McGill Robotics

RoboSub 2021

Authors:

Luai Abuelsamen,
Tommy Clark,
Vasily Fedotov,
Ben Hepditch,
Charles Lapierre,
Wissam Mantash,
Allison Mazurek,
Rafid Saif,
Lingzhi Zhang.

June, 2021

Abstract

The last several years have presented a unique set of challenges for the Mcgill Robotics team. Extremely large turnover coupled with poor documentation and systems failure has lead us to 'think smaller'. This year we have decided to get a fresh start and build solid foundations for the years to come. Our main values are now those of simplicity, reliability, and the development of a team where all members have a strong understanding of each component of the AUV. We want our AUV team to be more accessible to newcomers by operating under simple logic and by having information about the robot easily accessible. We also want to make sure that, in the future, the team will be able easily understand and verify the work that we are doing, such that they will be able to efficiently and easily build upon it.

This report discusses various aspects of our AUVs, old and new, including competition strategy, design creativity, and experimental results. In accordance with our values, our competition strategy placed a special emphasis on simplicity, which we believe leads to better robustness. We achieved this by heavily prioritizing the competition tasks we thought we would be able to achieve in the short term. This allowed us to put more energy towards making the robot perform its tasks using reliable methods rather than attempting to do everything with mediocre success rates. When it comes to design creativity, as we were planning for years ahead, we made it a priority to have a system that is modular and extensible. This can be seen in our mission planner which uses state machines to encapsulate tasks as states. This design will allow us to modify and add tasks without impacting work done previously. We also improve modularity by obeying the *separation of concerns* principle when designing our package system. Meanwhile, our mechanical design is made extendable by ensuring that the hardware of our AUV is easily accessible and replaceable. Finally, we report on some modest bench-top testing results, demonstrating the frequency response of our custom hydrophones filter board and the debugging outputs of our mission planner.

Competition Strategy

Our general competition strategy is to prioritize robustness and simplicity whenever we are presented with a design choice. Due to the small size and relative inexperience of our team, we have decided to focus on tasks that require the least amount of hardware and that we believe we will be able to complete with a high success probability. Consequently, we have decided not to attempt the bin and torpedo tasks.

Planning for years to come, we have decided to implement the overarching logic of the robot using a state machine. By having every task be self contained, the state machine allows us to easily add or remove tasks without impacting the rest of the logic. We go into more detail about the state machine and how it works in the design creativity section.

The rest of this section is used to describe our strategy for each task.

Gate Task

For the gate task we have chosen to accept a random starting direction based on the outcome of a coin flip. As per competition rules, it is possible to adjust the code running on the AUV after the outcome of the coin flip is known. Thus, we can hardcode the initial angle between the direction of the AUV and the direction of the gate. To orient ourselves towards the gate an Inertial Measurement Unit (IMU) is used to determine the current direction of the AUV relative to its initial orientation.

Completing this task is broken down into three steps: submerging the AUV; orienting it to face the gate; and surging through the gate. To move our robot, we will be using a series of PID controllers to arrive at the required setpoint in each degree of freedom. Our PID controllers will rely on a bank of sensors to measure our instantaneous displacement from our desired setpoint, and adjust the robot's effort accordingly.

We use the PID ROS package as a framework: it generates topics upon which the sensors publish state estimation data, which the PID subsequently uses to publish efforts on different topics to guide the AUV towards a predefined setpoint. Using this package allows us to not worry about the implementation of a PID controller and focus instead on the way it 'hooks up' the different parts of the system.

In implementing this state, as with other states, we determine a threshold within which the robot can be considered 'at the setpoint.' When the robot is within this threshold we wait for 10 consecutive readings that are within the threshold to ensure that the robot is stable at its target state before executing the next move.

Concretely, for descending the robot to the competition depth, a setpoint depth of 2.5 m is hard-coded with a threshold of 0.2 m. The current depth estimate is provided by a pressure-based depth sensor. The PID will receive the current depth from our onboard pressure-based depth sensor and use the difference between this reading and our setpoint to output the effort to our upward-facing thrusters which will apply a vertical force on the AUV. Similarly, for rotating the AUV towards the gate, the IMU provides the angular state to the PID, the angle setpoint is set after the coin toss outcome is known and a angle threshold of five degrees is used.

To move the robot through the gate, the robot is set to surge with a predetermined magnitude for a set amount of time so as to cover a hardcoded distance. We chose to pursue this ded-reckoning approach due to its simplicity and our confidence that we can - through pool testing - come up with appropriate hardcoded parameters that would achieve the task for the majority of competition situations.

Lane Detection

With the foreknowledge that we have just completed the gate task, we expect to be in close proximity to the lane marker. We begin by using our downward facing camera to obtain footage and check if there are any bright orange points using colour thresholding. If we identify a sufficiently large contour satisfying the colour threshold, we take this as an indication that we are directly above the lane. If not, we perform a raster scan to center ourselves above the lane. During this raster scan, we maintain a constant heading to ensure that we are still aligned to the first segment of the lane marker.

Once we identify that the lane is in the viewframe, we engage a PID controller to bring the centroid of the lane to the middle of the viewframe. Once this is complete, we are oriented directly above the lane.

