Cornell University
Autonomous Underwater Vehicle

Fall 2022

Controls

Semester Review

Nathaniel Navarro (nrn25)

December 18, 2022

Contents

1 Abstract

Presented is an overview of work done over the course of Fall 2022 which aimed to address various issues encountered throughout the summer and semester regarding our ability to control mini-sub well. Much of the work involved digging through source code and documentation, both AUV's and third-party vendors'. Additionally, much time was spent attempting to isolate components of our control system in order to detect sources of error.

While no new software was written, a deeper understanding of our control system was developed, and documentation in the form of this paper and improved wiki articles has been created.

2 Goals

Goals:

- Familiarize oneself with control systems in general. This includes best practices regarding system modularity and PID loops.
- Gain understanding of AUV's control stack in it's current iteration in terms of design, implementation, and architecture .
- Locate sources of error in our control system leading to unsatisfactory control of minisub, isolate, and fix them in a way that avoid future issues with the same component.
- Troubleshoot immediate issues regarding our control stack, including issues with thrusters, the gx, optimization algorithms, and dead code.

3 Previous Implementations

Our control stack in it's current iteration was initially brought about in Fall 2012 in a project called "auv-control3." Jeff Heidel's technical documentation offers the most comprehensive look at how the control system was devised and intended to be used. Since then, the control system has been iterated on and modified numerous times.

Spring 2013 added a roll PID loop (although some information from that semester's technical documentation there has since been made irrelevant). Spring 2014 added passive force modeling and quaternion PID, and modified our optimizer. Spring 2016 and Fall 2017 added a navigation layer to our controls stack, which is in use to this day. (Spring 2016 is recommended reading!)

4 Design

The following is an iteration on the "Control Flow" section found on our wiki[1].

Largely, the control system is set desires through either control helm or framework methods. Our navigator takes these desires, and passes them through to our PID constantly running PID loops. At this point, our optimizer takes PID loop outputs and and find thruster outputs which most closely align with our desired PID loop outputs. The optimizer uses data from `thruster_manager.py` and `thrusters.py` to perform optimizations.

See the implementation section below for a more in depth explanation.

[1] https://wiki.cuauv.org/en/Software/Subsystems/Control

5 Implementation

The control system translates desires set through framework methods or control helm in the following way:

Control helm and velocity controls writes desires to a `navigational_desires` shm group.

The navigator reads from this shm group. If positional controls are being used, errors based on current position and desired position are calculated based on the `kalman` group and `navigation_desires` group. Additional trigonometry is performed to correctly map these errors to the sub's orientation.

If we are using velocity controls the navigator simply passes velocity desires through to the `desires` shm group.

At this point, a `PIDLoop` instantiated in `auv-controld3` reads from `desires` and `kalman` groups and calculates outputs based on the sub's PID tuning. Our `PIDLoop` object then writes to `control_internal_{}` groups, where {} is one of `depth`, `pitch`, `roll`, `heading`, `velx`, and `vely`.

Our optimizer reads from `control_internal_{}` to get setpoints from PID loops. The optimizer then solves the matrix equation $A\vec{x} = \vec{b}$ where A is a sub-to-thrust matrix (converting sub coordinate desires to thrust outputs for our motors) and $\vec{b}$ is our desired-output vector as thruster forces. $\vec{x}$ ends up representing our motor desires to produce $\vec{b}$. We convert the values of $\vec{x}$ into a PWM ranging from -255 to 255, mapping to expected values of PWM in microseconds based on T200 data[2]. These PWMs are written to shm group `motor_desires`. Our lovely ECEs have set things up so that this is directly translated into our thrusters spinning at their expected speed, based on said PWM.

Our optimizer uses 2 components. A thruster manager and a thruster python module.

Our thruster manager contains most of the matrices and utilities used in our optimizers calculations. We use it to limit thruster outputs and calculate coordinate conversion. Abstractly, this is what allows our isolated thrusters, which work at a base level without knowledge of one another, to be "mapped" to certain force vectors and torques, based on how they are positioned and oriented on our sub.

The thruster module contains hard-coded data regarding our thrusters, such as PWM-thrust curves, location with respect to sub center, orienta-

[2] https://bluerobotics.com/store/thrusters/t100-t200-thrusters/t200-thruster-r2-rp/

tion, and a few utilities which makes such information easy to access in the thruster manager and elsewhere.

To recap, desires travel from either control-helm or a mission to our navigator, to our PID loops, to our optimizer, which uses a thruster manager and thruster module to convert these desires into PWMs for our thrusters, which directly control the speed, and therefore thrust of thrusters.

The above is an overview of how the control system works.

Additional work was done this semester on a number of side-issues. Measurement of our sub's behavior in water by timing the sub (from a moving start) from various set points in the pool. We also measured drift by measuring movement from set points after a predefined amount of time had passed. Drift was also measured while in motion by setting a predefined heading, telling the sub to move forward, and seeing how much the sub drifted traversing one pool width.

Ways to communicate the gx were also explored. At first we hoped to obtain a mini-dv9 cable, however such cables are hard to come by. As a result we looked into the MicroStrain drivers and API that exist in `sensors/3dmg/gx4/SDK/` and communicated with the gx4 through serial via a c++ `calibrate.cpp` script in `sensors/3dmg/gx4/`.

Contemplating ways to empirically measure our thrusters' PWM-thrust curves, the idea of creating a bollard-pull test rig was also raised, which may see further pursuit.

6 Results

Documentation regarding our control system is improved, and technical issues plauging our sub have become less frequent. A large source of issues seemed to be two-fold. Old thrusters likely had PWM-thruster curves which differed from expected. Additionally, gx bias likely played a part in causing our sub to drift. Understanding the gx calibration to a deeper extent will likely come in handy in the future. Additionally, a framework for empirically measuring thrusters in the future has been reintroduced, and may be useful going forward.

7 Future Improvements

A refactoring of our control system may be overdue. The current system is a pain to grok, with large amounts of "magic" reading and writing to shm occurring opaquely. A better system may hold the results of calculations within a single control daemon, and write/read to shm transparently within said daemon. This may, however, come with memory limitations. Such a direction should be explore further.

Additionally, more work can be done regarding optimizer and thruster manager documentation, a task to be tackled next semester.

Finally, a way to empirically measure the well/poorly our sub should be devised. Bollard-pull test-rig would be a good start.