Acoustically Optimized Propulsion Systems Using a Cause-Effect Controller

by

J. Ben Thomson

B.Eng, Royal Military College of Canada (2001)

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the degrees of

Master of Science in Mechanical Engineering

and

Master of Science in Naval Architecture and Marine Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© Massachusetts Institute of Technology 2017. All rights reserved.

Signature redacted

Department of Mechanical Engineering

Signature redacted

May 12, 2017

Certified by ......................

Themistoklis Sapsis

Associate Professor

Thesis Supervisor

Signature redacted

Rohan Abeyaratne

Chairman, Department Committee on Graduate Students

JUN 21 2017
Acoustically Optimized Propulsion Systems Using a Cause-Effect Controller

by

J. Ben Thomson

Submitted to the Department of Mechanical Engineering on May 12, 2017, in partial fulfillment of the requirements for the degrees of Master of Science in Mechanical Engineering and Master of Science in Naval Architecture and Marine Engineering

Abstract

The proposed Acoustic Response Control Process (ARCP) allows for a "Cause-Effect" Controller (CEC) to provide a method to optimize the trade-off between a marine vessel’s ability to maneuver and the radiated acoustic noise (or internal vibrations). Through the discretization of both a marine vessel’s motion and all possible configurations for the vessel’s propulsion and control systems, a finite number of unique states can be created. The ARCP uses hardware and software together with a CEC to learn the cause-and-effect relationship between the finite states of the ship and the resulting acoustic noise and vibrations which are generated. Once this relationship is determined, the same CEC can use these relationships in reverse order to predict the amount of acoustic noise that would be generated by a proposed propulsion change or maneuver. In this context, any maneuver or speed change is simply considered a state change. The CEC could then choose the optimal new end state, along with intermediate states which provides the greatest speed, acceleration, and ability to maneuver without exceeding the user-defined acoustic thresholds.

The ARCP offers a number of both civilian and military applications. The ARCP, for instance, can be used on merchant ships to avoid certain frequency and amplitude combinations which interfere with marine life. Vehicles that use acoustic sensors could optimize their search speeds while minimizing their own noise which interferes with their sensors. For military applications, the ARCP will give vessels the greatest ability to maneuver while maintaining stealth from hostile acoustic sensors.

Thesis Supervisor: Themistoklis Sapsis
Title: Associate Professor
Acknowledgments

First and foremost, I would like to thank my thesis advisor, Assoc. Prof. Themistoklis Sapsis. Despite coming to him with a vague and very ambitious project idea, he took me on as a student and was extremely supportive, trusting, and patient throughout this project.

Secondly, I would like to pass on my appreciation to John Leonard and Mei Cheung for letting me use one of their autonomous kayaks. I am especially appreciative of the time that Mei took to show me how the kayaks were configured, and her trust in letting me use and modify these highly specialized autonomous vehicles.

I would like to thank Prof. Michael Triantafyllou and Dixia Fan for their help and guidance with finding a space where I could work on this project, as well as letting me use the tow tank in which to conduct early testing.

I also like to thank Dr. Michael Benjamin and Dr. Michael "Misha" Novitzky for their technical assistance with MOOS-IvP. In addition to help with coding, they were also very supportive in helping me to find my way around the MIT Sailing Pavilion.

On a personal note, I would like to thank my dear wife Suzanne. She has been very supportive of the long hours spent on this project. She not only kept me hydrated and caffeinated, but even helped me by proof-reading this document.
# Contents

1 Introduction ........................................... 17

1.1 Terminology and Definitions ....................... 19

1.2 Motivation and Possible Applications ............... 32

1.2.1 Optional Additional ARCS Sub-Systems ............ 36

1.3 ARCP Description & ARCS Examples ................ 37

1.3.1 General Description of the ARCP ................. 37

1.3.2 Alternative Ad Hoc Process ...................... 51

1.3.3 Example of an Unmodified Propulsion and Maneuvering System 53

1.3.4 Example of Integrating an ARCS into an Existing Propulsion and Maneuvering System ................. 54

1.3.5 Example of an ARCS with a Bypass Switch ........... 56

2 Prior Research and Theory .......................... 57

2.1 Theory ................................................... 57

2.1.1 Ship Noise - Machinery ............................ 58

2.1.2 Propeller Noise .................................... 61

2.1.3 Non-Uniform Wake Fields .......................... 66
2.1.4 Effects of Changing Propeller Advance Ratio
2.2 Prior Research in Measuring Ship Noise
  2.2.1 Hull Vibration Noise
  2.2.2 Full-Scale Maneuering Acoustic Signature
  2.2.3 Predicting Ship and Submarine Noise using CFD
  2.2.4 Alternative Use of Sensors to Predict Cavitation Noise
2.3 Prior Research in Other Engineering Applications of Machine Learning
  2.3.1 Deep Belief Networks to Localize Plate Failure
  2.3.2 Machine Learning to Diagnose Diesel Engines
  2.3.3 Vibration Mitigation with a Fuzzy Logic PID
2.4 Research Summary

3 Prototype Development, Hardware & Software Configuration
  3.1 Introduction
  3.2 Hardware Configuration
    3.2.1 Baseline Hardware
    3.2.2 Hardware Modifications
  3.3 Software Configuration
    3.3.1 Existing Open-Source Software Used
    3.3.2 MOOS Software Modifications
    3.3.3 New Software Developed

4 Experimentation, Testing, Conclusions, and Future Work
4.1 Numerical Value Descriptions ........................................... 109

4.2 Incremental Testing & Results ........................................... 110

4.2.1 Tow Tank - Static Testing ........................................... 110

4.2.2 Tow Tank - Dynamic Testing ........................................ 131

4.3 Conclusions ............................................................... 137

4.4 Lessons Learned .......................................................... 137

4.5 Future Work and Improvements ........................................ 138

A Hardware Specifications & Data Sheets ................................ 139

A.1 Data Sheets and Technical Information for the Following Hardware - Listed In Order: ........................................... 139

B Software - Code ............................................................. 159

B.1 MOOS Mission Configuration File ....................................... 159

B.2 Customized Inertial Measurement Unit MOOS Application -C++ Code 170

B.2.1 Custom IMU Program Header File - SpecIMU.h .................. 170

B.2.2 Custom IMU Source Code - SpecIMU.cpp ........................ 174

B.3 Customized MOOS Application For Recording Self-Noise -C++ Code 189

B.3.1 Read Self Noise Application Header File - ReadSelfNoise.h ... 189

B.3.2 Read Self Noise Application Source Code - ReadSelfNoise.cpp 191

B.4 Cause-Effect Controller Application - C++ Code .................... 198

B.4.1 Cause-Effect Controller to PID Header File - CEPID.h .......... 198

B.4.2 Cause-Effect Controller to PID Source Code - CEPID.cpp ...... 203
List of Figures

1-1 Black Box Concept and Fine State Machine, Images from Gill, 1962 [8] 21
1-2 Marine Vessel As a Plant - Standard Control System Terminology . . . 40
1-3 Calibration Stage of the ARCP ................................................. 42
1-4 Example of a "Greedy" Recursive Approach to Selecting Intermediate Ship-Propulsion States .................................................. 48
1-5 Operate Stage of the ARCP .................................................... 49
1-6 Example of the Alternative "Ad Hoc" Process .............................. 52
1-7 Example of an Unmodified Ship Propulsion and Maneuvering Control System. Note: Ship has Three Inputs: Rudder, Engine, and CPP . . . 53
1-8 Shipboard Acoustic Response Control System (ARCS) - Fitted Between Autopilot User Interface and Processor .......................... 54
1-9 Shipboard Acoustic Response Control System (ARCS) - Fitted Before Autopilot, and Replacing User Interface .......................... 55
1-10 Shipboard Acoustic Response Control System (ARCS) - Fitted Between Autopilot and Individual System Controllers ...................... 56
1-11 Example with an ARCS Bypass Switch to Allow Direct System Control 56

2-1 Types of Propeller Cavitation - Image from Kerwin, 2010 [10] ....... 63
Propeller Characteristic Curves - Image from Ross, 1976 [14] ...... 70
Images from Trevorrow, 2008 [16] ........................................ 73
HoverGroup Autonomous Kayak With Cart ............................. 86
Thruster Pod and Skeg ....................................................... 87
Labeled CPU Box Enclosure ................................................. 88
Onboard Gumstix Computer ................................................. 90
Installation of the IMU ....................................................... 93
Hydrophone Suspended to the Side of the Kayak ...................... 96
CEPID Application Logic Flow ............................................. 108
Example of Audio Clipping - Image from wikipedia.com ............ 110
Static Tow Tank Arrangement .............................................. 111
Propulsion Step Test - Max Amplitude .................................. 112
Propulsion Step Test - RMS Amplitude ................................ 112
Propulsion Step Test - RMS Amplitude - MATLAB Smoothing Function 113
Propulsion Step Test - RMS Amplitude - "Savitzky-Golay" Filter .... 114
Propulsion Step Test - RMS Amplitude - Using Sector Mean ...... 114
Response Showing 0.6-Second Time Delay Between Thrust and RMS Amp ................................................................. 115
Propulsion Step Test #1 - Using Sector Mean - Multiple Tests .... 116
Propulsion Step Test #2 - Using Sector Mean - Multiple Tests .... 117
4-11 Rudder Step Test: Rudder Data Only
4-12 Rudder Step Test with Propulsion Step Test Data
4-13 Propulsion Step Test - 0.1-Second Sample Size
4-14 Propulsion Step Test - 0.15-Second Sample Size
4-15 Propulsion Step Test - 0.18 Second Sample Size
4-16 Propulsion Step Test - 0.2-Second Sample Size
4-17 Acoustic Response Averages For All Values
4-18 Acoustic Response Averages For Thrust >= -65%
4-19 Original and "Groomed" Data for Set 1, Thrust >= -65%
4-20 Original and "Groomed" Data for Set 2, Thrust >= -65%
4-21 "Groomed" Acoustic Response for Both Sets: Thrust >= -70%
4-22 Large Propulsion Changes Response for Set 1
4-23 Large Propulsion Changes Response for Set 2
4-24 Large Propulsion Change Response for Set 2 - 1.5 Second Delay
4-25 Kayak on Zip Line - Before Start of a Run
4-26 Kayak on Zip Line - During a Run
4-27 Non-Powered Drag Test to Assess Flow Noise
4-28 8-Second Dynamic Runs - Set 1
4-29 Zoomed-In Response for Dynamic Runs - Set 1
4-30 0 to 50%
4-31 0 to 50% to 60%
4-32 0 to 70%
THIS PAGE INTENTIONALLY LEFT BLANK
List of Tables

4.1 Sample Duration vs Effective Sampling Rate .................. 122
4.2 Acoustic Response Averages For Each Set of Step Tests .......... 123
4.3 Large Propulsion Change Comparison with 5% Step Test .......... 130
Chapter 1

Introduction

As the affordability, miniaturization, and processing power of modern computers continue to evolve, it opens up new options to address engineering problems. Experience-based approaches, such as machine learning, give practical solutions which can often provide real-world results that are comparable to traditional approaches using deductive methods. In many cases where there are a large number of complex interactions, analytical methods for solving equations, can become prohibitively complicated. Even experientially derived empirical equations still require a large number of experimentally derived coefficients that are unique to each application. This in turn requires multiple experiments or simulations for each application. Alternatively, using computer algorithms to directly map the stimulus or input (cause) to a response (effect) becomes a viable alternative. Different applications will still produce unique results, however the same algorithms can be re-used for multiple applications.

By directly data-mapping the cause and effect of complicated multi-variable systems, useful results can be obtained without having to identify all the contributing factors, and their potentially even more complex interdependences. Also, inherent within this approach is also the ability to dynamically adapt to multiple environments, or system deterioration, by re-running the same algorithms as needed. Additionally, the same cause-effect relationship can also be used as a health-monitoring system,
alerting users when the system is suddenly not performing as predicted.

One possible way to integrate a cause-effect approach into real-world engineering problems, is do so as a system controller. Specifically, a Cause-Effect Controller (CEC), is presented here as a method for a sea-going vehicle to learn how changes to its propulsion system affects the amount of acoustic noise generated. Once this relationship is determined, it can be used to predict the amount of noise that will be generated for the full range of the propulsion system. With complete predictions, a CEC can then be used to optimize the ship’s performance in terms of speed without exceeding user-defined acoustic noise thresholds where possible.

A detailed explanation of the proposed acoustically optimized shipboard propulsion system will be fully introduced here in Chapter 1. This will be done by first defining the terms and definitions used throughout this paper followed by motivations and applications specific to marine propulsion. With the intent of focusing on generalized algorithms that can adapt to a plethora of applications, a more-encompassing Acoustic Response Control Process (ARCP) will be introduced. Subsequently, a number of tangible examples of how CECs could be used to build an Acoustic Response Control System (ARCS) on board ships are given. By using specific examples, the primary stages of the ACFP are shown in how the cause-and-effect relationships between parameters are established, and subsequently how these relationships are then used for optimization.

Chapter 2 will discuss prior and relevant research conducted in these areas, and how it relates to more informed decisions in developing this proposed system. This chapter will also include other approaches to measuring and predicting the noise generated by ships.

Although the use of sophisticated algorithms were beyond the scope of this project, simplified algorithms were developed as a proof-of-concept that a CEC is indeed a viable approach to this application. Chapter 3 will go into detail about the hardware and software used in which to develop the cause-effect data map.
Chapter 4 will discuss how developmental testing was conducted and the results of validation experiments conducted to date. This is also where some areas for further research will be identified, as well as identifying ways that the current design could be improved. This will include how the same system can be used to measure and predict noise generated as the vessel accelerates, decelerates, and due to abrupt changes to the propulsion system.

1.1 Terminology and Definitions

Controllable Pitch Propeller (CPP): Also known as a Variable Pitch Propeller (VPP), it is a ship propeller in which the pitch angle of the propeller blades can change by rotating the blades on their longitudinal axis. For this discussion, CPP also includes Reversible Pitch Propellers (RPPs) where the pitch angle can be reduced to negative values, providing reverse thrust without changing the direction of rotation.

Propulsion System: For the purpose of this discussion, the propulsion system is specific to the controllable components of the ship that interact with water in such a way that acoustic noise or vibrations are generated. For propeller-driven systems, it would include controlling shaft speeds, shaft acceleration, and propeller pitch (if fitted with CPP). In the case of water jets, it includes impeller speeds, impeller accelerations, impeller pitch (if controllable), and the nozzle system.

Maneuvering Control System: The maneuvering control system is what allows watercraft to control and change their direction (heading), as well as depth for submersibles. The system can be independent of the propulsion system as is the case with a rudder system. It can also be incorporated into the propulsion system, such as with directional nozzles, directional pods, or differential drives.

Acoustic Response: In the context of this discussion, acoustic response refers to near-field and far-field acoustic noise that is generated outwardly from the ship,
as well as vibrations that are transmitted internally into the ship structure or within the equipment itself, depending on the application. The acoustic response can be a single variable such as only broadband noise amplitude or power or the response can also contain multiple single variables such as recording both Root Mean Square (RMS) amplitude and peak amplitude. Alternatively, the response can be multi-variable, where for instance the amplitude is measured separately, and sub-divided into several different frequency bands. The latter could be useful in applications where only specific frequencies are paramount for optimization.

Acoustic Sensors: For this discussion, acoustic sensors refer to the broad range of sensors that are used to measure the acoustic response as previously defined. In the case of internal ship vibrations, accelerometers are often used. For near- and far-field acoustic noise, pressure transducers/hydrophones are typically used. The acoustic sensors are often either fitted to the vessel such as with hull-mounted hydrophones, or towed by the vessel as with a towed array. However, acoustic response can also be measured using sensors that are geographically fixed in the water, using deployable buoy sensors, or even fitted or towed by another vessel.

Environmental Conditions: Information provided by the acoustic sensors is affected by environmental conditions such as water temperature, water salinity, and sea state. Sensor input data can be corrected for environmental conditions with data gathered directly by additional sensors, or instead using available historical environmental information based on geography.

Finite State Machine The "black box" conceptualization of a digital system is to view that system only in terms of the data going into the system and the resultant data coming out from the system. The details of how the system determines the correct outputs, based on the inputs, is intentionally omitted in a "black box" conceptualization, as it would occur inside the "black box". This conceptualization, then allows for any number of solutions internal to the black
box to be equivalent, provided that it consistently produces the correct data outputs based on the same inputs. This simplification neglects other factors such as timing, power consumption, and heat load. A finite state machine is a hierarchical approach to replicating the "black box" by first determining a finite number of states, which in most cases would correspond to the number of possible input combinations. It is also assumed that the required outputs for each of these input combinations can also be determined. This allows for the system inside the "black box" to be replaced by a finite state machine where the system can only be in one state at a time, as determined by the input, and will have the required corresponding outputs while in this state. State transitions are then handled with a knowledge of the current state, as well as the next state needed, as the inputs change [8].

Figure 1-1: Black Box Concept and Fine State Machine, Images from Gill, 1962 [8]

**Ship-Propulsion State:** In the context of this project, the ship-propulsion state is the set of input information that is needed to fully define what the ship, maneuvering control system, and propulsion system are doing at the time that the acoustic response is being measured. The ship propulsion state will vary in complexity based on the application. As per a finite state machine, it is important that the ship can only be in exactly one state at a given time. A simplified ship-propulsion state would be comprised of a single variable that simply defines
the amount of power being used for propulsion under steady speed conditions without maneuvering. A more complicated ship-propulsion state, as needed to measure the acoustic responses under unsteady conditions, would contain at least three variables: the ships speed, the power being exerted by the propulsion system, and the new desired ship speed. Depending on the application, there is flexibility in terms of which variables are used. For example, it may make more sense to use shaft speed [rpm] instead of propulsion power or to use new desired power instead of new desired ship speed. Although well beyond the scope of this project, an advanced system would have a ship-propulsion state that would include even more variables, such as the status of the maneuvering control system, the motion of the ship (in terms of yaw pitch and roll), the pitch angle of the CPP, and possibly even the associated rates of change for each of these variables. The complete list of parameters that make up the ship-propulsion state can be further sub-divided into plant input parameters, and plant output parameters. If unfamiliar with the term "plant" in the sense of control systems, please refer to the definition under "Marine Vessel as a Plant" that appears later in this section.

**Data Map:** In the context of this paper, data mapping is the process of linking two or more independent data models so that they can be cross-referenced. The data contained in the models, along with the method of cross-referencing together, comprise the actual data map. In the context of this project, the data map would link the ship-propulsion state to the corresponding acoustic response. Note that where the ship-propulsion state contains multiple variables, all permutations of these variables will need be measured and mapped separately or somehow interpolated. This data map is what defines the cause-effect relationship, in which the ship-propulsion state is the "cause" and acoustic response is the "effect". It may be necessary to modify or keep different data maps for different environmental conditions.
Response Threshold(s): These thresholds are effectively the acoustic response limits that the user has defined as being acceptable. Optimization in the context of this project is to maximize performance of the ship while staying just below these limits/thresholds. The data complexity of the threshold should match the complexity and form of data response measurements. For example, if the application is only concerned with broadband RMS amplitude, then the threshold may be a single value. If the application called for different limits depending on frequency, then the response threshold would be comprised of different values for each frequency band measured in the data map. For example, merchant ships may wish to restrict only the bandwidth of frequencies that interfere with sealife communications, while a military vessel may be concerned with broadband noise and specific tonals. As per acoustic response, this definition includes internal vibrations and both near- and far-field acoustic noise. The response threshold value can be either be a static absolute value, or a dynamic variable, such as a percentage of the ambient noise (either measured or historical).

Navigational Aids: In general terms, navigational aids refer to the broad range of equipment normally installed on most commercial and military vessels that assist in navigating the vessel and avoiding collision. In the context of this discussion, the relevant navigational aids are those systems that will provide the information for the aforementioned ship-propulsion state. Examples of common navigational aids include a GPS (position, heading, and ground speed), a Doppler speed log (speed through the water), and a gyrocompass (heading). Medium sized and larger vessels will also have radar and an echo sounder (to measure water depth). Larger commercial vessels and military vessels will be equipped with an electronic charts system, which automatically takes data from other navigational aids and incorporates the data with digital charts (maps) to help the crew navigate. In addition, military vessels will often have an inertial navigation system (INS) that provide heading, pitch, roll, ground speed, acceleration/deceleration, and positional information.
Own-Ship Noise (OSN):  *Own-Ship-Noise (OSN)* is the collective acoustic energy in form of pressure waves from the host platform’s *propulsion system*, flow noise, and auxiliary systems which are fed back into the host platform’s own sensors. This noise interferes with the sensor’s abilities to detect and track distant objects.

Acoustic Signature:  *Acoustic signature* is often used in a naval context to describe the amount of acoustic noise generated by a ship. The meaning changes slightly based on the context as it can refer to a generalized measure of the amount of noise being generated, or it can refer to specific combinations of frequency and amplitude which can be used by more sophisticated sonar systems to identify the specific ship, or class of ship.

Propeller Slip:  One way to understand this concept is to visualize a fixed-pitch mechanical screw advancing into a solid as it is rotated. The pitch of the screw relates to how far it will advance to how much it is rotated by. In this ideal analogy, we expect that the screw will advance exactly as the product of the pitch and the amount it was rotated. Screw propellers, however, operating in a much less efficient fluid medium will never achieve an advance exactly equal to the product of the pitch and rotation. *Propeller slip* is a measure of how much the screw propeller is falling short of the ideal condition. The mechanical screw in a solid will have no slip while a screw propeller will have a value greater than zero. There will be a steady amount of slip when neither the rotational speed nor the speed of advance are changing with respect to time. [6]

Forced Propeller Slip:  There will always be some *propeller slip* at steady conditions. However, the amount of *propeller slip* can be altered from the steady-state value by not allowing the rotational velocity of the propeller and the speed of advance of the propeller through the water to reach their normal equilibrium. For the purpose of this discussion, we introduce a new term, *forced propeller slip*, which refers to when the propeller is slipping more than the steady-state propeller slip because of a mismatch between the speed of advance and rota-
tional speed (angular velocity) of the propeller. The mismatch can occur in either order. For example, a ship operating at a steady speed alters course using a rudder. The increased drag of the rudder, together with the increased resistance of "side slip", will slow the vessel down as it turns, however the shaft maintains a constant rotational rate. The forced slower speed will reduce the amount the propeller advances, and therefore causes an increase Propeller slip. Another mismatch can occur when a ship tries to accelerate quickly. The propulsion system is mechanically linked and has considerably less mass than the rest of the ship and therefore the shaft will be able to reach the new rotational speed relatively quickly. The massive ship will have a lot more momentum and will take significantly more time to reach a new steady speed of advance. In this case, there will be a large amount of propeller slip initially, which will eventually decrease until the ship reaches the new steady speed. This is analogous to a car spinning its wheels when trying to aggressively accelerate; the wheel’s angular velocity increases much faster than the car’s axial velocity. The resultant increased acoustic response that can be seen during forced propeller slip can be shown as it relates to cavitation inception [14] as well as being directly measured via experimentation [16]. A more detailed explanation, along with associated equations, are given in Chapter 2.

**Cavitation Inception Speed (CIS):** This is the lowest sustained vessel speed at which the propeller(s) causes significant levels of propeller cavitation. As the vapor cavities that are generated above this speed collapse, they significantly increase the noise generated by the propeller. Therefore, Cavitation Inception Speed (CIS) can be considered as a threshold where there is a noticeable difference in generated noise above and below this speed. Note that significant propeller cavitation can still occur at slower speeds, but this is usually at speed transitions.
Unmanned Vehicle (UV): In general an unmanned vehicle (UV) can be land-based, aerial, or sea-going. In the context of this project, the focus is only on sea-going vehicles. Unmanned surface vehicles (USVs) refer to autonomous or remotely controlled vehicles that operate on the surface of the water. Unmanned undersea vehicles (UUVs) refer to autonomous or remotely based craft able to operate at range or depths below the ocean surface.

Cause-Effect Controller (CEC): A cause-effect controller (CEC) is based roughly on the concept of a black-box finite state machine, but is different in that for a given state, the levels of the controller outputs associated with that state are not known at the time that the controller is built. Additionally, the values of one or more input into the controller, which determines the states and when state transitions occur are also not yet established. Instead of using pre-determined linear states with predetermined outputs, the CEC creates an experience-based data map that directly links the state of the system to the predicted response. For example, in the context of this project, the CEC will conduct calibration runs to create a data map between the ship-propulsion state and the acoustic response. At this time, the CEC will also know the user-defined response thresholds. When the controller is given a new desired ship-propulsion state, for example an increase in the ship’s speed, the CEC will look in the data map to determine which ship-propulsion state best matches the new requested state and is also below the response thresholds. The CEC will then order the propulsion system to adopt this new state. This resultant state will be the optimal performance of the propulsion system without exceeding the response threshold. CECs are designed to be recursive in that they can effectively adopt many different ship-propulsion states while incrementally approaching the optimal steady-state. In other words, the system will not simply jump to the new desired speed as quickly as possible, but will instead control the rate at which the ship accelerates to ensure it accelerates as quickly as possible while refraining from exceeding thresholds unnecessarily at each step along the way.
CECs are more complex in terms of hardware than finite state machines as the relationship between input and outputs can not be hard-wired prior to the system being manufactured. The real advantages of determining these values post-manufacturing, is that the same CEC, with the same algorithms, can be used on many different platforms without the need to experientially determine specific values before construction. Additionally, the commonality allows for mass-production without the need to design and build systems custom to the individual platforms. A CEC is also more adaptive to changing conditions as it can be dynamically re-calibrated as required when platform or environmental conditions change.

**Marine Vessel as a Plant:** In which to make this discussion as straightforward as possible, established terminology specific to ship maneuvering and control systems is used where applicable. In the context of controllers, the basic elements that make up a control system are: a plant, inputs, outputs, and sensors. Inputs act on a plant in which to produce the desired outputs. Therefore, to be consistent with other work in the field with respect to maneuvering and control of ships and submarines, the marine vessel itself is the plant. The inputs to the plant are the systems or sub-systems that can be directly controlled, such as the propulsion power and rudder angle. The outputs would be the resultant effects on the plant, such as the actual ship's speed and the ship's heading [17]. With data mapping, certain signals may be an output at one stage of the data mapping process, but become an input signal at a later stage. Also, because a CEC uses data mapping, it is interested not only in the relationship between the inputs and resultant outputs, but also the relationships between different inputs and between different outputs. To avoid ambiguity between data going into and coming out of the CEC as inputs/outputs and what are plant inputs/outputs in the context of traditional controllers, the terms "input parameters" and "output parameters" are used to refer to the latter. The general term: "parameters", refers to input and output parameters of the plant.
Proportional-Integral-Derivative (PID) Controller: PID controllers are used in many applications across many industries. PID controllers are closed-loop feedback controllers that perform very well in real-world applications because of their ability to correct for errors. An error in this context is the difference of the desired output and the actual output of the plant. The proportional function of the controller will increase the correction feedback proportionally to the amount of error so that the controller is neither driving the system excessively, nor failing to provide enough correction to overcome the error. The derivative aspect of the controller measures how quickly the system is recovering, so that it can decrease the level of correction when near the desired output in order to reduce overshoot. Finally, the integral function examines error over time, and therefore corrects for slow, cumulative errors such as drift. When the PID is "tuned", the coefficients that determine the levels of each of these functions are set such that the controller is optimized for the specific application. The PIDs are built with an understanding of the physics that describe the system and tuning is done through modeling. However, there are many applications where there is insufficient knowledge of the physical system, or it is too difficult to model. This is where heuristic approaches are instead used [17]. Methods for tuning difficult-to-model systems have been around for many years and are well established [19], however these forms of tuning are still performed pre-construction. More recently, with developments in computer science, new heuristic methods are available to conducted tuning using algorithms, and with the use of firmware or software, the PID tuners can be tuned and re-tuned at run-time. PID controllers need to determine the error between the desired output and the actual output in which to provide corrections; this is done through feedback sensors.

Cause-Effect Proportional-Integral-Derivate (CEPID): While a CEC is able to determine what the new desired outputs should be, it has no ability for ensuring that it stays in the desired state, nor has the ability to correct for error and stabilize the system. This is where a CEC can be paired with a PID
controller; herein referred to as a Cause-Effect Proportional-Integral-Derivative (CEPID) controller. The CEC portion of the CEPID will resolve the states and determine what the desired state and outputs should be. The PID portion of CEPID would then in turn take the desired output and adjust these outputs to correct for errors, and in doing so, would ensure that the state of the CEC is persevered. With recent advances in heuristic methods of tuning PIDs, this allows for the entire CEPID to be adaptive so it can fully calibrate at run-time on different platforms using common hardware and software.

**Acoustic Response Control System (ARCS):** An *Acoustic Response Control System (ARCS)* is a general term for all integrated sub-systems which are used on board ships to provide optimization between propulsion system performance and the acoustic noise generated. With a CEC or CEPID at the core, the ARCS also includes other sub-systems, such as the *acoustic sensors*, the interfaces with *navigational aids*, the user-machine-interface, and the interfaces to the ship’s *maneuvering control and propulsion systems* which is usually via the autopilot system. The ARCS may also include data logging sub-systems. The amount of hardware, firmware, and software required for an ARCS will depend on what sensors and *navigational aids* the host platform has available, and how difficult it is to interface with them.

**Acoustic Response Control Process (ARCP):** Where as an ARCS refers to a system adapted to a type of platform, an *Acoustic Response Control Process (ARCP)* is a more generic term for how an ARCS determines the cause-effect relationship and how it is used. The ARCP allows ship operators and designers to implement an ARCS that maintains externally transmitted noise (or internally transmitted vibrations) below certain thresholds (*response thresholds*), while still maximizing the platform’s speed and maneuverability within these thresholds. Although discussed in more detail later in this chapter, for most applications, the ARCP has at least two distinct steps. The first stage is *"calibration"* where CEC develops the required data maps. The second stage is
"operate", where the CEC data map is used to predict and appropriately modify the command signals to the propulsion and maneuvering systems based on the given response threshold. Although more of a parallel or alternative process, an ARCS can also be operated in an "ad hoc" mode where instead of predicting the acoustic responses, it acts reactively to decrease propulsion orders and adjust maneuvers once they have exceeded the response thresholds.

**Input Parameters:** As previously discussed, in the context of a shipboard propulsion control system, the plant would be the vessel itself. The plant input parameters are propulsion and maneuvering control signals that the ship crew, or autopilot, normally have direct control over, and are used to indirectly control the platform's heading (direction) and speed. Depending on what propulsion system is fitted on the platform, the input parameters will differ slightly, but generally perform the same functions of indirectly controlling the vessel speed and heading. For example, in the case of a traditional ship propulsion system, the plant input parameters (which the crew would have direct control over) would include the shaft speed and rudder angle. By changing the shaft speed and rudder angle directly, the ship speed and heading are subsequently affected indirectly. For a platform equipped with a water jet system, impeller speed and nozzle angle would be the input parameters. Submersibles have additional hydroplane control surfaces which are also plant input parameters and are used to control the vessel's depth. Some vessels have maneuvering thrusters which would use different input parameters (thruster power and direction). On some platforms, propeller pitch can also be controlled directly, and in this sense would also be considered an input parameters for the plant. The term input parameter should not be confused with the line input(s) into the CEC, which includes both input and output parameters in the context of a plant.
Output Parameters: Output parameters refer to the plant outputs in the traditional controller context, and therefore are parameters that can be influenced and measured, but not directly controlled such as the ship’s speed, heading, or depth. The purpose of a control system like a PID controller is to provide a stable desired output, despite disturbances, through feedback obtained by continually adjusting the plant inputs as required. For example, a control system designed to help a ship maintain a desired heading (yaw), despite external disturbances such as wind and waves, would use a gyrocompass as a sensor to measure the actual heading (output) and provide this value in the form of feedback. This control system would then compare the actual heading to the desired heading to determine the error, and would then adjust the rudder angle input in which to correct the error. In this case, the focus of the control system is on the primary output being the ship’s heading, however changing rudder angle also causes secondary effects such as causing the ship to roll, increased flow noise, and a slight deceleration, some or all of which may be measured. A simple PID controller would not have the need to measure these secondary outputs as they would be considered separately in other stages of the ship design process. In the case of a CEC, in which to fully measure and identify the separate ship-propulsion states, many of these secondary outputs become relevant, even if they are not used for feedback to stabilize the plant. Therefore, output parameters refer to all relevant plant outputs, both primary and secondary that cannot be directly controlled. Note that some output parameters that are used by the CEC do not need to be measured independently, but can instead be calculated where derivative and integral relationships exist. For example, although heading would be measured by a gyrocompass, the turning rate would be the first derivative of the heading with respect to time. The term output parameter should not be confused with the line outputs from the CEC, which includes both input and output parameters in the context of a plant.
1.2 Motivation and Possible Applications

In which to optimize a ship propulsion system, the first step is to identify what parameters are the focus of the optimization. Some examples of the parameters of the propulsion system that designers and platform operators may wish to optimize include:

- **Performance Abilities**
  - Top Speed
  - Maneuverability / Turning Rate
  - Acceleration and/or Deceleration

- **Precision**
  - Maintain a Static Position
  - Precise Speed and/or Heading

- **Fuel Economy**
  - Endurance - Maximum Time on Station
  - Most Economic Speed

- **Reliability**
  - Mean Time to Failure
  - Redundancy / Robustness

- **Noise and Vibration**
  - As Experienced by Crew or Passengers
  - Transmitted into the Water
  - Interference with Own Sensors and Equipment

For each of these parameters, we need to distinguish between two categories of optimizations: 1) those which occur pre-construction during the design phase, and 2) operational optimizations which refer to how the system is employed post-construction. During the initial design phase, several of these parameters will be synthesized into what is considered the best overall design for the specific application, understanding that there will be many inherit trade-offs, as not all parameters can be optimized concurrently. For example, with large merchant ships, the designer will focus on fuel economy at moderately high sustained speeds, and will be less concerned with maneuverability or noise. Designers of warship propulsion systems,
however will focus on top speed, maneuverability, and the noise generated, while being less concerned with fuel economy.

Once a design is finalized, it will have specific capabilities and limitations with respect to the aforementioned parameters. For example, any given ship will have a single most economic speed: the speed where they can travel the largest distance for a given amount of fuel. The ship will also have a single top sustained speed: the highest speed it can maintain without risk of damage to the equipment. There will also be a specific speed where the vessel will have the greatest turning rate. In addition, the vessel will have a specific CIS and also a minimum speed at which the vessel is still able to effectively maneuver on its own. This is an instance where operators of these vessels have the ability to choose between a variety of settings and speeds in which to perform specific operations. The operators are of course confined by the propulsion system’s physical capabilities and limitations, but also by their abilities to perfectly time and coordinate multiple system changes under dynamic conditions. This subsequently introduces the concept of operational optimization, where the goal is to optimize how the vessel is best used at different times and in different roles with a fixed propulsion system design. The operational optimization can be quite trivial, such as simply adopting the most economic speed with which to optimize range, but can be much less obvious for multi-variable optimization.