Upon finding the lane, our first task is to reduce noise by applying a Gaussian blur to the image obtained. This prevents any specks from interfering with our detection process. Following this, we increase the red component of the entire image to ensure that the contrast is stark enough to be clearly separable for our color mask. We can then move on to applying a color mask, for which the parameters have been adjusted with testing. Furthermore, we have implemented a dynamic reconfigure service that allows a user to adjust these threshold values while the robot is active. These 'on-the-fly' adjustments allow us to respond to different ambient lighting conditions and significantly accelerates testing. At this point, we arrive at a binary image so we see the image in terms of parts that are orange and parts that are not. Again, we find contours in this image and check if any of these contours are large enough to be the lane. If we find contours of sufficient size, we apply Canny edge detection on the image. We then apply the Hough lines algorithm to interpret the edges output by the Canny edge detector as straight lines. We separate the identified lines into two groups corresponding to the two segments of the lane. Finally, we compute the angle between these two groups of lines and use this angle to orient the AUV toward the next task.

Buoy Task

Our procedure for the buoy task is broken up into two parts: the approach, and target identification. Upon achieving stable alignment with the lane, the AUV transitions to processing the feed from its front camera. Color thresholding is used to guide the sub towards the buoy, until it takes up a predetermined size in the viewframe, which is used as a rough proxy for distance. The RGB values of the buoy will be empirically determined prior to the competition, but will be further specified during the testing phase of the competition to account for environmental differences. With the buoy in frame, an ORB image classification algorithm picks up the scale invariant features of the image on the buoy, and uses K-Nearest Neighbours to determine if the image on the buoy is the target. If the image is not identified to be the target, the AUV will wait until the rotation of the buoy brings the target image into the viewframe. Once it correctly identifies the target image, it will surge towards it for an empirically determined time to touch the buoy and complete the task. Figure 1 gives an example of our ORB detector identifying a target image for a representative task.

Surfacing task

Our strategy for the final task is to do the bare minimum, but do it well. We plan to simply get to the pinger and surface within the octagon.

To locate the pinger's relative position to the robot once the previous task is complete, we use 4 Teledyne Marine RESON TC 4040 hydrophones. They are piezoelectric sensors, which we placed together in a custom pressure vessel attached the side of the robot. These sensors directly convert pressure waves from the pinger into voltage signals. We manipulate those signals using an in-house designed filtration board to isolate and amplify the known frequencies of the pinger. The processed signals are then simultaneously digitized by four Analog to Digital Converters (ADCs) on a nucleo microprocessor for the robot to use a Time Difference of Arrival (TDoA) algorithm to locate the position of the pinger relative to itself. The TDoA algorithm uses cross-correlation between the sinusoidal functions of every possible pair of the digitized signals to find the time delays between the signals. The algorithm then uses this cross-correlation to estimate an angle for the pinger relative to the robot. Once a relative angle is determined, the robot enables a PID controller to act on its yaw to align itself towards the pinger, or in other words, make its relative angle to the pinger zero.

Once the AUV is stably pointing towards the pinger, the robot will surge forward while keeping the yaw PID on to always be moving toward the pinger. Once it passes over the pinger, the robot should detect a very sudden change in angle. If we are indeed passing over the pinger, this change will be too fast for the PID to keep us on course. The robot will use that sudden change to deduce it has surpassed the pinger and stop surging. It will stay motionless for a few pings to make sure that the pinger is indeed behind it, and once that is confirmed, all motors can simply be turned off and the robot's positive buoyancy will make it float back to the surface.

Though this approach is quite simple, we expect it to also be robust. We plan on making the robot move slowly to avoid big changes in angle, except once we reach the pinger. We also confirm any deduction by staying motionless for multiple pings when we reach a noteworthy position (facing the pinger when we are rotating towards it, and surpassing the pinger when we are surging). This allows for better error detection.

Design Creativity

This year was the start of a rebuilding project for our team. We are currently building a new AUV from scratch, both in the mechanical and software aspects. Though our robot is not complete, we have still made good progress considering the current world situation. Had there been an in-person competition this year, we would probably have competed with our older model as it is still functional. We have decided to separate the content of this section into two subsections: the first explaining the most notable aspects of the older robot and what we managed to implement this year, and the second one describing the interesting features we plan on implementing in our upcoming iteration.

Currently Implemented

Hydrophones and Filter Board

The hydrophones are currently located on the side of our AUV. As mentioned in the previous section, they consist of four piezoelectric sensors, which convert pressure waves (the waves emitted by the pingers in the water) into voltage signals. Each sensor is connected to its own port on a filter board, which isolates and amplifies the known frequencies of the pinger in the voltage signal using standard op amp filter topologies. The modified signals are then sent to four Analog-to-Digital Converters (ADCs) on a nucleo microprocessor, and the digitized signals are finally analyzed using the TDoA algorithm to give us a relative position for the pinger with respect to the robot.

This approach is especially effective because the first step of signal-processing is not done by the brain of our robot, the Jetson. This reduced load on the "brain" of our robot allows it to concentrate on the logic we implemented and perform its tasks/decision-making with much less latency. Also, the nucleo board only sends data when there is a ping, which reduces the load on the Jetson even further.

An alternative design would be to implement the signal filtration digitally using our Jetson board, but this approach would be computationally expensive and vulnerable to discretization errors.

Depth Sensor

Our team created a custom PCB board with the purpose of determining the depth of the AUV. On the PCB there is a voltage regulator and a MEMS pressure sensor used to measure exerted water pressure. From this measurement, we are able to calculate the depth of the AUV using Bernoulli's Equation $P = \rho g h$. The sensor has both I2C communication and an SPI interface for communicating with an Arduino, where we read the data and publish it to a ROS topic. To improve our sensor's reliability, we incorporate a watchdog timer for automatic system reset in case of any hardware or software fault. A fault is assumed to have occurred if the main loop takes too long to execute once, at this point the 'watchdog timer' will be zero, indicating need for reboot of the pressure sensor and micro controller.

