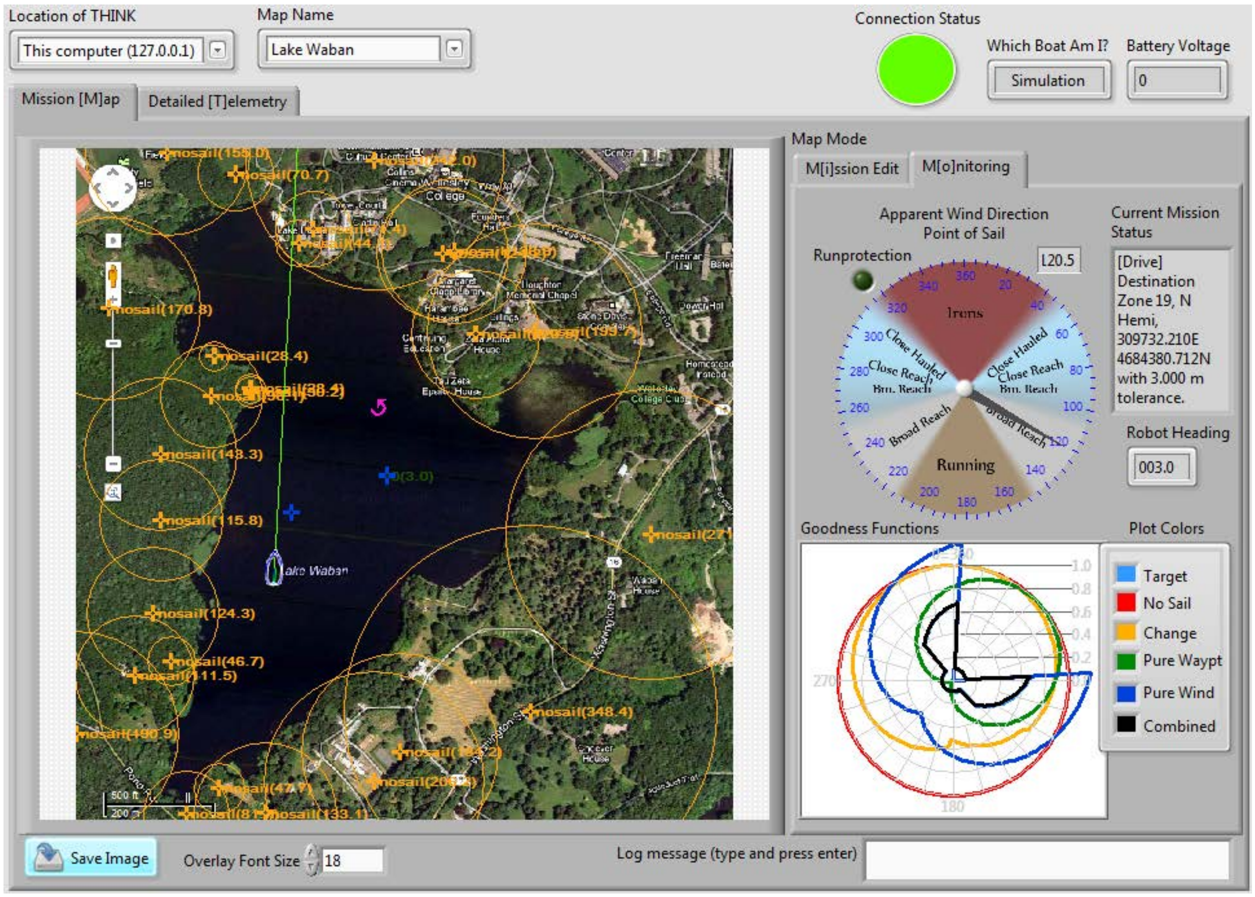

Location of THINK: This computer (127.0.0.1)
Map Name: Lake Waban
Connection Status
Which Boat Am I?: Simulation
Battery Voltage: 0
Mission [M]ap | Detailed [T]elemetry
Map Mode: M[i]ssion Edit | M[o]nitoring
Apparent Wind Direction Point of Sail
Runprotection
120.5
Current Mission Status: [Drive] Destination Zone 19, N Hemi, 309732.210E 4684380.712N with 3.000 m tolerance.
Robot Heading: 003.0
Goodness Functions
Plot Colors: 🟦 Target, 🟥 No Sail, 🟧 Change, 🟩 Pure Waypt, 🟦 Pure Wind, ⬛ Combined
Save Image
Overlay Font Size: 18
Log message (type and press enter)

Figure 27: the OCU in mission monitoring mode showing a simulated mission. The current mission is visualized, along with key data such as the goodness functions currently being considered. More detailed telemetry is available on a second tab.

