# Design and Implementation of Seabee: the University of Southern California's Autonomous Underwater Vehicle

Written By: Travis Mendoza, Jessica Freidin, Michael Kukar, and Ben Shiroma

**The USC Autonomous Underwater Vehicle (USC AUV) design team is a student run team that designs and builds a robotic submarine every year to compete in the International Robosub Competition. The competition calls for the completion of a number of tasks in a pool that demonstrate a submersible's capacity for autonomous control over its position and orientation, active decision making, image processing, torpedo firing, marker dropping, and acoustic recognition. However, the USC AUV submarine (Seabee) was kept simple and is not intended to demonstrate all of these capabilities.**

## I. INTRODUCTION

USC AUV is a student-run robotics team that allows its members to gain experience working on a multi-systems project while still in an educational environment. The team consistently participates in the Robosub Competition and strives to continually improve through exchange of ideas and involvement in the AUV community. The team uses a full-year design cycle between iterations of the Seabee AUV. From preliminary internal design reviews to a critical review attended by experts in the field, designs and changes to Seabee undergo a thorough analysis culminating in vital improvements to the AUV without wasteful or unnecessary expense. This year USC AUV has undergone a complete rebuild of Seabee's internals and pod design, keeping only the frame and original hull from last year.

## II. DESIGN STRATEGY

USC AUV is unique in that although it does technically have a faculty advisor, the team has opted to receive no help or guidance from him or any other advisor for that matter. The benefits of doing so are primarily that all aspects of the team that a faculty member would normally take care of are instead under student control e.g. design decisions, systems integration, and money accounts. By forcing members to handle these aspects of the team that are outside of their normal projects, it is hoped that they will gain a more well-rounded educational experience.

However, there are also disadvantages to this lack of a permanent advisor. Organization of the team's members, records, and knowledge banks changes along with nearly every turnover in leadership that occurs once a year – a corollary that has lost the team a significant amount of information.

Historically USC AUV has never excelled at the Robosub competition – probably a sub-corollary to the organizational challenges mentioned above – and the team has not qualified for the competition for at least two years.

With the team's historical performance in mind, the objective of this year's team was kept as simple as possible: to drive through the qualifying gate and to follow the path. In order to meet this goal and involve as many students as possible, each member was given responsibility over a single project that would benefit the submarine in some way – as is done every year – and every weekend an all hands building or testing session was held.

## III. VEHICLE DESIGN

### *A. Mechanical*

#### *1. Mechanical Overview*

The mechanical design philosophy for SeaBee is intended to create a modular, compact, and lightweight AUV. The single-hull design with an internal rack system enables easy removal and centralized access to electronics while the external frame provides support for long term development by allowing individual component redesigns with little to no effect on the overall structure of the AUV. Electrical connections are established by using waterproof connectors, supplied from Teledyne Oil & Gas, from the hull end cap to external components such as the IMU, cameras, and thrusters allowing simplified electrical reconfiguration. Similar wet-pluggable connectors are also in place to allow devices to be plugged into SeaBee without worrying

about water – saving time during wet tests and competition runs.