The sensor is exposed to the environment via a feedthrough in the AUV's body. In order to simultaneously expose the pressure sensors to the water and prevent leakages into the main pressure vessel, we recess the sensor into the aforementioned feedthrough, then encapsulate the sensor PCB in watertight stycast epoxy. Figure 2 shows the PCB layout for this board

State Machine

A state machine is a computational model that consists of a finite number of states with predetermined transitions between them. This machine is "a device which can be in one of a set number of stable conditions depending on its previous condition and on the present values of its inputs".[Lexico 2021] State machines are used in the mission planner which is the high level mind of the robot, deciding what the robot should do next. We use the Python SMACH library to create our state machine.

To implement our state machine, we follow a step-by-step process. First, we define what states represent. In our case, states represent tasks. These tasks can be overarching competition goals such as navigating through the gate, but also can be smaller operations such as detecting the lane marker. Secondly, for each state we define transitions into and out of the state; each transition is determined by the current state and inputs from the AUV's sensors and internal logic. Finally, in each state, we define a set of actions to be executed when the robot is within the state and the conditions for transition into other states.

We choose to utilize state machines since they allow us to encapsulate all the actions that need to be performed for a particular task in one state, making the code easier to maintain and understand. Moreover, it allows the code to be easily extensible since if we would like to perform more tasks in the future, we do not need to modify the existing code; we would only need to add new states and transitions. Finally, state-machines are an intuitive and easily visualizable method of mission planning, since all actions related to a particular task are encapsulated in a single state. This allows us to easily focus on the transitions between states without worrying about the internal architecture of any particular state.

Another option we explored in the past is algorithmically coding the tasks we intend to perform in order, while including conditions for failure. However, an issue we faced is that our code would get exponentially more complex and difficult to understand as the number of tasks increases. Moreover, unlike with state-machines, adding a task requires an in-depth understanding of the entire code. We found that this method is also prone to error since it is not as easy to visualize and test, whereas the events in a state-machine are easy to visualize as they are encapsulated in an intuitive manner.

In Progress

Mechanical Design

Bradbury features a cylindrical body with a transparent compartment containing all the electrical components, as well as five LED strips on top indicating the status of the robot as can be seen in Figure 3. With 8 thrusters strategically placed, Bradbury can move in all directions without rotating thrusters.

Dennis, our recent design, features an octagonal body with four thrusters evenly spaced around the perimeter of the main pressure vessel. Each thruster is attached to a servo allowing it to rotate 180 degrees. The remaining sides host the torpedo launchers and transparent access panels. Dennis also showcases a removable transparent top, allowing for easier disassembly and access to internals.

Thrust vectoring

A new feature we plan on implementing on our AUV is thrust vectoring. The frame of the new AUV is designed so that we can place four thrusters on four different side panels of the robot. The relative position of the thrusters allows us to only give each of them one degree of freedom to be able to surge, heave, sway, and control roll, pitch, and yaw (basically perform all possible movements in 3D space).In the appendix, Figure 4 shows pictures of the frame of the robot and how each thruster can rotate.

We decided to step away from our guiding principle of simplicity to do thrust vectoring because we found it a nice challenge in both the mechanical and software aspects of the robot. We need to design a sealed hinge for the thrusters and have to transform our movement vector (expressed as [x,y,z]) into an orientation and an effort for each thruster. The latter is especially difficult considering some movements require efforts from different thrusters to cancel out. The decision to implement thrust vectoring was taken as an opportunity to learn about control systems.

Modular Design of the Software

There are plans to refactor how code is organized in the repository such that it is more modular and abides by a self-consistent design architecture. Currently, the repository consists of several ROS packages; however, there is inconsistency in regards to the purpose of a ROS package within this project. Some packages like cv (computer vision) group modules together based on the underlying technology. Others, like controls, have suffered from scope-creep - what the package is responsible for is ill-defined and includes low-level code for the particular implementation of the propulsion mechanism.

These design decisions present code smells - characteristics of code indicating deeper problems that will cause downstream development issues. In this section, a new architecture will be formalized and advantages and disadvantages of this proposed design are considered.

Code Smells

Grouping code in packages based on using similar underlying technologies creates a close-coupling between what the code does and how it does it. As an example to illustrate why this should be avoided: if the team chooses to change the underlying logic for the buoy task such that it no longer uses computer vision then it would not make sense to have relevant code in the cv package anymore. Each time significant implementation changes are made, there is a possibility that things get moved between packages. This may cause confusion since locating lane-marker navigation code requires one to already be familiar with its particular implementation, and have to re-learn this if ever the implementation changes. Therefore, it may be advantageous to organize ROS packages in a way where code pertaining to a certain function of the robot always resides in the same package regardless of how the function is implemented.

Secondly, packages that balloon in scope lead to confusion. Code pertaining to commanding the thrusters should be in a propulsion package separate from the control logic because control and propulsion are orthogonal and independent concerns. If packages are grouped by function, the propulsion package does not need to be aware of how decisions are made regarding the effort that should be exerted by the thrusters. The propulsion package should only be concerned with providing the interface with the mechanical propulsion mechanism. Having these independent concerns in the same package could encourage people to push code for the propulsion system that makes explicit use of the concrete control system implementation, leading to more closely-coupled code that is harder to refactor. Additionally, overzealous grouping can cause the ‘hooks’ by which packages are connected to become ill-defined. This presents several problems: not knowing where to ‘hook into,’ hooks that do not follow a consistent convention, accidentally using the wrong hooks, or creating duplicate hooks; all of which lead to poor code readability and understanding. If there are clear definitions of the scope and interfaces of each package it is easier to define the extent of tasks, compartmentalize them, and document their interfaces.