Networking

In charge of this section: Jared

Network connections to-and-from the sailbot are implemented as reader and writer classes where each instance maintains its own separate channel of communication. Fundamentally, both the reader and the writer are Network Streams, a TCP/IP communication protocol written by National Instruments specifically for streaming high-throughput data. The reader and writer classes wrap around the Network Streams to provide the following additional functionality:

- Automatic packet time-stamping
- The ability to abort connection attempts with no finite timeout

- The ability to read either the most recent or the next packet received from the writer
- Persistent connections which can be re-established if the connection is lost or after the reader or the writer application is aborted and re-started

As with the logging functionality, the networking functionality has a subroutine which runs in the background and manages the network communication. Function calls to the writer store data in an internal queue which is sent across the connection by the background subroutine; function calls to the reader pull data from an internal queue which is written to by the background subroutine. This allows time-critical loops to call the networking functionality, as networking is a non-deterministic process.

Manual Override

In charge of this section: Jason

The OCU is also equipped with a manual override control. The user can take manual control of the rudder and sails individually, or both together. When manual override is activated, keyboard input defines rudder and sail setpoints that are sent do the boat.

We have also created a keyboard interface for the manual override which allows the operator to interface with the OCU without looking at the screen or using a mouse.

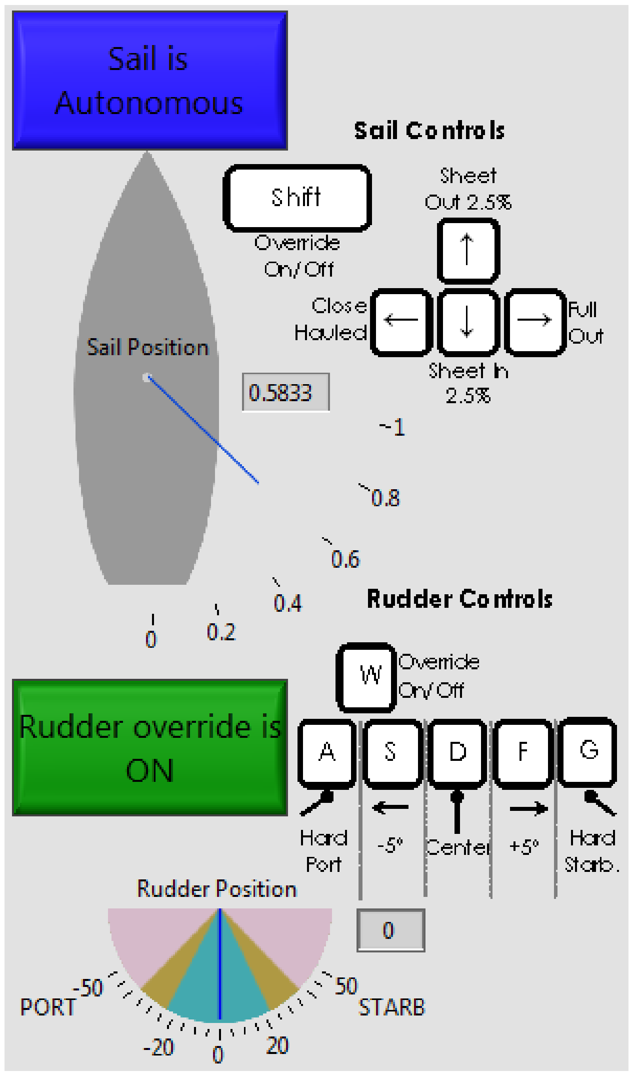
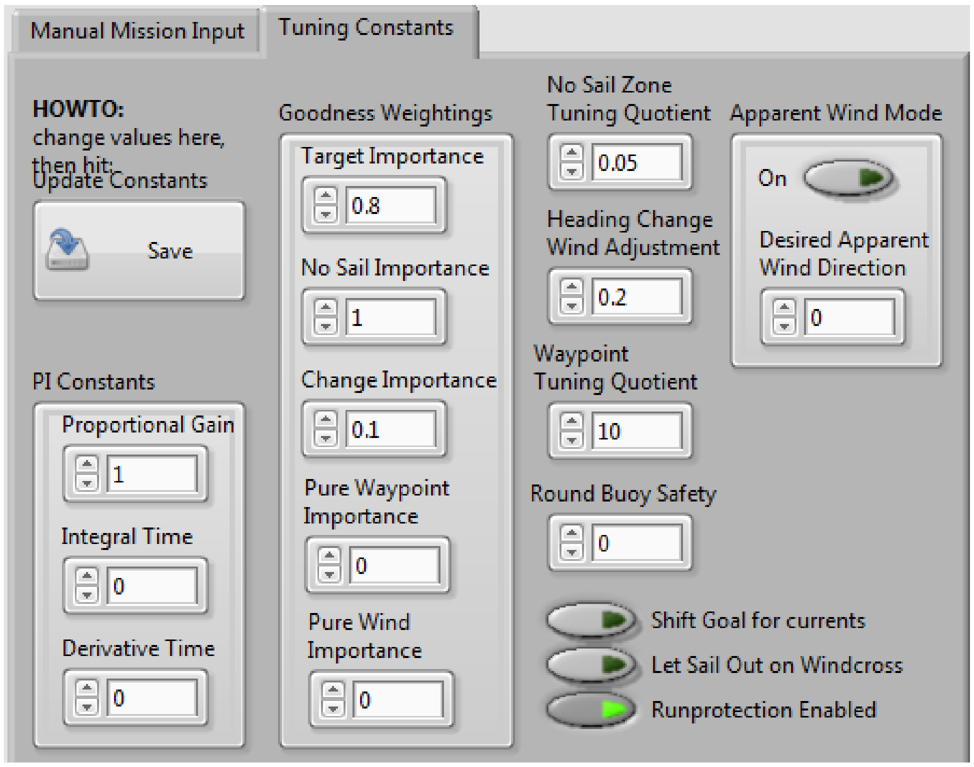


Figure 28: Graphical rudder and sail position indicators on the OCU

Tuning

The OCU also provides an interface for tuning constants used internally on the sailbot for navigation. These include the weightings for our fuzzy-logic goodness functions as well as various features we have implemented. This has allowed us to test different configurations, isolate problems and tune key parameters on-the-fly.


Manual Mission Input | Tuning Constants

HOWTO:
change values here,
then hit:
Update Constants

Save

PI Constants
- Proportional Gain: 1
- Integral Time: 0
- Derivative Time: 0

Goodness Weightings
- Target Importance: 0.8
- No Sail Importance: 1
- Change Importance: 0.1
- Pure Waypoint Importance: 0
- Pure Wind Importance: 0

No Sail Zone Tuning Quotient: 0.05

Heading Change Wind Adjustment: 0.2

Waypoint Tuning Quotient: 10

Round Buoy Safety: 0

Apparent Wind Mode
- On
- Desired Apparent Wind Direction: 0

Shift Goal for currents

Let Sail Out on Windcross

Runprotection Enabled

Figure 29: OCU tuning constants interface.