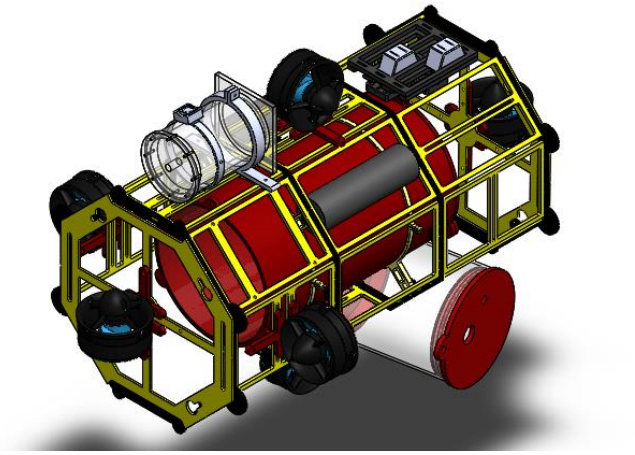

Figure 1: Overall CAD model of Seabee 2016

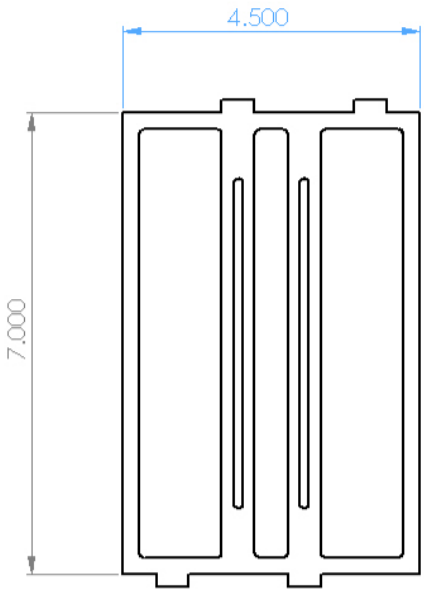


Figure 2: Basic external frame panel design

*2. Hull*

The hull for SeaBee is constructed out of 3/16" thick T6061 anodized aluminum. The enclosure employs a cylindrical design measuring 7.5" in diameter and 13" in length. It shields the internal electrical systems from water damage with a watertight design. In the hull cap design, the cap is covered with waterproof Teledyne Oil & Gas connectors allowing access to the electrical systems from components outside of the hull.

*3. External Frame*

The external frame consists of an octagonal cage surrounding the main hull. Each octagon measures 10.836" around an inscribed circle. Four octagons create three sections of the frame. Panels constructed of T6061 anodized aluminum measuring 7" x 4.5" x 3/16" make up the walls of the octagonal cage resulting in a total of twenty-four panels. Each panel is associated with a certain location of the frame and a component. In addition, panels can move around the frame easily if design changes are necessary. The front and back surfaces of the octagonal cage serve as mounts for the depth thrusters and the flotation structure.

These panels are designed around a common panel template in SolidWorks and can be quickly machined through a laser-cutting process and then anodized. The panel redesigns dramatically reduce the design and manufacturing time of any mechanical changes to the Seabee AUV.

*4. Motors/Thrusters*

SeaBee uses six BlueRobotics T100 thrusters arranged in three main groups: horizontal for forwards and backwards movement, vertical for depth, and strafing to enable six degrees of control.

*5. Battery Pods*

One battery enclosure is mounted to the underside of the vehicle's frame. The end caps are made of 6061 aluminum and the body is made of polycarbonate. This frees up space for the electrical systems inside the primary hull. Battery pods can be hot swapped, allowing for extended runtime. The battery enclosures feature a cylindrical design. They are sealed via a bore plug seal with two O-rings. They are also fitted with pressure relief valves as a safety measure to prevent the buildup of hazardous pressure levels.

*B. Electrical*

*1. Overview*

The Seabee electrical system provides the robot and its systems with power and provides the appropriate interface between all of the electrical devices. The main electrical system is made up of two backplanes, and three daughter cards each featuring its own Atmel 1280 microprocessor. This eliminates the need for excessive wires inside the submersible hull, giving the final product a cleaner look, and increasing

the overall reliability by alleviating the risk of loose wires and poor connections

*2. Power Distribution*

The Seabee Power Board is developed as a versatile platform on which to develop a wide array of electrical systems. In addition to supporting the vehicle's power management and regulation needs, the Power Board serves as an interface for any sensors not capable of directly interfacing with the XTX Carrier Board.

At the heart of the power board is the Parallax Propeller P8X32A microcontroller. The P8X32A has eight 32-bit processors, making it well-suited to the robot's high-precision, low- latency requirements. Nicknamed BeeSTEM, the microcontroller is responsible for maintaining communication with these sensors. As the BeeSTEM receives data from the sensor suite, it updates its control loops and forwards the data to the XTX Carrier Board. The BeeSTEM also produces the appropriate commands to initialize the sensors and place them in desired operating modes.

The Power Board produces regulated power at the common voltages of +3.3V, +5V, and +12V. To ease the process of swapping or adding sensors, the Power Board has auxiliary connectors. The Power Board incorporates nine motor drivers, which are controlled by BeeSTEM. Six of these provide power to the thrusters, and the other three support additional actuators, such as the Marker Dropper or Torpedo Launcher.