The specific goal of this project is to design a system that will optimize the trade-off between propulsion system performance and the amount of acoustic noise generated in the water. This will be operational optimization as described above, since the intent is not to re-design a propulsion system, but rather to optimize control of it within the established system’s physical capabilities and limitations. This is also considered multi-variable optimization, as both the aspects of propulsion system performance, such as speed, and the maximum allowable acoustic noise response are both controllable by the operator. As more aspects of performance are added, such as acceleration and deceleration, the number of variables increases accordingly.

The benefits of optimizing parameters such as the maximum sustained speed
or fuel economy are fairly evident. However, the benefits of optimizing a propulsion system such that certain response thresholds are not exceeded is less obvious. For this reason, an ARCS would be limited to possible applications where limiting the amount of acoustic noise being generated in the water is just as, if not more important, than the overall system’s performance. For the applications given, it is assumed that a CEPID controller is being used with a user-set acoustic response threshold. The applications may refer to only the CEC, CEPID controller, or the ARCS where the latter also includes all associated sub-systems and interfaces. Either way, the more encompassing ARCP is being used.

**Sensor Optimization:** Applicable to submarines, surface combatant ships, research vessels, survey vessels, and UVs. In general, OSN is reduced at lower speeds, however higher speeds are desired in which to maximize the search coverage. An ARCS with the response threshold set for the frequency and the anticipated returned signal strength for the item being searched for, can then control the host vessel at an optimized speed, thereby giving the best coverage while maintaining the maximum likelihood of sensors detecting the item. Specific examples could include a warship searching for a submarine, or a group of USVs searching for the black-box transmissions from an aircraft lost at sea. Note that in which to avoid certain combinations of amplitude and frequency, this does not necessarily require the host vessel to reduce speed, as an increase in speed may also provide a frequency shift in OSN, achieving the desired effect.

**Signature Reduction (Stealth):** Applicable to submarines, surface combatant ships, and military UVs. In which to maintain a tactical advantage, ship operators and designers want to minimize the likelihood of being detected by hostile vessels, while at the same time not overly restricting their own ability to maneuver (speed, acceleration, and depth/course changes). An ARCS, calibrated with a threshold that matches the anticipated hostile sensor capabilities, can give the host platform the maximum ability to maneuver while minimizing the likelihood of being detected by hostile platforms. Optimization is achieved by carefully
controlling the *propulsion and maneuvering control systems*, as well as avoiding certain conditions where excessive noise/vibration is generated, such as with propeller cavitation or hull resonances. A military-specific is with the use of acoustic (noise-making) countermeasures that act as decoys against hostile torpedoes. The platform employing the countermeasure will want to distance itself from the decoy as quickly as possible, while at the same time, it does not want to generate more noise than the countermeasure. This is where optimizing the ability for the ship to maneuver away from the countermeasure while staying below the response threshold of the countermeasure could provide a significant tactical advantage.

**Environmental Consideration:** Applicable to merchant vessels, drilling platforms, pleasure craft/cruise ships, and government-owned vessels that operate in or near environmentally sensitive waters. Noise generated by marine traffic, especially at certain frequencies, has negative effects on certain marine life. It is possible that in the near future, governments will impose prescriptive restrictions that will limit which types of vessels are permitted in these protected waters, and/or impose speed limitations [15]. As the acoustic noise produced by vessels varies greatly, even within the same class of ship, these prescriptive restrictions may not provide adequate protection to marine life in some cases, and may be unnecessarily prohibitive in other cases, forcing adequately quiet ships to unnecessarily circumnavigate the protected areas. Given specific frequency and amplitude thresholds for an environmentally protected area, an ARCS can determine if a vessel is able to operate within the required noise thresholds, and if so, allow ships to operate as efficiently as possible within these restrictions. Data logged in the ARCS can be used as evidence of environmental compliance.

**Crew and Passenger Comfort:** Applicable to all civilian platforms, especially passenger ships and pleasure craft. In addition to externally radiated noise, vibration and noise from the propulsion system is also transmitted inwards through the ship structure where it affects crew-occupied spaces. Health and safety regu-
lations limit crew exposure to noise and vibrations (1981 IMO Code). An ARCS would allow operators to optimize the balance between passenger comfort and platform performance.

1.2.1 Optional Additional ARCS Sub-Systems

Although not within the scope of this project, there are some additional improvements that can be made to an ARCS, and are worth presenting here to demonstrate additional applications and further areas for improvement.

**Active Noise Cancellation (ANC)** - In addition to optimally controlling a propulsion system, the ARCS could be used together with an Active Noise Cancellation (ANC) system. The CEC could then dynamically control both the propulsion system and the ANC. In which to do this, during the "calibration stage" of the ARCP (described in more detail later in this chapter), the CEC could also capture audio recordings. The specific matching noise-canceling signal could be generated and mapped to the matching ship propulsion state along with the acoustic response. Outputting the ANC signals would then be greatly simplified and regarded as an additional output from the CEC during the "operate stage" of the ARCP.

**Control of Internal Vibrations** - It was mentioned in Section 1.2 that an ARCS could also be used to manage internal vibrations for crew and passenger comfort. In which to do this, the ARCS would need to be modified in terms of both hardware and software. The hardware modifications would be to either replace the acoustic sensors with hull-mounted vibration measuring accelerometers, or to simply add these accelerometers as a secondary sensor. Operators would then need to specify an additional response threshold: vibration response thresholds. Similar to acoustic response thresholds, the vibration threshold values could be sub-divided by frequency. The sub-division by frequency simply allows for more precise control, as people are prone to greater discomfort at certain frequencies.
**Calibration Health Monitor** - Usually during the *operate stage* of the ARCP, the acoustic sensors are no longer required. As an alternative to turning these sensors off, they could be used to provide validation of the current calibration. It is possible that environment conditions would change, or mechanical defects will occur such that the real-time *acoustic response* will no longer match the values stored in the CEC. As a minimum, the system will alert the operators that the system has lost confidence in its ability to stay beneath the response thresholds. A more advanced health monitoring function would also either update the original calibration, or begin creating a new one for the new conditions. This function may also provide early warning indications to the operate that something has changed mechanically with their vessel and should be investigated.

### 1.3 ARCP Description & ARCS Examples

In this section, the Acoustic Response Control Process (ARCP) is discussed in more detail, as well as how this could process could be interfaced into a ship’s existing equipment to form an Acoustic Response Control Systems (ARCS).

#### 1.3.1 General Description of the ARCP

**Process Premise**

Although accurate, using analytical or numerical simulation methods to model and predict the acoustic responses of a marine propulsion system, are specific to the model. There are multiple sources of noise caused by numerous interactions between propulsion, hull, and control surfaces paired with dynamic water conditions that make even steady-state predictions cumbersome. Unsteady and non-uniform flow conditions are even more difficult to model and predict [4]. Analytical and numerical simulations are usually built to model one specific class of marine vessel and cannot be easily adapted to other platforms without new simulations or experiments. Additionally, numerical simulations and software-modeled predictions are very com-
putationally demanding, and the results are not available quickly enough to be used in pseudo real-time systems. This discussion presents the ARCP as a adaptive process that will use a combination of hardware, software, and already fitted equipment (where possible) to learn the cause-and-effect relationship between the propulsion and maneuvering systems, together with the ship's motion (ship-propulsion state), and the resultant acoustic responses generated. The ARCP, in effect, bypasses the need for modeling and a full understanding of all the aforementioned interactions by mapping the cause-and-effect relationships directly. Because the process includes learning the propulsion and control parameters of the host platform, it can be applied to any platform with minimal variation in the hardware/software configuration using the common ARCP. In other words, by using a process that uses the host vessel's actual movements and real-world measured response to directly develop an approximate relationship between them, a vessel-specific response prediction tool can be developed without the need for laboratory modeling or simulation. Additionally, with the use of general experienced-based algorithms, the same hardware and software can be used for different marine vessels, without the need to significantly vary the ARCS designs. Finally, the process will use the learned cause-and-effect relationship to control the host platform with optimal maneuverability so that the user-defined acoustic response thresholds are not exceeded (where possible).

One particularly difficult condition to predict and model is the propeller noise that occurs when the ship is transitioning between speeds. During aggressive maneuvers and quick speed changes there will be significant forced propeller slip as described in Section 1.1. Most ships are able to change the speed and direction of the propulsion system much more quickly than then the actual speed or direction of the entire vessel. This is not only because the propulsion system must change speeds first in which to develop the new thrust in order to change the speed of the ship, but also because of the relative mass differences between the moving parts of the propulsion system and the ship itself. Take for example a large merchant ship with a fixed-pitch propeller that is traveling forward at speed, but wishes to use reverse-thrust to slow
or stop. Reversing the direction of rotation of the shaft using the engines can be done relatively quickly, and the shaft will achieve a steady rotational speed in the reverse direction long before the ship slows to a halt due to momentum. A ship moving forward with the shaft turning in reverse will produce a very different flow across the propeller and hull than a ship moving in reverse with the shaft also turning in reverse. This is the reason why the state of the propulsion and control systems, as well as the movement of the entire ship all need to be considered and included together in which to create the various ship-propulsion states.

Most propellers are optimized for a steady flow as ships usually spend a lot more time at a given speed, rather than frequently changing speeds. The different flow and interactions will inevitably produce different acoustic responses. Each acoustic response would also be unique to each combination of hull and propulsion system. With so many different marine vessels, it becomes necessary to use a singular, common process that can adapt to each unique marine vessel. Before discussing the stages of the proposed ARCP, and how data is managed, it is important to first discuss the terminology and convention that is used for labeling data flow.

Marine Vessel as a Plant - Input and Outputs

Although briefly introduce in Section 1.1, this section provides more detail specific to marine vessel control systems and terminology. To create the cause-effect relationships required for the ARCP, the data mapping algorithms will at, different stages, combine signals in different ways. This makes it ambiguous as to what are strictly data inputs and outputs and what are plant inputs and outputs. The definitions of plant input parameters and plant output parameters will be used consistently as defined in Section 1.1. However, a plant output parameter may be used as either a data input or an output from the CEC. Making the terminology even more complicated, this same plant output parameter may change from being an input to an output of the CEC at different stages of the ARCP. In attempt to provide clarity, a system diagram showing a marine vessel equipped with a rudder for maneuvering, and a fixed-pitch propeller for controlling speed is given as Figure 1-2. This figure
shows the normal signal flow from user input to the system being commanded, but also shows the corresponding terminology used in control systems: plant, inputs/actuators, outputs, sensors, disturbances, sensor noise, and feedback. Figure 1-2 clearly shows what signals are going into and coming out of, the plant. These signals will continue to be referenced as plant input and plant output parameters regardless of how they are combined for data mapping.

Figure 1-2: Marine Vessel As a Plant - Standard Control System Terminology

Process Stages

Although the ARCP can have multiple stages and sub-stages added depending on the complexity of the system, and notwithstanding the "ad hoc" mode, the ARCP must have at least two distinct stages. The stages must be separate because the CEC needs to generate the vessel-specific data map before it can be used for optimization. Although the stages can be iterative, input and output parameters are used differently in the two stages, thereby making each stage distinct. The minimum two stages are herein referred to as the calibration stage where the CEC creates the data map, and the operate stage, where the completed data map is used by the system to find the optimal maneuvering and propulsion configurations. An ARCS can move
back and forth between the two modes as required, to either update the data map, or to make use of a partially complete calibration in the interim while still completing the full data map. The terms "stage" and "mode" can be used interchangeably, but to avoid ambiguity, "stage" will be used in the context of a process (ARCP), and "mode" will be used in context of a system (ARCS).

Calibration Stage

The primary purpose of the calibration stage of the ARCP is to build the cause-effect relationship between how the ship is moving, and the amount of acoustic noise generated. How the ship is moving, is captured as a discrete state as defined by the ship-propulsion state in Section 1.1. Depending on how many different ship movements the ARCS is trying to optimize, it will also determine how complex the system needs to be. In simpler cases such as merchant ships, where speed and course changes are infrequent, and the consequences of exceeding response thresholds during these occasions are minor, the ACRS can be fairly simple. For the application of a warship taking evasive maneuvers for example, speed and maneuvering interactions are important, and the consequence of exceeding response thresholds, even for short periods of time, are severe. Therefore, for the latter application, a more complex ARCS is required. The complexity of the system will determine how many variables need to be captured in the ship-propulsion state.

For every variable that is to be included in the ship-propulsion state, there needs to be a way to measure or calculate the values of these variables. The values for measured variables will come from both input parameters, such as rudder angle and throttle speed, and also from output parameters such as the ship's axial speed. The values for parameter outputs will come from sensors, while parameter input values can be taken from command signals, or from the sensors already built into the equipment. Calculated variables will be found by combining other variables, or by integrating/differentiating the variables with respect to time. An example of calculated values would be in finding acceleration from speed or vice versa. Another example would be to measure the turning rate of the ship based on how the heading
changes with respect to time.

The *acoustic response* would be considered a secondary plant output parameter as defined in 1.1. It is an output parameter in the sense that the value is controlled indirectly by changing plant input parameters such as speed and rudder angle. It is also considered a secondary output because, except for in the case of "ad hoc" mode, it is not used as a feedback signal by controllers onboard the ship. In the *calibration stage*, *acoustic response* is used as an input to the CEC. Figure 1-3 shows an example of what the data flow would look like going into the CEC during the *calibration stage*.

![Figure 1-3: Calibration Stage of the ARCP](image)

To better explain how information is being processed, take an example ARCS to be used on a merchant ship. In this hypothetical example, the owners of the ship wish for the vessel to be able to operate in environmentally protected waters. In which to do so, they intend to reduce low-frequency noise being generated by the ship at sustained speeds which interferes with marine mammals’ ability to communicate. In this case, with speed changes occurring infrequently, the short bursts of noise and vibration during these transitions are deemed to be acceptable. The ship does however need to make regular rudder movements in which to correct for sea currents and wind. An ARCS that has three variables is chosen for this case in which to define the applicable *ship-propulsion state*: engine throttle as percentage of full power $P$, rudder
angle $\delta$, and rudder angle rate. We note the rudder angle rate can be calculated as the first derivative of the rudder angle with respect to time. Therefore we can state that the ship-propulsion state (SPS), which is the "cause" in our data map, is a function of these three variables as shown in Equation 1.1.

$$Cause = SPS = f(P, \delta, \frac{d}{dt} \delta)$$ (1.1)

Building further onto this example, an equivalent expression is needed for the acoustic response. Because the ship owners have concerns about both amplitude and frequency, we define the acoustic response (AR), which is the "effect" as a function of root-mean-square amplitude ($ARMS$) and frequency (freq) as per Equation 1.2.

$$Effect = AR = f(ARMS, freq)$$ (1.2)

Before data mapping can begin, both the cause and effect variables need to be discretized into a finite number of elements. The number of elements will depend on the level of precision required and the amount of processing power and memory available. For instance, the rudder angle can be discretized into 3-degree divisions, 1-degree divisions, or into divisions that are only a fractions of a degree. Depending on the algorithms used, the divisions do not need to be of equal sizes, but combined, all divisions need to span the full range of motion. This will allow for interpolation. In this example, there would likely need to be at least three divisions for frequency: one division is needed for the frequency band where marine mammals communicate, which is a known entity [15], and the other two elements would be the frequency bands above and below in order to span the full range of relevant frequencies.

Note that for our example, all values for the variables in the ship-propulsion state can be measured or computed with information already available on the ship. The only sensor that needs to be added is to measure the acoustic response. During the calibration stage, the data map will be created as the ship adopts the various ship-propulsion states. Depending on the method and level of sophistication of the experience-based machine learning algorithms used, it will determine how many of
the possible permutations of the ship-propulsion states need to be explicitly adopted and measured. The specific algorithm will also determine for how long data needs to be gathered at each ship-propulsion state in order to provide adequate levels of confidence and error rejection. The calibration can occur over long periods of time, being measured as the ship is operated normally, or dedicated calibration maneuvers can be conducted in which to specifically feed the CEC with the required range of ship-propulsion states. Either way, the CEC will need to create a data map that matches the permutation of the ship-propulsion states to the measured acoustic response. Again, depending on the complexity of the algorithms used, the system might be able to operate (with reduced confidence) with incomplete data.

While the discussion of machine learning techniques such as neural nets are beyond the scope of this project, a simplified data map is discussed here using a "brute force" technique to populate a data array structure or the data vector equivalent. An array is simply an ordered list of elements. To create multidimensional arrays, additional arrays are simply stored as elements within a parent array. For instance, storing one-dimensional columns of data as elements into another array forms a two-dimensional array which is effectively just a table of rows and columns. In the example being discussed, we need a three-dimensional array to cover all the permutations of the three-variable ship-propulsion state. A one-dimensional array is needed to span all permutations of the acoustic response. When the data map is created, for every element of the three-dimensional "cause" array, a one-dimensional "effect" array will be assigned. In our example there are three frequency bands: Band1, Band2 and Band3 and therefore each effect array would contain these three elements. If we use the imperative assignment ":=" to represent "is assigned the value of", a data map assignment could look like:

\[ \text{Cause}[P, \delta, \frac{d}{dt}\delta] := [A_{\text{RMS,Band1}}, A_{\text{RMS,Band2}}, A_{\text{RMS,Band3}}, \] \quad (1.3)

To continue the earlier numerical example, with only acoustic sensors, we can now add hypothetical data. When our example ship has a steady rudder at 0-
degrees, and 50% throttle, the RMS amplitude values for each of the three bands were measured to be 0.123, 0.456, and 0.789 respectively. For the purpose of this example, the throttle array contained 201 elements spanning from -100% to 100% in 1% increments, and that the rudder had 91 elements spanning from -45° and 45° in 1° increments. The minimum and maximum rudder rates are -5 and 5 °/s respectively, and are spread over 11 divisions. Assuming the data structure uses the convention that the first element of an array is the zeroth element, then a throttle of 50% would be the 150th element of the throttle array. 0° of rudder angle would be the 45th element of the rudder array and 0 °/s would be the 5th element of the rudder rate array. Inserting these numerical values into Equation 1.3 gives Equation 1.4.

\[
\text{Cause}[150][45][5] := [0.123, 0.345, 0.567]
\] (1.4)

As an aside, it worth discussing how managing internal vibrations could also be done concurrently as discussed in Section 1.2 from a data management perspective. Adding an accelerometer as a sensor is simple enough. As for software, this would be done by adding an additional dimension to the "effect" array. Instead of having a single, one-dimensional array containing acoustic sensor data, it would now have a second one-dimensional array with the latter containing the measured values of the accelerometer. Now the "effect" array is essentially an array of arrays albeit with only two entries at this point, but still making it a two-dimensional array. As with the acoustic response, it would make sense to divide the vibration response up into several separate frequency bands with individually set limits, as people experience different levels of discomfort at different frequencies. The number of frequency divisions again corresponds to the number of elements in this new one-dimensional array. In our simplified data array mapping example, the vibration response array and the acoustic response array do not need to have the same number of frequency divisions. However, response threshold will need to be expanded to the same size and shape of the modified "effect" array so that they can be compared and optimized during the operate stage.
Operate Stage

The primary purpose of the operate stage of the ARCP is to make use of the cause-effect relationship between how the ship is moving and the amount of acoustic noise generated in which to optimize the command signals to the propulsions and maneuvering systems without exceeding the defined response thresholds. Within the operate stage of the ARCP, there can be several smaller stages, but there is intentionally flexibility given here for choices of algorithms and methods to be used. In general however, the following sub-stages could be used each time the vessel’s crew request a new desired speed or new heading:

1. Determine the current ship-propulsion-state

2. Convert the new desired speed/heading into a desired steady ship-propulsion state

3. Use the data map to determine which ship-propulsion state best matches the desired state and also has an assigned acoustic response that is less than the response threshold

4. Determine the intermediate ship-propulsion-states required to get from the current to the desired ship-propulsion state without also exceeding response thresholds

5. Incrementally command the maneuvering and propulsions systems with modified command signals corresponding to the next incremental state. Repeat this for each incremental state until the desired ship-propulsion-state is achieved.

It is worth further discussing some of the subtleties of these sub-stages. In the second sub-stage, the keyword "steady" is used. Most marine vessels, when transiting, try to operate at a constant speed and heading for each lag of their journey. With the exception of specialty vessels, like those with dynamic positioning systems, most ships are more fuel efficient and will spend a far greater amount of time at a
constant speed and heading then they do altering their heading or changing their speed. So in this context, a "steady" ship-propulsion state means a constant heading and speed. "Steady" in this context does not preclude feedback control to make regular error corrections, such as small rudder angles and small rudder movements to correct for external disturbances. It is intentional to capture the additional noise generated by these small movements during the calibration stage, as they will also be required here in the operate stage. A more sophisticated ARCS data-mapping algorithm would measure environmental conditions, such as sea state, and would include the measurements in the ship propulsion state. Doing so, would allow the ARCS to effectively adjust to different sea states.

When referring to the "best match" as described in the third sub-stage, it does not necessarily mean that if the desired state requested by the operator has an assigned acoustic response that exceeds the response thresholds that a slower or more conservative ship-propulsion-state will be selected by the ARCS. Generally speaking, more aggressive speeds tend to increase the broadband noise response, however, when looking at frequency bands with individually-set thresholds, adopting a more aggressive speed may be preferential. For example, take the merchant ship from section 1.3.1 that wanted to avoid frequencies used by marine mammals. So in this sense, a lot of noise at a higher frequency is more desirable than moderate noise levels at the frequencies to be avoided. In this case, having the ship drive at a slightly faster speed may provide enough of a frequency shift to satisfy the response thresholds. It is likely that for any given ARCS, there will be a way for the operator to set how far above or below the desired speed they are willing to accept. Although developing more sophisticated "best match" algorithms are beyond the scope of this project, it would be relatively easy for the system to select the "best match within the available range" and notify the operator that they are exceeding the response threshold, and by how much.

Choosing the best intermediate states while transitioning from the current steady ship-propulsion state to the desired ship-propulsion state is not trivial. Even
when both the current and desired ship-propulsion states are within the response thresholds, the states between them, may not be. A simplified "greedy" algorithm would simply choose the next best intermediate state, even if that precludes better options later on. In doing so, it may possibly have to temporarily adopt a state that exceeds the response thresholds, or choose not reach the desired state at all. A more complex algorithm would map out the entire path to the new desired steady ship-propulsion state. A complex algorithm would also consider the time spent to complete the transitions. For example, if both speed and heading changes were requested at the same time, there would be multiple decisions for the best path. To find the best overall path, it may be better to change both throttle and rudder angle concurrently at some points along the path and better to change them independently for other points. For instance, it may faster and quieter to speed up without applying rudder up to a speed where turning rate is optimal, stay at that speed to complete the turn, then resume speeding up after the turn is complete. Developing this type of advanced algorithm is left as possible future work, while the key takeaway point is that the ARCP does not simply try to jump to the desired steady state, but rather choses a more optimal path to get there. An example of a possible "greedy" recursive approach is given in Figure 1-4.

Figure 1-4: Example of a "Greedy" Recursive Approach to Selecting Intermediate Ship-Propulsion States
To provide deeper insight into how data is being managed by the CEC in the *operate stage*, a general description will be given, followed by a continuation of the example from section 1.3.1. Figure 1-5 shows the general flow of information to and from the CEC for the *operate stage* of the ARCP. What is different here is how the data map is accessed.

![Data Map Diagram](image)

**Figure 1-5: Operate Stage of the ARCP**

Continuing with simplified algorithms for the purpose of explaining concepts, a simple search algorithm can be used. At this point, the current *ship-propulsion state*, the desired *ship-propulsion state*, and the *acoustic response thresholds* are all known. What is not known is the optimal steady *ship-propulsion state*, or what intermediate *ship-propulsion states* should be used for the transition. This is specific to sub-stages 3 and 4 as presented at the beginning of Section 1.3.1.

The first task to be performed by the CEC in sub-stage 1 is to determine the current *ship-propulsion state* $SPS_{current}$. Using the same convention of per Equation 1.1, we define the $SPS_{current}$ as a function of engine throttle as percentage of full power $P$, rudder angle $\delta$, and rudder angle rate as shown in Equation 1.5.

$$SPS_{current} = f(P, \delta, \frac{d\delta}{dt})$$  \hspace{1cm} (1.5)
For this discussion, we will continue with our merchant ship example to make it easier to follow the data flow. Our hypothetical merchant ship has the autopilot set to travel north at 14 kts. There is a slight set to starboard, and the vessel is maintaining 3 degrees of starboard helm to correct for this and maintain course. This 3-degree adjustment comes from the PID controller as well as any minor rudder movements required to correct for disturbances such as waves. The operators of the vessel are using an ARCS to prevent interfering with marine mammals in the area. The ship operators decide they want to increase speed to 18 kts. The autopilot will know the relationship between power and speed and that for instance 14 kts equates to 40% of max power, and 18 kts equates to 70% of max power. The variable values before the speed change are: $P = 40, \delta = 3,$ and $\frac{d\delta}{dt} = 0$. We assign these values to $SPS_{Current}$ as per Equation 1.6

$$SPS_{Current} = [40][3][0] \quad (1.6)$$

The next sub-process stage of this simplified algorithm is to determine the desired steady ship-propulsion state ($SPS_{Desired}$). Knowing that the desired speeds equates to 70% of max power, we set $P = 70$. Next, we know that the ship was maintaining a steady heading at the current speed with 3 degrees of port helm, so we set that value as well $\delta = 3$. The value will be later adjusted by the PID as likely a smaller rudder angle will be required at the higher speed, but using $\delta = 3$ is sufficient for our purpose. We are moving to a steady ship-propulsion state, so $\frac{d\delta}{dt} = 0$. This defines our temporary target state, however, it is premature to simply assign these values to $SPS_{Desired}$ as it may exceed our response thresholds. The CEC logic will check the values for the acoustic responses that were mapped to our temporary target state during the calibration stage. For the sake of discussion, assume that the acoustic response does indeed exceed our response threshold. The algorithm will then begin searching for the nearest state that will satisfy our response thresholds. For our simplified algorithm search, the CEC logic will fix $\delta$ and $\frac{d\delta}{dt}$ and look for the nearest lower and nearest higher thrust (if it exists) that does satisfy our response thresholds. Again, for the sake of this discussion, the states corresponding to
68\% thrust and 78\% thrust do indeed have acoustic responses that are less than the response thresholds. Our simplified algorithm simply favors whichever state is closest, and therefore chooses $P = 68$. $SPS_{\text{desired}}$ then gets assigned this value as shown in Equation 1.7.

$$SPS_{\text{desired}} = [68][3][0]$$

(1.7)

For our example, the system designers deemed that exceeding response thresholds during transitions was acceptable, and therefore the CEC will simply pass $SPS_{\text{desired}}$ to the propulsion and maneuvering systems. If we were concerned about exceeding thresholds while changing speed, the ship-propulsion state would contain a forth variable: acceleration. In this case, the CEC would move along the acceleration dimension of our now four-dimensional data array, keeping the other variables fixed in which to find and select which accelerations should be used as intermediate states. Note that our system does not have direct control over acceleration of the ship. However, to "effectively" control the ship’s acceleration with use of intermediate states, the CEC will simply order a number of small increases in throttle to be spread out over time.

### 1.3.2 Alternative Ad Hoc Process

As mentioned under the definition in Section 1.1, the concept of an ad-hoc mode option was introduced. A more detailed description is as follows. The ad-hoc process, or mode, once incorporated into an ARCS, is a separate, stand-alone process that uses the same sensors and information as the combined calibration and operate stages, but bypasses the need for data mapping. Using the ad hoc stand-alone process, the real-time measurement of the acoustic response is compared against the response threshold. If the response threshold is being exceeded, the system will simply reduce the command signals of the current ship-propulsion state until the acoustic response is less than the response thresholds (as shown in Figure 1-6). In effect, it operates as a noise limiter by simply limiting propulsions and maneuvering command signals.
One advantage of this stand-alone process is that it does not require calibration and can be used at any time. Another advantage is that the data used is more up-to-date, and may be more accurate than the values stored in CEC when the system was last calibrated. There are however numerous disadvantages to an ARCS operating in an *ad hoc mode*. First of all, the mode will be more prone to errors: for instance, the system may mistakenly read the *acoustic response* of an overtaking ship, and needlessly limit the host ship accordingly. Secondly, in *ad hoc mode*, the system is limited to taking more conservative movements, and the ability to adopt a more aggressive speed that may also be within the *response threshold* can not be exploited. Finally, it is necessary for the host vessel to exceed the *response threshold* before the ARCS can take corrective action. This greatly increases the likelihood of exceeding the *response threshold*, and duration of time in which it will do so. Error rejection would require more time to integrate and filter incoming data, which would subsequently increase the amount of time the vessel will be exceeding the *response threshold* before corrective action is taken.

\[\text{Figure 1-6: Example of the Alternative "Ad Hoc" Process}\]
1.3.3 Example of an Unmodified Propulsion and Maneuvering System

Taking a step back from focusing on the CEC and data flow, the final objective of this chapter is to show how an ARCS could be integrated into existing ship systems. A simplified diagram of a ship equipped with a Controllable Pitch Propeller (CPP) and rudder is shown as Figure 1-7. There are a few different ways that an ARCS could be integrated into a ship. One option would be to replace the autopilot and all the individual controllers with a CEPID as described in Section 1.1. This may be the best approach when designing a new ship. A less intrusive option, however, would be to modify the existing autopilot, leaving most of the existing hardware unaltered. By inserting the ARCS before these controllers, individual system PID controller logic and feedback sensors can be left unaltered. The idea being that the controllers are unable to distinguish between commands signals coming directly from the unmodified autopilot or those coming from the ARCS.

Figure 1-7: Example of an Unmodified Ship Propulsion and Maneuvering Control System. Note: Ship has Three Inputs: Rudder, Engine, and CPP
1.3.4 Example of Integrating an ARCS into an Existing Propulsion and Maneuvering System

The first possible option for integrating the ARCS into the simplified ship diagram shown in Figure 1-7 is to place it between the user interface and before the autopilot logic as shown in Figure 1-8. This approach would only be feasible if the autopilot itself could be easily altered, and there is an ability to intercept signals passed between the user-interface part of the autopilot and the output interface part of the autopilot. This configuration has an advantage in that the most autopilot controls remain the same, and only a few new user-interfaces need to be added (such as a way to set response thresholds). Another advantage of this arrangement is that the ARCS would not require a custom output interface that has matching hardware and communications protocols for all the external controllers, as the autopilot’s existing hardware for this would be simply re-used.

Figure 1-8: Shipboard Acoustic Response Control System (ARCS) - Fitted Between Autopilot User Interface and Processor

Another option would be to place the ARCS entirely before the existing autopilot. In this case, the ARCS passes signals to the autopilot as the user would such that the entire autopilot functions as originally designed. This arrangement requires for the ARCS to have a completely new user interface for both ARCS-specific
functions, such as changing response thresholds, as well as replacing all user-input functionality of the original autopilot. Depending on the circumstances, this may be the preferred option, as it would provide commonality between all ships using the system. As with the previous possible implementation, this option keeps the existing autopilot hardware for communicating with the individual system controllers. A diagram of this possible arrangement is given as Figure 1-9.

Figure 1-9: Shipboard Acoustic Response Control System (ARCS) - Fitted Before Autopilot, and Replacing User Interface

The third possible implementation presented in this section is to insert the ARCS entirely after the original autopilot, as shown in Figure 1-10. Similar to the first option, this implementation allows the user to keep much of the existing user interface with only a few additions required. This option requires the ARCS to have output hardware and protocols for each controller. Even with the use of reconfigurable data ports, this still adds to the hardware complexity of the system. This however can also be an advantage; this is the only configuration that allows for the ARCS to control the CPP directly. Consider if the autopilot was built for fuel efficiency only, and not allow pitch to be changed at higher speeds. The ability to change pitch directly gives the ARCS one more degree of freedom in which to change the acoustic responses, providing more options for acoustic-performance optimization.
1.3.5 Example of an ARCS with a Bypass Switch

There will naturally be some hesitation by marine vessel owners, masters, and crew to allow a new system to alter the way their vessel moves in high-traffic environments and especially in the context of emergency maneuvers to avoid collision or grounding. In anticipation of those concerns, it is relatively simple to include an ARCS bypass switch that will allow the user to take direct control of the equipment as shown in Figure 1-11.

Figure 1-10: Shipboard Acoustic Response Control System (ARCS) - Fitted Between Autopilot and Individual System Controllers

Figure 1-11: Example with an ARCS Bypass Switch to Allow Direct System Control
Chapter 2

Prior Research and Theory

2.1 Theory

There are several sources for the far-field acoustic noise generated by marine vessels. The most common sources are: machinery noise, propeller noise, flow noise, and crew noise. The scope of this discussion will be limited to noise sources that can be influenced through diligent control of the propulsion and maneuvering systems. Limiting the discussion to these sources will maintain the underlying objectives of this project in optimizing speed and maneuverability without exceeding response thresholds for any given vessel. Therefore, crew noise will not be discussed, and the development of machinery noise sources will be limited to the relevant type that can be at least indirectly influenced by changes in the maneuvering and propulsion systems.

It is a goal of this project to work with existing equipment on ships as much as possible, while avoiding attempts to design better or quieter equipment. The aspects of ship design and naval architecture that focus on making ships quieter - while still maintaining high performance - is widespread, and many of the techniques and designs are now available to the general public. With this in mind, this chapter will not focus on designs of quieter ship hulls, propulsion systems, control surfaces, or
additional systems that can decrease ship noise. It will, however, focus on how ship noise is generated by interactions between the hull, propulsion system, and control surfaces. In doing so, this will provide insight into how an optimization system such as the ARCS could optimize ship performance while mitigating versus acoustic noise for already-constructed designs.

Finally, stepping back from focusing on ocean engineering and the mechanics that affect ship noise, included is a brief overview of some other engineering applications of machine learning. Currently, other engineering disciplines, somewhat similar approaches of trying to solve complex problems through machine-learning techniques are being developed.