Mission Map

In charge of this section: Jason

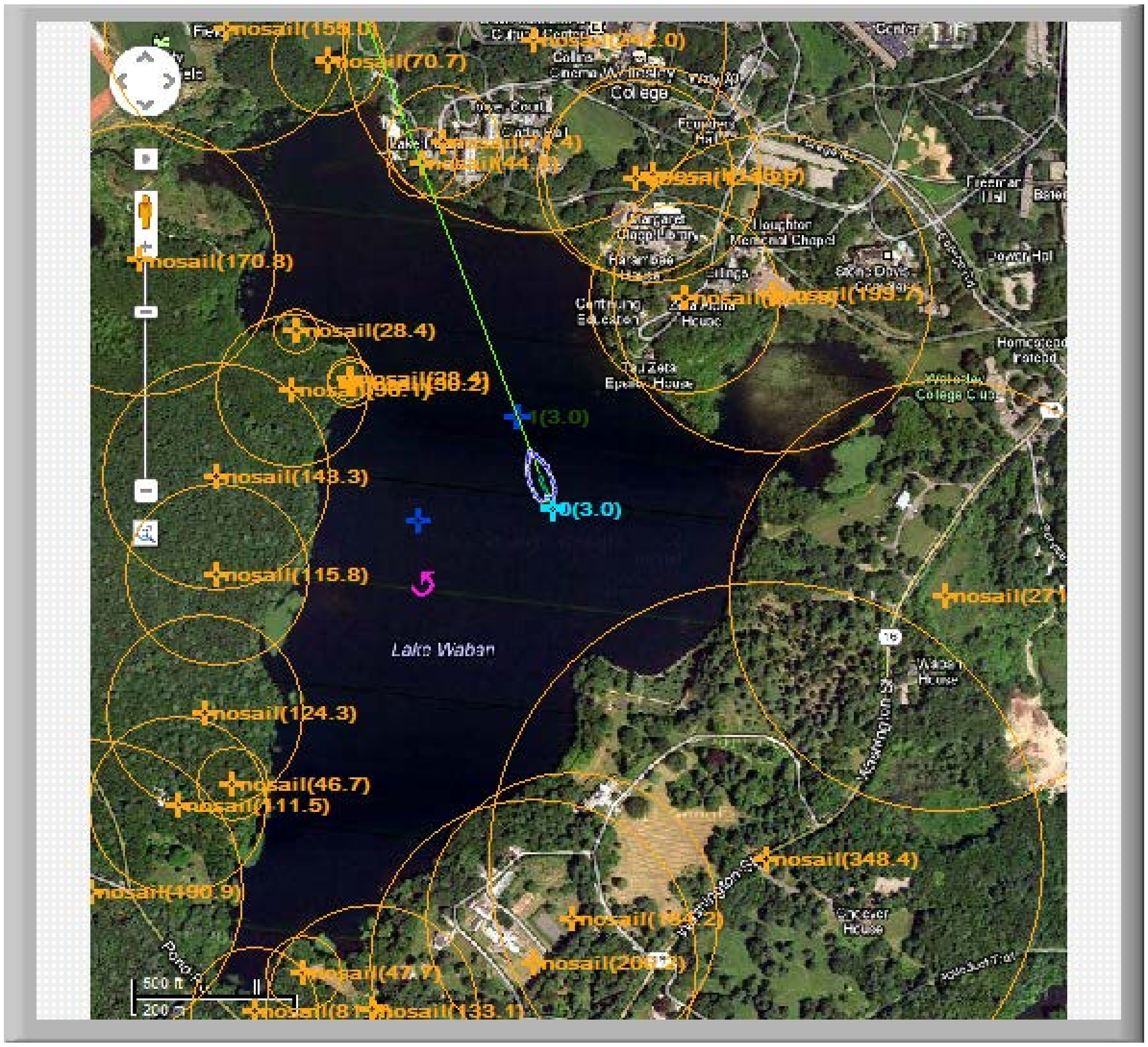


Figure 30: The OCU mission map, showing a typical test consisting of global no-sail zones, waypoints, and a round-buoy mission. The boat is located near the center, facing north-northwest, which is in agreement with the green "desired direction" vector. The light-blue waypoint has already been checked off. The parenthesized numbers indicate no-sail zone sizes and waypoint tolerances.

The mission map (shown in Figure 30) is a core feature of the OCU. It provides an indicator of the geographical location of the sailbot, as well as information about assigned missions and no-sail zones. The map also indicates the heading of the boat, its sail position, and displays the desired heading of the boat as a vector. All of this has been designed to provide key information about the mission status at a glance.