*3. Computing*

Computer design for Seabee is governed by the vehicle's need to perform complex computer vision and machine learning algorithms in real time in spite of restrictive space requirements. An XTX computer-on-module (COM) was selected for its balance between size and performance. The Seabee XTX Carrier Board was designed to break out important signals such as power, USB, VGA, and Ethernet, SATA, and USART.

*4. Sensors*

The Seabee sensor suite was designed to provide as close to a 6-DOF state-space solution as is possible within team budget. The Seabee is capable of actively monitoring the status of its own systems through the use of internal temperature sensors, internal pressure transducers, current monitoring in each motor driver channel (nine total), and coulomb counting in each battery pod. Four high-performance Internet-Protocol (IP) cameras equipped with wide-angle lenses are used to complete specific vision-related tasks and to provide an additional position estimate via visual odometry.

For the aforementioned 6-DOF localization the Seabee incorporates a Xsens MTi Attitude and Hearing Reference System (AHRS), a Honeywell HMC6343 3-axis compass, and a pressure transducer used to calculate depth. The primary advantage of using an AHRS like the MTi over a traditional inertial measurement unit (IMU) comes in the form of the unit's on-board signal processor, which allows it to be calibrated for the electromagnetic fields present on the Seabee and thus achieve virtually no drift.

Some of the sensors on the Seabee such as the IP cameras, are capable of directly interfacing with the XTI carrier board via USB. Most of the sensors on the robot, however, utilize serial protocols such as RS232, $I^2C$, and SPI. The BeeSTEM is responsible for meeting these bus requirements.

*5. Batteries*

The Seabee battery system consists of two custom +22.2V lithium polymer battery packs in parallel for a total of 20,000 mAh. Each pack contains a Seabee Battery Board based on the ATMEGA 406 microcontroller. In addition to regulating the LiPo cells to ensure even discharge, the Battery Boards actively monitor state of charge (SOC) through use of current integration, or “coulomb counting”. Each Battery Board incorporates an LED display to provide visual feedback to the operator.

*6. Killswitch*

The killswitch was designed with reliability as the primary focus. The design is robust and minimal, functional in the most demanding situations. A single reed switch is mounted inside the primary hull, actuated by a magnet tethered to the outer hull. A tiny logic-gate- based circuit ascertains the state of the reed switch and produces a 5V TTY “kill” signal to the microcontroller on the power board. The microcontroller is then responsible for placing the motor and actuator drivers into a low-power state.
One can place a high degree of confidence in the killswitch as its implementation, which takes place at the lowest level possible short of physically disconnecting the batteries, ensures that it will

function properly even if other systems on the sub malfunction. Additionally, the robot's software monitors the state of the killswitch via the power board microcontroller. This means that if the robot enters the “kill state”, processing can be halted until the state is exited and then proceed as normal, a helpful tool during development.

*7. Actuation*

The actuation daughtercard features twelve H-bridge motor controllers with PWM. Six controllers directly drive the robot’s thrusters, and are able to run them forwards and backwards, as well as to throttle them effectively via a PWM signal. The other six controllers can be used for manipulators on the robot, including the dropper and torpedo firing system. The current draw of each motor controller is monitored by a current sense resistor, providing the computer with vital information as to the power being drawn by any motor. The computer is also able to shut off any or all of the motors in the event of a malfunction.

*C. Software*

*1. Ubuntu*

Seabee uses Ubuntu Linux 10.10 “Maverick Meerkat” as its operating system. This selection was made based on the open-source nature of Ubuntu, its portability, and its good support for the team’s intended software architecture.

*2. ROS*

Seabee has used ROS, an open-source toolkit developed by Willow Garage, since the 2010 RoboSub competition. ROS, or the “Robot Operating System”, provides a language-generic, modular paradigm for the development of software systems.

System components are discretised into units called “nodes”, each of which has at least one dedicated thread and the ability to control parts of its life cycle.

When necessary, these nodes are able to communicate via explicitly defined, language-generic messages sent over a named, simplex channel, or “topic”. The direction of these topics is determined at compile time; a node can “advertise” an outgoing topic via a “publisher” object or “subscribe” to an incoming topic via a “subscriber” object. However, topics can be redirected or “remapped” at runtime, allowing for the creation of more loosely-defined distributed systems.