2.1.1 Ship Noise - Machinery

According to Ross [14], machinery noise is created by radiated structure-borne vibrations in mechanical equipment. Causes of these vibrations include mechanical imbalances, electromagnetic force fluctuation, impact, and friction. These vibrations occurring in the ship machinery, such as turbines, motors, transformers, gearing, and other rotating machinery such as engines and generators, become sources of ship noise which are in turn transmitted into the water. Due to advancement in power generation techniques, electromagnetic force fluctuation has not been a significant source of machinery noise since the 1930s, and therefore will be not discussed here. The pressure waves created by these vibrations contribute to the overall noise signature transmitted by the ship. Vibrations in the propellers, drive shafts, control surfaces, and machinery, along with other wetted components will transmit these vibrations directly into the water. Vibrations from internal machinery will also be transmitted into the water, but are done so indirectly by first transferring forces to their supporting structures, and then subsequently to the ship’s hull plating.
Mechanical Imbalances

Further elaborating on the causes of machinery noise, or phenomena as described by Ross, mechanical imbalances can be further categorized as "rotational" and "reciprocating" imbalances. Rotational imbalances are caused by a slight imperfection in aligning the rotating mass center of gravity with the center of rotation. There will also be rotational imbalances caused by imperfect symmetry in the mass being rotated, therefore causing fluctuating forces and moments.[14]

Reciprocal imbalances occur in machinery, which by design, is not symmetrical about the axis of rotation, such as with pistons in cylinders. These systems produce large imbalanced forces that are transferred through the crank shaft by their connecting rods. Vibrations, and therefore, acoustic noise transmission will therefore occur be at lower frequencies and correspond to the harmonics of the crank rotational speed.[14]

Impact Sounds and Gearing

As suggested by the name, impact sounds are caused by impact between hard surfaces - usually metal on metal. These sounds can be caused by equipment shift with the movement of the ship, or can be related to deliberate machinery movements such as a rudder hitting the mechanical stops when fully actuated. The noise caused from the movement of the teeth in the main propulsion reduction gearing also falls into this category of impact-sound machinery noise. The amount of noise generated and the frequency will depend on the meshing of the gears. Gear tooth impacts will cause tones at multiples of the frequency at which the gear teeth impact, which will vary directly with the angular velocities of the gearing. As the function of the reduction gearing is to reduce the high-speed, low-torque input from the main engines into the high-torque, low-speed propeller shafting, the angular velocity of the gearing will vary proportionally with the rotational rate of the propeller shafts. Another type of gearing noise is called "hobbing error" and has to do with the imperfections in the gears themselves created during manufacturing process. [14]
Piston Slap

The final type of machinery noise as presented by Ross is known as piston slap, which is caused by lateral movements of the pistons. As the piston travels through its full reciprocating cycle, it will also create impacts between the piston and the inner surface of the cylinder walls. Piston slap is a major source of mechanical noise for marine diesel engines in particular and will become more severe over the service life of the engine due to wear.[14]

Machinery Noise in Context of an ARCS

As an ARCS is intended to measure the full spectrum of radiated acoustic noise generated by the vessel, regardless of the source, it will include all forms of machinery noise in assessing the acoustic response. However, some machinery noise will be generated independent of the maneuvering and propulsion systems, such as the machinery used for producing fresh water, power-generation, and HVAC. These types of machinery noise will be present even when the engines are shutdown and control surfaces locked, and therefore will be present for all ship-propulsion states.

Other types of equipment such as axillary machinery which provide lubrication and cooling to the propulsion systems may only be operating under certain conditions such as a high sustained speed. This axillary equipment may also have binary modes such as on/off or low/high, but would still be linked, although indirectly, to the ship-propulsion states. However, as long as the axillary systems are operated and configured in the same way during the calibration and operate stages, the ARCS should still correctly include their contributions to the acoustic response. A more sophisticated ARCS could include axillary equipment monitoring and control as well.

The possibilities for acoustic response optimization come from the systems that are directly controlled (input parameters) such as engine throttle and rudder angle. If the response thresholds for the ARCS are configured such that they have more restrictive limits only at certain frequencies, then the ARCS can make decisions to either increase or decrease the input parameters in which to avoid those certain fre-
quencies. If gearing noise at a particular shaft speed for instance, created a harmonic that specifically interfered with another sensor, the ARCS would simply increase or decrease the shaft rotational speed accordingly. Another example might have to do with operating the rudder at low speed. Perhaps when the rudder is moved to 40°, it hits mechanical stops, causing an impact sound. The ARCS would simply limit the ordered rudder angle to 40° instead. Either way, without knowing any of these details of how the noise is being generated, the ARCS should still be able to reduce the occurrences of machinery noise.

2.1.2 Propeller Noise

Propeller noise is transmitted as pressure waves emanating from the propeller itself, the water as it is affected by propeller, and the vessel hull in response to pressure fluctuations originating at the propeller. Carlton [4], identifies five sources of propeller noise:

1. Pressure waves are caused by the displacement of water by the thickness of the propeller blades.

2. The pressure difference created between the pressure side of the blade (in the direction of rotation) and the suction side of the blade (opposite the direction of rotation) also creates pressure waves.

3. Flow noise is generated by boundary conditions of the blade surface as the water moves from the leading edge to the trailing edge across each blade.

4. Noise is generated by the periodic fluctuations of fixed-cavity volumes due to changes in the wake field. Fixed-cavity volumes in this context refer to cavities that remain attached to the propeller, such as sheet cavitation, and mid-chord cavitation. These terms will be explained in more detail later.

5. A significant amount of noise is generated by the sudden collapse of cavitation bubbles and vortex cavities as the cavities move out of the low pressure region.
Carlton further divides these sources into the two general categories of "non-cavitating" for sources 1 though 3, and "cavitating" for sources 4 and 5. Carlton’s sources of propeller noise are for marine vessels in general, and therefore apply to surface and submarine vessels. However, in the case of submarines, the list must be modified slightly. For instance, source number 4 will need to be expanded beyond periodic fluctuations to include transient fluctuations caused by movement of the submarine’s rudder and after hydroplanes, as unlike surface ships where the rudder is usually placed behind the propeller, a submarine’s control surfaces are located ahead of the propeller. The other submarine-specific modification required to the above list would be to add another source of non-cavitating noise: the noise caused by the vibration of the propeller blades at their resonant frequencies as they rotate through periodic variations of the inflow pressure field, which in the current context, is the wake field created by the hull and appendages. Examples of the differences in wake fields between surface and submarine vessels will be described later in this section in the context of non-uniform flow.

**Cavitation Noise**

Pressure waves created by a cavitating propeller can be several orders of magnitude stronger than the other aforementioned sources of propeller noise. From research conducted during World War II, the differences in sound levels measured from submarines increased by 40 dB when cavitating [14]. For this reason, the point at which cavitation first occurs (commonly referred to as "cavitation inception speed" (CIS) is a condition which marine vessels will try to avoid completely while trying to remain covert. There are several ways in which cavitation can occur. Figure 2-1 provides a useful diagram of the different types of cavitation that can occur.
Following an analytical approach to describing propeller cavitation, we will start with description of cavitation inception using the Bernoulli equation as described by Kerwin (2010) [10]. Beginning with a hydrofoil at an arbitrary angle of attack, that is subject to a uniform flow \( U_\infty \hat{i} \) and ambient pressure \( p_\infty \). In this context, the uniform speed is the vessel’s forward speed through the water with the coordinate system fixed at the propeller. Because the hydrofoil is rotating at some angle of attack, the hydrofoil will have an induced tangential velocity that when combined with the uniform flow has a resultant velocity vector being \( V = (U_\infty + u)\hat{i} + \hat{v} \). Assuming inviscid, incompressible flow, from Bernoulli’s equation, the pressure everywhere can be found as per Equation 2.1.

\[
P + \frac{1}{2}\rho V^2 = P_\infty + \frac{1}{2}\rho U_\infty^2
\]  

(S.1)

Kerwin then defines the pressure coefficient \( C_P \) as:

\[
C_P = \frac{p - p_\infty}{\frac{1}{2}\rho U_\infty^2} = \left[ 1 - \frac{V^2}{U_\infty^2} \right]
\]  

(S.2)
Cavitation will occur when the pressure anywhere is below the pressure at which water turns to vapor: \( p \leq p_v \). If we define the minimum pressure point along the hydrofoil as the \( p_{\text{min}} \), then this is point where cavitation will first occur when \( p_{\text{min}} \leq p_v \). Kerwin defines the **cavitation number** \( \sigma \) in relation to the **pressure coefficient** where cavitation will first occur as \( \sigma \leq -[C_{p,\text{min}}] \) and therefore the **cavitation number** is defined as per Equation 2.3. The significance of the **cavitation number** is that it characterizes the susceptibility of the given flow cavitating, as well as the degree to which cavitation will develop [10].

\[
\sigma = \frac{p_{\infty} - p_v}{\frac{1}{2} \rho U_{\infty}^2}
\]  

(2.3)

Revisiting Ross (1976) [14] moves discussion towards what happens with respect to propeller noise due to cavitation under the condition of **forced propeller slip** as defined in Section 1.1. Ross also uses the same approach to find a non-dimensional constant for susceptibility to cavitation as did Kerwin. What Ross has defined as the **advance cavitation parameter** \( K_a \) as in Equation 2.4 is similar to Kerwin’s Equation 2.3. Note that \( U_a \) means the velocity at which the vessel is advancing, which is equivalent to Kerwin’s variable \( U_\infty \).

\[
K_a = \frac{p_{\infty} - p_v}{\frac{1}{2} \rho U_a^2}
\]  

(2.4)

However, Ross decided it was also useful to focus on the **tip cavitation parameter** because it allows for analysis when not operating at the designed **advance ratio**, which would be the case for **forced propeller slip**. Before continuing with the discussion of **tip cavitation parameter**, it is first necessary to introduce and explore the **advance ratio**. **Advance ratio** is like **propeller slip** as discussed by Froude [6] for screw propellers in that it is a measure of how much the propeller advances for each rotation; however modern propellers using lifting line theory do not have a constant pitch. Instead, to create a non-dimensional value for how much modern propellers are "slipping", **advance ratio** \( J \) is used as shown in Equation 2.5. Here, \( n \) is the rotational frequency, and \( D \) is the diameter of the propeller.
\[ J \equiv \frac{\frac{U_a}{n}}{D} = \frac{U_a}{nD} \quad (2.5) \]

Returning to the discussion of the tip cavitation parameter \( K_a \), a more comprehensive representation that includes resultant velocity at the tip, (with separate velocity components,) in terms of advance ratio is given as Equation 2.6.

\[ K_t \equiv \frac{p_\infty - p_v}{\frac{1}{2} \rho U_t^2} = \frac{p_\infty - p_v}{\frac{1}{2} \rho (U_a^2 + (\pi n D)^2)} = \frac{p_\infty - p_v}{\frac{1}{2} \rho (\pi n D)^2 ((\frac{\pi}{\sqrt{J}})^2 + 1)} \quad (2.6) \]

This leads to determining that \( K_a \) will always be much greater than \( K_t \) by the relationship shown in Equation 2.7

\[ K_a = \left[ \left( \frac{\pi}{\sqrt{J}} \right)^2 + 1 \right] K_t \quad (2.7) \]

Because \( K_t \) is much smaller than \( K_a \), especially at smaller advance ratios, we can say that the rotational component of the tip velocity (\( \pi n D \)) is more meaningful in terms of cavitation inception. This allows for a situation where the shaft accelerates quickly (large increase in \( n \)), but due to momentum the vessel’s speed of advance has not yet had time to reach the new corresponding speed (\( U_a \)). Because the \( \pi n D \) term is dominate, the propeller can cavitate even at low vessel speeds provided it has ample shaft rotational speed. This creates the result that a vessel aggressively changing speeds can cavitate before actually reaching the vessel speed (\( U_a \)) that would be understood to be the CIS.

It should be noted that for propellers operating near the surface of the water, it is not valid to assume that the ambient pressure is uniform. For surface vessels where the propeller is near the surface, there is a significant difference in hydrostatic pressure from the shallowest to the deepest part of the propeller. This would give the same propeller different cavitation inception speeds, depending on the angle and depth that it is mounted beneath the vessel. This creates more vessel-specific data requirements in which to predict cavitation using analytical or empirical methods.

Specific to this project, this theory provides two significant insights: The first is that the vessel’s speed through the water (\( U_a \)) alone cannot be used to accurately
describe the *ship propulsion state*, but needs to be paired with a second variable such as shaft speed, or the vessel’s acceleration. This will in turn allow the ARCS to handle speed transitions differently than steady state speed conditions. The second insight from this theory section is that ambient pressure has a significant impact on when cavitation occurs. This means that for a submarine or a UUV, that the vessel’s depth below the surface, which is related directly to the ambient hydrostatic pressure, would need to be also included in the *ship propulsion state*.

### 2.1.3 Non-Uniform Wake Fields

Although open-water propeller testing can provide some insight into predicting the onset of propeller cavitation, the uniform flow calculations are not well-suited for propellers once coupled with the vessel hull because of the non-uniform wake field. The interactions between the propeller and the hull of a ship are a thoroughly researched area of study due to the importance of properly matching the propeller to the hull. The "no-slip" boundary layer conditions at the ships hull will cause the wake field as seen by the propeller, to be uneven. It is intuitive that the non-uniform wake field will influence the in-flow to the propeller. What is less intuitive (but has also been thoroughly researched), is the effect that the presence of the propeller has on the flow forward of it. The resulting combination hull-and propeller-model of the wake field is the "self-propulsion" model. For the context of this discussion, unless otherwise stated, when referring to the wake field at the inflow of the propeller, it is the combined self-propulsion model that is to be considered.

To provide context of the effects of the wake field on propeller noise, the general concept of a wake field and the differences unique to submarine hull design will be discussed. The "no-slip" boundary conditions exist along the surface of the hull. Figure 2-2 shows the axial velocity field located just before the propeller for a single-propeller surface ship. This particular hull has a narrow vertical skeg (sternward extension of the keel) which is directly in line with the propeller. The in-flow velocity has been slowed the most at the 12 o’clock position where the boundary layer effects
from the underside of the ship are the strongest. The effect of the skeg creates additional boundary layer effects at the 12 and 6 o’clock positions. The least affected regions (highest velocity) are at the 5 and 7 o’clock positions, where the edges of the propeller are the farthest away from the hull (and skeg).

The wake field for a submarine is given as Figure 2-3. As is typical of most submarines, the propeller is concentric with the hull. The most predominant effect of the hull boundary layer is seen where the hull tapers to the propeller hub. The boundary layer is almost uniform all the way around the hull, except for the notable increases at the 3, 6, 9 and 12 o’clock positions. These four positions correspond respectively to the starboard-aft hydroplane, lower rudder, port-aft hydroplane, and the top rudder. The 12 o’clock position has a notably larger hull boundary effect because the submarine’s sail (fin) is also located at this position.

Figure 2-2: Surface Ship Wake Field - Image from Kerwin, 2010 [10]
To develop the concept of propeller tones, we now introduce the concept of the Blade Passing Frequency (BPF) which is defined as how often any propeller blade passes by a fixed point as the propeller rotates. BPF will therefore be a function of both the angular velocity of the propeller ($n$) and the number blades ($Z$). For example, a propeller with only two blades would have the same BPF as a propeller with four blades turning at half the speed. As a blade moves through a full rotation of the wake field shown in Figure 2-3, each blade will experience flow fluctuations at the 3, 6, 9 and 12 o’clock positions. As the propeller angular velocity, is usually held constant to achieve the required axial velocity, these pressure fluctuations will be periodic, occurring four times per revolution per propeller blade. It is these periodic pressure fluctuations that cause the far-field radiated tones at the BPF and its higher harmonics [18]. Note that the submarine wake field is still much closer to being constant over each rotation than the ship wake field, due to the symmetry of the concentric propeller and hull design. This gives the submarine a significant advantage over the surface ship in terms of a more uniform flow, and therefore a reduction in acoustic noise generation.
2.1.4 Effects of Changing Propeller Advance Ratio

Ross conducted research to determine how changes to the propeller Advance Ratio affects propeller performance in terms of thrust, efficiency, and torque. A brief summary of Ross’s findings are provided here in order to bring more context to what occurs during force propeller slip. Continuing with using non-dimensional values, Ross defines the thrust coefficient \( (C_T) \) as show in Equation 2.8 and the torque coefficient \( (C_Q) \) as 2.9.

\[
C_T = \frac{T}{\rho n^2 D^4} \quad \text{(2.8)}
\]

\[
C_Q = \frac{Q}{\rho n^2 D^5} \quad \text{(2.9)}
\]

Ross distinguishes between the power required to turn the propeller, and useful propulsive power in which to determine the propeller efficiency \( (\eta_p) \). Power required to turn the propeller is the product of the torque \( (Q) \) and the angular speed \( (n) \). Useful propulsive power is product of the thrust \( (T) \) and the speed of advance \( (U_a) \). The propeller efficiency \( (\eta_p) \) is given as Equation 2.10 [14].

\[
\eta_p = \frac{U_a \cdot T}{2\pi n \cdot Q} = \frac{U_a \cdot C_T}{2\pi n D \cdot C_Q} \quad \text{(2.10)}
\]

Substituting the definition for the advance ratio as per Equation 2.5 into Equation 2.10 yields Equation 2.11.

\[
\eta_p = \frac{J \cdot C_T}{2\pi \cdot C_Q} \quad \text{(2.11)}
\]

Using open water-propeller tunnel measurements, Ross produced the following characteristic curves as shown in Figure 2-4.
The significance of how propeller efficiency, torque, and thrust change with the \textit{advance ratio} provides more insight into how the corresponding acoustic noise will also change under the conditions of \textit{forced propeller slip}. The hull design of a ship will determine the speed versus power curve. A propeller is matched to the hull and propulsion system based on key parameters, one of which is the advance ratio. Operating the propeller under \textit{forced propeller slip} conditions will provide different results with respect to efficiency, torque, and thrust, as propeller matching assumes steady speed conditions. Specific to this project, this informs ARCS designers of which variables should be included in the \textit{ship propulsion state} to accurately create the right cause-effect relationships. For example, we can see analytically, that just using shaft speed as a single input parameter is not sufficient for fully defining the state of the propulsion system. Instead we should use at least one other variable: such as acceleration, or speed through the water, in which to allow for different discrete \textit{ship propulsion states} for different \textit{advance ratios} and in doing so, correct for \textit{forced}
propeller slip. Experimental data supporting the differences in the acoustic response with changes to the advance ratios will be discussed later in the next section.

2.2 Prior Research in Measuring Ship Noise

2.2.1 Hull Vibration Noise

In addition to the effects of varying ambient pressure, along with flow fluctuations of a non-uniform wake field, there is also interaction between the propeller and hull plating providing additional layers of complications when predicting ship noise. Also, as previously discussed, machinery noise will also be transmitted through the ship's structure to the hull plating. Transmission of machinery vibrations can be reduced through effective use of vibration-limiting securing mounts that provide dampening between the machinery and the hull. Propeller-induced hull plate vibrations, however, are more difficult to isolate mechanically, yet are especially disruptive to the crew of smaller ships, or when accommodation spaces are located near the propeller.

A.C. Nilsson's [13] research on propeller-induced hull plate vibrations, provides more insight into how propeller-induced hull plate vibrations are generated, and offers some perspective for ship designers on how they may be able modify a ship’s configuration to limit the disruptive effects. In which to provide substantiation to these insights, Nilsson performed both theoretical and experimental research and compared results.

The results show that for any use of hull-mounted hydrophones, the pressure effects near the hull, due to the response of vibrating hull plate, are significant and need to be considered. For ship designers, it is important to note that for applications where propeller-induced hull vibrations need to be reduced, it can be achieved by either increasing the plate thickness, or by decreasing the space between frames.

The theoretical model for the response in hull plates above the propeller was found to be valid and the "simply support" boundary conditions gave results close to
the experimental values. The effects of the pressure response in the hull plating are significant for frequencies above 40 Hz and less than the coincident frequency.

The significance of this research is both in accounting for the hull plate vibrations as another source of noise to be considered, but also provides insight as to where acoustic sensors should be located. Hull-mounted hydrophones near the propeller will show significantly higher amplitude pressure fluctuations due to the vibrations in the hull plating to which it is mounted. For this reason, to better estimate far-field acoustic noise levels, hull mounted hydrophones should not be used unless hull-plate vibrations are specifically corrected for.

### 2.2.2 Full-Scale Maneuvering Acoustic Signature

As part of a study performed by Trevorrow (2008) [16], a full-sized research vessel was used to measure changes in the acoustic noise signature of a ship as it conducted different maneuvers. A series of GPS-equipped buoys were used to record data as the vessel maneuvered around them. This experiment showed that at full-scale, there was a significant increase in the acoustic noise during turns, although the propeller shaft speed remained constant. Trevorrow et al believes that the increased noise was due to the reduced propeller *advance ratio* due to the turn-imposed decrease in vessel speed. This is the same effect that has been referred to previously in this discussion as *forced propeller slip*.

The vessel used for the experiment was the Canadian Coast Guard Ship (CCGS) Vector, with a length of 39.7 m and a 560-tonne displacement. The propulsion system is a single shaft, 3-bladed CPP with a diameter of 1.8 m. Power was provided by a 825 HP (600 KW) diesel engine.

The sensors used for acoustic recording were a series of Broadband Underwater Recording Buoys (BURB). A procedure was used to correct the recorded signals in terms of propagation distance loss and aspect angle.

Although not explicitly addressed by Trevorrow, the loss of speed during a
turn is expected in ship maneuvering. *Drift angle* is a measure of how much a vessel "skids" during a turn. The speed loss in the turn is due to the added mass of the vessel moving laterally at this *drift angle* [17].

During the experiments conducted by Trevorrow, a series of straight runs were conducted at various speeds, followed by a series of 90 and 180 degree turns. Figure 2-5 shows how the ship speed was reduced during turn at various turn rates, and also the corresponding increases in the spectral source level (SSL). Note that these significant increases in SSL are still present even after all corrections were made for speed dependence and aspect angle to the buoys.

![Figure 2-5: Images from Trevorrow, 2008 [16]](image)

Trevorrow wrote that although the shaft remained at the same speed before and during the turn, that the ship's forward speed reduced to 72% of the inbound speed while in the turn. This resulted in a reduction in the propeller *advance ratio* \((J)\) from 0.674 to 0.488. Citing the same research conducted by Ross [14] as discussed in Section 2.1.4, they noted that as the *advance ratio* decreased, the propeller efficiency also decreased, while the propeller thrust and torque increased. This can be seen again from Figure 2-4 in Section 2.1.4. Trevorrow presumed that the increased sound levels were due to this increased propeller thrust and torque.

This research by Trevorrow et al, not only provides insight into how to better capture the *ship propulsion state*, but also recommends that a sampling rate of at least 1 Hz would be needed in which to obtain accurate measurements of the changing
advance ratio. Also, by using sensor buoys and the same methods for correcting for distance from the sensors and changing ship aspect angles, it makes it possible to accurately calibrate an ARCS without requiring sensors to be fitted to the ship. Instead, recording buoys such as the BURB, or even fixed arrays could be used instead for measuring acoustic response.

2.2.3 Predicting Ship and Submarine Noise using CFD

Another area of using modern processes to solve the problem of accurately predicting ship noise is to look at simulation programs such as Computational Fluid Dynamics (CFD). Based on how long it takes for results of these simulations to be rendered, and the amount of assumptions and explicitly defined values required, this approach is not yet adaptive, nor are they ready to be used in near real-time systems. Despite this, there are still advances being made, specifically with submarines and modeling the noise from propellers. Significant progress has been made recently in coupling CFD analysis with experimental results for generic submarine propellers and hulls. The following is a brief synopsis of selected current and relevant research.

Zhou (2013), [18]: The primary focus of this paper was to investigate how tone noise generated by the submarine propeller relates to a non-uniform flow. In the parametric comparison, researchers were able to draw direct connections proving that the number of submarine hull appendages directly affects Blade Passing Frequency (BPF) harmonics.

As a notable secondary effect of this study, the researchers have also verified that both the "DARPA Suboff" hull model and "NACA 66" are very good generic representations of modern submarine designs. They were able to show through experimental and CFD predictions (κ − ϵ approach) the effect of varying propeller design parameters such as the skew angles, blade numbers, and shaft rates. By varying these design parameters independently during testing, the mathematical relationships between changes to these parameters and the overall Sound Power Level (SPL) were established. Their predictions led to the optimal configuration being a seven-bladed,
highly skewed propeller with broad blades. Although this is not a surprise, as this correlates with what is commonly known of Cold-War-era propeller designs, it has proven that the modern approach of CFD predictions are able to reproduce what was determined decades ago through very expensive experimental modeling of incremental prototypes.

Chase (2013), [5]: This research investigated which CFD models most accurately correlate with experimental results for submarine propellers. In this case, the ENSEAN 1619 propeller was used with the DARPA Suboff submarine hull. Knowing which CFD models most accurately predict results is important when superposing multiple and concurrent effects to the wake field, such as accelerating while applying a strong rudder angle.

As expected, certain CFD models performed better than others under different circumstances. The simulations were performed using "CFDShip-Iowa V4.5" which has Reynolds Averaged Navier-Stokes (RANS), Detached Eddy Simulation (DES), and Delayed Detached Eddy Simulation (DDES) capabilities. The RANS and DES approaches are based on a blended $\kappa - \omega/\kappa - \epsilon$ approach with a Shear Stress Transport (SST) turbulence model.

In general, both DES and DDES performed significantly better than RANS, and in most cases DDES performed slightly better than DES (this was concluded as the RANS model is ill-suited for resolving tip vortices). DES allows capture of the tip vortices better than RANS because the turbulent viscosity is reduced where the grid is fine enough to capture large vortices. All three models (DES, DDES and RANS) consistently under-predicted thrust and torque by about 7.8% and 5.9% respectively when compared with experimental results. This was especially true at high loads.

The relevance of this research shows that although CFD is an excellent tool for learning some of the underlying physical effects that are occurring, it has not yet evolved to the point where a control system can predict and correct for propeller noise. This supports the cause-effect ARCP approach as a more achievable and pragmatic
interim solution.

2.2.4 Alternative Use of Sensors to Predict Cavitation Noise

As an alternative approach, Han et al. (2016) [9] conducted evaluation experiments to see if an accelerometer could be used as a sensor to predict Cavitation Inception Speed (CIS). By installing both accelerometers and hydrophones at the suction pipe of a waterjet, they were able to conduct a side-by-side comparison of the sensor performance.

Using Detection of Envelope Modulation on Noise (DEMON) analysis to demodulate cavitation noise with the shaft rotation speed and blade passing frequency, the authors were able to approximate where propeller cavitation was occurring. Being able to conduct this analysis with hydrophone data is not new, however they were also able to conduct a comparable analysis using accelerometer data that was measuring vibrations on the hull adjacent to the propeller as an alternative to using hydrophones. The authors proposed that accelerometers can be used as an alternative to hydrophones in which to provide real-time CIS monitoring. It should be noted that the authors acknowledge that real-time CIS monitoring would still be required, as the value will change over time based on the condition of the hull and propeller.

The relevance of this research with respect to this project is that if properly configured, it is possible that accelerometers measuring vibrations can be used as the acoustic sensors for not only internal vibration analysis as already proposed, but also for measuring the acoustic response.
2.3 Prior Research in Other Engineering Applications of Machine Learning

2.3.1 Deep Belief Networks to Localize Plate Failure

Georgoulas (2016) [7], presents the use of a machine learning technique known as *deep learning* as a method of real-time detection of small defects in the metal plating of ships. When metal undergoes irreversible changes to its internal structure, such as cracking due to aging, it will emit acoustic waves in the solid structure known as Acoustic Emissions (AE). If a system were to be installed that could measure and interpret AE being emitted by the ship structure, then it would be able to provide operators with early warning of an imminent failure. The method presented also allows for determining the location where the AE originated, and therefore the ability to localize where the failure occurred in the structure.

The machine learning approach presented uses Deep Belief Networks (DBN), which is an alternative type of deep neural network used to map the raw input space into a lower dimension feature space. With the simplified preprocessing stage developed by Georgoulas, the researchers conducted simulation experiments of a stiffened plate model that was partially submerged in water. The simulation produced strong results of achieving 94% AE localization rates using only a single sensor, despite the extremely complicated ship structure. To the best of their knowledge they were the first to use DBNs for localization of AE events.

Put in the context of this project, Georgoulas research shows how experience-based machine learning can be used to greatly simplify complex problems. Although still restricted to a noise-free environment, simulation-only, and with a defined model, the authors have shown the benefits of approaching a very complex engineering problem with the use of DBNs. Also, some of the pre-processing techniques used could be adapted for use in other advanced machine learning applications such as the cause-effect relationships for an ARCS. Georgoulas’ application and approach is very dif-
ferent than the proposed ARCP as with localizing AE, their DBN cannot learn for real-world operations of the ship. It would not be feasible to create cracks and damage in a ship's hull in which to train a machine-learning system to recognize the signs of structural failures.

2.3.2 Machine Learning to Diagnose Diesel Engines

Briefly moving away from ocean engineering, to investigate wider engineering applications for machine learning, we find that other researchers are also using somewhat similar approaches to solving complex problems, but with very different applications.

In which to provide an alternative approach to detecting and diagnosing faults in diesel engines, Kowalski et al. (2017) [11], uses an Extreme Learning Machine (ELM) which is a form of neural network. In doing so, the researchers have built a fully automatic machine-learning based system for engine fault detection. Their in lab experiment employed a diesel engine that could simulate malfunctions. Then, using a cause-and-effect analysis they separated diagnostic signals in the form of $NO_x$, $CO$, $CO_2$ and $O_2$ fractions in the exhaust gas along with their temperatures. This allowed for the training of an experienced-based machine-learning approach to learn the relationship between different diagnostic signals to the corresponding faults.

Using machine learning, Kowalski et al. were successful in training a system that could correctly identify 15 different faults based on only the diagnostic signals as inputs. To put this in the context of this project and the terminology used herein, they effectively were able to train the system by injecting a known fault type as a "cause", and let the system "learn" the corresponding "effects" (or response) comprised of diagnostic signals. This would be analogous to the calibration stage of the ARCP. Once this mapping is complete, the system can look only at the "effects" in terms of diagnostic signals and determine the "cause", which in the context of a diesel engine, would be the status: healthy or otherwise the type of fault. This reversal from effects to cause, would be analogous to the operate stage of the ARCP.
The application presented by Kowalski et al. is similar to the approach proposed in this project, as the system is not as concerned with understanding the meaning of diagnostic signals, nor is it providing any insight into the underlying chemical and physical behaviors that are causing the changes. Instead, the system simply looks at the inputed variables, and based on experienced-based learning, is able to determine if the system is healthy or not. If it is not healthy, it is able to determine what the fault is.

However the application presented by Kowalski et al. is also significantly different from the proposed ARCP, not only in the application, but in how the cause-effect relationship is used, and also in the ability for the process to be adaptive. In terms of how the data map is used, the system as presented by Kowalski, although very useful, is only a diagnostic system. This would not be a good use of a cause-effect controller (CEC) as there is no way to operate the diesel engine differently based on the diagnostic signals in which to gain a useful benefit - such as to operate the diesel engine such that the fault clears or it can be operated in a more fuel-efficient manner. Also, there does not seem to be an equivalent of an ARCS ability for the user to change the acoustic response threshold in real-time and still optimize plant inputs without the requirement to re-conduct the learning/calibration.

With respect to adaptability, Kowalski was able to train the system using a laboratory diesel engine that was able to introduce faults of specific types, allowing the machine learning to map the fault as defined by the researchers to the specific diagnostic variable state. Other diesel engines would have different operating temperatures and gas fractions that would be considered healthy, and therefore this relationship would need to be established for each diesel engine. In addition, most non-laboratory diesel engines, especially those already installed and in use, would have no way to introduce faults with which to create the engine-specific data maps. Although not discussed, it is possible that the diagnostic signals may change over time as the engine parts wear, which would require the system to have some way to re-calibrate.
2.3.3 Vibration Mitigation with a Fuzzy Logic PID

Focusing on control systems, it is worth investigating other types of hybrid controllers such as the CEC and the CEPID proposed in this paper. There is active research being conducted in the fields of developing controllers which integrate machine learning to be more adaptive after construction.

Fuzzy logic is a form of programming that doesn't restrict the system to a binary, true or false type of logic. Fuzzy logic is often used in applications that humans already perform well, but the desire is for a machine to perform the function instead. This allows for the programmer to pass on general rules or guidelines, and then have the system make appropriate decisions based on those guidelines. As such, intelligent Fuzzy Logic Controllers (FLC) are more adaptive to dynamic situations than hard-wired controllers. FLCs are especially useful for applying heuristic knowledge in situation with uncertainty.

Azeloglu (2017) [3], presents using a Self-Tuning Fuzzy Logic PID Controller (STFLPIDC) to reduce vibrations induced by an earthquake on a crane structure. This research is the most recent in a series of progressive uses of FLCs for active seismic control in structural applications. Therefore, Azeloglu compares simulated performance of the STFLPIDC to that of a classical FLC in a similar application.

In the simulations, the combined STFLPIDC was able adapt the control of the active seismic actuators by allowing the ability to change the FLC's scaling factors, membership functions, and rules after the system is brought online. According to the authors, the superior qualities of their method include its simplicity, satisfactory performance, and its robust character.

In the context of this project, this research shows how more versatile controllers are being developed, and can be modified after being brought online. The FLC and STFLPIDC are controllers in the classic sense that they stabilize a system against disturbances. In this case, the controller drives actuators (plant inputs) to cancel earthquake-induced seismic vibrations (disturbances), to keep the crane struc-
urally stable (plant output). This is different from what a CEC does in the sense of control, where the CEC does not to try maintain the system at a stable state, but instead determines the optimized method to change from one state to another desired state - (i.e to change heading and speed, not to maintain it). In Section 1.1, a CEPID was defined as a hybrid controller that combined a CEC for determining how to optimally change states, together with a PID intended to provide the function of maintaining stability in the new state. It would be this latter part of the controller, the stabilizing portion, that a STFLPIDC could replace. In fact, doing so would result in a notable improvement as it would allow the PID portion of the CEPID to be as flexible and dynamic after being brought online as is the CEC.

2.4 Research Summary

Although a great deal of work is currently being done in engineering applications of machine learning, there does not appear to be specific applications associated with acoustically optimized propulsions and maneuvering systems. Also, based on searches conducted to date, there does not seem to be any current research that is proposing the use of cause-effect relationships to both learn a system and also to optimize how it used based on variable user thresholds. However, there is significant work that overlaps with portions of this project, and therefore allows for improvements in specific implementations of the ARCP to incorporate more sophisticated and adaptive algorithms as well as to simply integration with existing ship systems.
Chapter 3

Prototype Development, Hardware & Software Configuration

3.1 Introduction

In which to develop a "proof of concept" of the ARCP, a simplified prototype was constructed using an Unmanned Surface Vehicle (USV). There were several reasons that a USV was chosen as the test platform for the prototype ARCS:

**Consistency:** Although robustness, data filtering, and error rejection are all goals for later prototypes, the first step was to limit input errors so that the underlying data relationships could be first identified and incorporated into the CEC algorithms. In which to achieve this, consistency was required in the ability to tightly control timing of movements, throttle settings, and rudder movements. This would be much more challenging with a "human-in-the-loop" trying to directly control a throttle, at the right rate, and with the exact same timing for multiple consecutive runs. A ship's autopilot is also able to do this, but it would be more challenging to synchronize data gathering and computational processing components of the ARCS.
Configurability: USVs that have been custom-built by MIT students will have accessible hardware and non-proprietary software. This makes changing hardware, hardware settings, firmware, software, and even making structural changes much easier than working with a fully-contained, built-to-specification vehicle. This allows for use of open-source software as well as access to the many "do-it-yourself" tools and resources that are available online. The USV would also have a processor onboard that could be used to consistently control the vehicle’s movements, in addition to performing synchronous data gathering and analysis.

Funding / Resources: This project did not have any funding or sponsors external to MIT. As there were neither grants nor research funding linked specifically to this project, resources were limited to what platforms were readily available within the Department of Mechanical Engineering. Without additional funding, the purchasing, renting, or hiring of a test platform was not a feasible option. The construction of a new USV (including design and construction) was also prohibitive in terms of both cost and time. As such, any hardware modifications and additional equipment needed had to be either available in the labs, or inexpensive enough that it could be reasonably purchased without exceeding a typical student project budget.