Mission Input

In charge of this section: Jason

The OCU provides 3 methods of input: Mission Definition File (MDF) loading, mission map input, and GPS input.

MDF Loading

Any mission created on the OCU can be saved as an MDF, or Mission Definition File. Our MDFs use a combination of binary and XML data to store copies of our mission objects and no-sail zones for later retrieval. Upon retrieval, the missions are sent to the boat immediately.

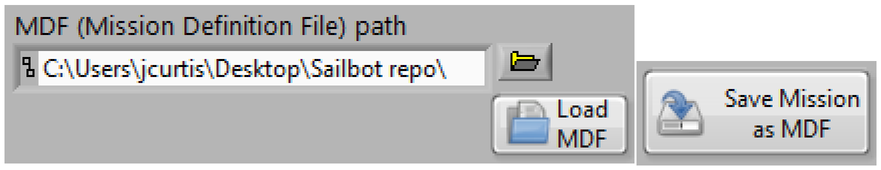
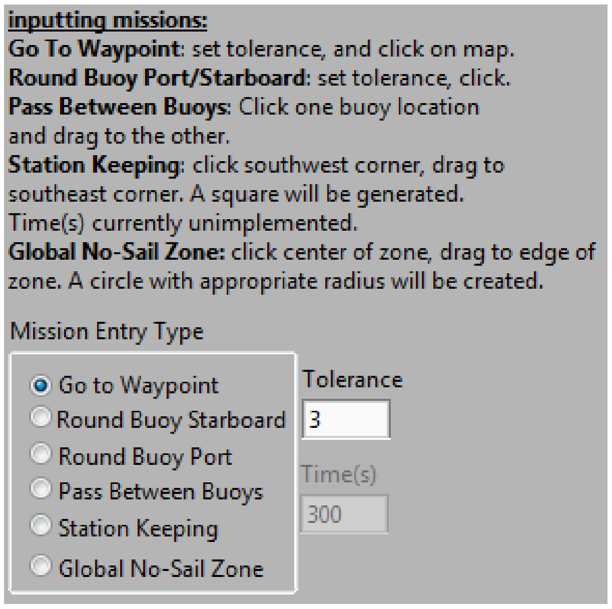

MDF (Mission Definition File) path
C:\Users\jcurtis\Desktop\Sailbot repo\
Load MDF
Save Mission as MDF

Figure 31: The MDF loading and saving interface elements.

Mission Map Input

The OCU also has an interface that allows the user to input any type of mission by clicking on the mission map:



<u>**inputting missions:**</u>
Go To Waypoint: set tolerance, and click on map.
Round Buoy Port/Starboard: set tolerance, click.
Pass Between Buoys: Click one buoy location and drag to the other.
Station Keeping: click southwest corner, drag to southeast corner. A square will be generated. Time(s) currently unimplemented.
Global No-Sail Zone: click center of zone, drag to edge of zone. A circle with appropriate radius will be created.

Mission Entry Type
- [✓ Go to Waypoint]
- Round Buoy Starboard
- Round Buoy Port
- Pass Between Buoys
- Station Keeping
- Global No-Sail Zone