Arbitrary levels of complexity can be achieved through the creation of multiple publishers and subscribers within a node. More complex paradigms, such as “actions” or “preemptable tasks”, have been implemented to take advantage of this fact.

At a low level, inter-node communication is accomplished by message serialization within publisher objects, transmission of serialized data, and message deserialization within subscriber objects. However, if two nodes can be run on the same machine, and furthermore within the same process, this communication can instead be performed without serialization via shared pointers, thus allowing for high-performance, low-overhead communication.

Perhaps more importantly, ROS provides fairly generic implementations of many common algorithms known to the field of robotics, including such categories as sensing, navigation, planning, and visualization.

*3. QuickDev*

While ROS provides a solid foundation on which to build, working within the toolkit often involves creating unnecessary redundancies across implementations. Furthermore, the serialization-free communication described above is traditionally time consuming to set up, as modules utilizing it must follow the “nodelet” paradigm rather than the more common “node” paradigm. This issue is resolved by maintaining a set of scripts and generic wrappers around the more commonly-used ROS components in an open-source package called “quickdev” available in USC's “usc-ros-pkg”.

*4. Vision Pipeline*

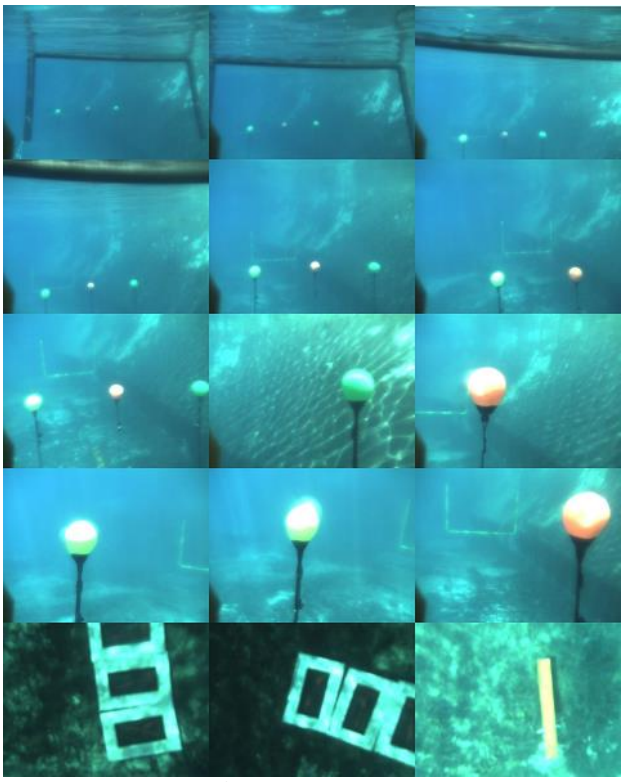

Figure 3: Seabee's view of the competition pool

In its current state, Seabee views the world through two PointGrey Firefly USB cameras: one facing forward and one facing downward. Both cameras are configured to stream bayered 320x240 images at 30 hz. While the hardware interface to these cameras is USB, they support the IIDC 1394-based Digital Camera Specification over USB, allowing for the use of "firewire" camera drivers. Conveniently, ROS provides an acceptable firewire camera driver node, which is used to read images from Seabee's cameras. Figure 3 shows an example of some of the competition objects as seen by Seabee's cameras.

Images streamed from the cameras are bayered and distorted by various optical effects, including those from the camera lenses and the results of different physical mediums surrounding the sensor tubes containing the cameras. To compensate for these effects, another useful community-developed ROS node is used called "image_proc", which utilizes several common OpenCV functions to de-bayer and un-distort (given a camera calibration) the incoming raw images.

Following basic low-level image filtering, a color-space conversion from BGR to HSL is performed. In order to optimize the vision algorithms farther down the pipeline, the code seeks to mimic neural adaptation and avoid unnecessarily processing pixels that are not changing by a sufficient amount. This value is determined by calculating a weighted sum of the differences between each pixel in each channel in the current image and the value of that pixel at the point in time that it was last determined to have changed. Pixels found to have changed by a minimum amount are recorded in a binary mask, which is published along with the corresponding HSL image. This mask-image pair is called an "adaptation image."