Availability: This project was limited to completing all data gathering between the months of November 2016 to April 2017. The MIT Sailing Pavilion, which is where most of the autonomous vehicle testing is conducted, is closed during the winter months. A test platform had to be chosen that was portable enough to be moved around the MIT campus, and was small enough to fit into one of the testing tanks on campus that is available for year-round use.

Size: In addition to the requirements for the platform to be small enough to use in the testing tanks, it was also important for the vehicle to be as large as possible. The reason being, is that unless the vehicle has enough mass, it would not have the momentum to create the necessary force propeller slip conditions as defined
in Section 1.1. Additionally, it was important that the test platform had a large enough propeller with sufficient torque and shaft speed to create measurable amounts of acoustic noise and a definitive CIS. The USV platform ultimately used for this project was not ideal with respect to all the aforementioned aspects and is further discussed under Lessons Learned in Section 4.4.

**Expertise and Support:** A significant amount of effort is required to learn the hardware, drivers, operating system, sensors, and components for a specific autonomous vehicle. This familiarity is required not only so that the equipment can be used and maintained properly, but is also a prerequisite to conducting hardware and software modifications. The baseline software that is used to control autonomous vehicles is not trivial, and having to create that software before starting the design and integrating of an ARCS would have been a prohibitive level of additional effort. Fortunately, the USV selected is well-documented, and well-known by other students working in Ocean Engineering. Additionally, this USV uses MOOS-IvP for the autonomous software, which was developed at MIT, and is widely used within the department, making software support readily available. The people who provided invaluable help and support with the USV and software are mentioned in the acknowledgments.

### 3.2 Hardware Configuration

The USV that was selected is a HoverGroup Autonomous Kayak. These USV kayaks were constructed by the Hover Research Group (aka HoverGroup) under the supervision of Dr. Franz Hover at MIT, and are currently under the custody of Professor John Leonard, a senior member of the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL). Much of the technical work in constructing, configuring, and documenting the kayaks was done by Joshua Leighton.
3.2.1 Baseline Hardware

Kayak Hull

The HoverGroup Autonomous Kayaks were built as a testbed for multi-vehicle control using through-water acoustic communications. The specific vehicle used has the name "Kestrel". The hull of the kayak is a *WaveSport Fuse 35*. The kayak is 1.8m (5.9 ft) long and weighs 40 kg (88 lbs) [12]. The hull is shown as Figure 3-1.

![Figure 3-1: HoverGroup Autonomous Kayak With Cart](image)

Propulsion and Maneuvering

Propulsion and steering are both controlled via a pod thruster located near the front of the kayak. A small skeg is located near the stern to provide added directional stability. The pod contains a *Minn-Kota Riptide 55* trolling motor and is controlled using a *RoboteQ LDC1430* motor controller. This allows for a top speed of 2 m/s (4kts) when not towing. Directional control of the pod, which is used to steer the kayak, is from a *ServoCity MEGA Servo MS530-1* [12]. A photo of the thruster pod and skeg are given as Figure 3-2.
Power Storage and Distribution

The batteries used are LiFePo4 100 Ah batteries with a 12.8V nominal voltage stored inside waterproof plastic ammo cans. These batteries will provide an endurance of about four hours when operated at moderate speeds (drawing no more than 25 amps). The batteries are recharged using a FMA Powerlab 6 charger from ProgressiveRC which provide up to a 40 amp charging rate and a 1 amp balance current.

Contained in the electronics enclosure is a power distribution board that includes the DC transformers to produce the required 6V and 12V outputs. This board is controlled by an Arduino. The power distribution system was designed to accommodate a sustained load of 50 amps. This value exceeds the actual current being drawn at maximum thrust, but provides assurance that there is adequate insulation and cooling.

CPU Box

The CPU Box is a watertight plastic enclosure with watertight connectors. This enclosure holds and protects most of the vehicle's electronics, with exception of the OS5000 compass, motor controller, and Freewave radio. The components will be individually explained, but to first provide a brief overview, Figure 3-3 show the mod-
ified CPU Box with key components labeled. The CPU Box enclosure has electronic components separated into two levels. The top level with the smaller components is shown as Figure 3-3a while the bottom level is given as Figure 3-3b.

![Labeled CPU Box Enclosure](image)

(a) Top Level
(b) Bottom Level

Figure 3-3: Labeled CPU Box Enclosure

**Sensors**

**Ocean Server OS5000 Compass:** This is a tilt-compensated compass that is easy to calibrate. Because of electromagnetic interference, the compass sensor was moved out of the CPU Box and now resides in a separate enclosure at the very front of the kayak. The compass is converted to USB and is connected to the watertight USB connector of the CPU Box, and then internally to the USB hub.

**U-Blox 6 GPS Module:** This GPS module is connected via USB to the USB hub. It is also used to provide timing information to the WHOI micromodem via a separate connection. The antenna being used is a *Novatel GPS-701-GG* [12].

**Communications**

**WHOI Micromodem:** Underwater data communication between vehicles is through a WHOI Micromodem (25 KHz transducer, WH-BT 2) with a 4-element receive
array (HTI-96 MIN). Although the use of these underwater communications devices was the primary reason for the construction of the HoverGroup Autonomous Kayaks, because these modems were not used for this project, they will not be discussed in detail. Information is available on the HoverGroup wiki [12].

**WiFi:** A 2.4 GHz Bullet M from Ubiquiti provides 802.11 WiFi with an increased range over standard WiFi. The bullets are connected to the CPU Box via the RJ-45 watertight connection labeled "bullet". In addition to using the ethernet for bi-directional data, this hardware configuration uses Power-Over-Ethernet (POE) so that the power for the antenna comes from the CPU Box as well.

**Freewave Radio:** A Freewave FGR2-PE radio has also been added with a separate antenna. Unfortunately because of mutual interference, only the Freewave or the Ubiquiti bullet can be used at one time. The Freewave uses a 900 MHz ethernet radio. Inside the CPU Box, the XBee module was connected to the USB hub.

**RC controller:** This is a Radio Controlled (RC) remote and receiver as sold for hobby remote-controlled boats, cars, and planes. The receiver is a Futaba R6203SB and is connected to the Arduino. The system is configured such that if the remote is powered on, the Arduino will give overriding control to the RC radio over commands from the onboard processor.

**Processing and Data Storage**

The onboard computer is a Gumstix DuoVero Crystal (GUM4430C) mounted on a Parlor breakout board. A detailed specifications data sheet is included in Appendix A. This computer contains a microSD card for storage, and runs on a Debian build of Linux. At the time of this project, this computer was only a few years old, however there are already a number of more capable computers available for an equivalent price. A photo of the Gumstix as mounted in the CPU Box is shown as Figure 3-4. The computer is connected to the USB hub via the USB micro port, and also to
the ethernet switch via the RJ-45 connection on the breakout board. The processor uses only a heat sink and ambient air for cooling. Power is provided from the power distribution board.

![Figure 3-4: Onboard Gumstix Computer](image)

### 3.2.2 Hardware Modifications

In which to adapt the HoverGroup Autonomous Kayak to a configuration that is useful for developing and testing a prototype Acoustic Response Control System (ARCS), a few hardware modifications were required. As the HoverGroup Autonomous Kayak is being used for active research by other students, it was important that all modifications could be easily reversed, and where possible, all inactive hardware was disconnected but left installed.

#### Hardware Disconnected and/or Removed

**WHOI Micromodem:** Initially it was hoped that the WHOI micromodem, or at least the array of passive hydrophones that are part of this system, could be used for measuring the acoustic response for the ARCS. However, after many attempts to reconfigure, it was found that the modem has a hard-wired bandpass filter that blocks all non-communication frequencies. With only being able to measures amplitudes in the 10 - 25 kHz range, it was ill-suited for measuring the
kayak’s self-noise. The array and the towed transducer were disconnected at the through-hull connectors on the kayak. The pins were protected with dummy plugs. The internal cabling was tied back and sealed. Water-tight caps were placed over the now unused CPU Box watertight connectors for the array and Micromodem. Internally, all connections, including the 4-port serial to USB converters were left intact and unaltered.

**Ocean Server OS5000 Compass:** In which to measure acceleration, an accelerometer is needed. Given that Internal Measurement Units (IMUs) are relatively inexpensive and can perform the combined functions of a magnetic compass, gyrocompass, and accelerometer, it made the OS5000 compass redundant. Also, as the OS5000 compass was external to the CPU Box, this made the USB watertight connector available to be used for other applications. The USB cable coming into the CPU Box was tied back and made waterproof, but otherwise the compass along with its separate enclosure in the kayak bow, was left unaltered.

**Freewave Radio:** As the kayak will be operated only in water near the MIT Sailing Pavilion, the extended range of the Freewave is not required. The Freewave antenna was disconnected, and the cabling both inside and outside the hull was tied back and sealed. Internal to the CPU Box, the USB cable going to the XBee module was disconnected.

**Hardware Installed and/or Modified**

**Installation of an Inertial Measurement Unit (IMU):** Although the GPS can also give an estimate of speed and acceleration, it does so by differentiating position with respect to time, and therefore is less accurate at measuring quick changes. As the ARCS should be able to evaluate when the vessel is accelerating or decelerating (in which to capture the *forced propeller slip*), the decision was made to install an IMU.

Based on the performance, modest price, and the availability of open-source firmware, the SparkFun 9 Degrees of Freedom - Razor IMU - Model:
SEN-10736 was selected. Out of the box, the IMU does not come with any easy way to power the board or to interface with the kayak’s onboard computer. To resolve this, pin headers were bent 90 degrees and soldered to the Future Technology Devices International (FTDI) connectors on the IMU. Using male-male jumper pins, the pin headers from the IMU were connected to a 3.3v FTDI basic breakout board. This small breakout board works as an adapter and has a micro-USB interface which can be used for both powering the IMU and communications via the USB hub to the onboard computer. Specifications and the pinouts for both the IMU and the FTDI basic breakout board can be found in Appendix A.

There was very limited space left on the top level of the CPU Box as shown in Figure 3-3a. The only location where both the IMU card and the FTDI breakout board could be installed was in the top left corner. In which to use this space, the GPS card had to be rotated 180 degrees so that its separate connection to the WHOI micromodem would be pointed outward and not be in the way. Figure 3-5 shows the section of the CPU Box before and after the modifications were made. Due to the space restrictions, the IMU was installed such that the forward direction for the IMU, was 90 degrees out from the forward direction for the kayak. This required that software corrections be applied to heading, pitch, roll, and axial accelerations. The software code for these corrections are included in the SpecIMU MOOS application that can be found in Appendix B.
Sound Card: Once it was determined that the WHOI micromodem and array could not be used for measuring self-noise, the next option was to use a sound card. Sound cards—either USB or onboard the computer—are supported by most operating systems, and will have all the digital-to-analog converters and signal processing hardware required to handle a single microphone input. Since only one audio input is required to measure self-noise, this makes using a sound card a good option.

The next option that was explored, but ultimately not chosen, was the
use of the onboard sound card that is built into the Gumstix and uses a 3.5mm audio connector on the Parlor breakout board. This option was not implemented for a few reasons: The first issue encountered was that the Gumstix uses a very uncommon OTMP standard for the 3.5 mm audio port making it difficult to find a matching amplifier. Secondly, using the Gumstix’s onboard audio required a lot of software configuration that is computer-specific. This means that if the Gumstix were to be replaced with an upgraded computer, the software would also need to be modified accordingly. Finally, using the onboard audio would require the CPU Box be modified to include another watertight connection.

Instead of using the onboard audio adapter, a very inexpensive (<$7 USD) Sabrent AU-MMSA USB sound card was selected. This card has separate 3.5mm connection for a microphone, making it much easier to find a matching amplifier. As a generic sound card, no drivers are required. Also, generic sound card settings could be used, allowing the software being written for this project to not be operating system dependent. Also, by using the USB watertight connector that was previously used by the OS5000 compass, no modifications to the CPU Box were required. Specifications for the USB sound card are given in Appendix A.

Hydrophone: The hydrophone that was selected was the Aquarian Audio Products H2a-XLR hydrophone. The reason for selecting this particular hydrophone was because it had the required bandwidth (<10 Hz to >100kHz) for measuring broadband acoustic noise, and also because it was available and unused within the department. This is a omni-directional hydrophone that is waterproof up to a depth of 80m. The hydrophone is designed to be powered by an amplifier that is able to provide +48V Phantom Power, which increases both the gain as a standard amplifier would, but also provides power to the hydrophone.

Amplifier: Using an independently powered amplifier allows electrical power to be supplied from a different battery than the one providing power to the propulsion system. This is important, as electrical noise from the propulsion system (which
is a common result from the use of electric motors) will also cause electrical noise on all systems powered from the same power source, unless specifically filtered. To avoid having any electrical noise affecting the quality of measured acoustic data, an amplifier that has a separate power source was selected. The *Armonic SmartRig Audio Adapter* was chosen for the amplifier as it meets the requirements of having *Phantom Power* and is powered by a separate 9V battery. This amplifier has the connectors needed to interface with both the USB sound card’s 3.5 mm audio port, and the hydrophone’s XLR connection. The hydrophone has a gain potentiometer which allows for the gain to be adjusted manually as required, which is very helpful during initial experimentation. The amplifier limits bandwidth to the audio range, and therefore measured frequencies are between 20 Hz and 20 kHz which is sufficient for the application. Technical information on both the hydrophone and the amplifier can be found in Appendix A.

The hydrophone was suspended in the water next to the kayak so that it was far enough away from the kayak in which to measure far-field noise. It was suspended to the side in which to prevent it from being subjected to the turbulent flow caused by vortex shedding from the propeller and hull as this would cause localized pressure variations, which were not representative of far-field acoustic noise. By the nature of being suspended, there should be minimal vibrations transmitted to the hydrophone from the kayak structure vibrations as is the case with hull-mounted hydrophones as discussed in Section 2.2.1. This arrangement, however, will have some flow noise across the hydrophone itself, as well as vortex-induced vibrations in the cable from which it is suspended. Given that the system does not need to be exact, this sources of error were deemed acceptable. The mounting and suspension arrangement is shown as Figure 3-6.
Firmware Modifications

Although software will be discussed separately in Section 3.3, firmware modifications will be discussed here. In this context, firmware refers to reconfigurable persistent software stored in individual components of hardware, such as on the GPS receiver and the IMU. The only firmware that was modified for this project was that for the IMU.

A very capable open-source firmware for this IMU was developed and published by Peter Bartz (2013), and Quality & Usability Lab, Deutsche Telekom Laboratories, TU Berlin (2011-2012). The firmware is widely available under the name "Razor AHRS Firmware v1.4.2". It is open source code and was released under GNU GPL (General Public License) v3.0. This firmware was very well written and robust, but it only had the ability to output sensor data in a few different modes. After the IMU was calibrated and with the firmware as distributed, the data sent to the CPU was limited to either angles or sensors data, but not both.

Angle Output: In this mode, the IMU combines the measurements from the accelerometers, gyros, and magnetometers to produce tilt-corrected values for heading (yaw), pitch, and roll. The algorithms are fairly sophisticated, and all the complex calculations are done on the IMU itself, which takes some of the computational load
away from the kayak's onboard computer. The heading value is needed to replace the compass that was disconnected. The ARCS is only interested in acceleration that is due to changes in the kayak's motion. Therefore, in which to separate motion-induced acceleration from gravity as the kayak pitches and rolls while maneuvering, the ARCS will also need to know the pitch and roll at any given time so that the contribution of acceleration from gravity can be subtracted. The data output mode from the IMU is selected by sending the message "#ot" over the serial interface to the IMU. An example of two successive output messages in the "angles mode" appear in the order of yaw, pitch, roll as:

#YPR=-48.76,1.20,0.61
#YPR=-48.75,1.20,0.63

Sensor Data: In sensor data mode, the accelerometers, gyros, and magnetometers produce either calibrated or raw data for each of the X,Y, and Z axes. The three axial measurements, over the three different types of sensor comprises the "9 degrees of freedom" advertised by the IMU manufacturer. The only values here that are needed for this project are the accelerometer readings. The output is selected by sending the message "#osct" over the serial interface to the IMU. Two successive outputs messages in "sensor data mode" are given here in the format: Accelerometer X,Y,Z; Magnetometer X,Y,Z; Gryro X,Y,Z:

#A-C=-5.32,2.91,251.10
#M-C=90.62,116.83,327.23
#G-C=-1.95,-1.94,25.05
#A-C=-6.29,3.88,251.10
#M-C=91.62,121.47,324.18
#G-C=-1.95,-2.94,24.05

In which for the ARCS to measure acceleration, correct for gravity, and function as a compass, data is needed from both of these modes, however the firmware as distributed only allows one or the other output mode to be selected. To address this, the firmware code was modified to include a "hybrid mode". This hybrid mode is selected by sending the message "#osht" over the serial interface to the IMU. In
this new mode, the IMU outputs alternating lines of data. The first line is output angle message. The next line is the calibrated sensor data message, but only the line for acceleration is included. Two successive outputs now appear as:

#YPR=-47.33,1.14,0.73
#A-C=-2.76,2.82,274.71
#YPR=-47.32,1.10,0.75
#A-C=-2.76,2.82,273.75

The modified Razor AHRS Firmware was uploaded into the IMU. The IMU was calibrated, and the correction values were entered directly into the firmware. This IMU allows for correction of electromagnetic interference provided that it is calibrated in the same environment in which it is used. This project uses this modified firmware with "hybrid mode" selected. Although open source, the original code (Ver 1.4.2) developed by Peter Bartz is copyrighted. When working with open-source code, version control is important, and therefore, under normal circumstances, only the author releases new and modified versions of the code. For these reasons, neither the original nor the modified code that was developed for this project will be included with this paper.

3.3 Software Configuration

The majority of the effort in creating the prototype ARCS, was spent developing software. A significant amount of existing software tools were used (when possible), however, in which to create the algorithms and data-handling abilities required for a CEC prototype, three Mission Oriented Operating Suite (MOOS) applications were created. There are over 3,200 lines of code between these three applications. Despite the abundance of code, the prototype is not sophisticated in terms of machine learning, nor in how well it defines the ship propulsions state, as the objective is only a "proof-of-concept". In that regard, the data mapping algorithm simply creates the cause-effect relationship between the power output of the thruster pod (measured as "percentage thrust"), and the Root-Mean-Square (RMS) amplitude across all mea-
surable frequencies (broadband noise). The ARCS will then use the CEC's data map to determine the optimal speed for any user-defined response threshold.

Descriptions of third-party software will only be briefly introduced here. If more detail is required on any the third-party software, links are provided to the software developers web pages in the bibliography.

3.3.1 Existing Open-Source Software Used

MOOS MOOS-IvP

The Mission Oriented Operating Suite (MOOS) was developed by Paul Newman, a postdoc working for Professor John Leonard at MIT. Building upon the MOOS architecture, Interval Programing (IvP) was developed by Mike Benjamin to allow for multi-objective optimization. MOOS-IvP is a set of open-source modules, applications, and utilities, written in C++. MOOS and MOOS-IvP are used extensively for autonomous marine vehicles in MIT's Ocean Engineering and Sea Grant Programs.

MOOS is a cross-platform middleware that operates between the host computer's operating system and a large number of available applications which provide functionality to autonomous vehicles. The architecture allows for most applications to work on any platform running MOOS. The MOOS architecture uses a common database called "MOOSDB" that holds all the pertinent information that is needed to be passed between applications, and also controls the timing and flow of information. One vehicle, running a single MOOSDB with any number of applications is called a "community". More details on MOOS and MOOS-IvP can be found on the reference webpage [1].

By the nature of MOOS architecture, it uses asynchronous timing. This method is very efficient for multi-tasking, as each application will be run in succession, thus ensuring that CPU resources are routinely made available to all applications. This prevents any one task from monopolizing the CPU resources, effectively starving the other applications. As part of the MOOS construct, each application
can be configured to run and communicate more or less frequently than the other applications. This configurability allows more time-sensitive applications to be run more frequently as desired. There are, however, physical limitations as to how quickly the applications can be run based on available processing power. If the combined application load does not allow for all applications to run as often as configured, the applications won't achieve their desired refresh rate. Another advantage of the asynchronous architecture is that it allows for programs to join or leave the community seamlessly without affecting the timing of the other programs.

However, there are some notable drawbacks to an asynchronous architecture as it relates to this project, as the timings are neither exact nor predictably periodic. A synchronous architecture would ensure that each data entry occurs at the same time each cycle; there may still be timing delays, but the timing delays would remain constant. This is not the case for MOOS. Being asynchronous, when one application sends a value to the MOOSDB, there will be some time delay before another application receives the message. When precise timing is required, the differences can be significant, and determining the values of multiple variables at any one specific point in time can be challenging. MOOS would lose a lot of capability if it were limited to being synchronous, so there are different approaches to achieving more precise timing while remaining asynchronous. One option is to take advantage of the feature where the MOOSDB records not only the messages, but when the messages were posted. This will allow for some corrections to be made to timings. This feature was used to some degree for the new MOOS applications created for this project.

Another limitation of MOOS being asynchronous is that data may be coming in from a sensor (or another data source) much more quickly to one particular MOOS application than how often the MOOS application itself is run. This was indeed the problem with the IMU initially; it would update the acceleration readings at 20Hz, but the application that reads the data is only run five times a second (5 Hz); maybe more, or maybe less depending on the CPU loading. If not addressed, this would constitute a huge amount of data loss when the IMU application is suspended
periodically so that all the other MOOS applications can be run. The solution that was used in this project was to buffer the data when the application wasn’t active. When the application does become active, it will gather up all the data from the buffer and pass the data as a packet of processed information to the MOOSDB. For the example of the IMU, all data will be stored into the buffer at the full 20Hz. When the application runs, it will produce an average acceleration from the entire contents of the buffer, and post this average to the MOOSDB as often as the application runs (about 5Hz). This will ensure minimal data loss, albeit at the cost of some data resolution. However, a sampling rate of 5 Hz, or every 200 milliseconds, provides ample precision for the purpose of capturing the kayak’s movements.

**Third-Party MOOS Applications Used**

There are several MOOS applications that were required to be run in which for the vehicle to operate autonomously. The complete list of applications can be read from the MOOS Mission file which is provided at Appendix B. However a few notable applications are mentioned here:

**pLogger:** This MOOS application is part of "core MOOS" and is an essential tool for debugging. pLogger is a program that creates data logs of all information passed to the MOOSDB. Generally the focus is on the asynchronous log function, where messages are recorded to a log file as often as they change. There is also a synchronous log function that will create periodic entries. It is this synchronous log (SLOG) that is used by the prototype ARCS. Other applications store sensor data in the SLOG file, and the CEC uses these data entries to create the data maps. Based on testing, it was determined that setting the SLOG file to record every 200ms was the right balance between obtaining reasonable precision while not overwhelming the processor. It is important to note that what the pLogger application does is record all the last known values periodically. "Last known" is significant, as if a value changes multiple times between SLOG entries, only the last entry will be recorded. Also, if the value was up-

101
dated immediately after the last reading, it can be up to 200 ms off from the
time it will be listed with in the SLOG. These margins of error are acceptable
in the case of the ARCS prototype. As previously discussed in Section 2.2.2,
any sampling rate better than 1 Hz should be sufficient.

**pMarinePID _Hover:** This is a MOOS application that works as a PID controller.
When the application is given a desired speed, it will act as a PID controller
and adjust the percentage thrust in which to maintain the desired speed. As
a PID controller, it will automatically correct for disturbances such as waves,
drift errors from water currents, and varying battery voltages. For this project,
a CEPID is created by using a new MOOS application that first creates a CEC,
and then modifies this PID controller MOOS application accordingly. In this
very simple proof-of-concept design, the CEPID application does this by simply
sending pMarinePID _Hover new acoustically optimized values for maximum
allowable thrust whenever the user changes the *response threshold.*

**pHelmIvP:** This application acts for an autonomous vehicle in the same capacities
as a combined autopilot and navigation system does on a ship. This program
determines the speed and heading changes that the autonomous vehicle takes
in accordance with the behaviors that have been specified. This application is
relevant to this project as it acts as the full-scale autopilot equivalent for the
ARCS prototype. The MOOS-IvP website is an excellent source to learn more
about pHelmIvP and behaviors [1].

**Third-Party Software Used**

Other than MOOS and the firmware for the IMU as discussed in Section 3.2,
the only other third-party software used was a free software program called "Sound
eXchange", or "SoX". This application can be called from the command line of
multiple operating systems and can record, convert, and play back audio files. SoX
is used by one of the new MOOS applications developed for this project that reads
and analyses self-noise. It is relatively easy to use the command line from within
a C++ application to call another program and read back the results. In addition to being able to record audio, SoX is able to conduct an analysis of the recording, including maximum amplitude, RMS amplitude, and rough frequency among others. So rather than create an entirely new program to gather all the statistics from the audio being recorded by the ARCS acoustic sensor, the MOOS application can simply call the SoX program to take short period recordings (samples), and then use the SoX analysis tools to return the pertinent statistics for this project: RMS Amplitude and Max Amplitude.

SoX is produced mainly by sourceforge, and is free to use under the GNU General Public License version 2.0 (GPLv2) and the GNU Library General Public License version 2.0 (LGPLv2). More information on the product can be found on their web page [2].

3.3.2 MOOS Software Modifications

In addition to the new applications that were created, the only modification to other MOOS applications, was to "pMarinePID_Hover". In which to make the CEC and this PID application function as a CEPI D, a small modification was made to pMarinePID_Hover. As written, the maximum allowable thrust that pMarinePID_Hover will use is read from the MOOS mission file on startup only. As part of this project, pMarinePID_Hover was modified slightly to subscribe to the MOOS message "MAXIMUM_THRUST" and "MINIMUM_THRUST", and would allow for the internal values of maximum and minimum thrust to be changed by these MOOS messages. This allows for run-time changes, but the remainder of the code otherwise remains unaltered. The CEPI D application will now be able to control this PID controller application through the two aforementioned MOOS messages. Because this change was fairly minor, and not interfere with MOOS version control, the modified pMarinePID_Hover is not included in this project.
3.3.3 New Software Developed

In which to build the working ARCS prototype, three new MOOS applications were built. All three are included in Appendix B. This section will give a brief overview of the functions that they perform.

Specialized IMU Driver - SpecIMU

This code takes the modified messages from the serial port in the asynchronous MOOS environment and outputs the heading, pitch, and roll, along with acceleration on all three axes to the MOOSDB. It performs the asynchronous serial operations as described in Section 3.3.1. As the IMU replaces the functionality provided by the recently removed OS5000 compass, vessel heading functionality was also added. This application has the ability to "zero" the IMU. As the IMU will not be kept perfectly flat, there will be some component of gravity acting on all axes, even when the kayak is motionless in the water. The kayak will also settle at some non-zero pitch and roll when motionless in the water. The "zeroing" procedure finds the average values over time while the kayak is still, then subtract those average values as a correction to all further outputs. This effectively corrects for the IMU not being flat in the kayak, as well as the kayak not floating perfectly flat in the water. Another unique function performed by this MOOS application is to correct for pitch and roll in real-time as the kayak moves. Although "zeroed", as the kayak pitches and rolls due to wave action, gravity will again provide non-motion related accelerations to the IMU. This additional function uses trigonometry to remove the gravity component in real time for the changing angles of pitch and roll, such that the only acceleration being recorded are indeed from the kayak's motion. A brief summary of the key functions performed by this MOOS applications are listed as follows:

- Reads all the vehicle-specific configuration data from the mission file, such as the USB port and baud rate of the IMU
- Asynchronous serial port read-and-write functionality
- Corrects from being installed 90 degrees from "forward" by swapping pitch and
roll, and also renaming directional axes. Manual heading offsets can also be entered via the mission (configuration) file for correction of permanent misalignments and declination.

- Updates pLogger via MOOSDB with heading, pitch, roll, and gravity-corrected axial accelerations
- Has the ability to change modes, including calibrating/zeroing via commands from the MOOSDB
- Creates a separate IMU log in the pLogger directory for additional data or de-bugging

**Read Self-Noise Program - ReadSelfNoise**

This MOOS application configures and reads acoustic information as measured by the hydrophone, then outputs those key values to MOOSDB. As described earlier, "SoX" is used to capture very short audio recordings and then perform statistical analyses on those recordings. The application can record and retain small wave files (.wav) together with passing along the statistical analysis. Alternatively, if desired, this application is able to perform only the statistical analysis, without creating audio files. After some experimentation, it was found that the best duration for audio recordings was 0.1 seconds, with a new recording started every 0.2 seconds. This equates to a duty cycle of about 50%, which should be enough data for the system to approximate the *acoustic response*. Recording and analyzing the audio files is very processor intensive, and this duty cycle allows the other applications to run as intended. A sample duration of 0.1 seconds also ensures that at least one full wavelength will be recorded for signals down to a frequency of 10Hz, which seems sufficient as the amplifier is only rated for frequencies above 20Hz. A brief summary of the key functions performed by this MOOS applications are listed as follows:

- Reads and applies the configuration information for the sound card from the mission file, such as the USB port number
- Will automatically reconfigure the sound card with the right settings, such as turning off "Automatic Gain Control" and setting microphone levels
• Reads and applies the configuration settings to call SoX for the specific type of recordings requested, and reads back only the statistical data requested
• Can be turned on and off via MOOSDB messages to save CPU resources when acoustic data is not needed
• Updates MOOSDB when it is successfully taking readings
• If requested in the mission file, can retain audio file records in the "wav" format
• Outputs to the MOOSDB, the RMS Amplitude, Max Amplitude, and the Rough Frequency of each sample

Cause-Effect Controller with PID Interface - CEPID

The CEPID application is the core of the ARCS prototype as it controls all other ARCS-specific MOOS applications and sensors. This application also finds the cause-effect relationship, and then uses that relationship to optimize propulsion, within the user-specified limits, by sending updated thrust minimum and maximum values to the PID application (pMarinePID_Hover). There is some terminology that has been reused in this application that can be ambiguous in the context of this project. Therefore, in the context of this MOOS application only, the term "initialize" means to send the signal to the "IMU" for it to "zero". The term "calibrating", refers to creating the cause-effect relationship between the ship-propulsion-state (which is "DESIRE__THRUST"), and the acoustic response (which is the broadband RMS Amplitude). The term "response curve" refers to the 2D plot (or curve) that relates "DESIRE__THRUST" to RMS amplitude. This same curve is the "data map", and once generated, will be used to find the maximum allowable thrust values for any given response threshold. A brief summary of the key functions performed by this MOOS applications are listed as follows:

• Can either create a "response curve", use a previously created and stored "response curve", or can bypass the CEC altogether to simply pass unrestricted operations to the PID. These values can be set in the MOOS mission file
• The application can perform noise monitoring where it records and populating the MOODB with self-noise statistics regardless if the CEC is being used or
bypassed. This can be turned on or off, either after calibration, or in PID-only mode, to save CPU resources.

- Will automatically find the pLogger SLOG file director and use it to extract information for generating data maps.

- During "initialization" while the IMU is "zeroing", this application will measure and record the ambient noise level. This allows for response thresholds to be set in terms of ambient noise instead of just fixed values.

- When a new response threshold is sent, the system will automatically find the new values for minimum and maximum thrust and pass it to the PID controller.

- The CEPID application also has a simulation mode, where previously gathered data from a SLOG file can be used to test the algorithms for generating the cause-effect data map (which is the response curve for this ARCS prototype).

- This application will synchronize timing through the different MOOS applications by reading the configuration file to determine the start order, and then provide time corrections and offsets. This will make the data analysis much more accurate. There is also a time offset applied to recorded data as the data isn’t posted to the MOOSDB until some time after the causal event occurs. The value can be changed via the mission file.

- The application includes some rudimentary error correction capability. When multiple calibration runs are conducted (at least five are recommended), it will recursively replace outlier data that was likely due to an external noise source. A data point is considered an "outlier" if its value on a single calibration run is significantly different than the average of the same point in the other calibration runs. The configuration file allows for the user to change the "percent-difference" threshold that defines an outlier. In the code, this process is referred to as "data grooming".

- As this is prototype code, the system creates a temporary debugging file where programmers can choose to write data at any number of test points to the debugging file to confirm the validity of the data handling. The pre-established test points are commented as "TEST CODE".
Figure 3-7 is provided to give some insight into the logic flow between different modes in the CEPID application. It also shows the conditions under which the system will change modes. The term "state" is used interchangeably with the term "mode", and the term "state change" is equivalent to the term "mode change".

Figure 3-7: CEPID Application Logic Flow
4.1 Numerical Value Descriptions

As this is a prototype ARCS, there is no need to calibrate sound levels in terms of a reference value and then measure dB from that reference. Instead, to keep the data easier to understand, dimensionless quantities are being used for both Max Amplitude $A_{\text{Max}}$ and the RMS Amplitude $A_{\text{RMS}}$. Conveniently, as part of the digitalization of audio signals performed by the sound card, the values are dimensionless by default. To make the audio sample recordings, a 16-bit depth was used, meaning that the number of discrete levels that can be recorded is $2^{16} = 65536$. As the sound is a signed integer (can be both positive and negative), this means that all sound will fall into discretized levels between $-32769$ and $+32768$. The values used for this project are made dimensionless at the sound card by dividing the quantity being measured by the maximum possible quantity that can be measured as shown in Equation 4.1.

$$A_{\text{Max}} = \frac{\text{Amplitude}}{\frac{1}{2}2^{16}} \quad \text{Where} \quad -\frac{1}{2}2^{16} < \text{Amplitude} \leq \frac{1}{2}2^{16} \quad (4.1)$$
Therefore, according to Equation 4.1, $A_{Max}$ and $A_{RMS}$ (which is calculated in the same way) will always have values between -1 and 1. Although $A_{RMS}$ is used for data mapping as it is a better measure of acoustic noise energy, $A_{Max}$ needs to be monitored as well to avoid "clipping". As shown in Figure 4-1, clipping is when signal information is lost because parts of the signal exceed the maximum amplitude quantity that can be captured. The values will instead be truncated to the maximum value creating an inaccurate discretization of the input signal. Even though the $A_{RMS}$ may never go all the way up to a value of 1.0, if the $A_{Max}$ does, even briefly, it indicates that clipping has occurred, and the value of $A_{RMS}$ is not accurate. In other words, $A_{RMS}$ is a useful non-dimensional measure of the acoustic response, provided that the amplifier gain is adjusted such that $A_{Max}$ never reaches a value higher than 0.99.

![Figure 4-1: Example of Audio Clipping - Image from wikipedia.com](image)

### 4.2 Incremental Testing & Results

#### 4.2.1 Tow Tank - Static Testing

The first set of tests were conducted with the kayak tied to the side of the tow tank. This placed the kayak and propeller in a constant state of forced propeller slip. The first objective of these early tests was to determine what the gain setting should be on the amplifier. The goal was to set the gain high enough to get useful data at low values of thrust, and also low enough to not exceed $A_{Max} = 0.99$, as this would distort the associated value of $A_{RMS}$. The setup used in the tow tank with the
kayak secured in place is shown as Figure 4-2. The hydrophone was suspended next to the kayak such that it wouldn’t be subject to direct wake effects.