Proposed Architecture

The proposed architecture aims for the repository to consist of a series of loosely-coupled ROS packages that each have a clearly defined scope and purpose. The packages group code based on their functionality/concerns and operate akin to an interface such that each package may contain several implementations of the same concern. For example, the propulsion package may have several implementations for different thruster configurations; however, only one of those implementations will be used in competition. Importantly, other upstream packages (ie. mission-planning, controls) need not be concerned with how the propulsion system is implemented, only that it works to propel the AUV as intended. Tentatively, the packages that would comprise the repository include:

- **propulsion**: responsible for the propulsion mechanism interface
- **controls**: responsible for calculating effort based on a setpoint and state-estimate it receives from upstream

- **state-estimation**: responsible for determining the pose and kinematic state of the AUV. Notably, this includes low-level implementation of devices such as hydrophones, depth sensors, and IMU. Furthermore, responsible for computer vision implementations for determining orientation based on lane-markers
- **mission-planning**: responsible for making high level decisions about the actions the AUV needs to take, particularly the steps involved in completing a task and transitions between tasks

This architecture could be extended in the future to include packages for logging, debugging and testing. Separating out these orthogonal concerns ensures consistency across packages without polluting the code inside other packages.

In this loosely-coupled architecture, the controls package may require the current depth from the state-estimation package; however, the way in which the state-estimation package works is a black box - the controls package need not be concerned with how the depth is determined as long as it trusts that the value provided at the interface is correct. This gives freedom to refactor the concrete implementation of the depth-estimation node in the state-estimate package (ie. by using a DVL sensor as opposed to a pressure based sensor) without any disruption to downstream dependencies - the controls system would never be the wiser that anything changed.

This naturally leads into the concept of package scope since each package is concerned with providing a set of functionalities with clearly defined expectations. Then, implementing or refactoring a functionality is restricted to only use the interfaces of external packages and must adhere to providing exactly what is expected of it - no more, no less. As a concrete example, the surge controller node, part of the controls package, expects to receive a setpoint from the mission-planner package and an estimate of the current position from the state-estimate package. Using this data, it provides a thrust vector to the propulsion package. The controls package should not try to make mission-level decisions about what to do and in what order (that is the mission-planner's job) and should not make any assumptions about how the thrusters are configured (that is abstracted away in the propulsion package).

Advantages

- Several different or historical implementations can be included in each package. This could be useful to have as a reference, to assess performance or to defer back to without having to revert git history by setting the build system to use a particular implementation.
- Having scoped, modularized components makes it easier to plan out the tasks. Each task can consist of implementing a particular functionality, the scope and extent for which is restricted by the design architecture. This avoids increasingly long tasks that don't have a clear end goal.
- Packages that are loosely-coupled allow for group collaboration without stepping on toes, since individuals making changes to a particular package need not worry about the implementation changes being made to dependencies.

Disadvantages

- The benefits of this design philosophy persist as long as it is adhered to. Over time there may be areas where old anti-patterns are used, or cases where the proposed approach seems too laborious as opposed to a more convenient closely-coupled solution. In these cases, adhering to the architecture may be more difficult to understand.
- May be difficult for new members to understand and adopt this philosophy since many things are handled in the abstract as opposed to dealing with concrete implementations. Likewise, there will be more overhead with the build system and getting things to hook together in a loosely coupled manner which makes working with the code less approachable.

- The architecture does not promote code reuse since two modules may have different functionalities and therefore live in separate packages but use very similar underlying frameworks. Whenever such a framework must be refactored changes may have to be made in several areas which presents opportunities for inconsistencies between these frameworks across packages.

Experimental Results

This year being entirely online, the great majority of our tests have been targeting software, but we still managed to test some physical components. Unfortunately, we have not been able to test our robot in the water, so we do not have any real ”situational” tests.

Tuning of control system

Before the onset of the pandemic, we were able to test our depth PID. We intended to tune the PID parameters using the Ziegler-Nichols method, but upon testing we had difficulty attaining a sufficiently clear oscillation to execute the tuning algorithm. Furthermore, during our testing, we found that a wide range of gain parameters produced acceptable setpoint attainment. We settled on relatively low gains, reflective of our prioritization of robustness over speed.

Lane Detector

While we could not do any pool testing for our lane detector, we worked around it with two approaches. We captured live footage from a webcam and placed orange objects in the frame to see if our blur, edge detection and color masks worked. We moved the object around to test if it still worked. The results from these tests were highly encouraging. We tested for various orientations of the two lanes, and we received the output angles. We checked these and they appear to give good estimates of our headings. In the appendix, a sample output of the lane detector is displayed in Figure 5

Hydrophones

We did a simulation using LTspice for the hydrophones filter board. We were unable to do any testing in a pool for the hydrophones, but we were able to make some small tests. First, we put the hydrophones in a bathtub with a pinger and managed to get a reading. The hydrophones and the associated ROS software return a heading, but we are unable to verify the accuracy due to the nature of our testing environment. We also tested the frequency response of our filter-board in isolation. After testing, we found out that they were filtering for 130kHz, which is quite far from the 40kHz from the pingers. We believe we would only need to change passive circuit elements on our filter board to change this filter frequency. In the appendix, Figure 6 shows a circuit schematic and LTSpice simulated frequency response for the hydrophones signal processing board. It consists of an input and output gain stage and fourth order filter. Figure 7 shows the measured magnitude and phase response of the board.

State Machine, Logic, and State Transitions

To test the state machine we manually input data and see how the state responds through debug messages. Once a state is implemented, thorough testing is conducted through a wide range of possible inputs values. Afterwards we define transition properties and ensure transitions occur in the correct order under the right conditions.