Newly-generated adaptation images are fed through a color classification node which utilizes a native Bayes classifier trained on actual camera footage. The classifier identifies newly- changing pixels, calculates the likelihood that a given pixel should be classified under each of a set of colors, and then publishes these probabilities as an array of images, with each image representing the classification for the corresponding color. Subsequent color-sensitive feature extraction algorithms in the pipeline can utilize these probability images in their calculations. Figure 3 shows an example of a probability image for the orange color produced by the color classifier next to the input image as well as the image on which the orange color model was trained.

*5. Recognition Pipeline*

Recognition and subsequent localization relative to landmarks, or unique competition objects, is critical to success in the RoboSub competition when the robotic platform used is not aided by a Doppler Velocity Logger; however, these landmarks are subject to change each year. From a software standpoint, it is desirable to interact with a generic landmark recognition interface rather than directly calling upon multiple specialized interfaces. In order accomplish this, Seabee utilizes an extensible landmark recognizer that accepts a landmark filter and calls on child modules to perform specialized landmark recognition. This ensures that, with the exception of drastic changes to the competition, landmark-dependent algorithms can remain mostly unchanged, while specialized recognition algorithms can be easily developed, tested, and deployed through a standard, familiar interface.

Where possible, it is desirable to be able to recognize landmarks via an adaptive, generic system, thereby avoiding as much specialization-related overhead as possible. SeaBee accomplishes this by calling on a scale- and rotation-invariant feature recognition system which utilizes OpenCV 2D feature extraction,

or contour extraction, performed on color- classified images produced by the vision pipeline. A set of template contour features, in the form of normalized, rotation-aligned histograms, is trained for any landmarks or landmark components and passed along with any candidate contour features located within incoming images to a generic recognition algorithm, which calculates and returns match qualities for each template-candidate pair. In this way, specialized landmark recognition algorithms can offload a significant amount of specialization to a central, generic system, yet still ensure the use of alternate, arbitrarily specialized recognition methods.

In its current state, SeaBee uses a specialized recognition algorithm for each unique landmark, excepting landmarks differing only in color. Furthermore, recognition is specialized based on the expected sensor source of landmarks relative to the sub; for example, it is assumed that pipelines and bins will only enter the field of view of the downward-facing camera, while buoys, hedges, and windows are expected to be found only in the field of view of the forward-facing camera. Given this assumption, the sub only searches for the former set of landmarks in the images streamed from the corresponding source, and so on.

It is assumed that certain landmarks are only located within certain parts of the competition pool, and further assumed that given an arbitrary set of goals, only some subset of these landmarks need to be recognized. Given these assumptions, it is possible to search for only some subset of landmarks dependent on the submarine's current location and/or goal. Therefore, it is desirable to utilize some simple means of applying a landmark filter, with either narrowing or widening constraints, through recognition algorithms in order to improve performance. Such a filter is accomplished via a custom color- and shape-based filtering API that accepts a list of filter items, each specifying either a narrowing or widening constraint to be applied to the color or type of a landmark. For example, when Seabee is attempting to locate a buoy, it looks for orange pipelines and buoys of any color on approach, then looks for buoys of a single color on each buoy-touching attempt, then looks for only orange pipelines and yellow hedges as it attempts to locate the first hedge, etc.

*6. Sensing and Localization*

Currently, Seabee's most advanced sensor is an XSens MTi IMU. This inertial measurement unit provides the robot with "drift-free" 3D heading and acceleration data calculated by an on-board EKF fed by the device's accelerometers, gyroscopes, and magnetometers in real-time at 100 Hz. The team used this device effectively at the 2011 RoboSub competition to maintain a surprisingly accurate heading while navigating un-assisted within the competition pool.

Many other teams rely on a Doppler velocity logger (DVL), a very precise sonar device which can provide the linear components of velocity and pose of the sensor with respect to its environment. This sensor is currently out of USC AUV's price range, so the team must rely on alternate, often noisy sources of odometry. Furthermore, both the sensor and thruster configuration of the Seabee platform is almost always subject to change; depending on the progress of the electrical and mechanical teams, the team may or may not have a variety of sensors at its disposal, each with varying capabilities in terms of both functionality and measurement noise. For this reason, it was necessary to develop both a generic realtime simulation of the vehicle's dynamics, as well as a generic Bayesian measurement fusion system capable of combining all components of all observables, whether simulated or actual, into corresponding "filtered" measurements.