![Figure 4-2: Static Tow Tank Arrangement](image)

**Amplifier Gain**

A propulsion step test was conducted that increased from 0 to 100% thrust in 5% increments, returned to 0 thrust, and then decreases in 5% increments to -100% thrust. Figure 4-3 shows the max amplitude vs the kayak thrust, and Figure 4-4 shows the RMS Amplitude. In this first set of data, clipping occurred starting at -65% thrust. This tells us that we should have reduced confidence in the accuracy of the associated RMS value. However, despite the clipping, with increased amplitude, the RMS Amplitude did still show an increasing trend, but the rate at which it increased slowed (which was likely due to the clipping).
Figure 4-3: Propulsion Step Test - Max Amplitude

Figure 4-4: Propulsion Step Test - RMS Amplitude

The results from this first analysis indicate that the manual gain should be adjusted such that clipping first occurs at around -85% thrust, and does not occur at any point while going forward. This seems like the best compromise in which to still have accurate readings at low speeds. As it is unlikely that, for any practical
reason, the kayak will actually go faster astern than -85% thrust, the corresponding 
acoustic response can be treated as effectively having a value of 1. This setting on the 
amplifier gain potentiometer was marked for consistency going forward. The manual 
gain setting had to be adjusted lower again after a later experiment involving dynamic 
(moving) tests as discussed in Section 4.2.2.

**Acoustic Response Smoothing / Filtering**

These tethered tests were also the first set of data that could be used to 
develop a method of smoothing, or "filtering" the input data. It is apparent from 
Figures 4-3 and 4-4, that some form of data filtering or smoothing is required as 
the acoustic response is sporadic. Two different localized-average data smoothing 
algorithms were used on the acoustic response data in an attempt to find more con-
sistency. Using different data smoothing techniques that are built into MATLAB, 
two smoothed response data graphs were produced. Figure 4-5 shows the use of the 
MATLAB "smoothing" function and Figure 4-6 shows the "Savitzky-Golay" filter.

![Figure 4-5: Propulsion Step Test - RMS Amplitude - MATLAB Smoothing Function](image)
Neither smoothing technique was consistent enough to be used. However, as finding the best filter to manage incoming data is outside the scope of this project, a more "brute-force" method was used that takes the mean across an entire sector. A sector in this context is the time domain over which the thrust remains constant. This same data is presented using sector averages as shown in Figure 4-7.

Figure 4-7: Propulsion Step Test - RMS Amplitude - Using Sector Mean
Looking closely at each thrust transition, it was clear that the acoustic response followed a change in thrust by about 0.6 seconds. This is most clearly seen when the thrust goes from 100% to 0, after which the response should drop to the ambient level. A zoomed-in view of this data is shown as Figure 4-8. This response delay is a combination of the time it takes between when the new thrust is requested and the Arduino receives the message, the time it takes to generate the new thrust, plus the time to generate, analyze, and post the audio recording statistics. The delay was consistently about 0.6 seconds. Therefore, an algorithm is used that shifts the response data forward by 0.6 seconds so that the data better correlates, and the calculated sector means are more accurate. This algorithm is included as part of the method used by the CEPID in the C++ code to create the data map. The time shift correction of 0.6 seconds can be adjusted from the MOOS mission file.

Figure 4-8: Response Showing 0.6-Second Time Delay Between Thrust and RMS Amp

Repeatability of Data Results

Although developing a more advanced machine learning algorithm was beyond the scope of this project, a method had to be developed to create a response curve that was consistent. To address this, multiple step tests were run in succession
to compare the consistency of the sector mean RMS amplitude between runs. The first set of five step tests is shown as Figure 4-9. There are two anomalies that were known to be caused by external sound source contamination as marked with red circles. The tow tank is located near a subway line, and passing trains were audible at the two times where anomalous data was recorded. The technique used to correct for these errors will be discussed later in this section. However, it was also notable that the sector means are not perfectly consistent outside of the anomalies either. It can also be seen that in a few instances, the sector mean was lower at 100% thrust than it was at 95% thrust.

![Medium Sized Propulsion Step Test](image)

Figure 4-9: Propulsion Step Test #1 - Using Sector Mean - Multiple Tests

For comparison, a second set of five runs was also conducted as shown in Figure 4-10. During this set, there were again two passing subway trains, which are also marked with a red circles. The solution to resolving the discrepancy between runs within each set was to average the sector means between runs. This is what was implemented in the CEPID in the C++ code. The results of averaging the sector means, with error rejection, is given later in this section where the results from each set of five runs are compared.
Rudder Noise

Because the kayak uses a directional thruster pod to provide steering control during later testing, it was important to determine if moving the pod was going to produce a significant *acoustic response* on its own that could inject errors into the CEC. In which to keep terminology consistent with the MOOS-IvP terminology, the directional thruster is considered a "rudder". Recall that for this proof-of-concept data map, the pod (rudder) angle is not included in the *ship propulsion state*. To measure the *acoustic response* of only the changing thruster pod angle, a rudder step test was performed. The rudder movements were: 0, -15, 15, 0, -30, 30, 0, -45, 45, and 0 in 4-second increments. Figure 4-11 shows the RMS amplitude recorded during the movements.
It can be seen in Figure 4-11 that, as expected there is an increase in the RMS Amplitude immediately following a rudder movement. However, the *acoustic response* (RMS amplitude) values are very low in comparison to those of propulsion. For comparison, the same rudder step function is presented again along with the propulsion RMS amplitude for the same period of time as shown in Figure 4-12.
With the *acoustic response* (RMS Amplitude) from the rudder only shown in green, and the *acoustic response* of propulsion only shown in red, it is clear that even at very low thrust, the propulsion noise far exceeds that of the rudder movements. For this reason, the self-noise contributions from the small rudder movements required for the kayak to keep course in open water can be neglected.

**Variation in Sample Duration**

As described in Section 3.3.3, the audio samples that are used to determine the *acoustic response* were chosen to have a duration of 0.1 seconds, and the statistics are uploaded to the pLogger SLOG file every 0.2 seconds. Experiments were performed based on the same propulsion step test to determine the effects of varying the sampling duration. The original 100 ms sample is shown as Figure 4-13, with 150 ms, 180 ms, and 200 ms, shown as Figures 4-14, 4-15, and 4-16 respectively.

Figure 4-13: Propulsion Step Test - 0.1-Second Sample Size
Figure 4-14: Propulsion Step Test - 0.15-Second Sample Size

Figure 4-15: Propulsion Step Test - 0.18 Second Sample Size
From this data, there was no clear increase in the consistency of the *acoustic response* in that it still fluctuates around the sector mean. However, this does provide two different observations. The first observation is that at longer sampling durations, the raw signal is periodically flat, showing no change between recorded readings. As can be verified by examining the log file, what is happening is that at larger sampling sizes, the processor is not able to analyze the audio sample and pass it onto the pLogger quickly enough before the next line of the SLOG file is written. This means that the same old data is passed forward. The goal was to have a pLogger sampling rate of 0.2 seconds (5 Hz). When looking at the SLOG file, where old data is repeated as the new data did not arrive in time, we find that the effective sampling rate suffers. Table 4.1 shows the comparison between the sample duration size and the effective sampling rate. The highest sampling duration that did not decrease the effective sampling rate occurred at 150 ms. Although a sample duration of 150 ms could be used, it was decided to continue to use 100 ms to avoid future CPU-loading problems, as the CPU will be more heavily used during open-water operations when more MOOS applications are running.
<table>
<thead>
<tr>
<th>Sample Duration [ms]</th>
<th>Actual Sampling Period [ms]</th>
<th>Effective Sampling Frequency [Hz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>250.9</td>
<td>3.9862</td>
</tr>
<tr>
<td>150</td>
<td>253.7</td>
<td>3.9423</td>
</tr>
<tr>
<td>180</td>
<td>419.7</td>
<td>2.3829</td>
</tr>
<tr>
<td>200</td>
<td>420.6</td>
<td>2.3778</td>
</tr>
</tbody>
</table>

Table 4.1: Sample Duration vs Effective Sampling Rate

The second observation is that as the sample size increased, there were unexpected results where the acoustic response at 100% thrust is actually less than it was at 95% thrust. This could be an error induced by noise, or it could be a measure of what is actually occurring. It is possible that 95% thrust causes propeller-induced hull vibrations that are closer to one of the natural frequency modes of the kayak hull than at 100% thrust or another equivalent phenomenon. More research could be conducted in this area, including frequency analysis, to determine precisely what is occurring. It is also possible that a portion of the subsequent zero-thrust acoustic response data is being included in the sector mean for 100% thrust, and is therefore artificially driving down the value of the sector mean. If the decrease at 100% thrust is accurate, then more work should be done with a more powerful processor that is able to take longer samples more quickly. Ideally, this could be achieved by using a multi-core processor and dedicating an entire core to audio recording alone. This would prevent the large drain on CPU resources affecting other aspects of the kayak’s operation. That being said, a duty cycle greater than 100% cannot be achieved without double-counting data. With a 0.2-second recording being conducted every 0.2 seconds, the duty cycle is already 100%, and increasing the processor power would not allow going beyond 0.2 second recordings.
Step Test - Response Curve

Based on taking the average of the sector means over the five propulsion step test runs, the response curve was generated. The following data was found that relates the thrust to the broadband acoustic response. Numeric values are given in Table 4.2, and are shown graphically in Figure 4-17. Both the table and the figure show each of the two sets of five propulsion step tests.

<table>
<thead>
<tr>
<th>Thrust</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Thrust</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Thrust</th>
<th>Set 1</th>
<th>Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-100</td>
<td>0.7206</td>
<td>0.7462</td>
<td>-50</td>
<td>0.0141</td>
<td>0.0116</td>
<td>0</td>
<td>0.0033</td>
<td>0.0037</td>
</tr>
<tr>
<td>-95</td>
<td>0.6980</td>
<td>0.7212</td>
<td>-45</td>
<td>0.0122</td>
<td>0.0092</td>
<td>5</td>
<td>0.0032</td>
<td>0.0035</td>
</tr>
<tr>
<td>-90</td>
<td>0.6590</td>
<td>0.6722</td>
<td>-40</td>
<td>0.0095</td>
<td>0.0076</td>
<td>10</td>
<td>0.0033</td>
<td>0.0036</td>
</tr>
<tr>
<td>-85</td>
<td>0.6082</td>
<td>0.6260</td>
<td>-35</td>
<td>0.0071</td>
<td>0.0060</td>
<td>15</td>
<td>0.0034</td>
<td>0.0040</td>
</tr>
<tr>
<td>-80</td>
<td>0.5377</td>
<td>0.5325</td>
<td>-30</td>
<td>0.0067</td>
<td>0.0049</td>
<td>20</td>
<td>0.0036</td>
<td>0.0048</td>
</tr>
<tr>
<td>-75</td>
<td>0.4166</td>
<td>0.3724</td>
<td>-25</td>
<td>0.0086</td>
<td>0.0041</td>
<td>25</td>
<td>0.0038</td>
<td>0.0062</td>
</tr>
<tr>
<td>-70</td>
<td>0.2740</td>
<td>0.2209</td>
<td>-20</td>
<td>0.0153</td>
<td>0.0038</td>
<td>30</td>
<td>0.0041</td>
<td>0.0084</td>
</tr>
<tr>
<td>-65</td>
<td>0.1194</td>
<td>0.0972</td>
<td>-15</td>
<td>0.0161</td>
<td>0.0037</td>
<td>35</td>
<td>0.0045</td>
<td>0.0141</td>
</tr>
<tr>
<td>-60</td>
<td>0.0460</td>
<td>0.0381</td>
<td>-10</td>
<td>0.0098</td>
<td>0.0035</td>
<td>40</td>
<td>0.0047</td>
<td>0.0182</td>
</tr>
<tr>
<td>-55</td>
<td>0.0217</td>
<td>0.0168</td>
<td>-5</td>
<td>0.0063</td>
<td>0.0034</td>
<td>45</td>
<td>0.0049</td>
<td>0.0138</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td>0.0054</td>
<td>0.0080</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Acoustic Response Averages For Each Set of Step Tests

Figure 4-17: Acoustic Response Averages For All Values
Because of the significant increase in noise at high negative thrusts, resolution at smaller values is lost in which to accommodate the full scale. Figure 4-18 is given with better resolution, having limited the graph to only values above -65% thrust. There is notably an anomaly in Set 1 at around -15% thrust and another in Set 2 at around +40% thrust. Because the *acoustic response* values were very low at these two points, any external noise sources, such as a passing subway train, would heavily skew the data. These values do indeed correspond to the subway-induced anomalies as shown back in Figures 4-9 and 4-10.

![Figure 4-18: Acoustic Response Averages For Thrust >= -65%](image)

**Step Test - Response Curve Error Correction**

Given that in the tow tank, two passing subways trains were able to skew the data, it was apparent that even as a prototype, some type of error-handling algorithm was needed. The method that was integrated into the C++ code measures percent difference between each sector mean for a given thrust, and the average of all runs for the same thrust value. An assumption is made that it very unlikely that any unrelated noise source (contamination) will cause significant destructive interference (same frequency, 180-degree phase shift), and therefore all external noise sources will
increase the acoustic response. As the acoustic response is an RMS value, it will always be positive. Therefore, if a single measured sector mean for a given thrust on a given test is significantly higher than the average for the same sector across all tests combined, then the single value is likely erroneous. The exact algorithm used is a recursive function that can be found in source code at Appendix B.

As understanding this method simply from reading the C++ code can be difficult, an overview of the error-handling algorithm is presented here. The algorithm uses percent difference as a comparison, which will quantify the difference between each sector mean (for a given thrust), and the average between all runs. Because injected noise error will increase the acoustic response over a brief interval, it will also increase the average value of all points over the same interval. The error outliers would be the only sector means which would still have values significantly higher than the average, as valid data would all be less than the skewed average. The algorithm will then replace the outlier data with the average across all runs. It will then calculate a new average across all runs, and repeat this procedure until there is no singular value significantly driving the average away from the other points. The percent difference that is deemed acceptable can be adjusted in the MOOS mission file configuration. Figure 4-19 shows the original and "groomed" data for the first set of runs (Set 1), and 4-20 shows the same for second set of runs (Set 2).
As seen in Figure 4-20, the curve for Set 2 has been smoothed, correcting both subway-induced errors at -15% (Run 4) thrust and 55% thrust (Run 5) as shown in Figure 4-10. The same can be seen in Figure 4-19, corresponding to the subway-induced errors at 40% (Run 4) thrust and 95% thrust (Run 5) shown in Figure 4-9.
However, it is apparent that there is still an *acoustic response* decrease in the Set 1 data as the thrust is increased from 95% to 100%—even after external data source errors are removed. This correlates with the entirely independent data that showed the same effect occurring in Section 4.2.1 when varying sample duration. Although it will not be further investigated now for this project, it is worth gathering more data at these two thrusts, preferably with a processor capable of capturing longer sample durations to confirm the decrease in *acoustic response*. With more supporting data, this would strongly build the case that an ARCS can reduce the acoustic signature by increasing speed in certain applications as was proposed in Chapters 1 and 2. To show comparison and consistency between the two sets of runs (noting the difference at 100% thrust), both sets of groomed data curves are shown as Figure 4-21.

---

**Figure 4-21: "Groomed" Acoustic Response for Both Sets: Thrust >= -70%**

Based on these results, the algorithm is able to effectively create the cause-effect relationship between thrust % and the *acoustic response*. Although difficult to depict in this section, it can be seen that by running the attached C++ code that for any given response threshold sent to the MOOSDB, that the CEPID application very effectively reads the internally-stored data-map equivalent of these curves, and using interpolation for continuity, correctly sets the minimum and maximum thrust values.
Rapid Propulsion Changes

In addition to the propulsion step test, large propulsion changes were also conducted while the kayak was secured to the side of the tow tank. The purpose of these tests were for comparison later with dynamic tests during changes in propulsion. What is unique to the static tests is that the propeller is always in a state of complete *forced propeller slip* and there ia also minimal flow noise created. Later, during the dynamic tests, the propeller will be in the *forced propeller slip* condition only temporarily until the kayak has reached a steady speed. This comparison will hopefully provide some insight into the effects that *forced propeller slip* has on the *acoustic response*.

A series of rapid propulsion changes were conducted with large changes to the percentage of thrust. Two sets of five runs were conducted with each run including the following propulsion thrust changes:

- 0 to 50 to 100%
- 0 to -50 to -75%
- 75 to -75%
- 0 to 100%
- 100 to -75%

Although a total of two sets of five runs were conducted, the graphs contained too much information to clearly show results. For this reason, only the first two runs are shown for each set of runs, as depicted in Figures 4-22 and 4-23.
It is apparent in Figures 4-22 and 4-23 that the *sector mean* is visually less than the average (middle) of the response data of each sector. This is because it takes longer for the propeller to change speed with such large changes in thrust as compared to those earlier in this section where only 5% thrust increments were used.
Figure 4-23 was modified with the first 1.5 seconds of data removed from calculating the sector means, so as to not include data while the propeller was in transition. This is instead shown as Figure 4-24.

![Figure 4-24: Large Propulsion Change Response for Set 2 - 1.5 Second Delay](image)

With Figure 4-24, it is clear that the data lines up better with the sector means as expected. For matching values of thrust, the *acoustic response* values for both data sets containing rapid changes (both with and without the 1.5-second time shift), are shown compared to the *acoustic response* values of the 5% thrust step tests in Table 4.3. With the 1.5-second correction, the resulting values are as expected and much closer to the values of the 5% thrust step test.

<table>
<thead>
<tr>
<th>Thrust</th>
<th>5% Step Test 1</th>
<th>5% Step Test 2</th>
<th>Large Change Unmodified Set 1</th>
<th>Large Change Unmodified Set 2</th>
<th>Large Change Less First 1.5s Set 1</th>
<th>Large Change Less First 1.5s Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-75</td>
<td>0.4166</td>
<td>0.3724</td>
<td>0.3110</td>
<td>0.3322</td>
<td>0.3613</td>
<td>0.3380</td>
</tr>
<tr>
<td>-50</td>
<td>0.0124</td>
<td>0.01048</td>
<td>0.0125</td>
<td>0.0097</td>
<td>0.0101</td>
<td>0.0134</td>
</tr>
<tr>
<td>50</td>
<td>0.0054</td>
<td>0.00513</td>
<td>0.0057</td>
<td>0.0060</td>
<td>0.0062</td>
<td>0.0058</td>
</tr>
<tr>
<td>75</td>
<td>0.03218</td>
<td>0.0375</td>
<td>0.0376</td>
<td>0.0353</td>
<td>0.0379</td>
<td>0.0404</td>
</tr>
<tr>
<td>100</td>
<td>0.1201</td>
<td>0.1006</td>
<td>0.0885</td>
<td>0.0903</td>
<td>0.0964</td>
<td>0.0958</td>
</tr>
</tbody>
</table>

Table 4.3: Large Propulsion Change Comparison with 5% Step Test
4.2.2 Tow Tank - Dynamic Testing

For the dynamic (moving) tow tank tests, the kayak was tethered to a zip line using carabiners as shown in Figures 4-25 and 4-26 so that it would be free to travel forwards and backwards, but would not come in contact with the sides of the tow tank. Thrust duration was limited to only 8 seconds to prevent the kayak from running out of space and making contact with the ends of the tow tank. These tests provided an opportunity to assess the dynamic conditions—especially with measuring axial acceleration in calm water. However, with only 8 seconds of data, there was not enough run time to get very useful time-averaged data. Nevertheless, some key observations were made.
Flow Noise Test

Before beginning with powered dynamic runs, the kayak was dragged (un-powered) through the water at a speed roughly equivalent to 50-70% thrust. The purpose of this test was to get an idea of how much the flow noise contributes to the overall acoustic response. The results of the drag test are given as Figure 4-27.

![Dynamic Tests - Drag Test](image)

Figure 4-27: Non-Powered Drag Test to Assess Flow Noise

Even without power to the propeller, the flow noise was significant in that it was about the same as 70% thrust when the kayak was tethered to the side of the tow tank. For these reasons, the gain on the amplifier was turned down to ensure that there would be minimal audio clipping with the increased flow noise during dynamic tests. The new gain setting on the amplifier's potentiometer was marked for consistency with future experiments. The implication of having adjusted the manual gain setting is that data for dynamic testing cannot be compared directly with the static test results. As shown in Figure 4-27, not only is the flow noise significant, but it is also very erratic. This will further complicate results from additional short runs as even more time averaging would be needed to produce more constant response data.
Dynamic Acoustic Response

Three sets of powered dynamic runs were conducted. Each set contained five runs as follows:

- 0 to 50% (8 seconds)
- 0 to 60% (8 seconds)
- 0 to 50% (4 seconds), 50 to 60% (4 seconds)
- 0 to 70% (8 seconds)
- 0 to 80% (8 seconds)

The first set of five dynamic runs is given as Figure 4-28. The large time intervals between runs was to allow time for the water in the tow tank to become calm again. Because of the large scale of the thrust data shown on the graph, it is hard to see the details of the acoustic response for these runs. To provide better resolution, Figure 4-29 is provided which zooms in on each of the five powered runs for Set 1.

Figure 4-28: 8-Second Dynamic Runs - Set 1
Although the data is more erratic than it was when the kayak was tied to the side of the tow tank, the sector means are still increasing as expected. It is noted that there is a more significant delay in the acoustic response, which is more in line with the delays seen in Section 4.2.1. This suggests that increasing the current delay from 0.6 seconds to 1.2 seconds may provide better data for dynamic testing.

**Acceleration Measurements**

Although including acceleration in the ship-propulsion state was beyond the scope of the this project, showing how it would be done was indeed a project goal and part of the proof-of-concept validation. This section shows how acceleration can be measured and subsequently integrated into a more advanced version of the prototype.
CEC. With all the modifications and corrections incorporated into the Specialized IMU MOOS application (as described in Section 3.3.3) identifying and isolating states of transition between speeds is fairly straightforward using the changes in the axial acceleration of the kayak. Now including acceleration values (indicated in black), zoomed-in transitions are presented as Figures 4-30, 4-31, and 4-32.

![Figure 4-30: 0 to 50%](image)

![Figure 4-31: 0 to 50% to 60%](image)
All three figures show a clear action-reaction in the acceleration where it first goes negative, and then positive when the kayak propeller "digs in". All three figures also show a clear action-reaction in the acceleration where it first goes positive and then negative when the kayak suddenly stops providing thrust. These "pulses" are clear markers that could be used by a computer program to indicate when a speed transition has occurred. In all examples, as a steady speed is achieved, the amplitude of the acceleration "settles" to a nearly flat line, indicating when a speed transition has ended and the new desired speed has been achieved. Note that Figure 4-31 has a second speed increase in the middle, which is why this figure does not show the acceleration settling between the start and stop pulses as it did for the other figures. Again, this is a clear indication that a computer algorithm can use as a clear marker to determine when a speed transition has ended. These transitions could then be included as an additional dimension of the ship propulsion state and therefore have unique acoustic responses associated with each speed transition.
4.3 Conclusions

The objective of this project was achieved in that a functional prototype Acoustic Response Control System (ARCS) was built and tested. Using the onboard computer, the ARCS on this autonomous kayak, was able to successfully map the amount of noise being generated by the kayak to what the propulsion system was doing at the time. The ARCS was able to then use that same data to select the optimal propulsion thrust percentage that would be just below the user-defined response threshold. When the user changes the threshold while the kayak is operating, the ARCS responds with a new optimal thrust without the need to recalibrate. Although speed transitions and conditions of forced propeller slip were not included in this prototype, the sensor configurations and methods for doing so were developed and shown to be feasible. Although further work can be done to improve several aspects of the system, including the ability to handle complicated speed transitions, error-rejection, and more efficient machine-learning and calibration algorithms, the prototype was nonetheless successful. This research shows that with modern processors able to quickly gather and assess information, that a Cause-Effect Controller (CEC) using common adaptive software is feasible for use in similar engineering application. A CEC is best used in applications where it is difficult to predict outcomes using analytical methods, and experimentation is impractical due to the large number of experiments that would be needed to account for slight variations between applications.

4.4 Lessons Learned

During the experiments, visual propeller cavitation was not observed. It was also noted that momentum effects such as forced propeller slip and side slip during maneuvers were slight. Graduating to a larger vehicle would address both of these issues. In which to capture acoustic cavitation effects, which may have been present but not measured, an amplifier that includes frequencies up to 26 kHz is needed.
4.5 Future Work and Improvements

**Machine-Learning Algorithms:** This prototype used a simple calibration curve. Improved experience-based methods of machine learning, such as neural nets, could be used instead. Additionally, better error-rejection and response-filtering methods, such as an informed Kalman Filters could be incorporated.

**Improved Calibration Methods:** When the decision is made for the ship to conduct dedicated maneuvers to calibrate the ARCS as quickly as possible, better options could be developed than the step tests used for this prototype. Using statistical methods in which multiple parameters can be correlated without having to do all possible permutations could save a lot of time and fuel.

**Larger-Scale Testing:** In which to further validate the use of ARCS on large ships, a medium-sized ship implementation could be done. Using an autopilot, the required consistency of not having a human in the loop should be achievable.

**Advanced ARCS:** There are a lot of improvements that can be made on this current prototype with respect to the ship-propulsion state to include rudder effects, speed transitions, roll, and the effects of sea-state and other environmental factors such as water temperature. There could also be a lot of improvements on the *acoustic response* to include differentiation based on frequency.

**Tactical Maneuvering:** Although more relevant to military applications, the ability to find the optimal use of propulsion and control surfaces (within the response threshold) could be paired with an algorithm that also chooses an optimal tactical maneuver based on a threat's sensors. The optimal maneuver could be automated and even combined with the use of automated countermeasures.

**Applications Outside of Ocean Engineering:** Although not researched as part of this project, there are likely multiple applications outside of ocean engineering that might benefit from using a Cause-Effect Controller as proposed herein.
Appendix A

Hardware Specifications & Data Sheets

A.1 Data Sheets and Technical Information for the Following Hardware - Listed In Order:

- Electronics Enclosure Wiring Diagrams - Hovergroup Wiki [12]
- Gumstix DuoVero Crystal COM Model: GUM4430C
- Gumstix Parlor Breakout Board Model: B40002
- Ocean Server Compass Model: OS5000
- U-blox Neo-6 GPS Module Model: NEO-6T
- SparkFun 9 Degrees of Freedom - Razor IMU - Model: SEN-10736
- SparkFun SparkFun FTDI Basic Breakout - 3.3V Model: DEV-09873
- Sabrent USB External Stereo 3D Sound Adapter - Model: AU-MMSA
- Aquarian Audio Products H2a-XLR Hydrophone
- Sarmonic SmartRig Audio Adapter
Motor Driver Box:
- LED Control:
  - +5V taken from LED driver
  - DOUT1 - pin1 on motor driver DB15 connector
- Motor Driver:
  - Roboteq I/O connector:
    - 1 - DOUT1 - see led control above
    - 2 - TX - green - top left on cpu connector
    - 3 - RX - blue - top right on cpu connector
    - 4 - DIN1 - e-stop - top right on e-stop connector
    - 13 - GND - power switch led +
    - 14 - 5V - power switch led + and e-stop top left
- CPU Box Connector:
  - Top left - green - roboteq tx - roboteq I/O pin 2
  - Top right - blue - roboteq rx - roboteq I/O pin 3
  - Bottom left - power ground
  - Bottom right - power vbatt

Power Switch Wiring:
- C1 - power control
- NO - vbatt
- NC - ground
- LED+ - roboteq I/O pin 14 (+5V)
- LED- - roboteq I/O pin 13 (gnd)

LED Control:
- External connector shared with e-stop:
  - Bottom right - LED +
  - Bottom left - LED -
Modifications and Bodes:

Power Board:
- Vbat control (unlabeled between 12v and 6v) must be wired to pin 1 of its optorelay (immediately below the left main power connector)
- Radio power control must be manually wired to a 2-pin KK connector (along with 5V) in the proto area
- Optorelay for 6v control must be doubling, and its associated resistor (vertical and to the left) halved to ~300 ohms

Arduino Shield:
- Radio power control and VBat control must be manually wired (see tables above)
- 12v control line transistor can be bypassed
- Motor driver is wired to serial uart 2
- Gumstix is wired to serial uart 1
- Tmp102 temp sensor resides on i2c bus
- Sbus wired to uart3
- Wiznet wired to default uart (shared with USB)

Self Power Reset

On Keestrel, this has been modified as follows to prevent the positive input from hitting the positive rail.

3.3V power for radio box
External Connectors on cpu box

FTID USB-COM232-PLUS4

- 2 - rxd
- 3 - txd
- 5 - gnd

Modem port A
- db9 pin 2 <-> top right
- db9 pin 3 <-> bottom middle
- db9 pin 5 <-> bottom left

Freewave port B
- db9 pin 2 <-> solid blue
- db9 pin 3 <-> striped blue

External/altimeter port C
- db9 pin 2 <-> bottom left (viewed from outside)
- db9 pin 3 <-> bottom right

Additional Information:

Tritech Depth Sounder:
- Draws approximately 110mA at 12v
- 3 pin impulse connector with pin 1 higher than the others and closer to pin 2 than pin 3
  - 1 - gnd
  - 2 - signal (rs232, 9600 baud)
  - 3 - power (v_batt ok, 10.5-20v)

Cruzpro Depth Sounders:
- red - power
- shield - gnd
- green - signal
- white - unused

Duovero Header:
- USB host is the port located farther from ethernet port
- 40 pin header information:
  - USB host: http://pubs.gumstix.com/boards/P4FLORPCB40002-F39002/44002.pdf
    - We use gnd, vcc_1.8, v_batt_5, and uart2 tx/rx

Radio Box

RJ45 reference:
In 100BaseT link only orange and green pairs are used, blue and brown are spare.

External Connector
See cpu section above

Wiznet (WIZ108R)
Make sure serial debug mode is turned off in configuration tool

Freewave:
RS485 pinouts:
D+ short pins 5 and 6
D- short pins 7 and 8
gnd on pin 4
RS232 pinouts:
pin 4 gnd
pin 5 RX
pin 6 TX
COM1 used for RS485
COM2 used for RS232
pin 5 - rx - striped blue
pin 6 - tx - solid blue

Moxa rs232/rs485 converter (TCC-80)
db9 rs232 side uses
tx - 2 - brown
rx - 3 - striped brown
rs485 side must have ground connected
power plug must have ground and +5v
moxa dip switches: 1 and 2 on (rs485 two wire), 3 off (no termination)

Bullet PoE
POE on spares uses
blue and striped blue for DC +
brown and striped brown for DC -

Overall:
Double up on processing cores while keeping the same tiny footprint. The DuoVero Crystal offers a dual-core ARM Cortex-A9 processor and twice the RAM of its predecessors, alongside SGX540 graphics for stunning 1080p HD video output and SLR-like image processing up to 20 megapixels.

**Mating connector**
- 2 x 70-Pin DF40 Connectors (Male DuoVero COMs)

**What’s included**
- 5 x Spacers (1.5mm 0-80 retaining 48/48)
- 1 x DuoVero Crystal COM (board)
- 1 x DuoVero Yocto System Card (8GB)

**Key component**
- 1 x Texas Instruments TWL6040 Audio Codec
- 1 x Texas Instruments OMAP 4430 Processor

**Product links**
- Technical Specifications
- Programming Reference
- Geppetto@Workspace
Introduction to the Tiny Compass Family

The OceanServer Precision 3-Axis Tilt Compensated Compass products use state of the art technology to provide outstanding performance and ease of use in a low cost design.

The OS5000 family of compasses are a new class of sensor components providing best in class performance for under $250.00 (USD) in volume.

Features:
- Compass accuracy, 0.5 degrees RMS heading while level, 1° RMS <±30° Tilt, 1.5° RMS <±60° Tilt, undisturbed field, .1 Degree resolution
- Roll & Pitch full rotation, typical 1° accuracy <±30° tilt
- Pitch Angles +/-90 degrees, Roll Angles +/-180 degrees
- Tilt-compensated (electronically gimbaled)
- Tiny size, 1"x1"x0.3", less than 2 grams weight
- Low Power Consumption, <30ma @3.3V
- Hard and soft-iron compensation routines
- Optional support for a high resolution Depth or Altitude sensor (24 bit AD)
- Serial Interface:
  - RS232 or USB (or TTL see OS4000-T)
  - Baud rate programmable 4,800 to 115,000 baud
- Rugged design
  - 10,000 G shock survival
  - -40°C to 80°C operating temperature (Accuracy specified for 0°C to 50°C)
- ASCII sentence output, in several formats, NMEA checksum
- High Data Update Rate to 40HZ
- Support for True or Magnetic North Output
- Precision components
  - 3 Axis magnetic sensors from Honeywell
  - 3 Axis Accelerometers from ST Microelectronics
  - 24 bit differential Analog to Digital converters from Analog Devices
OCEANSERVER COMPASS MANUAL

- 50 MIPS processor supporting IEEE floating point math

OS5000-S — Signals: RS-232 Levels: 3-Axis Compass, 1" x 1" size with full RS-232 support, single connector on the topside (note arrow imprinted on this side) of the module.

OS5000-USD — Signals USB or RS-232: 3-Axis tilt compensated compass, 1" x 1". Has both Serial & USB direct interface and has connectors on both sides of the module.

Always mount compass with 7 pin side 1 up even when using USB connector other mounting positions available with firmware version 1.5.

Additional Note: the OS5000-USD version supports pressure measurement for depth (Pictures may not reflect latest release).

OS Compass Design Features

Hardware Variants:
- OS5000-S (RS232 Only), OS5000-USD (RS232 & USB) note OS5000-USD (Standard Features includes Depth Option)

Magnetic Sensors:
1 Functional description

1.1 Overview

The NEO-6 module series is a family of stand-alone GPS receivers featuring the high performance u-blox 6 positioning engine. These flexible and cost effective receivers offer numerous connectivity options in a miniature 16 x 12.2 x 2.4 mm package. Their compact architecture and power and memory options make NEO-6 modules ideal for battery operated mobile devices with very strict cost and space constraints.

The 50-channel u-blox 6 positioning engine boasts a Time-To-First-Fix (TTFF) of under 1 second. The dedicated acquisition engine, with 2 million correlators, is capable of massive parallel time/frequency space searches, enabling it to find satellites instantly. Innovative design and technology suppresses jamming sources and mitigates multipath effects, giving NEO-6 GPS receivers excellent navigation performance even in the most challenging environments.