In the appendix, Figure 8 shows the data published to the hydrophones/pose topic and Figure 9 shows the behaviour of the state machine in reaction to that published data. We have chosen to display a simple case of the robot achieving its goal without any hiccups, but are aware that there is a big possibility for things to go wrong. We have tested some of those possibilities, and even though we know our logic is not perfect, we have chosen not to modify anything yet. Our logic is robust enough for most situations, and we are waiting for pool testing to see if the implemented logic is indeed robust enough.

Appendix

This section contains the figures referenced in the main text.

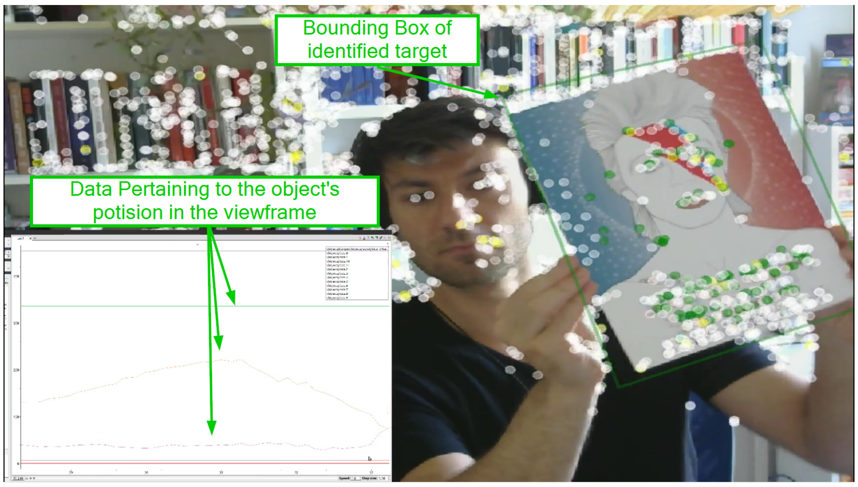

Bounding Box of identified target
Data Pertaining to the object's potision in the viewframe

Figure 1: An example of our ORB image detector in action. The small circles are scale invariant features, the green rectangle is the bounding box of the detected image, and the plot in the lower left is a livestream of the target's position and orientation. The target here is an image of David Bowie, but changing the target with this scheme is very straightforward.

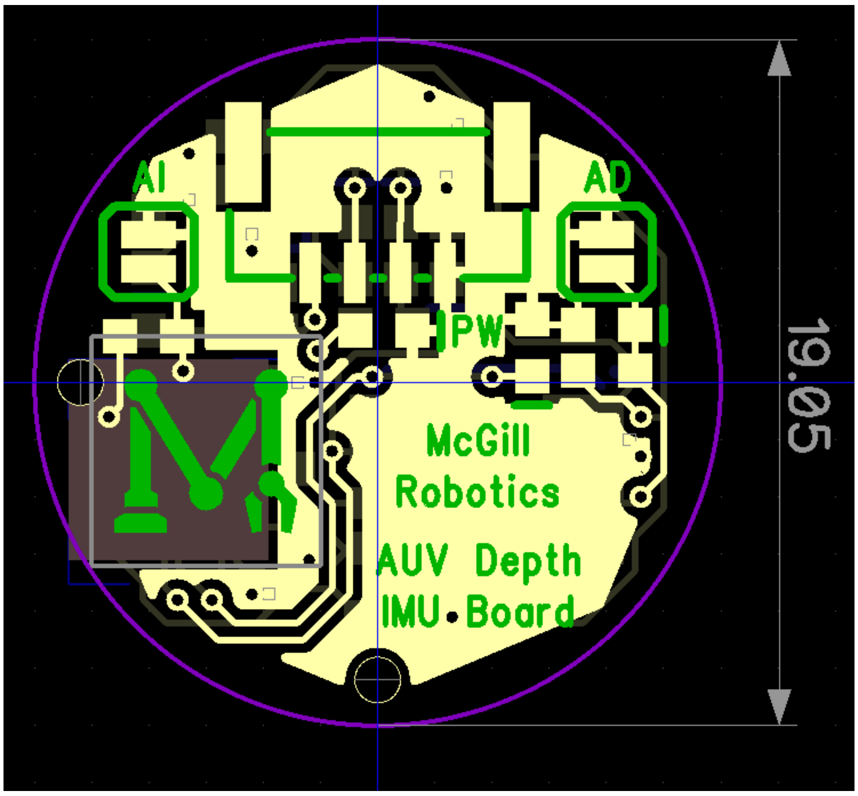


Figure 2: The PCB layout for our current depth sensing solution. This PCB is placed in a feedthrough in the main pressure vessel, then encapsulated in stycast epoxy. Originally, this board was also responsible for measuring the pose of the robot via an IMU, but this has been superseded by a dedicated IMU in the main pressure vessel.

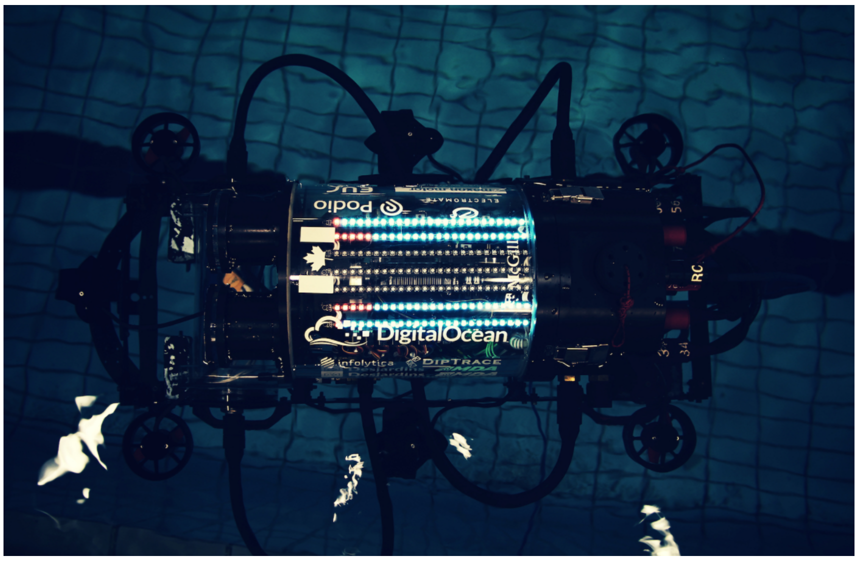


Figure 3: A top-down photograph of the previous iteration of our AUV.The four upward facing thrusters can be seen at the corners of the AUV, and the remaining four are along the sides.