Given Seabee's current sensing capabilities, the linear components of pose and velocity (those that would be trivially provided by a DVL) are the most difficult to obtain. In order to compensate, a realtime simulation of the sub's dynamics is run using a software library called BulletPhysics. Given the vehicle's mass, per-axis linear drag coefficients, the current thruster configuration (per-thruster capabilities and relative pose), and the motors value being set on each thruster, Seabee is not able to calculate all components of pose and velocity with moderate accuracy. This output and any other odometry measurements are fused together on a per-axis basis into a complete odometry estimate.

When combined, the filtering and simulation modules allow for maximum functionality and modularity within the constantly changing constraints of the Seabee platform; as sensors are added and removed, the accuracy of the corresponding measurements will vary accordingly. For example, if a DVL were to be integrated into the platform, the accuracy of the linear components of odometry would be expected to increase significantly. As an added bonus, these systems also allow for the advanced testing of other software modules via arbitrary levels of simulation of sensor values and other information, in circumstances when a desirable real-world testing environment is

not practical or is entirely unavailable, or when the effects of a theoretical change to the system need to be studied.

*7. Navigation Pipeline*

Seabee's navigation system is built from modules with varying levels of specialization connected by generic interfaces. At a high level, its current implementation accepts a series of navigation constraints in the form of "waypoints," generates a trajectory with an arbitrarily high "temporal resolution", and then attempts to follow the trajectory within an additional set of constraints. This functionality is distributed over several modules including a trajectory planner, a high-level trajectory follower, a low- level velocity-based controller, and a platform-specific serial interface. Any module wanting to accomplish trajectory-based control of the vehicle must provide a trajectory to be followed. Within this system, a trajectory is composed of discrete intervals, each containing a constant acceleration over that interval and the desired state of the vehicle at the beginning of that interval, in the form of a waypoint (pose and velocity). Low-level velocity-based control is also possible; indeed, it is utilized by the trajectory following module, which is discussed in a later section.

A trajectory-planning paradigm was developed to simplify the creation of these trajectories while following waypoint-based constraints. In general, a trajectory planner accepts a list of two or more waypoints to be traversed in order, including the initial state of the vehicle, any intermediate states, and the final state of the vehicle, and returns a trajectory that meets the given constraints as closely as possible. The trajectory planner also accepts a temporal resolution parameter, which determines the maximum length of intervals over changing accelerations, as well as constraints on the maximum velocity and acceleration of the resulting trajectory (used to specialize the trajectory for the capabilities of a given platform). Currently in use is a linear trajectory planner, which iteratively generates a trajectory between each waypoint, ignoring intermediate velocities for simplicity. Starting with the initial vehicle state, the planner attempts to accelerate at the maximum given acceleration, up to the maximum given velocity, and generates a new trajectory interval for each change in acceleration, following a simple set of rules: if the error in position to the current waypoint is below some threshold, the planner returns an empty trajectory; otherwise, if the error in position is below some threshold, the planner attempts to strafe to the desired position, then attempts to face the desired heading; otherwise, if the error in position is above some threshold, the planner attempts to rotate the vehicle to face the location of the desired waypoint, then attempts to translate the vehicle to the desired position along its forward axis, and finally attempts to face the desired heading. After the final waypoint is reached, the planner returns the complete trajectory.

After a trajectory is generated by a trajectory planner, it is passed on to a trajectory follower for realization. In this case, the trajectory follower is generic due to the generic trajectory design; regardless of the implementation of the planner that generated the trajectory, the trajectory follower can use the same algorithm to traverse the given trajectory. Along with a trajectory, this module also accepts constraints related to the accuracy of the realization of the given trajectory, including the maximum deviation from the trajectory, both in pose and time, as well as the planner to use, if any, to attempt to recover in the event that the given constraints are not able to be met. If this failure occurs, the trajectory follower notifies the module that initiated the current goal, then attempts to re-plan from the vehicle's current pose to the beginning of the given trajectory, if a recovery planner was specified. In this way, the trajectory follower can be passed a trajectory that does not necessarily start at the vehicle's current state, and it will automatically prepend a path that brings the vehicle to the beginning of the original trajectory, if possible. For a given interval in the trajectory, the follower interpolates between the starting velocity and the calculated ending velocity according to the acceleration specified by the interval, for the duration specified by the interval, publishing a desired velocity at each step; this interpolation occurs at a user- specified rate, and can therefore be specialized for a given platform. Realization of a trajectory, then, is as simple as iterating over all intervals in the trajectory, interpolating, and performing any recovery applicable recovery behaviors.