1.2 Product features

<table>
<thead>
<tr>
<th>Model</th>
<th>Type</th>
<th>Supply</th>
<th>Interfaces</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEO-6G</td>
<td>GPS</td>
<td>HW</td>
<td>Power</td>
<td>UART</td>
</tr>
<tr>
<td>NEO-6Q</td>
<td>GPS</td>
<td>PP</td>
<td>Power</td>
<td>UART</td>
</tr>
<tr>
<td>NEO-6M</td>
<td>GPS</td>
<td></td>
<td>Power</td>
<td>UART</td>
</tr>
<tr>
<td>NEO-6P</td>
<td>GPS</td>
<td></td>
<td>Power</td>
<td>UART</td>
</tr>
<tr>
<td>NEO-4V</td>
<td>GPS</td>
<td></td>
<td>Power</td>
<td>UART</td>
</tr>
<tr>
<td>NEO-4T</td>
<td>GPS</td>
<td></td>
<td>Power</td>
<td>UART</td>
</tr>
</tbody>
</table>

O = requires external components and integration on application processor

Table 1: Features of the NEO-6 Series

All NEO-6 modules are based on GPS chips qualified according to AEC-Q100. See Chapter 5.1 for further information.
### 1.3 GPS performance

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver type</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td>Time-To-First-Fix</td>
<td></td>
</tr>
<tr>
<td>Cold Start</td>
<td>26 s</td>
</tr>
<tr>
<td>Warm Start</td>
<td>28 s</td>
</tr>
<tr>
<td>Hot Start</td>
<td>1 s</td>
</tr>
<tr>
<td>Aided Start</td>
<td>&lt;3 s</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td>Tracking &amp; Navigation</td>
<td>-162 dBm</td>
</tr>
<tr>
<td>Reacquisition^</td>
<td>-160 dBm</td>
</tr>
<tr>
<td>Cold Start (without aiding)</td>
<td>-148 dBm</td>
</tr>
<tr>
<td>Hot Start</td>
<td>-157 dBm</td>
</tr>
<tr>
<td>Maximum Navigation update rate</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td></td>
<td>5 Hz</td>
</tr>
<tr>
<td>Horizontal position accuracy</td>
<td>GPS</td>
</tr>
<tr>
<td>SBAS</td>
<td>2.0 m</td>
</tr>
<tr>
<td>SBAS + PPP^</td>
<td>&lt; 1 m (2D, 50%)</td>
</tr>
<tr>
<td>SBAS + PPP^</td>
<td>&lt; 2 m (3D, 50%)</td>
</tr>
<tr>
<td>Configurable Timepulse frequency range</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td></td>
<td>0.25 Hz to 1 kHz</td>
</tr>
<tr>
<td>Accuracy for Timepulse signal</td>
<td>RMS</td>
</tr>
<tr>
<td>99%</td>
<td>&lt;60 ns</td>
</tr>
<tr>
<td>Granularity</td>
<td>21 ns</td>
</tr>
<tr>
<td>Compensated^</td>
<td>15 ns</td>
</tr>
<tr>
<td>Velocity accuracy</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td></td>
<td>0.1 m/s</td>
</tr>
<tr>
<td>Heading accuracy</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td></td>
<td>0.5 degrees</td>
</tr>
<tr>
<td>Operational Limits</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td>Dynamics</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td>Altitude</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td>Velocity</td>
<td>NEO-6G/QT</td>
</tr>
<tr>
<td>CEP, 50%, 48 hours static, -130 dBm, SEP: &lt;3.5 m</td>
<td></td>
</tr>
<tr>
<td>Demonstrated under following conditions: 24 hours, stationary, first 600 seconds of data discarded. HDOP &lt; 1.5 during measurement period, strong signals. Continuous availability of valid SBAS correction data during full test period.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: NEO-6 GPS performance

---

1. All satellites at -130 dBm
2. Without aiding
3. Dependent on aiding data connection speed and latency
4. Demonstrated with a good active antenna
5. For an outage duration ≤10 s
6. CEP, 50%, 48 hours static, -130 dBm, SEP: <3.5 m
7. DEMO-6P only
8. Demonstrated under following conditions: 24 hours, stationary, first 600 seconds of data discarded. HDOP < 1.5 during measurement period, strong signals. Continuous availability of valid SBAS correction data during full test period.
9. Quantization error information can be used with NEO-6T to compensate the granularity related error of the timepulse signal
10. Assuming airborne <4g platform
INTERFACE

POWERSHIFT Basic

AUR

TITLE: 9DOF-Razor-v22
Document Number: REU4
Date: 6/1/2011 4:05:35 PM Sheet: 2/2

Released under the Creative Commons Attribution Share-Alike 3.0 License
http://creativecommons.org/licenses/by-sa/3.0
Design by: AUp
<table>
<thead>
<tr>
<th><strong>Product Info.</strong></th>
<th><strong>Value</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>0.8 oz</td>
</tr>
<tr>
<td>Dimensions</td>
<td>4.20 x 2.75 x 0.60 in</td>
</tr>
<tr>
<td>Model</td>
<td>AU-MMSA</td>
</tr>
<tr>
<td>UPC</td>
<td>819921011138</td>
</tr>
<tr>
<td>Color</td>
<td>Black</td>
</tr>
<tr>
<td>Material</td>
<td>Plastic</td>
</tr>
<tr>
<td>Number of Audio Channels</td>
<td>2.1</td>
</tr>
<tr>
<td>Total Microphone Ports</td>
<td>1</td>
</tr>
<tr>
<td>Total Auxiliary Ports</td>
<td>1</td>
</tr>
<tr>
<td>Compatibility</td>
<td>Windows Compatible, Mac OS X Compatible, USB 2.0 Compatible</td>
</tr>
<tr>
<td>Operating System Supported</td>
<td>Linux, Mac, Windows</td>
</tr>
</tbody>
</table>
Aquarian Audio Products

H2a-XLR Hydrophone User's Guide

Thank you for purchasing your Aquarian Audio Products H2a-XLR hydrophone. This hydrophone is designed to provide high-quality audio performance in a low-cost device. It is very durable and will interface directly with professional audio microphone preamps. It offers very good sensitivity and low noise in the human auditory range. The H2a-XLR's streamlined shape and high specific gravity will help to maintain a low working depth in a moving water column. Its compact size and the easy hand of its cable make it very portable and simple to use.

Using the H2a-XLR

The H2a-XLR is terminated with a 3-pin male XLR plug. Wiring is standard: pin 1 is ground, pin 2 is hot and pin 3 unused. This configuration should be compatible with any standard female XLR microphone jack. Phantom power is required. Any standard phantom power supply voltage will work. Do not exceed +48V when powering the H2a-XLR. With the connection firmly made and phantom power switched on, there's nothing more to do but adjust levels and take in the sounds of the deep.

Hydrophone care

No special care is required for the H2a-XLR. It is designed to withstand corrosion and the impact of accidental drops, but making an attempt to keep the output plug clean and dry and avoiding unnecessarily rough handling will help to ensure the long-term stability of the product. It is best not to store the hydrophone in a waterproof enclosure. Doing so will trap moisture, salts and minerals that are left on the hydrophone and cable after deployment and increase corrosion problems with the output plug. Making an extra effort to coil the cable nicely when retrieving the hydrophone will help avoid problems with tangles as the cable ages. Most importantly, protect the cable from cuts and abrasions! The H2a-XLR uses a custom-made cable with a very durable urethane jacket. However, it is also designed to be compact and flexible. Kinking the cable, walking on it, or dragging it over a sharp or abrasive surface may damage the cable sheath and eventually cause the hydrophone to fail. Both aquatic and terrestrial animals may attack the cable in an unattended application. Using some kind of cable shroud, such as plastic tubing, can help protect the hydrophone in long-term installations.

1 Most XLR-terminated microphones are balanced and use Pin 3 for a cold (or low) side of the signal, which has the same output impedance and is typically complimentary to the signal on Pin 2. This configuration can be beneficial for common-mode noise reduction when used with a balanced input device. The H2a-XLR runs single-ended. It is a simple output plug variation of our standard H2a. Single-ended operation is preferable in most hydrophone applications because common-mode noise is seldom a problem and the hydrophone consumes less power and has a quieter output with the single-ended drive. The H2a-XLR is very well shielded. The single-ended drive should not be an issue even in very noisy environments and should work well with all phantom powered balanced mic inputs.
Specifications

The H2a-XLR is intended to be a lower-cost and easy-to-use alternative to military and lab-grade hydrophones. Deriving high sensitivity and low noise were made a priority over maintaining strict tolerances. The following specifications are typical of a limited sample group and are not guaranteed. They are for basic comparison information only.

Specifications are dependent upon the audio device to which the H2a-XLR is connected. The hydrophone sensor is capable of picking up sounds from below 20Hz to above 100KHz. The output impedance of the H2a-XLR is set in part by the phantom power supply from the audio device with which the hydrophone is used. Hi-frequency performance is also limited by the output impedance of the hydrophone and the cable impedance—which is a function of length. Please also note that further limitations in your overall system may result from the sampling rate of digital recorders and by the input stage of your audio device’s microphone preamp. Despite the uncertainties of above, you should expect to easily capture the entire human auditory range of 20Hz to 20KHz.

The following specifications are based upon using the H2a-XLR with a classic 48V phantom power supply (48V with 6.8K pull-up resistors):

- **Sensitivity:** -180dB re: 1V/µPa
- **Useful range:** <10 Hz to >100KHz
- **Polar Response:** Omnidirectional
- **Operating depth:** <80 meters
- **Output impedance:** 1 KΩ
- **Power:** 0.6 mA

**Physical:**
- **Dimensions:** 25mm x 46mm
- **Mass:** 105 grams
- **Specific Gravity:** 5.3

Warranty Statement

*Aquarian Audio Products* warrants the H2a-XLR Hydrophone from electrical failure or defects in workmanship for a period of one year following the date of purchase. Warranty claims or repairs can be made directly through *Aquarian Audio Products*. Corroded output plugs are not covered under warranty. Additional terms and conditions apply. For full details please view our general warranty statement at [http://www.aquarianaudio.com/warranty.html](http://www.aquarianaudio.com/warranty.html)
Audio Adapter

The SmartRig from Saramonic is an easy solution to connect any professional microphones to iPhone, iPod touch, iPad and Android devices.

User Manual

Specifications

<table>
<thead>
<tr>
<th>Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency Response</td>
<td>20Hz to 20kHz (+/-1.5dB)</td>
</tr>
<tr>
<td>Noise</td>
<td>-98 dB full band, phantom power ON</td>
</tr>
<tr>
<td>Maximum output level</td>
<td>2 Vrms</td>
</tr>
<tr>
<td>Distortion</td>
<td>0.025% THD</td>
</tr>
<tr>
<td>Phantom max current</td>
<td>13mA</td>
</tr>
<tr>
<td>Dimensions</td>
<td>L 86 x W 41 x H 42 mm</td>
</tr>
<tr>
<td>Weight</td>
<td>75g (2.57 oz)</td>
</tr>
</tbody>
</table>

Product Structure

PRODUCT INTRODUCTION

General Introductions

The SmartRig from Saramonic is an easy solution to connect any professional microphones to iPhone, iPod touch, iPad and Android devices.

As a high-quality microphone preamp, SmartRig allows you to use professional stage microphones or high-end studio microphones to create music with your iOS or Android devices.

Simply plug the microphone into the standard XLR connector of SmartRig and connect it to your device with the 3.5mm output cable.

The provided gain control thumbwheel makes it easy to set precise levels. Processed sound can be monitored through a 3.5mm headphone out.

Highlights

- XLR microphone input connector
- +48V phantom power
- 3.5mm headphone output for monitoring
- +48V/OFF/ON switch
- Gain control
- Power/Phantom power provided by 9V battery
- Includes free app
THIS PAGE INTENTIONALLY LEFT BLANK
Appendix B

Software - Code

B.1 MOOS Mission Configuration File

This the configuration file for MOOS, also known as a mission file that was used onboard the kayak. This code show what parameters were used to initialize the different MOOS Applications that were called.

```plaintext
ServerHost = 192.168.1.105
ServerPort = 9500
Community = KESTREL

// MIT Sailing Pavilion
LatOrigin = 42.358456
LongOrigin = -71.087589

Processconfig = ANTLER
{
    MSBetweenLaunches = 200

    Run = MOOSDB
    Run = pReadSelfNoise
        be run 2nd
    Run = pLogger
    Run = iSpecIMU
    Run = pCEPID
    Run = uTimerScript
    Run = iHoverKayak
    Run = iGPS_Hover
`
Run = pMarinePID_Hover @ NewConsole = false
Run = pHelmIvP @ NewConsole = false

// Run = pShare @ NewConsole = false
// Run = pProtoReporter @ NewConsole = false
// Run = iRTKNAVI @ NewConsole = false
// Run = iOS5000_Hover @ NewConsole = false
// Run = pEchoVar @ NewConsole = false
// Run = iAcommsDriver @ NewConsole = false
// Run = pNavManager @ NewConsole = false
// Run = pResourceMon @ NewConsole = false
// Run = pAckedCommsVehicle @ NewConsole = false
// Run = pScheduledTransmit @ NewConsole = false
}

ProcessConfig = uTimerScript
{
  AppTick = 2
  CommsTick = 2

  paused = false
  reset_time = all-posted
  reset_max = unlimited

  event = var=SN_TAKE_READINGS, val="true", time=1

  // allow time for calibration of IMU approx 12 secs
  event = var=DESIRED_THRUST, val=0, time=4
  event = var=DESIRED_THRUST, val=0, time=8
  // end calibration

  // signal ready for testing
  event = var=DESIRED_THRUST, val=10, time=12
  event = var=DESIRED_THRUST, val=-10, time=14
  event = var=DESIRED_THRUST, val=0, time=16

  // Position for 1st test – wait 92 secs
  // 0 -> 50
  event = var=DESIRED_RUDDER, val=2, time=98
  event = var=DESIRED_THRUST, val=50, time=100
  event = var=DESIRED_RUDDER, val=2, time=102
  event = var=DESIRED_THRUST, val=50, time=104
  event = var=DESIRED_THRUST, val=0, time=108

  // Position for 2nd test – wait 84 secs

// 0 -> 60
event = var=DESIRED_RUDDER, val=2, time=198
event = var=DESIRED_THRUST, val=60, time=200
event = var=DESIRED_RUDDER, val=2, time=202
event = var=DESIRED_THRUST, val=60, time=204
event = var=DESIRED_RUDDER, val=2, time=206
event = var=DESIRED_THRUST, val=0, time=208

// Position for 3rd test - wait 84 secs
// 0 -> 50 -> 60
event = var=DESIRED_RUDDER, val=2, time=298
event = var=DESIRED_THRUST, val=50, time=300
event = var=DESIRED_RUDDER, val=2, time=302
event = var=DESIRED_THRUST, val=60, time=304
event = var=DESIRED_RUDDER, val=2, time=306
event = var=DESIRED_THRUST, val=0, time=308

// Position for 4th test - wait 92 secs
// 0 -> 70
event = var=DESIRED_RUDDER, val=2, time=398
event = var=DESIRED_THRUST, val=70, time=400
event = var=DESIRED_RUDDER, val=2, time=402
event = var=DESIRED_THRUST, val=70, time=404
event = var=DESIRED_THRUST, val=0, time=408

// Position for 5th test - wait 92 secs
// 0 -> 80
event = var=DESIRED_RUDDER, val=2, time=498
event = var=DESIRED_THRUST, val=80, time=500
event = var=DESIRED_RUDDER, val=2, time=502
event = var=DESIRED_THRUST, val=80, time=504
event = var=DESIRED_THRUST, val=0, time=508

// Position for 6th test - wait 92 secs
// 0 -> 70 -> 80
event = var=DESIRED_RUDDER, val=2, time=598
event = var=DESIRED_THRUST, val=70, time=600
event = var=DESIRED_RUDDER, val=2, time=602
event = var=DESIRED_THRUST, val=80, time=604
event = var=DESIRED_THRUST, val=0, time=608

}

ProcessConfig = iSpecIMU
ProcessConfig = pReadSelfNoise
{
    AppTick = 10
    CommsTick = 10
    PORT = /dev/ttyUSB0
    BAUD_RATE = 57600
    BUFFER_SIZE = 1024
    CREATE_LOG = true
    USE_PLOGGER_DIR = true
    LOG_PATH = /home/mit/hoverland/moos-ivp-ben/logs/
    IMU_MODE = compass_IMU
    PRE_ROTATION = -90.0
    SENSOR_AVG_TIME = .38
    SENSOR_CAL_TIME = 10
    CAL_ON_STARTUP = false
    CORR_FOR_GRAVITY = true
}

ProcessConfig = pCEPID
{
    AppTick = 10
    CommsTick = 10
    USE_CEPID = true
    THRESHOLD = 0.1
    MARGIN_FACTOR = 1.0
    USE_EXISTING_CAL = true
    NOISE_MONITORING = true
    EXISTING_CAL_FILE = /home/mit/hoverland/moos-ivp-ben/calibration/kestrel_cal.txt
CALIB_PATH = /home/mit/hoverland/moos-ivp-ben/calibration/
NUM_CAL_RUNS = 5
USE_SIM_DATA = true
SIM_DATA_FILE = /home/mit/hoverland/moos-ivp-ben/data/
Med_Steps_Run_1.slog
RESPONSE_TIME_LAG = 0.6
TIME_BETWEEN_RUNS = 3.0 // should be around 10

ProcessConfig = pLogger
{
  AppTick = 10
  CommsTick = 10
  File = SN_LOG_KESTREL
  PATH = /home/mit/hoverland/moos-ivp-ben/logs/
  SyncLog = true @ 0.2
  AsyncLog = true
  WildCardLogging = false
  FileTimeStamp = true
  Log = SN_READ_ITERATIONS @ 0.2
  Log = SN_MAX_AMPLITUDE @ 0.2
  Log = SN_RMS_AMPLITUDE @ 0.2
  Log = X_ACCEL @ 0.2
  Log = DESIRED_THRUST @ 0.2
  Log = DESIRED_RUDDER @ 0.2
  //WildCardPatter = READ_SELF_NOISE, SN_PEAK
  //WildCardOmitPattern = PLOGGER_STATUS, DB_VARSUMMARY
}

ProcessConfig = iKST
{
  IterateMode = 0 // regular iterate and mail
  AppTick = 4
  output_path = "/home/mit/hoverland/moos-ivp-ben/logs/
           kst_KESTREL.csv"
  LOG = NAV_X
  LOG = NAV_Y
  LOG = NAV_HEADING
  LOG = NAV_SPEED
LOG = DESIRED_HEADING
LOG = DESIRED_SPEED
LOG = ARDUINO_RUDDER
LOG = ARDUINO_THRUST
LOG = COMPASS_ROLL
LOG = COMPASS_PITCH
LOG = ALTIMETER_DEPTH

ProcessConfig = pShare
{
    IterateMode = 0    // regular iterate and mail
    AppTick = 4

    output = PROTO_REPORT_LOCAL->PROTO_REPORT
                 :192.168.1.100:9001
    output = VIEW_RANGE_PULSE->192.168.1.100:9001
    output = ACKEDCOMMS_RETURN_ACK->192.168.1.100:9001

    input = route=192.168.1.105:9501
}

ProcessConfig = uSimMarine
{
    AppTick = 10

    // START_POS = 20, -15, 180, 0
    start_x = 0
    start_y = -20
    start_heading = 180
    start_speed = 0

    PREFIX = NAV
}

ProcessConfig = pProtoReporter
{
    IterateMode = 0    // regular iterate and mail
    AppTick = 1

    PLATFORM_TYPE = KAYAK
}

ProcessConfig = pMarinePID_Hover
{  
  AppTick = 10  

  Verbose = false  
  DEPTH_CONTROL = false  
  ACTIVE_START = true  
  MAXRUDDER = 45  
  MAXTHRUST = 100  

  // SPEED_FACTOR  
  // - A non-zero SPEED_FACTOR overrides use of SPEED_PID  
  // - Will set DESIRED_THRUST = DESIRED_SPEED * SPEED_FACTOR  
  SPEED_FACTOR = 30  

  // 25-30 for no modem  
  // 40 with modem  

  // Yaw PID controller  
  YAW_PID_KP = 0.45  
  YAW_PID_KD = 0.3  
  YAW_PID_KI = 0.01  
  YAW_PID_INTEGRAL_LIMIT = 500  
  YAW_PID_TAU = 1  

  // Speed PID controller  
  SPEED_PID_KP = 0.0  
  SPEED_PID_KD = 0.0  
  SPEED_PID_KI = 4  
  SPEED_PID_INTEGRAL_LIMIT = 2.5  

  SPEED_CONTROLLER = fit_pid  
  // factor, pid, fit_pid  

  SPEED_SLOPE = 60.6  
  SPEED_OFFSET = -1.21  
  ANGLE_LIMIT = 25  
  TIME_DELAY = 5  
}

ProcessConfig = pHelmIvP  
{
  AppTick = 2  

  ok_skew = any

165
Behaviors = KESTREL.bhv

Verbose = true
Domain = course:0:359:360
Domain = speed:0:3:61

IVP_BEHAVIOR_DIR = /home/mit/hovergroup/ivp-extend/trunk/lib/
IVP_BEHAVIOR_DIR = /home/mit/hovergroup-extend/trunk/lib/
BHV_DIR_NOT_FOUND_OK = true

ProcessConfig = pEchoVar
{
  IterateMode = 1 // comms driven iterate and mail
  AppTick = 4 // iterate lower bound
  MaxAppTick = 0 // no limit
  Echo = MOOS_MANUAL_OVERRIDE -> MOOS_MANUAL_OVERRIDE
}

ProcessConfig = iOS5000_Hover
{
  IterateMode = 0 // regular iterate and mail
  AppTick = 1

  Port = /dev/ttyUSB5
  Speed = 57600
  PreRotation = -15 // -15 for Boston (magnetic declination)

  FilterTimeConstant = 1
}

ProcessConfig = iRTKNAVI
{
  IterateMode = 0 // regular iterate and mail
  AppTick = 1

  HOST=192.168.1.105
  PORT=50001
}

ProcessConfig = iRemote
325 { 
326   IterateMode = 0  // regular iterate and mail
327   AppTick = 5
328 
329   CustomKey = 1 : HELM_VERBOSE @ "verbose"
330   CustomKey = 2 : HELM_VERBOSE @ "terse"
331   CustomKey = 3 : HELM_VERBOSE @ "quiet"
332   CustomKey = 4 : DEPLOY @ "true"
333   CustomKey = 5 : DEPLOY @ "false"
334   CustomKey = 6 : RETURN @ "true"
335   CustomKey = 7 : RETURN @ "false"
336 }
337
338 ProcessConfig = iHoverKayak
339 { 
340   IterateMode = 1  // comms driven iterate and mail
341   AppTick = 20  // iterate lower bound
342   MaxAppTick = 0  // no limit
343 
344   INVERT_RUDDER = false
345   RUDDER_OFFSET = 0
346   PORT_NAME = /dev/ttyO1
347 }
348
349 ProcessConfig = iAcommsDriver
350 { 
351   IterateMode = 0  // regular iterate and mail
352   AppTick = 5
353 
354   PortName = /dev/ttyUSB1
355   ID = 5
356 
357   PSK_minipackets = false
358   enable_ranging = true
359   show_range_pulses = true
360 
361 }
362
363 ProcessConfig = iAltimeter
364 { 
365   IterateMode = 0  // regular iterate and mail
366   AppTick = 1
367 
368   PORT_NAME = /dev/ttyUSB3
369 }
ProcessConfig = iAltimeter_cruzPro
{
    IterateMode = 0  // regular iterate and mail
    AppTick = 1
    PORT_NAME = /dev/ttyUSB3
    BAUD_RATE = 4800
}

ProcessConfig = pNavManager
{
    IterateMode = 1  // comms driven iterate and mail
    AppTick = 4  // iterate lower bound
    MaxAppTick = 0  // no limit
    timeout = 5  // timeout before abandoning a nav
    source1 = rtk_fix
    source2 = rtk_float
    source3 = gps
    source4 = rtk_single
}

ProcessConfig = iGPS_Hover
{
    IterateMode = 0  // regular iterate and mail
    AppTick = 20
    //PORT_NAME = $(GPSPORT)
    //BAUD_RATE = $(GPSBAUD)
    TCP_SERVER = localhost
    TCP_PORT = 50000
    USE_TCP = true
}

ProcessConfig = pResourceMon
{
    IterateMode = 0  // regular iterate and mail
    AppTick = 1
}
IterateMode = 1  // comms driven iterate and mail
AppTick = 1  // iterate lower bound
MaxAppTick = 0  // no limit

IterateMode = 0  // regular iterate and mail
AppTick = 20

period = 40
offset = 30

ProcessConfig = uXMS
{  
    AppTick = 4
    CommsTick = 4

    var = DEPLOY, RETURN, MISSION_MODE
    var = SN_TAKE_READINGS, SN_READ_ITERATIONS
    var = SN_MAX_AMPLITUDE, SN_RMS_AMPLITUDE,
    SN_ROUGH_FREQUENCY
}
B.2 Customized Inertial Measurement Unit MOOS Application - C++ Code

A quick description of this application can be found in section 3.3.3

B.2.1 Custom IMU Program Header File - SpecIMU.h

```cpp
#include "MOOS/libMOOS/MOOSLib.h"
#include <boost/asio.hpp>
#include <boost/thread.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <math.h>

class SpecIMU : public CMOOSInstrument
{
public:
  SpecIMU();
  ~SpecIMU();

protected:
  bool OnNewMail(MOOSMSG_LIST &NewMail);
  bool Iterate();
  bool OnConnectToServer();
  bool OnStartUp();
  void RegisterVariables();

  // Public Methods
```
# Configuration variables

```cpp
private:
    std::string m_port_name;
    int m_baud_rate;
    int m_buffer_size;
    bool m_output_accel;
    bool m_output_compass;
    bool m_create_log;
    bool m_use_plogger_dir;
    std::string m_log_path;
    double m_pre_rotation;
    bool m_calibrating_accel;
    bool m_calibrating_compass;
    bool m_corr_for_gravity;
    double m_sensor_avg_time;
    double m_calibrate_time;
```

# State variables

```cpp
unsigned int m_iterations;
double m_timewarp;
bool m_calibration_started;
```

# Data Integration and Management

```cpp
std::string m_pre_calibration_mode;
double m_last_accel_output_time;
double m_last_compass_output_time;
int m_avg_accel_counter;
int m_avg_compass_counter;
double m_avg_heading;
double m_avg_pitch;
double m_avg_roll;
double m_avg_accel_x;
double m_avg_accel_y;
double m_avg_accel_z;
double m_pitch_corr;
double m_roll_corr;
double m_accel_x_corr;
double m_accel_y_corr;
double m_accel_z_corr;
```
// Serial Port Specific
boost::asio::io_service m_io;
boost::asio::serial_port m_port;

// Asynchronous Specific
boost::thread m_serial_thread;
bool m_stop_requested;
boost::asio::deadline_timer m_timeout;

// Async Read & Write
std::vector<unsigned char> m_read_buffer, m_write_buffer;
boost::mutex m_write_buffer_mutex;
int m_bytes_to_write;
int m_async_bytes_read;
bool m_data_available;

// Data Flow
bool m_log_initialized;
std::ofstream m_IMU_log;
std::string m_string_buffer;

// Private Methods
void writeLine(std::string sLine);
void serialLoop();
void processWriteBuffer();
void processReadBuffer();
void avgAccel(double new_x, double new_y, double new_z);
void avgCompass(double new_heading, double new_pitch, double new_roll);

// Asynchronous serial port methods
void readHandler(bool& data_available,  
                 boost::asio::deadline_timer& timeout,  
                 const boost::system::error_code& error,  
                 std::size_t bytes_transferred);
void waitCallback(boost::asio::serial_port& ser_port,  
                  const boost::system::error_code& error);
void nullHandler(const boost::system::error_code& error,  
                 std::size_t bytes_transferred);

// parsing and updating methods
void ParserAndUpdate(std::string msg);
void updateAccel(std::string accel_msg);
void updateCompass(std::string compass_msg);
};
127
128  #endif
B.2.2 Custom IMU Source Code - SpecIMU.cpp

```
/* NAME: Ben Thomson */
/* ORGN: MIT */
/* FILE: SpecIMU.cpp */
/* DATE: Spring 2017 */
/* Limited IMU file that will read from Serial Port for */
/* the sparkfun 9DOF IMU and return specialized information*/
/* *******************************************************************/

#include <iterator>
#include "MBUtils.h"
#include "SpecIMU.h"

using namespace std;
using namespace boost::asio;
using namespace boost::posix_time;

// Constructor

SpecIMU::SpecIMU() : m_port(m_io), m_timeout(m_io)
{
    m_iterations = 0;
    m_timewarp = 1;

    // Configuration variables
    m_port_name = "/dev/ttyUSB0";
    m_baud_rate = 57600;
    m_buffer_size = 1024;
    m_output_accel = true;
    m_output_compass = true;
    m_create_log = true;
    m_use_plogger_dir = false;
    m_log_path = "/home/mit/hoverland/moos-ivp-ben/logs/";
    m_pre_rotation = -90.0;
    m_calibrating_accel = true;
    m_calibrating_compass = true;
    m_corr_for_gravity = true;
    m_sensor_avg_time = 0.38;
    m_calibrate_time = 10;
    m_calibration_started = false;