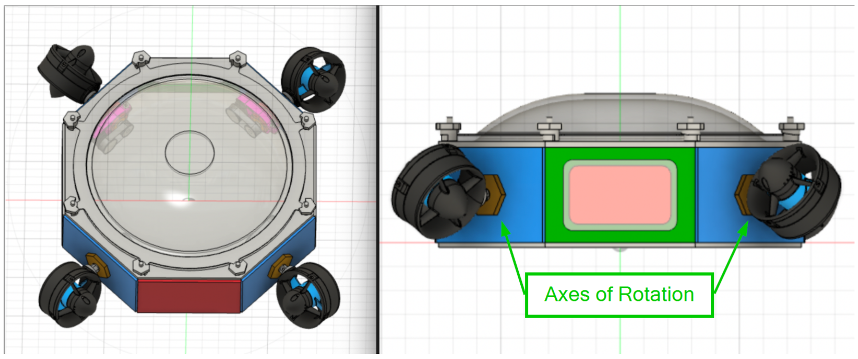
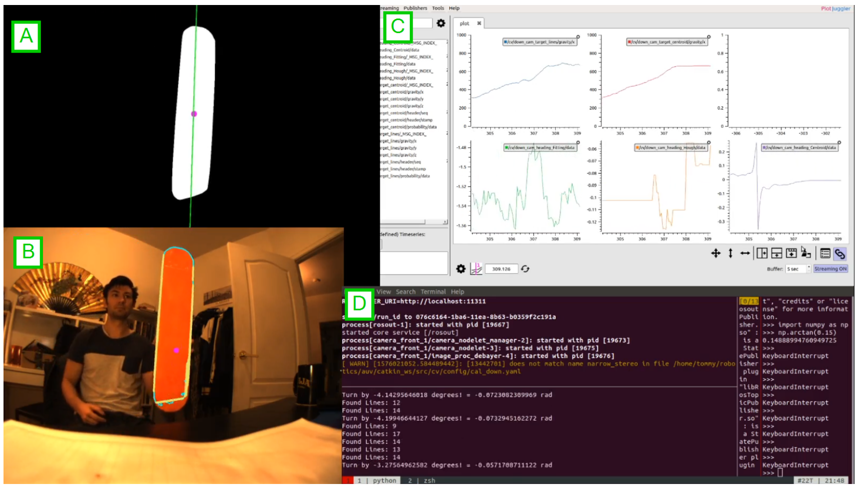


Figure 4: CAD images for the newest iteration of AUV, which features the ability to rotate the thrusters. The bronze feedthroughs, indicated with green arrows, are free to rotate and are driven by servo motors on the inside of the main pressure vessel.



```
ER_URI=http://localhost:11311

/run_id to 076c6164-1ba6-11ea-8b63-b0359f2c191a
process[rosout-1]: started with pid [19667]
started core service [/rosout]
process[camera_front_1/camera_nodelet_manager-2]: started with pid [19673]
process[camera_front_1/camera_nodelet-3]: started with pid [19675]
process[camera_front_1/image_proc_debayer-4]: started with pid [19676]
[ WARN] [1576021052.584489442]: [13442701] does not match name narrow_stereo in file /home/tommy/robo
tics/auv/catkin_ws/src/cv/config/cal_down.yaml

Turn by -4.14295646018 degrees! = -0.0723082309969 rad
Found Lines: 12
Found Lines: 14
Turn by -4.19946644127 degrees! = -0.0732945162272 rad
Found Lines: 9
Found Lines: 17
Found Lines: 14
Found Lines: 13
Found Lines: 14
Turn by -3.27564962582 degrees! = -0.0571708711122 rad
```

```
t", "credits" or "lice
nse" for more informat
ion.
>>> import numpy as np
>>> np.arctan(0.15)
0.14888994760949725
>>>
KeyboardInterrupt
>>>
KeyboardInterrupt
>>>
KeyboardInterrupt
>>>
KeyboardInterrupt
>>>
KeyboardInterrupt
>>>
KeyboardInterrupt
>>>
KeyboardInterrupt
>>>
KeyboardInterrupt
>>>
```

Figure 5: a) A binary image produced by colour thresholding demonstrating isolating the orange lane from its surroundings. The purple dot is the image centroid, and the green line is a least-squares-optimal line fit. b) The raw image upon which the color thresholding acts c) A livestream of several characteristics of the thresholded image, including the centroid position and the orientation of the line fit. d) A terminal window reporting some derived values, including the angle the camera must turn to be aligned with the line of best fit.