The output format of the trajectory follower (velocities) was selected to ensure generic means of controlling a given platform. However, generic conversion from desired velocity to platform-specific motor values was not handled, though it is likely feasible. Instead, a PID-based controller over error between current and desired position is in use; specifically, for each movement axis, both an independent PID controller and a specialized conversion function. In the current implementation, these axes include all individual linear and angular axes.

The final actuation is handled by a custom serial driver written specifically for the vehicle's current hardware implementation. This driver is able to read sensor values, including internal pressure, external pressure, and kill switch state, as well as set the voltage of all motor drivers on the vehicle's power board.

In order reduce the significant overhead required to interact with the vehicle's complex control systems, a set of motion primitives was implemented that allows the vehicle to be commanded through a short list of simple yet powerful functions. These functions will automatically invoke the functionality of the appropriate navigation components mentioned earlier, and include the ability to move to the position and/or orientation encoded in a pose, align to a position, orbit a given position, and activate the torpedo launchers and marker droppers. Furthermore, the pose of a named object (whether static or dynamic), such as a buoy or any other landmark, can be trivially looked up and passed to these functions, enabling the creation of short, easily-readable high-level navigation code.

*8. Competition AI*

The final, highest-level controller in the software architecture is the competition AI. This component is responsible for directing Seabee to perform the most effective possible action at any given time, given all known goals and constraints. It is also responsible for enabling/disabling lower-level, situational modules and filters, such as those related to movement, object recognition, and behavior production. These complex requirements are fulfilled by discretizing all actions into subtasks, each containing any actions to perform as well as the cost (in terms of distance and estimated completion time) and reward (in terms of points earned) associated with the task. Then a task tree of arbitrary height is built, thereby allowing for an arbitrary level of task specificity, and feed this structure into a custom hierarchical cost-based decision algorithm, which attempts to maximize the overall reward earned in the allotted time. Any distance-based costs are converted into estimated completion time using the vehicle's current pose and any distance that would be accumulated while traversing the tree down to that task.

## IV. EXPERIMENTAL RESULTS

No log of hours spent testing in a pool was kept, but a number can be roughly estimated. Starting the second semester of classes, a test was held every weekend. Minus holiday weekends, that amounts to about 12 tests. Considering each test was approximately 3-4 hours long, that’s roughly 42 hours testing, and if about a third of that was spent in the water, that amounts to 14 hours testing in the pool.

During these hours the submarine was confirmed waterproof and positively buoyant by design, as well as mechanically balanced for neutral orientation in the water. Electrical systems were proven sound, and software proved capable of stabilizing the submarine and controlling the submarine’s movement. However, the defects of the submarine were also highlighted during these tests. Namely that the submarine is mechanically far from a rigid body, electrically its components are noticeably non-uniform, e.g. thrusters, and the integrator of the PID control system had to be eliminated for control reasons.

## V. ACKNOWLEDGEMENTS

Support from The University of Southern California, iLab, the USC Dornsife College of Letters, Arts, and Sciences Machine Shop, and the Viterbi School of Engineering allows USC AUV to continue to be an integral part of student research at the USC Viterbi School of Engineering.

Thank you to our industry sponsors: the Boeing Company, Northrop Grumman, Digi-Key Corporation, and Lockheed Martin.

## VI. REFERENCES

M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an open-source Robot Operating System,” in International Conference on Robotics and Automation, ser. Open-Source Software workshop, 2009.

## VII. APPENDIX – OUTREACH ACTIVITIES

The USC AUV team actively participates in inspiring a younger generation of students interested in STEM fields. Team members spoke with middle school and high school girls at a Women In Engineering (WIE) event on USC’s campus, volunteered at USC Viterbi’s Robotics open house, and served as counselors during the USC Viterbi Robotics Invitational.

The 2016 USC AUV team