    // State variables
```
m_log_initialized = false;
m_stop_requested = false;
m_bytes_to_write = 0;
m_async_bytes_read = 0;
m_data_available = false;
m_string_buffer = "";

// Private Variables
m_pre_calibration_mode = "compass IMU";
m_last_accel_output_time = MOOSTime();
m_last_compass_output_time = MOOSTime();
m_avg_accel_counter = 0;
m_avg_compass_counter = 0;
m_avg_heading = 0;
m_avg_pitch = 0;
m_avg_roll = 0;
m_avg_accel_x = 0;
m_avg_accel_y = 0;
m_avg_accel_z = 0;
m_pitch_corr = 0;
m_roll_corr = 0;
m_accel_x_corr = 0;
m_accel_y_corr = 0;
m_accel_z_corr = 0;

// Destructor
SpecIMU:: ~ SpecIMU ()
{
}

// Procedure: OnNewMail
bool SpecIMU:: OnNewMail (MOOSMSG_LIST &NewMail)
{
    MOOSMSG_LIST:: iterator p;
    for (p=NewMail.begin(); p!=NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key = msg.GetKey();
        if (key == "IMU_MODE"){
            string sval = msg.GetString();
        }
    }
}
if (toupper(sval) == "COMPASS_IMU"){
    m_pre_calibration_mode = toupper(sval);
    m_output_compass = true;
    m_output_acc = true;
}
if (toupper(sval) == "COMPASS_ONLY"){
    m_pre_calibration_mode = toupper(sval);
    m_output_compass = true;
    m_output_acc = false;
}
if (toupper(sval) == "IMU_ONLY"){
    m_pre_calibration_mode = toupper(sval);
    m_output_compass = false;
    m_output_acc = true;
}
if (toupper(sval) == "CALIBRATE"){
    if (!m_calibration_started){
        m_calibration_started = true;
        m_calibrating_acc = true;
        m_calibrating_compass = true;
        resetCalibration();
    }
}
else if (key == "LOGGER_DIRECTORY"){
    if (m_create_log && !m_log_initialized){
        m_log_path = msg.GetString();
        m_log_path += "/";
        createLogFile();
    }
}
return(true);

Procedure: OnConnectToServer

bool SpecIMU::OnConnectToServer()
{
    RegisterVariables();
    return(true);
}
Procedure: Iterate()

happens AppTick times per second

bool SpecIMU::Iterate()
// Seperate thread interates though serialLoop. No intrate needed
{
    return(true);
}

Procedure: OnStartUp()

happens before connection is open

bool SpecIMU::OnStartUp()
{
    list<string> sParams;
    m_MissionReader.EnableVerbatimQuoting(false);
    if (m_MissionReader.GetConfiguration(GetAppName(), sParams))
    {
        list<string>::iterator p;
        for (p=sParams.begin(); p!=sParams.end(); p++) {
            string original_line = *p;
            string param = stripBlankEnds(toupper(biteString(*p, '=')));
            string value = stripBlankEnds(*p);
            if (param == "PORT")
                m_port_name = value;
            else if (param == "BAUD_RATE")
                m_baud_rate = atoi(value.c_str());
            else if (param == "BUFFER_SIZE")
                m_buffer_size = atoi(value.c_str());
            else if (param == "CREATE_LOG")
            {
                if (toupper(value)=="TRUE")
                    m_create_log = true;
                else if (toupper(value)=="FALSE")
                    m_create_log = false;
            }
            else if (param == "USE_PLOGGER_DIR")
            {
                if (toupper(value)=="TRUE")
                    m_use_plogger_dir = true;
                else if (toupper(value)=="FALSE")
            }
        }
    }
    return(true);
}
m_use_plogger_dir = false;

else if (param == "LOG_PATH")
    m_log_path = value;
else if (param == "IMU_MODE")
    {
        m_pre_calibration_mode = toupper(value);
        if (toupper(value) == "COMPASS_IMU"){
            m_output_compass = true;
            m_output_accel = true;
        }
        if (toupper(value) == "COMPASS_ONLY"){
            m_output_compass = true;
            m_output_accel = false;
        }
        if (toupper(value) == "IMU_ONLY"){
            m_output_compass = false;
            m_output_accel = true;
        }
    }
else if (param == "CAL_ON_STARTUP")
    {
        if (toupper(value) == "TRUE"){
            m_calibrating_accel = true;
            m_calibrating_compass = true;
        }
        if (toupper(value) == "FALSE"){
            m_calibrating_accel = false;
            m_calibrating_compass = false;
        }
    }
else if (param == "PRE_ROTATION")
    m_pre_rotation = atof(value.c_str());
else if (param == "SENSOR_AVG_TIME")
    m_sensor_avg_time = atof(value.c_str());
else if (param == "SENSOR_CAL_TIME")
    m_calibrate_time = atof(value.c_str());
else if (param == "CORR_FORGRAVITY")
    {
        if (toupper(value) == "TRUE")
            m_corr_for_gravity = true;
        else if (toupper(value) == "FALSE")
            m_corr_for_gravity = false;
    }
m_readbuffer = vector<unsigned char>(m_buffer_size, 0);
m_writebuffer= vector<unsigned char>(m_buffer_size, 0);
m_timewarp = GetMOOSTimeWarp();

RegisterVariables();

m_Comms.Notify("IMU_MODE", m_pre_calibration_mode);

// If a log file was requested but not using pLogger, make the log file now.
if (m_create_log && !m__use_ploggerdir) // Create IMU Log File
createLogFile();

if (!m_create_log) // Open port if not done by creating log files method
openPort(m_port_name, m_baud_rate);

//Note: If using plogger, the the port will be opened on new mail
return(true);

// Procedure: RegisterVariables

void SpecIMU::RegisterVariables()
{
  m_Comms.Register("LOGGER_DIRECTORY", 1);
m_Comms.Register("IMU_MODE", 0);
}

// Additional Methods

// Resets Calibration Data
void SpecIMU::resetCalibration()
{
  m_last_accel_output_time = MOOSTime();
m_last_compass_output_time = MOOSTime();
  m_avg_accel_counter = 0;
m_avg_compass_counter = 0;
m_avg_heading = 0;
m_avg_pitch = 0;
m_avg_roll = 0;
m_avg_accel_x = 0;
m_avg_accel_y = 0;
m_avg_accel_z = 0;
m_pitch_corr = 0;
m_roll_corr = 0;
m_accel_x_corr =
m_accel_y_corr =
m_accel_z_corr =

// Creates Log file and opens serial port
void SpecIMU::createLogFile() {
    stringstream filename, ss;
    time_t now = time(0); // Time using ctime in secs
    tm *ltm = localtime(&now); // Time converted to clock format
    filename << m_log_path <<"IMU_LOG_" << setw(2) << ltm->tm_hour << 
    "_" << setw(2) << ltm->tm_min << 
    "_" << setw(2) << ltm->tm_sec << ".txt";
    cout << "Opening_Log_file:" << filename.str().c_str() << endl;
    m_IMU_log.open(filename.str().c_str());
    m_IMU_log << "IMU_Log_File_" << filename.str() << "_ 
    Created:_" << endl;
    m_log_initialized = true;
    openPort(m_port_name, m_baud_rate);
}

// General Serial Port Aysnchronous Handling Methods

// Opens the serial port and creates Serial Loop in separate thread
void SpecIMU::openPort(string port_name, int port_baud_rate) {
    if (m_port.is_open())
    
}
return;

// open the serial port
m_port.open(port_name);

// serial port must be configured after being opened
// values are: no flow control, no parity, 1 stop bit, 8-bit chars
m_port.set_option(serial_port_base::baud_rate(port_baud_rate));
m_port.set_option(serial_port_base::flow_control(serial_port_base::flow_control::none));
m_port.set_option(serial_port_base::parity(serial_port_base::parity::none));
m_port.set_option(serial_port_base::stop_bits(serial_port_base::stop_bits::one));
m_port.set_option(serial_port_base::character_size(8));

// start the background thread
m_serial_thread = boost::thread(
    boost::bind(&SpecIMU::serialLoop, this));

// Closes the serial port and rejoins the main thread
void SpecIMU::closePort() {
    m_stop_requested = true;
    m_serial_thread.join();
    m_port.close();
}

// Allows communication back to the serial device
void SpecIMU::writeData(unsigned char *ptr, int length) {
    m_write_buffer_mutex.lock();
    memcpy(&m_write_buffer[m_bytes_to_write], ptr, length);
    m_bytes_to_write += length;
    m_write_buffer_mutex.unlock();
}

// Checks if data is available, and if so sends it to async read
void SpecIMU::readHandler(bool& data_available, 
                           deadline_timer& timeout, 
                           const boost::system::error_code& error, std::size_t bytes_transferred) {

181
if (error || !bytes_transferred) {
    // no data read
    m_data_available = false;
    return;
}

m_data_available = true;
m_async_bytes_read = bytes_transferred;
timeout.cancel();
}

void SpecIMU::waitCallback(serial_port& ser_port, const boost::system::error_code& error) {
    if (error) {
        // data read, timeout cancelled
        return;
    }
    m_port.cancel();
}

void SpecIMU::processWriteBuffer() {
    // take out lock
    m_write_buffer_mutex.lock();
    if (m_bytes_to_write > 0) {
        // if there is data waiting, copy it to a local buffer
        vector<unsigned char> local_write_buffer(
            m_bytes_to_write, 0);
        memcpy(&local_write_buffer[0], &m_write_buffer[0],
            m_bytes_to_write);
        m_bytes_to_write = 0;
        // release lock to prevent outside write requests from blocking on serial write
        m_write_buffer_mutex.unlock();
        // simple synchronous serial write
        m_port.write_some(buffer(local_write_buffer, local_write_buffer.size()));
    } else {
        // no data to write, release lock
        m_write_buffer_mutex.unlock();
    }
}

// This Serial Loop is effective the main interaction occurring in a separate thread
void SpecIMU::serialLoop() {

while (!m_stop_requested) {
    processWriteBuffer(); // handle outgoing commands to serial device

    // set up an asynchronous read that will read up to 100 bytes, but will return as soon as any bytes are read. Bytes read will be placed into m_read_buffer starting at index 0
    m_port.async_read_some(buffer(&m_read_buffer[0], m_buffer_size),
        boost::bind(&SpecIMU::readHandler, this, boost::ref(m_data_available), boost::ref(m_timeout), boost::asio::placeholders::error),
        boost::asio::placeholders::bytes_transferred);

    // setup a timer that will prevent the asynchronous operation for more than 100 ms
    m_timeout.expires_from_now(boost::posix_time::milliseconds(1000));
    m_timeout.async_wait(
        boost::bind(&SpecIMU::waitCallback, this, boost::ref(m_port), boost::asio::placeholders::error));

    // reset then run the io service to start the asynchronous operation
    m_io.reset();
    m_io.run();

    if (m_data_available) { // gather one line from the buffer
        m_string_buffer += string(m_read_buffer.begin(), m_read_buffer.begin() + m_async_bytes_read);

        while (m_string_buffer.find("\n", 1) != string::npos) {
            int index = m_string_buffer.find("\n", 1);
            ParserAndUpdate(m_string_buffer.substr(0, index));
            m_string_buffer = m_string_buffer.substr(index + 1, m_string_buffer.size() - index - 1);
        }
    }
}
Serial Output Data Formatting and Handling

// Parser Lines into separate variables
void SpecIMU::ParserAndUpdate(string msg) {
  if (msg[0] != '#' || msg[4] != '=') // do not include partial messages
    return;

  // Output time stamped raw line to log file
  if (m_create_log) {
    writeLine(msg);
  }

  // If finished calibrating revert to previous mode
  if (m_calibration_started && !m_calibrating_accel && !m_calibrating_compass) {
    m_Comms.Notify("IMU_MODE", m_pre_calibration_mode);
    m_calibration_started = false;
  }

  // Handle vector based on the type
  if (msg[1] == 'A' && m_output_accel) // Accelerometer Message
    updateAccel(msg);
  if (msg[1] == 'Y' && m_output_compass) // Magnetometer Message
    updateCompass(msg);
}

// Creates a time stamped line entry into the log file
void SpecIMU::writeLine(std::string sLine) {
  boost::posix_time::ptime pt(BOOST_POSIX_TIME::microsec_clock::universal_time());
  m_IMU_log << boost::posix_time::to_simple_string(pt) << " ":";
  m_IMU_log << sLine << endl;
}

// Parser and Update Accelerometer Output
void SpecIMU::updateAccel(string accel_msg) {
  bitString(accel_msg, '}');
  vector<string> accel_vector = parseString(accel_msg, ',');
// Order changed and "-" added to correct for IMU rotated 90 degree
double y_a = atof(accel_vector[0].c_str());
double x_a = -atof(accel_vector[1].c_str());
double z_a = -atof(accel_vector[2].c_str());

double time_now = MOOSTime();

// Update the average value
avgAccel(x_a, y_a, z_a);

// Check to see if calibrating and when finished set the corrections
if (m_calibrating_accel && (time_now -
    m_last_accel_output_time) > m_calibrate_time ){
    m_accel_x_corr = -m_avg_accel_x;
    m_accel_y_corr = -m_avg_accel_y;
    m_accel_z_corr = -m_avg_accel_z;
    m_last_accel_output_time = time_now;
    m_avg_accel_counter = 0;
    m_calibrating_accel = false;
}

// Based on output time: post and reset
if (!m_calibrating_accel && (time_now -
    m_last_accel_output_time) > m_sensor_avg_time ){
    m_Comms.Notify("X_ACCEL", m_avg_accel_x);
    m_Comms.Notify("Y_ACCEL", m_avg_accel_y);
    m_Comms.Notify("Z_ACCEL", m_avg_accel_z);
    m_last_accel_output_time = time_now;
    m_avg_accel_counter = 0;
}

// Parser and Update Magnetometer (compass) Output
void SpecIMU::updateCompass(std::string compass_msg){
    bitString(compass_msg, '=');
    vector<string> compass_vector = parseString(compass_msg, ',', '
');

    // Order changed and "-" added to correct for IMU rotated 90 degree
double heading = atof(compass_vector[0].c_str());
double roll = -atof(compass_vector[1].c_str());
double pitch = atof(compass_vector[2].c_str());
// Corrections for specific orientation and use
heading += m_pre_rotation;
while (heading >360)
    heading -= 360.0;
while (heading <0)
    heading += 360.0;

double time_now = MOOSTime();

// Update the average value
avgCompass(heading, pitch, roll);

// Check to see if calibrating and when finished set the corrections
if (m_calibrating_compass && (time_now - m_last_compass_output_time) > m_calibrate_time ){
    m_pitch_corr = - m_avg_pitch;
    m_roll_corr = - m_avg_roll;
    m_last_compass_output_time = time_now;
    m_avg_compass_counter = 0;
    m_calibrating_compass = false;
}

// Based on output time: post and reset
if (!m_calibrating_compass && (time_now - m_last_compass_output_time) > m_sensor_avg_time ){
    m_Comms.Notify("COMPASS_HEADING", m_avg_heading);
    m_Comms.Notify("COMPASS_PITCH", m_avg_pitch);
    m_Comms.Notify("COMPASS_ROLL", m_avg_roll);
    m_last_compass_output_time = time_now;
    m_avg_compass_counter =0;
}

// Average incoming sensor output for the IMU acceleration
void SpecIMU::avgAccel(double new_x, double new_y, double new_z){
    const double g=-240;
    double new_total_x, new_total_y, new_total_z;
    // Correct for gravity in pitch and roll if requested and not calibrating
if (m_corr_for_gravity && !m_calibrating_compass &&
    !m_calibrating_accel)
{
    new_total_x = (m_avg_accel_x * m_avg_accel_counter) +
                   new_x + m_accel_x_corr
    - g*sin(m_avg_pitch * M_PI/180);
    new_total_y = (m_avg_accel_y * m_avg_accel_counter) +
                   new_y + m_accel_y_corr
                   + g*sin(m_avg_roll * M_PI/180);
}
else {
    new_total_x = (m_avg_accel_x * m_avg_accel_counter) + new_x
                         + m_accel_x_corr;
    new_total_y = (m_avg_accel_y * m_avg_accel_counter) + new_y
                         + m_accel_y_corr;
}
new_total_z = (m_avg_accel_z * m_avg_accel_counter) + new_z
                         + m_accel_z_corr;
    m_avg_accel_counter++;
    m_avg_accel_x = new_total_x / m_avg_accel_counter;
    m_avg_accel_y = new_total_y / m_avg_accel_counter;
    m_avg_accel_z = new_total_z / m_avg_accel_counter;
}

// Average incoming sensor output for the compass
void SpecIMU::avgCompass(double new_heading, double new_pitch,
                           double new_roll)
{
    double new_total_heading, new_total_pitch, new_total_roll;
    new_total_heading = (m_avg_heading * m_avg_compass_counter) +
                         new_heading;
    new_total_pitch = (m_avg_pitch * m_avg_compass_counter) +
                         new_pitch + m_pitch_corr;
    new_total_roll = (m_avg_roll * m_avg_compass_counter) +
                         new_roll + m_roll_corr;
    m_avg_compass_counter++;
    m_avg_heading = new_total_heading / m_avg_compass_counter;
    m_avg_pitch = new_total_pitch / m_avg_compass_counter;
    m_avg_roll = new_total_roll / m_avg_compass_counter;
m_avg_roll = new_total_roll / m_avg_compass_counter;
B.3 Customized MOOS Application For Recording Self-Noise -C++ Code

This code uses a USB configured sound card to take and process acoustic data from the hydrophones and reored them to the MOOS synchronous log file.

B.3.1 Read Self Noise Application Header File - ReadSelfNoise.h

```cpp
#ifndef ReadSelfNoise_HEADER
#define ReadSelfNoise_HEADER

#include <iostream>
#include <cstdlib>
#include <sstream>
#include <cstring>
#include <fstream>
#include <stdexcept>
#include <sys/wait.h>
#include <ctime>
#include "MOOS/libMOOS/MOOSLib.h"
#include "MBUtils.h"

using namespace std;

class ReadSelfNoise : public CMOOSApp
{
public:
    ReadSelfNoise();
    ~ReadSelfNoise();

protected:
    bool OnNewMail(MOOSMSG_LIST &NewMail);
    bool Iterate();
    bool OnConnectToServer();

#endif
```


bool OnStartUp();
void RegisterVariables();
string BuildCmd();
void ConfigureMixer();
string ExecuteCmd(const char* cmd);
bool ParserAndUpdate(string cmd_return);

private: // Configuration variables
    string m_sound_card;
    string m_device_num;
    string m_bit_depth;
    string m_channels;
    string m_sample_rate;
    string m_sample_duration;
    string m_create_file;
    string m_file_path;

private: // State variables
    unsigned int m_iterations;
    double m_timewarp;
    bool m_get_reading;
    string m_cmd;
    string m_cmd_return;
    double m_sample_rms;
    double m_sample_max_amp;
    double m_sample_freq;
    int m_buffer_size;
    string m_wav_filename;
};

#endif
B.3.2 Read Self Noise Application Source Code - ReadSelfNoise.cpp

```cpp
/**
 * NAME: Ben Thomson
 * ORGN: MIT
 * FILE: ReadSelfNoise.cpp
 * DATE: Feb 2017
 * This application uses the sound card and a program called SoX, called from the cmd line, to analyse
 * self-noise and send stats to the MOOSDB.
 */

#include <iterator>
#include "ReadSelfNoise.h"

using namespace std;

// Constructor
ReadSelfNoise::ReadSelfNoise()
{
    m_iterations = 0;
    m_timewarp = 1;
    m_get_reading = false;
    m_sound_card = "2";
    m_device_num = "0";
    m_bit_depth = "16";
    m_channels = "1";
    m_sample_rate = "48000";
    m_sample_duration = ".05";
    m_create_file = "false";
    m_file_path = "/logs/"
    m_wav_filename = "";
    m_cmd = "";
    m_cmd_return = "";
    m_sample_rms = 0;
    m_sample_max_amp = 0;
    m_sample_freq = 0;
    m_buffer_size = 1024;
}
```

//
// Destructor

ReadStreamNoise::~ReadStreamNoise()
{
}

// Procedure: OnNewMail

bool ReadSelfNoise::OnNewMail(MOOSMSG_LIST &NewMail)
{
    MOOSMSG_LIST::iterator p;
    for (p=NewMail.begin(); p!=NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key = msg.GetKey();
        if (key == "SN_TAKE_READINGS"){
            string sval = msg.GetString();
            if (toupper(sval) == "FALSE")
                m_get_reading = false;
            if (toupper(sval) == "TRUE")
                m_get_reading = true;
        }
    }
    #if 0 // Keep these around just for template
    string key = msg.GetKey();
    string comm = msg.GetCommunity();
    double dval = msg.GetDouble();
    string sval = msg.GetString();
    string msrc = msg.GetSource();
    double mtime = msg.GetTime();
    bool mdbl = msg.IsDouble();
    bool mstr = msg.IsString();
    #endif
    return(true);
}

// Procedure: OnConnectToServer

bool ReadSelfNoise::OnConnectToServer()
{
    // register for variables here
}
RegisterVariables();
return(true);
}

// Procedure: Iterate()
// happens AppTick times per second
bool ReadSelfNoise::Iterate()
{
    if (m_get_reading){
        m_cmd = BuildCmd();
m_cmd_return = ExecuteCmd(m_cmd.c_str());
m_iterations++;
m_Comms.Notify("SN_READ_ITERATIONS", m_iterations);
ParserAndUpdate(m_cmd_return);
    }
return(true);
}

// Procedure: OnStartUp()
// happens before connection is open
bool ReadSelfNoise::OnStartUp()
{
    list<string> sParams;
m_MissionReader.EnableVerbatimQuoting(false);
if (m_MissionReader.GetConfiguration(GetAppName(), sParams))
{
    list<string>::iterator p;
    for (p=sParams.begin(); p!=sParams.end(); p++) {
        string original_line = *p;
        string param = stripBlankEnds(toupper(biteString(*p, '=')));
        string value = stripBlankEnds(*p);
        if (param == "SOUNDCARD")
m_sound_card = value;
else if (param == "DEVICE_NUM")
m_device_num = value;
else if (param == "BIT_DEPTH")
m_bit_depth = value;
else if (param == "CHANNELS")
    m_channels = value;
else if (param == "SAMPLE_RATE")
    m_sample_rate = value;
else if (param == "SAMPLE_DURATION")
    m_sample_duration = value;
else if (param == "CREATE_FILE")
    m_create_file = value;
else if (param == "SAMPLE_FILEPATH")
    m_file_path = value;
}

if (toupper(m_create_file) == "TRUE") {
    // Create Directory for wav_files
    stringstream directory, file_path, ss;
    time_t now = time(0);  // Time using ctime in secs
    tm *ltm = localtime(&now);  // Time concerted to clock format
    directory << "wav_files_" << setfill('0') << setw(2) <<
    "_" << setfill('0') << setw(2) << ltm->tm_hour <<
    "_" << setfill('0') << setw(2) << ltm->tm_min <<
    "_" << setfill('0') << setw(2) << ltm->tm_sec;
    file_path << m_file_path << directory.str() << "/";
    m_file_path = file_path.str();  //Update file path for all wav files
    ss << "mkdir_-" << file_path.str() << endl;
    system(ss.str().c_str());
}

ConfigureMixer();

m_timewarp = GetMOOSTimeWarp();
RegisterVariables();
return(true);

// Procedure: RegisterVariables

void ReadSelfNoise::RegisterVariables() {
    Register("SN_TAKE_READINGS", 0);
string ReadSelfNoise::BuildCmd()
{
    stringstream ss, filename;
    // If configured, make sequential wav filenames
    filename << setfill(\'0\') << setw(8) << m_iterations;
    m_wav_filename = filename.str();
    ss << "AUDIODV=plughw:\" << m_sound_card << ",\" <<
        m_device_num
    << "_AUDIODRIVER=alsa\_rec\_r\" << m_sample_rate << "\_b-
        " << m_bit_depth
    << "\c\" << m_channels << "\q\";
    if (toupper(m_create_file) == "TRUE") // If config to make files
        ss << m_file_path << m_wav_filename << ".\wav";
    else
        ss << "-n"; // no file, just stats
    ss << "\trim\_0\" << m_sample_duration << "\stat\" << "\-
        2\&1";
    string str = ss.str();
    return(str);
}

void ReadSelfNoise::ConfigureMixer()
{
    string str="\";
    stringstream speaker, mic, agc;
    speaker << "amixer\_c\" << m_sound_card << "\q\sset\ Speakers\_90\%\_unmute\";
    str = speaker.str();
    system(str.c_str());
mic = "amixer -c " + m_sound_card + " -qs Mic 90% mute_cap";
str = mic.str();
system(str.c_str());

age = "amixer -c " + m_sound_card + " -qs 'Auto_Gain Control' mute";
str = age.str();
system(str.c_str());

------ Procedure: ExecuteCmd ------

string ReadSelfNoise::ExecuteCmd(const char* cmd)
{
    char buffer[m_buffer_size];
    string result = "";

    FILE* pipe = popen(cmd, "r");
    if (!pipe) throw std::runtime_error("popen() failed!");
    try {
        while (!feof(pipe)) {
            if (fgets(buffer, m_buffer_size, pipe) != NULL) {
                result += buffer;
            }
        }
    } catch (...) {
        pclose(pipe);
        throw;
    }
    pclose(pipe);
    return result;
}

------ Procedure: ParserAndUpdate ------

bool ReadSelfNoise::ParserAndUpdate(string cmd_return)
{
    string max_amp_str = tokStringParse(cmd_return, "Maximum_amplitude", "\n",":");
string rms_amp_str = tokStringParse(cmd_return, "RMS___amplitude", 'n', ':' );
string freq_str = tokStringParse(cmd_return, "Rough___frequency", 'n', ':' );

m_sample_max_amp = atof(max_amp_str.c_str());
m_sample_rms = atof(rms_amp_str.c_str());
m_sample_freq = atof(freq_str.c_str());

m_Comms.Notify("SN_MAX_AMPLITUDE", m_sample_max_amp);
m_Comms.Notify("SN_RMS_AMPLITUDE", m_sample_rms);
m_Comms.Notify("SN_ROUGH_FREQUENCY", m_sample_freq);

return(true);
B.4 Cause-Effect Controller Application - C++ Code

This code acts as the cause-effect controller (CEC) and will call the SpecIMU and ReadSelfNoise applications as required. This program will create the data maps and will then provide modified signals to the PID application.

B.4.1 Cause-Effect Controller to PID Header File - CEPID.h

```cpp
using namespace std;

class CEPID : public CMOOSApp
{
    public:
        CEPID();
        ~CEPID();

    protected:
        bool OnNewMail(MOOSMSG_LIST &NewMail);
```

bool Iterate();
bool OnConnectToServer();
bool OnStartUp();
void RegisterVariables();

private: // CEPID Configuration variables
bool m_use_CEPID;
string m_calib_file_path;
int m_number_cal_runs;
double m_sync_period;
bool m_use_existing_cal;
string m_existing_file;
bool m_noise_monitor;
bool m_use_sim_data;
string m_data_file_name;
double m_response_time_lag;
double m_time_between_runs;
double m_rms_threshold;
double m_threshold_margin_factor;
double m_config_max_thrust;
double m_config_min_thrust;

private: // CEPID State variables
double m_plogger_time_estimate;
unsigned int m_iterations;
double m_timewarp;
string m_current_state;
bool m_cal_curve_generated;
vector vector< double > > m_cal_curve;
vector vector vector< double > > m_cal_response_tables;
vector vector< double > > m_cal_groomed_table;
string m_next_state;
bool m_calibration_complete;
bool m_cal_run_complete;
int m_cal_runs_completed;
string m_calib_file_name;
ofstream m_cal_file;
bool m_existing_cal_read;
ofstream m_temp_file; //FOR TESTING ONLY
string m_pLogger_dir;
bool m_pLogger_time_correction_set;
double m_pLogger_time_correction;
string m_slog_file_name;
string m_IMU_mode;
string m_read_SN;
double m_current_SN_iteration;
double m_last_SN_iteration;
double m_current_accel;
double m_last_accel;
int m_response_element_shift;
int m_elements_between_runs;
double m_init_start_time;
double m_init_end_time;
vector<double> m_sim_start_times;
vector<double> m_sim_end_times;
double m_cal_start_time;
double m_cal_end_time;
bool m_ambient_noise_lvl_found;
double m_min_thrust;
double m_max_thrust;

private: // CEPID Data Management variables
int m_time_col;
int m_read_col;
int m_max_col;
int m_rms_col;
int m_x_accel_col;
int m_heading_col;
int m_thrust_col;
int m_rudder_col;
double m_max_max_amp;
double m_max_rms_amp;
double m_max_x_accel;
double m_cal_ambient_noise_lvl;
double m_init_ambient_noise_lvl;

private: // Class Methods / Functions
void createCalibrationFile();
string createInputFileName();
void measureAmbientNoise();
void createSimCalTimes();
vector<vector<double>> createDataArray(string filename,
                                      double start_time, double end_time);
vector<double> formatAndParser(string line);
double getAppTimeDelay(string status_msg,
                       double msg_time);
void shiftResponseData(vector<vector<double>> &data_array, int column, int num_elements);

double getSectorMean(vector<vector<double>> &data_array, int column, int start_element = 0, int end_element = 0);

vector<int> getTransitionElements(vector<vector<double>> &data_array);

vector<double> getTransitionTimes(vector<vector<double>> &data_array);

vector<int> getSpecTransElements(vector<vector<double>> &data_array, double val1, double val2);

vector<double> getSpecTransTimes(vector<vector<double>> &data_array, double val1, double val2);

vector<vector<double>> getResponseTable(vector<vector<double>> &data_array);

void outputMaxValues();

vector<vector<double>> getCalibrationCurve(vector<vector<double>> &groomed_table);

vector<vector<double>> getGroomedTable(vector<vector<vector<double>>> rough_tables);

double getPercentDiff(double val1, double val2);

void setThrustMaximums(vector<vector<double>> &cal_curve);

double interpolateThrust(vector<double> thrust_curve, vector<double> response_curve);

vector<vector<double>> getGroomedTableFromFile(string filename);

// OVERLOAD + and for vector element operations

template<typename T>
std::vector<T> operator+(const std::vector<T>& a, const std::vector<T>& b){
    assert(a.size() == b.size());
    std::vector<T> result;
    result.reserve(a.size());

    std::transform(a.begin(), a.end(), b.begin(),
        std::back_inserter(result), std::plus<T>() );
    return result;
# endif
B.4.2 Cause-Effect Controller to PID Source Code - CEPID.cpp

/*************************************************/
/* NAME: Ben Thomson */
/* ORGN: MIT */
/* FILE: CEPID.cpp */
/* DATE: Mar 2017 */
/* This application is the "Cause-Effect Controller"(CEC) */
/* it controls the SpecIMu and ReadSelfNoise apps and */
/* relates RMS Amplitude to Thrust, then bases on the */
/* acoustic threshold will send the PID Application new */
/* minimum and maximum thrusts */
/* ***********************************************************/

#include <iterator>
#include "MBUtils.h"
#include "CEPID.h"

using namespace std;

// Constructor
CEPID::CEPID()
{
    m_iterations = 0;
    m_timewarp = 1;

    // Configuration variables
    m_use_CEPID = false;
    m_use_sim_data = false;
    m_use_existing_cal = false;
    m_noise_monitor = true;
    m_calib_file_path = "/home/mit/hoverland/moos-ivp-ben/calibration";
    m_time_between_runs = 10.0;
    m_rms_threshold = 0;

    // State variables
    m_iterations = 0;
    m_current_state = "START-UP";
    m_next_state = "";
    m_calibration_complete = false;
    m_existing_cal_read = false;
    m_cal_run_complete = false;
m_cal_curve_generated = false;
m_cal_runs_completed = 0;
m_number_cal_runs = 5;
m_calib_file_name = "calibration_data";
m_pLogger_dir = "";
m_pLogger_time_correction_set = false;
m_IMU_mode = "COMPASS_ONLY";
m_read_SN = "TRUE";
m_current_SN_iteration = 0;
m_last_SN_iteration = 0;
m_current_accel = 0;
m_last_accel = 0;
m_ambient_noise_lvl_found = false;
m_response_time_lag = 0.6;
m_sync_period = 0.2;
m_time_col = 0;
m_min_thrust = -100; // assume initially no restrictions
m_max_thrust = 100;
m_config_min_thrust = -100;
m_config_max_thrust = 100;
m_threshold_margin_factor = 1.0;

// Destructor
CEPID::~CEPID()
{
}

// Procedure: OnNewMail
bool CEPID::OnNewMail(MOOSMSG_LIST &NewMail)
{
    MOOSMSG_LIST::iterator p;
    for (p = NewMail.begin(); p != NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key = msg.GetKey();
        if (key == "CEPID_STATE"){
            string sval = msg.GetString();
            m_current_state = toupper(sval);
        }
        else if (key == "NEXT_CEPID_STATE"){
            // Handle NEXT_CEPID_STATE message
        }
    }
}
string sval = msg.GetString();
m_next_state = toupper(sval);

else if (key == "SN_READ_ITERATIONS"){
    double dval = msg.GetDouble();
m_current_SN_iteration = dval;
}
else if (key == "SN_TAKE_READINGS"){
    string sval = msg.GetString();
m_read_SN = sval;
}
else if (key == "X_ACCEL"){
    double dval = msg.GetDouble();
m_current_accel = dval;
}
else if (key == "LOGGER_DIRECTORY"){
    m_pLogger_dir = msg.GetString();
}
else if (key == "PLOGGER_STATUS"){
    if (!m_pLogger_time_correction_set)
        m_pLogger_time_correction = getAppTimeDelay(msg.GetString(), msg.GetTime());
}
else if (key == "SET_THRUST_MIN"){
    double dval = msg.GetDouble();
m_min_thrust = dval;
}
else if (key == "SET_THRUST_MAX"){
    double dval = msg.GetDouble();
m_max_thrust = dval;
}
else if (key == "SET_THRESHOLD"){
    // Threshold must be greater than ambient noise during calibration. If a value less than ambient is requested, set it to ambient lvl
    double dval = msg.GetDouble();
    if (dval >= m_cal_ambient_noise_lvl)
        m_rms_threshold = dval;
    if (m_cal_curve_generated)
        setThrustMaximums(m_cal_curve);
    else
        m_Comms.Notify("SET_THRESHOLD",
                        m_cal_ambient_noise_lvl);
}
#if 0 // Keep these around just for template
string key  = msg.GetKey();
string comm = msg.GetCommunity();
double dval = msg.GetDouble();
string sval = msg.GetString();
string msrc = msg.GetSource();
double mtime = msg.GetTime();
bool mdbl = msg.IsDouble();
bool mstr = msg.IsString();
#endif

return(true);
}

// Procedure: OnConnectToServer
bool CEPID::OnConnectToServer()
{
    RegisterVariables();
    return(true);
}

// Procedure: Iterate()
//     happens AppTick times per second
bool CEPID::Iterate()
{
    m_iterations++;

    //CEPID State Machine
    // Handle "Start-up" state
    if (m_current_state == "START-UP"){
        // Immobilize kayak
        m_Comms.Notify("DESIRED_THRUST", 0.0);
        m_Comms.Notify("DESIRED_RUDDER", 0.0);
    }
    // Handle "PID_ONLY" state
    if (m_current_state == "PID_ONLY"){

// No restrictions on movement; do nothing

// Handle "Initialization" state
if (m_current_state == "INITIALIZATION"){
    // Immobilize kayak
    m_Comms.Notify("DESIRED_THRUST", 0.0);
    m_Comms.Notify("DESIRED_RUDDER", 0.0);
}

// Handle "Pre-Calibration" state
if (m_current_state == "PRE-CALIBRATION"){
    // No restrictions on movement
    // Wait for next state to "CALIBRATION RUN" if not sim or not existing cal

    // If not yet done, find ambient noise
    if (!m_ambient_noise_LVL_found)
        measureAmbientNoise();

    // Check for existing cal and if valid data
    if (m_use_existing_cal && !m_existing_cal_read)
        m_groomed_table = getGroomedTableFromFile(m_existing_file);
    // set next state to "fully-calibrated"
    m_Comms.Notify("NEXT_CEPID_STATE", "FULLY-CALIBRATED");
}

// If using sim data, do not wait for operator to say go
if (m_use_sim_data && !m_use_existing_cal)
    m_Comms.Notify("NEXT_CEPID_STATE", "CALIBRATION_RUN");

// Handle "Calibration run" state
if (m_current_state == "CALIBRATION_RUN"){
    // Conduct movements deliberately to produce calibration data
    if (m_use_sim_data){
        m_cal_run_complete = true;
        m_cal_end_time = m_sim_end_times[m_cal_runs_completed];
        m_next_state = "CAL_ANALYSIS";
    }
    else if (!m_cal_run_complete){

// continue run ; send m_cal_run_complete when finished

if (m_cal_run_complete && !m_use_sim_data ){
    m_cal_end_time = MOOSTime() - GetAppStartTime() +
    m_pLogger_time_correction;
    m_next_state = "CAL_ANALYSIS";
}

// Handle "Cal_analysis" state
if (m_current_state == "CAL_ANALYSIS"){
    // No restrictions on movement
    if (!m_calibration_complete){
        vector< vector<double> > data_array;
        data_array = createDataArray(m_slog_file_name, 
            m_cal_start_time, m_cal_end_time);

        vector< vector<double> > response_table =
            getResponseTable(data_array);
        m_cal_response_tables.push_back(response_table);
        vector<double> thrust_values = response_table[0];
        vector<double> rms_values = response_table[1];

        // CAL FILE OUTPUTS
        m_cal_file << "%Cal_Run_" << m_cal_runs_completed +1 << endl;
        m_cal_file << "%THRU: ";
        for (int i = 0; i < thrust_values.size(); i++)
            m_cal_file << thrust_values[i] << ",";
        m_cal_file << endl;
        m_cal_file << "%RMS: ";
        for (int i = 0; i < rms_values.size(); i++)
            m_cal_file << setprecision(4) << rms_values[i] << ",";
        m_cal_file << endl;

        // TEMP FILE OUTPUTS TEST CODE
        m_temp_file << "m_cal_start_time: " << m_cal_start_time << endl;
        m_temp_file << "m_cal_end_time: " << m_cal_end_time << endl;
    }
}

// Print data array to temp File
for (int row = 0; row < data_array[m_time_col].size(); row++) {
    m_temp_file << "Line_Number:" << setfill('0') << setw(5) << row << "\n";
    for (int col = 0; col < data_array.size(); col++)
        m_temp_file << data_array[col][row] << ',';
    m_temp_file << endl;
}

vector<int> thrust_transition_elements = getTransitionElements(data_array);
vector<double> thrust_transition_times = getTransitionTimes(data_array);
vector<double> cal_start_times = getSpecTransTimes(data_array, 0, 5);
vector<double> cal_end_times = getSpecTransTimes(data_array, -100, 0);

m_temp_file << "transition_elements:\n";
for (int i = 0; i < thrust_transition_elements.size(); i++)
    m_temp_file << thrust_transition_elements[i] << ',';

m_temp_file << "transition_times:\n";
for (int i = 0; i < thrust_transition_times.size(); i++)
    m_temp_file << thrust_transition_times[i] << ',';

m_temp_file << "cal_start_times:\n";
for (int i = 0; i < cal_start_times.size(); i++)
    m_temp_file << cal_start_times[i] << ',';

m_temp_file << "cal_end_times:\n";
for (int i = 0; i < cal_end_times.size(); i++)
    m_temp_file << cal_end_times[i] << ',';

  //END TEST CODE
m_cal_runs_completed++;
}
if (m_cal_runs_completed >= m_number_cal_runs) {
    m_calibration_complete = true;
}

// If last cal run go to fully-calibrated, otherwise next cal run
if (m_calibration_complete) {
    // Analyze all runs and produce single response vs thrust table
    m_next_state = "FULLY-CALIBRATED";
}
else
    m_next_state = "PRE-CALIBRATION";
}//end state

// Handle "Fully-Calibrated" state
if (m_current_state == "FULLY-CALIBRATED") {
    // only noise related restrictions on movements
    if (!m_cal_curve_generted) {
        m_cal_curve = getCalibrationCurve(m_cal_groomed_table);
        setThrustMaximums(m_cal_curve);

        // Finished with output files
        if (m_cal_file.is_open()) m_cal_file.close();
        if (m_temp_file.is_open()) m_temp_file.close();

        if (m_rms_threshold > m_cal_ambient_noise_lvl)
            m_Comms.Notify("SET_THRESHOLD", m_rms_threshold);
        else
            m_Comms.Notify("SET_THRESHOLD",
                          m_cal_ambient_noise_lvl);
    }
}

//STATE CHANGES: Check next state, if conditions (checks) for changing state are met
// then change the current state to match
if (m_current_state != m_next_state) {
    bool check_1 = false;
    bool check_2 = false;
    bool check_3 = false;
    bool check_4 = false;
// Handle START_UP state change to "initialization"
if (m_current_state == "START-UP" && m_next_state == "INITIALIZATION"){
    // check that reading self noise
    check_1 = (m_current_SN_iteration > m_last_SN_iteration);
    // check that IMU reading is changing
    check_2 = (m_current_accel != m_last_accel);
    // check that a pLogger file was created
    check_3 = (m_pLogger_dir != "");

    // If conditions are met, change state, reset last readings
    if (check_1 && check_2 && check_3){
        // Conduct one-time outputs/actions on state change
        m_Comms.Notify("CEPID_STATE", m_next_state); // change current state to next
        m_Comms.Notify("NEXT_CEPID_STATE", "PRE-CALIBRATION"); // change next to new next
        m_Comms.Notify("IMU_MODE", "CALIBRATE"); // tell IMU to calibrate
        m_init_start_time = MOOSTime(); // No time correction applied as not calculated yet
        m_slog_file_name = createInputFileName();
    }
}

// Handle "initialization" state change to "pre-calibration"
if (m_current_state == "INITIALIZATION" && m_next_state == "PRE-CALIBRATION"){
    // check that reading self noise
    check_1 = (m_current_SN_iteration > m_last_SN_iteration);
    // check that IMU reading is changing - signifies that calibration is complete
    check_2 = (m_current_accel != m_last_accel);
// check that time delay between this app and PLogger has been calculated
check_3 = m_pLogger_time_correction_set;

// wait at least 2 sec to ensure IMU has time to receive calibrate message
check_4 = ((MOOSTime() - m_init_start_time) >= 1);

// If conditions are met, change state, reset last readings
if (check_1 && check_2 && check_3 && check_4){
    // Conduct one-time outputs on state change
    m_Comms.Notify("CEPID_STATE", m_next_state); // change current state to next
    m_Comms.Notify("SN_TAKE_READINGS", "FALSE"); // turn off reading SN to save resources
    // Mark initialization end time and apply time delay corrections to synchcronize with pLogger
    m_init_start_time = m_init_start_time - GetAppStartTime() + m_pLogger_time_correction;
    m_init_end_time = MOOSTime() - GetAppStartTime() - m_pLogger_time_correction;
    // No restrictions on movement during pre-calibration
    m_Comms.Notify("SET_THRUST_MIN", m_config_min_thrust);
    m_Comms.Notify("SET_THRUST_MAX", m_config_max_thrust);
}

// Handle "pre-calibration" CASE 1: state change to Calibration Run
if (m_current_state == "PRE-CALIBRATION" && m_next_state == "CALIBRATION_RUN"){
    // check that reading self noise
    check_1 = (m_current_SN_iteration > m_last_SN_iteration);
    if (!check_1) m_Comms.Notify("SN_TAKE_READINGS", "TRUE");
    // check that IMU reading is changing - signifies that calibration is complete
    check_2 = (m_current_accel != m_last_accel);
// check that pre-calibration has generated a value for
// ambient noise lvl
check_3 = m_ambient_noise_lvl_found;
// do not calibrate if config has selected an existing
calibration file
check_4 = !m_use_existing_cal;
@EnableConditions are met, change state, reset last
readings
if (check_1 && check_2 && check_3 && check_4){
    // Conduct one-time outputs on state change
    m_Comms.Notify("CEPID_STATE", m_next_state); // change current state to next
    // Mark initialization start time and apply time delay
    // corrections to synchronize with pLogger
    m_cal_start_time = MOOSTime() - GetAppStartTime() +
                   m_pLogger_time_correction;
    if (m_use_sim_data)
        m_cal_start_time = m_sim_start_times[
            m_cal_runs_completed];
    double calibration_run = m_cal_runs_completed + 1;
    m_Comms.Notify("CALIBRATION_RUN", calibration_run);
}
// Handle "pre-calibration" CASE 2: state change to "
// fully_calibrated"
if (m_current_state == "PRE-CALIBRATION" && m_next_state == "FULLY-CALIBRATED"){
   // No conditions, simply carry out state change
   // Conduct one-time outputs on state change
   m_Comms.Notify("CEPID_STATE", m_next_state); // change current state to next
   m_Comms.Notify("NEXT_CEPID_STATE", m_next_state); // change next stat to match
   if (m_noise_monitor)
        m_Comms.Notify("SN_TAKE_READINGS", "TRUE"); // reading SN only if requested
   else
        m_Comms.Notify("SN_TAKE_READINGS", "FALSE"); // turn off reading SN to save resources
}
// Handle "Calibration Run" state change to "cal
analysis"
if (m_current_state == "CALIBRATION_RUN" && m_next_state == "CAL_ANALYSIS"){

    // check that calibration is complete
    check_1 = m_cal_run_complete;

    // check that reading self noise is off
    check_2 = (m_current_SN_iteration == m_last_SN_iteration);
    if (!check_1) m_Comms.Notify("SN_TAIE_READINGS", "FALSE ");

    // If conditions are met, change state, reset last readings
    if (check_1 && check_2)
        // Conduct one-time outputs on state change
        m_Comms.Notify("CEPID_STATE", m_next_state); // change current state to next
        m_cal_run_complete = false;
    }

    // Handle "cal analysis" CASE 1: state change to "pre-calibration"
    if (m_current_state == "CAL_ANALYSIS" && m_next_state == "PRE-CALIBRATION"){

        // check that calibration is not complete
        check_1 = !m_calibration_complete;

        // check that run_complete flag is reset
        check_2 = !m_cal_run_complete;

        // If conditions are met, change state, reset last readings
        if (check_1 && check_2)
            // Conduct one-time outputs on state change
            m_Comms.Notify("CEPID_STATE", m_next_state); // change current state to next
        }

    // Handle "cal analysis" CASE 2: state change to "fully-calibrated"

if (m_current_state == "CAL_ANALYSIS" && m_next_state == "FULLY-CALIBRATED") {
    // check that calibration is not complete
    check_1 = m_calibration_complete;
    // check that reading self noise is off
    check_2 = (m_current_SN_iteration == m_last_SN_iteration);
    if (!check_1) m_Comms.Notify("SN_TAKE_READINGS", "FALSE");
    // If conditions are met, change state, reset last readings
    if (check_1 && check_2) {
        // Conduct one-time outputs on state change
        m_cal_groomed_table = getGroomedTable(
            m_cal_response_tables);
        m_Comms.Notify("CEPID_STATE", m_next_state); // change current state to next
        m_Comms.Notify("NEXT_CEPID_STATE", m_next_state); // change next state to match
    }
    m_last_SN_iteration = m_current_SN_iteration;
    m_last_accel = m_current_accel;
    return(true);
}

// Procedure: OnStartUp()
// happens before connection is open
bool CEPID::OnStartUp() {
    // Get information from own config
    list<string> sParams;
    m_MissionReader.EnableVerbatimQuoting(false);
    if (m_MissionReader.GetConfiguration(GetAppName(), sParams)) {
        list<string>::iterator p;
        215
for(p=sParams.begin(); p!=sParams.end(); p++) {
    string original_line = *p;
    string param = stripBlankEnds(toupper(biteString(*p, '=')));
    string value = stripBlankEnds(*p);

    if (param == "USE_CEPI D") {
        if (toupper(value) == "TRUE")
            m_use_CEPI D = true;
        else if (toupper(value) == "FALSE")
            m_use_CEPI D = false;
    }
    else if (param == "USE_EXISTING_CAL") {
        if (toupper(value) == "TRUE")
            m_use_existing_cal = true;
        else if (toupper(value) == "FALSE")
            m_use_existing_cal = false;
    }
    else if (param == "NOISE_MONITORING") {
        if (toupper(value) == "TRUE")
            m_noise_monitor = true;
        else if (toupper(value) == "FALSE")
            m_noise_monitor = false;
    }
    else if (param == "THRESHOLD")
        m_rms_threshold = atof(value.c_str());
    else if (param == "MARGIN_FACTOR")
        m_threshold_margin_factor = atof(value.c_str());
    else if (param == "EXISTING_CAL_FILE")
        m_existing_file = value;
    else if (param == "CALIB_PATH")
        m_calib_file_path = value;
    else if (param == "NUM_CAL_RUNS")
        m_number_cal_runs = round(atof(value.c_str()));
    else if (param == "USE_SIM_DATA") {
        if (toupper(value) == "TRUE")
            m_use_sim_data = true;
        else if (toupper(value) == "FALSE")
            m_use_sim_data = false;
    }
}

216
else if (param == "SIM_DATA_FILE")
    m_data_file_name = value;

else if (param == "RESPONSE_TIME_LAG")
    m_response_time_lag = atof(value.c_str());

else if (param == "TIME_BETWEEN_RUNS")
    m_time_between_runs = atof(value.c_str());
}

// Get information from pLogger config
list<string> sParams_pLogger;
int col_counter=1;

m_MissionReader.EnableVerbatimQuoting(false);
if (m_MissionReader.GetConfiguration("pLogger", sParams_pLogger)) {
    // Mission file reader reads bottom to top. Reverse into right order:
    sParams_pLogger.reverse();
    list<string>::iterator p;
    for (p=sParams_pLogger.begin(); p!=sParams_pLogger.end(); p++) {
        string original_line_pLogger = *p;
        string param_pLogger = stripBlankEnds(toupper(biteString(*p,"=")));
        string value_pLogger = stripBlankEnds(*p);

        if (param_pLogger == "SYNCLOG"){
            string sync_log_value = value_pLogger;
            string preamble = stripBlankEnds(toupper(biteString(sync_log_value,"@")));
            m_syncperiod = atof(sync_log_value.c_str());
        }
        else if (param_pLogger == "LOG"){
            string log_value = stripBlankEnds(toupper(biteString(value_pLogger,"@")));
            if (log_value == "SN_READ_ITERATIONS"){
                m_read_col = col_counter;
                col_counter++;
            }
            else if (log_value == "SN_MAX_AMPLITUDE"){
                m_max_col = col_counter;
                col_counter++;
            }
            else if (log_value == "SN_RMS_AMPLITUDE"){
        }
m_rms_col = col_counter;
    col_counter++;
}
else if (log_value == "X ACCEL"){
    m_x_accel_col = col_counter;
    col_counter++;
}
else if (log_value == "COMPASS_HEADING"){
    m_heading_col = col_counter;
    col_counter++;
}
else if (log_value == "DESIRED_THRUST"){
    m_thrust_col = col_counter;
    col_counter++;
}
else if (log_value == "DESIRED_Rudder"){
    m_rudder_col = col_counter;
    col_counter++;
}
}
}

// Get information from pMarinePID_Hover (if Present) or pMarinePID config
list<string> sParams_PID;
m_MissionReader.EnableVerbatimQuoting(false);
if(m_MissionReader.GetConfiguration("pMarinePID_Hover",
sParams_PID)) {
    list<string>::iterator p;
    for(p=sParams_PID.begin(); p!=sParams_PID.end(); p++) {
        string original_line_PID = *p;
        string param_PID = stripBlankEnds(toupper(biteString(*p , '=')));
        string value_PID = stripBlankEnds(*p);
        if(param_PID == "MAXTHRUST"){
            m_config_max_thrust = atof(value_PID.c_str());
        }
    }
}
else if(m_MissionReader.GetConfiguration("pMarinePID",
sParams_PID)) {
    list<string>::iterator p;
    for(p=sParams_PID.begin(); p!=sParams_PID.end(); p++) {
string original_line_PID = *p;
string param_PID = stripBlankEnds(toupper(bitString(*p,
  '=')));
string value_PID = stripBlankEnds(*p);

if (param_PID == "MAXTHRUST"){
  m_config_max_thrust = atof(value_PID.c_str());
  m_config_min_thrust = -atof(value_PID.c_str());
}

// TEST CODE - CREATES A SEPERATE FILE FOR DE-BUGGING
string temp_filename = m_calib_file_path + string("temp_file.txt");
m_temp_file.open(temp_filename.c_str());
m_temp_file << "%TEMPORARY_FILE_" << temp_filename << "_Created:
<< MOOS::TimeToDate(MOOSTime()) << endl;

// END TEST CODE

m_timewarp = GetMOOSTimeWarp();
RegisterVariables();
m_response_element_shift = round(m_response_time_lag/
m_sync_period);
m_elements_between_runs = round(m_time_between_runs/
m_sync_period);

// Check to see if going to PID_ONLY or Initialize state
if (!m_use_CEPID) { // CEPID disabled, use PID
  m_current_state = "PID_ONLY";
  m_next_state = "PID_ONLY";
  m_IMU_mode = "COMPASS_IMU";
  // No restrictions on movements, therefore use PID config values
  m_Comms.Notify("SET_THRUST_MIN", m_config_min_thrust);
  m_Comms.Notify("SET_THRUST_MAX", m_config_max_thrust);
  // If noise monitoring read self noise; otherwise save resources
  if (m_noise_monitor){
    m_read_SN = "TRUE";
    m_IMU_mode = "COMPASS_IMU";
  }
  else{
    m_read_SN = "FALSE";
  }
}
m_IMU_mode = "COMPASS_ONLY";
}

// Using CEPID therefore turn on
else {
    m_next_state = "INITIALIZATION";
    m_read_SN = "TRUE";
    m_IMU_mode = "COMPASS_IMU";
}

// If required, create the output calibration file
if (m_use_CEPID && !m_use_existing_cal)
    createCalibrationFile();

// Send out initial settings to MOOSDB
m_Comms.Notify("NEXT_CEPID_STATE", m_next_state);
    m_Comms.Notify("CEPID_STATE", m_current_state);
    m_Comms.Notify("SN_TAKE_READINGS", m_read_SN);
    m_Comms.Notify("COMPASS_IMU", m_IMU_mode);
    m_Comms.Notify("DESIRED_THRUST", 0.0);
    m_Comms.Notify("DESIRED_RUDDER", 0.0);
    return(true);

// Procedure: RegisterVariables
void CEPID::RegisterVariables()
{
    m_Comms.Register("CEPID_STATE", 0);
    m_Comms.Register("NEXT_CEPID_STATE", 0);
    m_Comms.Register("LOGGER_DIRECTORY", 0);
    m_Comms.Register("PLOGGER_STATUS", 0);
    m_Comms.Register("IMU_MODE", 0);
    m_Comms.Register("SN_READ_ITERATIONS", 0);
    m_Comms.Register("SN_TAKE_READINGS", 0);
    m_Comms.Register("X_ACCEL", 0);
    m_Comms.Register("SET_THRUST_MIN", 0);
    m_Comms.Register("SET_THRUST_MAX", 0);
    m_Comms.Register("SET_THRESHOLD", 0);
}
Additional Class Methods

// Creates calibration file
void CEPID::createCalibrationFile() {
    stringstream filename, ss;
    time_t now = time(0); // Time using ctime in secs
    tm *ltm = localtime(&now); // Time converted to clock format

    ss << setfill('0') << setw(2) << ltm->tm_hour << "_" <<
       setfill('0') << setw(2)
       << ltm->tm_min << "_" << setfill('0') << setw(2) <<
       ltm->tm_sec;

    filename << m_calib_file_path << "CAL_DATA_" << ss.str() <<
              ".txt";

    cout << "Opening Calibration file: " << filename.str().c_str() << endl;
    m_cal_file.open(filename.str().c_str());
    m_cal_file << "%Calibration_File_" << filename.str() << "_"
               << MOOS::TimeToDate(MOOSTime()) << endl;
}

// Create the full file name for the slog file that contains the data
string CEPID::createInputFileName() {
    if (m_use_sim_data){
        createSimCalTimes();
        return m_data_file_name;
    }
    else {
        string str;
        stringstream ss;
        string dir = m_pLogger_dir;
        vector<string> str_vector = parseString(dir, '/');
        string filename = str_vector.back();
        ss << m_pLogger_dir << "/" << filename << ".slog";
        str = ss.str();
        return str;
    }
}
analyze data during intitilaztion to determine ambient noise level