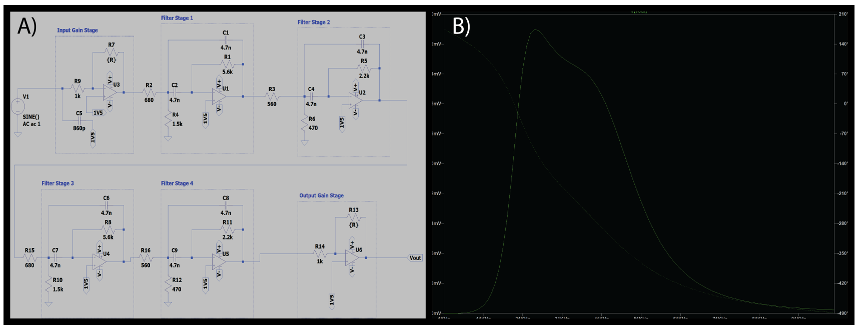


Figure 6: a) The schematic for our LTSpice simulation of our hydrophones filter board. b) The frequency response of this board to a 1mV input. The solid line is the amplitude response, (note the linear scale) and the dotted line is the phase response. The frequency response exhibits a a clear bandpass behaviour in the frequency range of the pingers; it will selectively amplify pings and reject other noise at frequencies outside the passband.

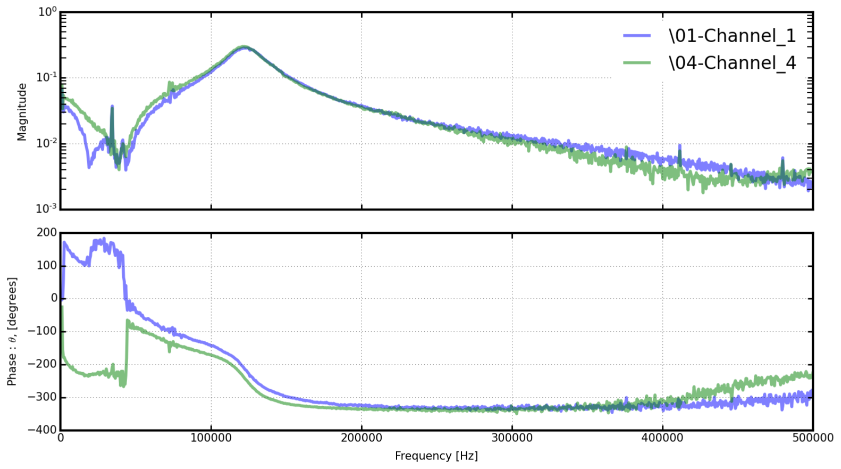
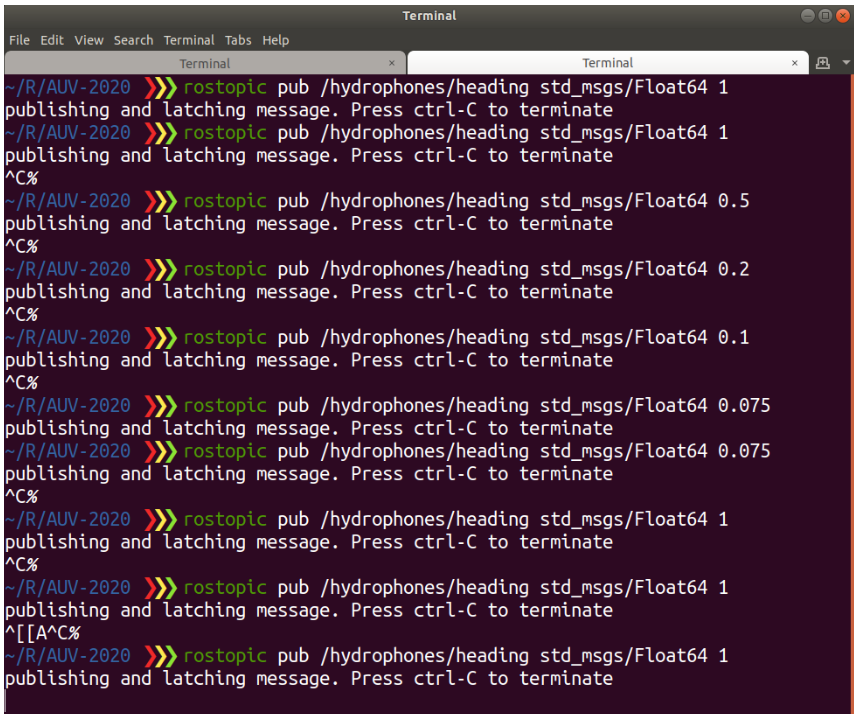
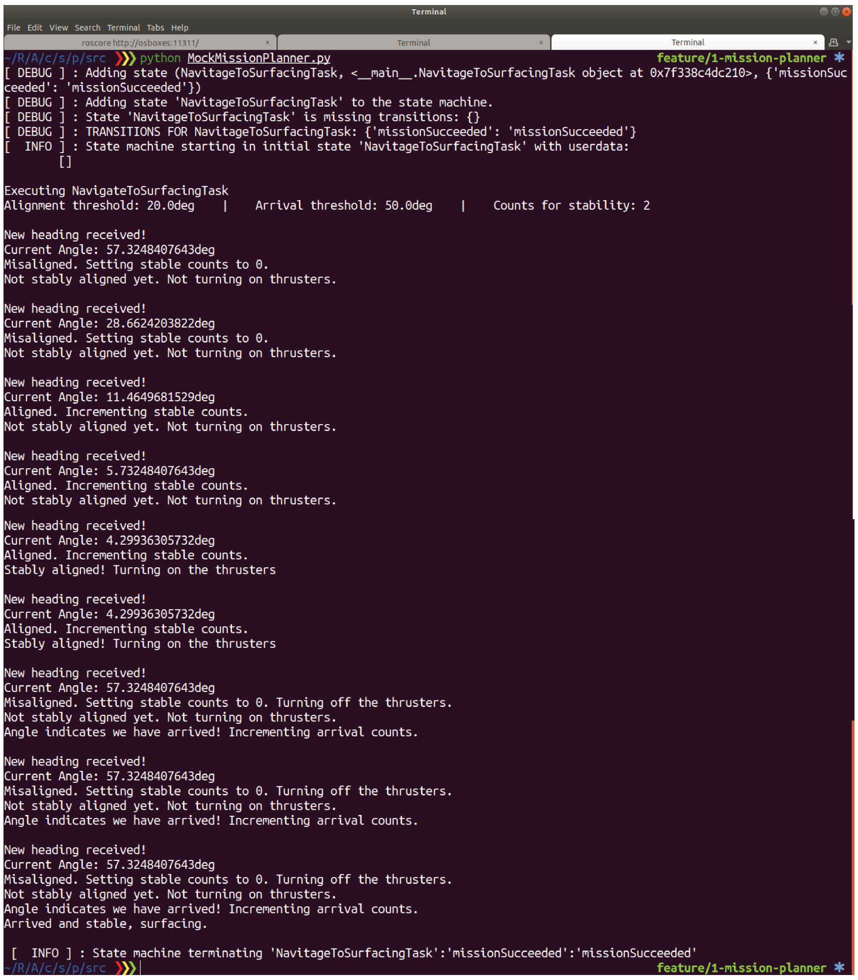