Tolerance: 3

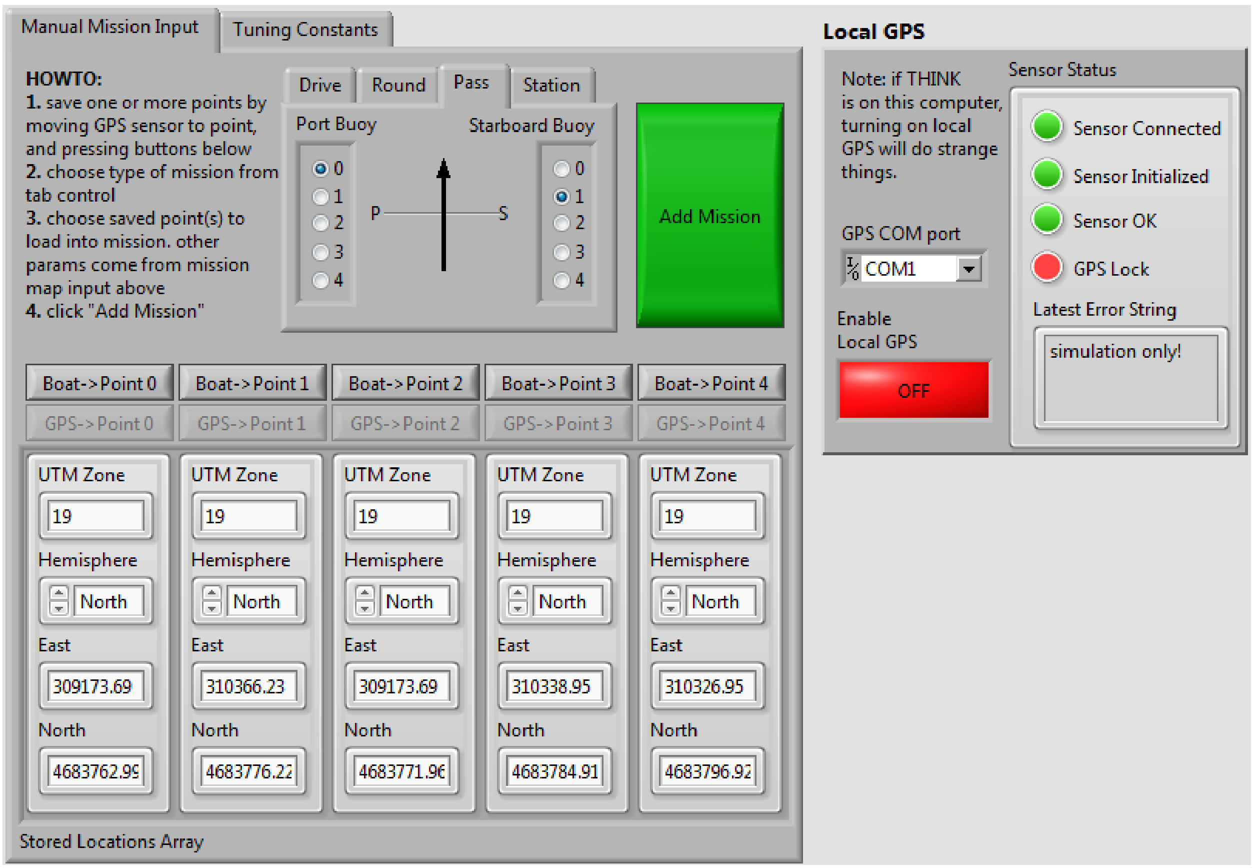
Time(s): 300

Figure 32: Mission Map input panel instructions

This provides an easy way to input a set of missions when exact placement is not necessary.

GPS Input

For more precise mission input, for example input of missions by tagging buoys, a GPS-coordinate-based mission input interface is also exposed on the OCU (Figure 33).



HOWTO:
1. save one or more points by moving GPS sensor to point, and pressing buttons below
2. choose type of mission from tab control
3. choose saved point(s) to load into mission. other params come from mission map input above
4. click "Add Mission"

Port Buoy: [✓ 0] 1 2 3 4

Starboard Buoy: 0 [✓ 1] 2 3 4

UTM Zone	UTM Zone	UTM Zone	UTM Zone	UTM Zone
19	19	19	19	19
Hemisphere	Hemisphere	Hemisphere	Hemisphere	Hemisphere
North	North	North	North	North
East	East	East	East	East
309173.69	310366.23	309173.69	310338.95	310326.95
North	North	North	North	North
4683762.9	4683776.2	4683771.9	4683784.91	4683796.92

Stored Locations Array

Local GPS

Note: if THINK is on this computer, turning on local GPS will do strange things.

GPS COM port: COM1

Enable Local GPS: OFF

Sensor Status

🟢 Sensor Connected
🟢 Sensor Initialized
🟢 Sensor OK
🔴 GPS Lock

Latest Error String: simulation only!

Figure 33: GPS mission input interface

This interface allows the operator to save up to 5 GPS coordinates using either the sailbot GPS or a separate COM-port attached GPS. Those coordinate sets can then be used to input any type of mission. This allows us to quickly tag buoys and use them to construct missions.