```cpp
void CEPID::measureAmbientNoise() {
    vector<vector<double>> data_array;
    double rms_mean, max_mean, start_time, end_time;
    data_array = createDataArray(m_slog_file_name,
                                  m_init_start_time, m_init_end_time);
    rms_mean = getSectorMean(data_array, m_rms_col);
    max_mean = getSectorMean(data_array, m_max_col);
    // OUTPUT to calibration file: preamble: rms_mean=VALUE, max_mean=VALUE
    if (!m_use_existing_cal)
        m_cal_file << "Ambient Noise: " << "rms_mean=" <<
                     rms_mean << ", max_mean=" << max_mean << endl;
    m_init_ambient_noise_lvl = rms_mean;
    m_ambient_noise_lvl_found = true;

    // TEST CODE
    m_temp_file << "m_init_start_time: " << m_init_start_time << endl;
    m_temp_file << "m_init_end_time: " << m_init_end_time << endl;
    for (int row = 0; row < data_array[m_time_col].size(); row++){
        m_temp_file << "Line_Number: " << setw(5) << row << ":\n";
        for (int col = 0; col < data_array.size(); col++)
            m_temp_file << data_array[col][row] << " ,";
        m_temp_file << endl;
    }
    //END TEST CODE

    // Create the full file name for the slog file that contains the data
    void CEPID::createSimCalTimes() {
        ifstream sim_file (m_data_file_name.c_str());
        vector<vector<double>> temp_data_array;
        vector<double> line_doubles, last_valid_data;
```
double line_time, first_valid_time = 0, last_valid_time = 0;
int num_columns;
string line;

if (sim_file.is_open()) {
    // get first line of data
    getline(sim_file, line);
    while (line[0] != '%') getline(sim_file, line);

    // parser and get format of first line of data
    line_doubles = formatAndParser(line);
    last_valid_data = line_doubles;
    line_time = line_doubles[m_time_col];
    num_columns = line_doubles.size();

    // advance until "end_of_file"
    while (!sim_file.bad() && !sim_file.eof()) {
        getline(sim_file, line);
        if (line[0] != '%') { // check if data or comment

            line_doubles = formatAndParser(line);
            line_time = line_doubles[m_time_col];

            // replacing empty "nan" values with previous values
            for (int i = 0; i < num_columns; i++) {
                if (isnan(line_doubles[i])) // check if "NaN"
                    line_doubles[i] = last_valid_data[i]; // push valid data fwd
                else last_valid_data[i] = line_doubles[i]; // update valid data
            }

        } // check if this is the first valid line of data (i.e. no Nans)

        if (!first_valid_time) {
            bool all_valid_data = true;
            for (int i = 0; i < num_columns; i++)
                all_valid_data = (all_valid_data && !isnan(line_doubles[i])); // check if "NaN"

            // If data all valid, make this first valid time
            if (all_valid_data)
                first_valid_time = line_doubles[m_time_col];
        }

        if (line_time > last_valid_time)
last_valid_time = line_time;  // update until exit
while loop
}
}
sim_file.close();
}
else cout << "Unable_to_open_sim_file";

temp_data_array = createDataArray(m_data_file_name,
   first_valid_time, last_valid_time);

// find all start and end times for cal runs based on
specified transitions
m_sim_start_times = getSpecTransTimes(temp_data_array, 0, 5);

m_sim_end_times = getSpecTransTimes(temp_data_array, -100, 0);

// update number of cal_runs to reflect number of runs in
sim file
m_number_cal_runs = m_sim_end_times.size();
}

// Creates a data array from the plogger SLOG file for given
time
vector< vector<double> > CEPID::createDataArray(string
   filename, double start_time,
   double end_time)
{
   vector< vector<double> > return_array;
   string line;
   double line_time, first_valid_time = 0;
   vector<double> line_doubles, last_valid_data,
   thrust_transitions_times;
   vector<int> thrust_transitions_elements;
   int num_columns;
   bool found_start_line = false;
   ifstream slog_file (filename.c_str());
   if (slog_file.is_open()) {
   
   // If a calibration run, apply min time between runs to
both ends
if (m_ambient_noise_lvl_found) {
    start_time -= m_time_between_runs;
    end_time += m_time_between_runs;
}

// get first line of data
getline(slog_file, line);
while (line[0] == '%') getline(slog_file, line);

// parser and get format of first line of data
line_doubles = formatAndParser(line);
lst_valid_data = line_doubles;
line_time = line_doubles[m_time_col];
num_columns = line_doubles.size();

// build array based on number of columns needed
vector<vector<double>> data_array(num_columns);

// advance until "endtime"
while (line_time < end_time && !slog_file.bad() && !slog_file.eof()) {
    getline(slog_file, line);
    if (line[0] != '%') {
        line_doubles = formatAndParser(line);
        line_time = line_doubles[m_time_col];

        // replacing empty "nan" values with previous values
        for (int i = 0; i < num_columns; i++) {
            if (isnan(line_doubles[i])) // check if "NaN"
                line_doubles[i] = last_valid_data[i]; // push valid data fwd
            else last_valid_data[i] = line_doubles[i]; // update valid data
        }

        // update global max values
        if (line_doubles[m_max_col] > m_max_max_amp)
            m_max_max_amp = line_doubles[m_max_col];
        if (line_doubles[m_rms_col] > m_max_rms_amp)
            m_max_rms_amp = line_doubles[m_rms_col];
        if (line_doubles[m_x_accel_col] > m_max_x_accel)
            m_max_x_accel = line_doubles[m_x_accel_col];
    }
}
// check if this is the first valid line of data (i.e. no Nans)
if (!first_valid_time){
    bool all_valid_data = true;
    for (int i = 0; i < num_columns; i++)
        all_valid_data = (all_valid_data && !isnan(line_doubles[i]));  // check if "NaN"

    // If data all valid, make this first valid time
    if (all_valid_data)
        first_valid_time = line_doubles[m_time_col];

    // If start time is before the first valid time, move start time
    if (start_time < first_valid_time)
        start_time = first_valid_time;

    // If in range add to data array
    if (line_time > start_time && line_time < end_time){
        for (int i = 0; i < num_columns; i++)
            data_array[i].push_back(line_doubles[i]);
    }
}
}
}
slog_file.close();

// correct the times for the response data by the given time lag
shiftResponseData(data_array, m_rms_col, m_response_element_shift);
shiftResponseData(data_array, m_max_col, m_response_element_shift);
return_array = data_array;

} else cout << "Unable_to_open_slog_file";
return return_array;

} // Takes a string, insert ',', into first whitespace, parser in string vector
vector<double> CEPIID::formatAndParser(string line){
    vector<double> line_doubles;

}
char c, last_c;
int i=1; // start at 2nd char
while (line[i]){
    c=line[i];
    last_c=line[i-1];
    if (isspace(c) && !isspace(last_c) && last_c != ',') c=',,';
    line[i]=c;
    i++;
}
vector<string> line_strings = parseString(line, ',', ');
for (int i = 0; i < line_strings.size(); i++) {
    line_doubles.push_back(atof(line_strings[i].c_str()));
}
return line_doubles;

// get the time offset between pLogger and this app. Make it a member correction
double CEPID::getAppTimeDelay(std::string status_msg, double msg_time){
    string uptime_str = tokStringParse(status_msg, "Uptime", ',', '="');
    double pLogger_app_start = msg_time - atof(uptime_str.c_str());
    double app_delay = GetAppStartTime() - pLogger_app_start;
    m_pLogger_time_correction_set = true;
    return app_delay;
}

// Shift the response data forward by elements equal to response delay
void CEPID::shiftResponseData(vector<vector<double>> &data_array, int column, int num_elements){
    double last_data = data_array[column].back();
    for (int i = 1; i <= num_elements; i++){
        // delete first entry
        data_array[column].erase(data_array[column].begin());
        // add element to end with value of last element
data_array[column].push_back(last_data);
}

// get the mean of a specified column of data array for given element range
double CEPID::getSectorMean(vector<vector<double>> &data_array, int column,
    int start_element, int
    end_element){
  // default values for both star and end elements is 0
  if (end_element <= 0) // make default value the size of the vector
    end_element = data_array[column].size();
  vector<double>::iterator start = data_array[column].begin() + start_element;
  vector<double>::iterator end = data_array[column].begin() + end_element;
  double sum = accumulate(start, end, 0.0);
  double average = sum / (end_element - start_element);
  return average;
}

// Find the element numbers of thrust changes (select element left of change)
vector<int> CEPID::getTransitionElements(vector<vector<double>> &data_array){
  vector<int> transition_elements;
  // define first element as a transition element
  transition_elements.push_back(0);
  for (int i = 0; i < data_array[m_thrust_col].size(); i++){  
    if(data_array[m_thrust_col][i] == data_array[m_thrust_col][i+1])
      transition_elements.push_back(i);
  }
  return transition_elements;
}
// Find the times of thrust changes (select element left of change)
vector<double> CEPID::getTransitionTimes(vector< vector<
double> > &data_array){
vector<double> transition_times;

// define first element as a transition time
transition_times.push_back(data_array[m_time_col][0]);

for (int i = 0; i < data_array[m_thrust_col].size(); i++){
  if (data_array[m_thrust_col][i] != data_array[m_thrust_col][i+1])
    transition_times.push_back(data_array[m_time_col][i]);
}

return transition_times;
}

// find elements when thrust goes from specified value to
// another specified value
vector<int> CEPID::getSpecTransElements(vector< vector<
double> > &data_array, double val1, double val2){

vector<int> transition_elements;

for (int i = 0; i < data_array[m_thrust_col].size(); i++){
  if (data_array[m_thrust_col][i] == val1 &&
    m_thrust_col[i+1] == val2)
    transition_elements.push_back(i);
}

return transition_elements;
}

// find elements when thrust goes from specified value to
// another specified value
vector<double> CEPID::getSpecTransTimes(vector< vector<
double> > &data_array, double val1, double val2){

vector<double> transition_times;

for (int i = 0; i < data_array[m_thrust_col].size(); i++){
  if (data_array[m_thrust_col][i] == val1 &&
    m_thrust_col[i+1] == val2)
    transition_times.push_back(i);
}

return transition_times;
}
if (data_array[m_thrust_col][i] == val1 && data_array[m_thrust_col][i+1] == val2)
transition_times.push_back(data_array[m_time_col][i]);
}
return transition_times;

// OUTPUT to calibration file: preample: max_amp=VALUE,
max_rms=VALUE, max_x_accel=VALUE
void CEPID::outputMaxValues()
{
if (!m_use_existing_cal)
m_cal_file << "MAX_VALUES: " << "max_amp=" <<
m_max_max_amp << ", " << "max_rms=" <<
m_max_rms_amp << ", " << "max_x_accel=" <<
m_max_x_accel << endl;
}

// Takes in a data array, returns array with columns: thrust, 
avg_rms
vector<vector<double>> CEPID::getResponseTable(vector<vector<double>> &data_array)
{
vector<vector<double>> response_table;
vector<double> thrust_values;
vector<double> rms_values;
vector<int> transitions;

// Find transition elements for calibration data array
transitions = getTransitionElements(data_array);

// populate the response curve columns with thrust and 
average RMS Amplitude
for (int i = 0; i < transitions.size() - 1; i++)
{
    // transitions[i] is the element immediately before the
    // next transition.
    // start and stop define the sector between transitions
    int start = transitions[i];
    int stop = transitions[i+1];

    double mean_rms = getSectorMean(data_array, m_rms_col, 
    start, stop);
    double thrust = data_array[m_thrust_col][stop];
// avoid thrust = 0 data after initial: data skewed due to momentum
if (i <= 1 || thrust != 0 ){
    rms_values.push_back(mean_rms);
    thrust_values.push_back(thrust);
}

// populate the response table with the inputs (thrust) and responses (mean rms)
response_table.push_back(thrust_values);
response_table.push_back(rms_values);
return response_table;

// Create a Cal curve with elements ordered from smallest to largest from table
vector< vector<double> > CEPID::getCalibrationCurve(vector<
    vector<double> > &groomed_table){
vector< vector<double> > cal_curve(2);
vector<double> thrust_values = groomed_table[0];
vector<double> response_values = groomed_table[1];
bool table_transferred = false;

// build two ordered tables from the groomed table
while (!thrust_values.empty()){

    // Find smallest element in thrust (input) column
    int smallest_element = 0;
    for (int i = 1; i < thrust_values.size(); i++){
        if (thrust_values[i] < thrust_values[smallest_element])
            smallest_element = i;
    }

    // move the input and reponse values at the smallest element to cal curve
    cal_curve[0].push_back(thrust_values[smallest_element]);
    cal_curve[1].push_back(response_values[smallest_element]);

    // remove smallest element from the lists
    thrust_values.erase(thrust_values.begin() + smallest_element);
}

return response_table;
}
response_values.erase(response_values.begin() + smallest_element);

// Ambient noise is assumed at thrust = 0. Use the corresponding value of the response curve to ensure users can not set thresholds less than this value as it would produce erroneous results
if (thrust_values[smallest_element] == 0)
    m_cal_ambient_noise_lvl = response_values[smallest_element];

// TEST CODE
m_temp_file << "CAL_CURVE_THRUST: \\
for (int i = 0; i < cal_curve[0].size(); i++)
    m_temp_file << cal_curve[0][i] << ',
    m_temp_file << endl;
for (int i = 0; i < cal_curve[1].size(); i++)
    m_temp_file << cal_curve[1][i] << ',
    m_temp_file << endl;
// END TEST CODE

// Update Cal file with Cal Curve
if (!m_use_existing_cal){
    // Update Cal File with table information
    m_cal_file << "#THRUST"
    for (int i = 0; i < cal_curve[0].size(); i++)
        m_cal_file << ', ' << cal_curve[0][i];
    m_cal_file << endl;
    m_cal_file << "#RMS_AMP"
    for (int i = 0; i < cal_curve[1].size(); i++)
        m_cal_file << ', ' << cal_curve[1][i];
    m_cal_file << endl;
    outputMaxValues(); // Finally add the maximum values to the cal file
}

m_cal_curve_generated=true;
return cal_curve;
vector< vector<double> > CEPID::getGroomedTable(vector<
    vector< vector<double> > > rough_tables){
    const float threshold = 30;
    bool data_groomed = false;
    vector< vector<double> > groomed_table(2);
    vector<double> response_average, response_table,
       percent_error;

    while (!data_groomed){
        // find average of response
        response_average = rough_tables[0][1]; // first response entry
        for (int i = 1; i < rough_tables.size(); i++){
            response_table = rough_tables[i][1];
            response_average = response_average + response_table;
        }
        // divide sum by entries to find average
        transform(response_average.begin(), response_average.end()
           () , response_average.begin(),
           std::bind2nd(std::divides<double>(),
           rough_tables.size()));

        // TEST CODE
        m_temp_file << "response_avg:";
        for (int i = 0; i < response_average.size(); i++)
            m_temp_file << response_average[i] << ',';
        m_temp_file << endl;
        // END TEST CODE

        // find percent errors
        for (int i = 0; i < rough_tables.size(); i++){
            response_table = rough_tables[i][1];
            for (int j = 0; j < response_table.size(); j++){
                double percent_diff = getPercentDiff(response_table[j]
                     , response_average[j]);
                if (percent_diff > 30){
                    percent_error.push_back(percent_diff);
                    // TEST CODE
                    m_temp_file << "\WARNING\: \Percent\_Difference\_of\:"
                        << percent_diff << "\_on\_run\:"
                        << i+1 << "\_thrust\_of\:" << rough_tables
                            [i][0][j] << endl;
            }}}}
// END TEST CODE
// replace outlier with average value
rough_tables[i][1][j]=response_average[j];
}
}

// TEST CODE
m_tempfile << "percent_diff:");
for (int i = 0; i < percent_error.size(); i++)
  m_tempfile << percent_error[i] << ", ";
m_tempfile << endl;

if (percent_error.size() == 0)
data_groomed = true;
else
  percent_error.clear();


 stressed_table[0] = rough_tables[0][0]; // add thrust column
stressed_table[1] = response_average; // add groomed average


// END TEST CODE


double CEPI::getPercentDiff(double val1 , double val2){

double diff , avg , result;

diff = (val1 - val2);
avg = (val1 + val2)/2;

if (avg == 0)
  result = 0.0;
else
  result = (diff/avg)*100;

return result;
}

void CEPI::setThrustMaximums(vector< vector<double> > &
cal_curve){

double thrust_min , thrust_max;
double threshold = m_rms_threshold;
vector<double> left_thrust_curve , left_responseCurve,
right_thrust_curve, right_response_curve;

// Generate separate Curves
for (int i = 0; i < cal_curve[0].size(); i++){
double thrust_value = cal_curve[0][i];
if (thrust_value <= 0){
    left_thrust_curve.push_back(cal_curve[0][i]);
    left_response_curve.push_back(cal_curve[1][i]);
}
if (thrust_value >= 0){
    right_thrust_curve.push_back(cal_curve[0][i]);
    right_response_curve.push_back(cal_curve[1][i]);
}
}

// reverse left curves as to be processed the same as the right
reverse(left_thrust_curve.begin(), left_thrust_curve.end());
reverse(left_response_curve.begin(), left_response_curve.end());

// Interpolate to find values
thrust_min = interpolateThrust(left_thrust_curve,
    left_response_curve);
thrust_max = interpolateThrust(right_thrust_curve,
    right_response_curve);

m_Comms.Notify("SET_THRUST_MIN", thrust_min);
m_Comms.Notify("SET_THRUST_MAX", thrust_max);

double CEPID::interpolateThrust(vector<double> thrust_curve,
    vector<double> response_curve){
double target_thrust, target_response;
double thrust_left, thrust_right, response_left,
    response_right;
bool brackets_found = false;
int i = 1;

// set target threshold as the specified threshold x the threshold margin
target_response = m_rms_threshold *
    m_threshold_margin_factor;

235
// first check if restricted at all, if not go back to config value
if (target_response > response_curve.back()){
    if (thrust_curve.back() > 0) // positive thrust values
        return m_config_max_thrust;
    else // negative thrust values
        return m_config_min_thrust;
}

// If restricted find bracket where threshold is exceeded
while (!brackets_found){
    if (response_curve[i] > target_response) {
        response_left = (response_curve[i-1]);
        response_right = (response_curve[i]);
        thrust_left = (thrust_curve[i-1]);
        thrust_right = (thrust_curve[i]);
        brackets_found = true;
    }
    i++;
}

// Find max thrust value through interpolation within the bracket
target_thrust = ((target_response - response_left) /
    (response_right - response_left) *
    (thrust_right - thrust_left) ) +
    thrust_left;
return target_thrust;

// Takes a goomed table from an pre-existing cal file so that curve can be built
vector<vector<double>> CEPID::getGroomedTableFromFile(string filename) {
    vector<vector<double>> groomed_table(2);
    vector<string> thrust_values_str, response_values_str;
    vector<double> thrust_values, response_values;
    ifstream in_file (filename.c_str());
    string line;
    if (in_file.is_open()) {
        while (!in_file.bad() && !in_file.eof()){
            getline(in_file , line);
        }
    }
if (line[0] != '%') // ignore comments (start with '
')
    string label = biteStringX(line, '.');
    if (label == "#TRUST")
        thrust_values_str = parseString(line, '.');
    if (label == "#MAM")
        response_values_str = parseString(line, '.');
}

in_file.close();

// convert string vectors to double vectors
for (int i = 0; i < thrust_values_str.size(); i++) {
    thrust_values.push_back(atof(thrust_values_str[i].c_str()));
    response_values.push_back(atof(response_values_str[i].c_str()));
}

// build table from vectors
groomed_table[0] = thrust_values;
groomed_table[1] = response_values;
}

else cout << "Unable_to_open_slog_file";

// TEST CODE
m_temp_file << "file_in_thrust:";
for (int i = 0; i < groomed_table[0].size(); i++)
    m_temp_file << groomed_table[0][i] << ', ';
m_temp_file << endl;
m_temp_file << "file_in_response:";
for (int i = 0; i < groomed_table[1].size(); i++)
    m_temp_file << groomed_table[1][i] << ', ';
m_temp_file << endl;

// END TEST CODE

m_existing_cal_read = true;
return groomed_table;
Bibliography