Figure 7: The measured frequency response of two channels of our filtration board. Importantly, it appears that the center frequency of the band pass filter is significantly different than simulation. This board will *attenuate*, rather than amplify, the pinger signals we are interested in. Further testing is required to determine the reason and correct the next iteration of this board.



```
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 1
publishing and latching message. Press ctrl-C to terminate
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 1
publishing and latching message. Press ctrl-C to terminate
^C%
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 0.5
publishing and latching message. Press ctrl-C to terminate
^C%
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 0.2
publishing and latching message. Press ctrl-C to terminate
^C%
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 0.1
publishing and latching message. Press ctrl-C to terminate
^C%
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 0.075
publishing and latching message. Press ctrl-C to terminate
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 0.075
publishing and latching message. Press ctrl-C to terminate
^C%
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 1
publishing and latching message. Press ctrl-C to terminate
^C%
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 1
publishing and latching message. Press ctrl-C to terminate
^[[A^C%
~/R/AUV-2020 ❯❯❯ rostopic pub /hydrophones/heading std_msgs/Float64 1
publishing and latching message. Press ctrl-C to terminate
```

Figure 8: An example of manual publishing to one of our topics (here the hydrophones) to test our mission planning system. Eventually, we should implement a simulator, but this allows for 'quick and dirty' tests of our system.



```
~/R/A/c/s/p/src ❯❯❯ python MockMissionPlanner.py                                        feature/1-mission-planner ✱
[ DEBUG ] : Adding state (NavitageToSurfacingTask, <__main__.NavitageToSurfacingTask object at 0x7f338c4dc210>, {'missionSuc
ceeded': 'missionSucceeded'})
[ DEBUG ] : Adding state 'NavitageToSurfacingTask' to the state machine.
[ DEBUG ] : State 'NavitageToSurfacingTask' is missing transitions: {}
[ DEBUG ] : TRANSITIONS FOR NavitageToSurfacingTask: {'missionSucceeded': 'missionSucceeded'}
[  INFO ] : State machine starting in initial state 'NavitageToSurfacingTask' with userdata:
        []

Executing NavigateToSurfacingTask
Alignment threshold: 20.0deg    |    Arrival threshold: 50.0deg    |    Counts for stability: 2

New heading received!
Current Angle: 57.3248407643deg
Misaligned. Setting stable counts to 0.
Not stably aligned yet. Not turning on thrusters.

New heading received!
Current Angle: 28.6624203822deg
Misaligned. Setting stable counts to 0.
Not stably aligned yet. Not turning on thrusters.

New heading received!
Current Angle: 11.4649681529deg
Aligned. Incrementing stable counts.
Not stably aligned yet. Not turning on thrusters.

New heading received!
Current Angle: 5.73248407643deg
Aligned. Incrementing stable counts.
Not stably aligned yet. Not turning on thrusters.

New heading received!
Current Angle: 4.29936305732deg
Aligned. Incrementing stable counts.
Stably aligned! Turning on the thrusters

New heading received!
Current Angle: 4.29936305732deg
Aligned. Incrementing stable counts.
Stably aligned! Turning on the thrusters

New heading received!
Current Angle: 57.3248407643deg
Misaligned. Setting stable counts to 0. Turning off the thrusters.
Not stably aligned yet. Not turning on thrusters.
Angle indicates we have arrived! Incrementing arrival counts.

New heading received!
Current Angle: 57.3248407643deg
Misaligned. Setting stable counts to 0. Turning off the thrusters.
Not stably aligned yet. Not turning on thrusters.
Angle indicates we have arrived! Incrementing arrival counts.

New heading received!
Current Angle: 57.3248407643deg
Misaligned. Setting stable counts to 0. Turning off the thrusters.
Not stably aligned yet. Not turning on thrusters.
Angle indicates we have arrived! Incrementing arrival counts.
Arrived and stable, surfacing.

 [  INFO ] : State machine terminating 'NavitageToSurfacingTask':'missionSucceeded':'missionSucceeded'
~/R/A/c/s/p/src ❯❯❯                                                                       feature/1-mission-planner ✱
```

Figure 9: An example of the responses returned from our mission planning system when receiving the manually published data demonstrated in the previous figure.

Bibliography

Lexico (2021). *State machine*. URL: https://www.lexico.com/en/definition/state_machine (visited on 26th June 2021).