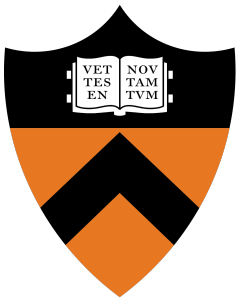
Knot The Average Boat: Development of a Semi-Autonomous Sailing Platform

Frederick "Jeb" Carter, '23

Submitted to the
Department of Mechanical and Aerospace Engineering
Princeton University
in partial fulfillment of the requirements of
Undergraduate Independent Work.

Final Report

April 26, 2023



Adviser:
Clarence Rowley
Reader:
Michael Littman
MAE 442
94 pages
File Copy

This thesis represents my own work in accordance with University regulations.

Abstract

The problem of autonomous sailing is a complex and challenging one. There are nearly countless possible environmental and telemetry sensors that can be implemented, each with varying challenges and benefits, and similarly many possible implementations of feedback control and navigation. This project approached the problem from a bottom-up approach, first creating a sensory and computational architecture that met capability requirements, then subsequently building a programmatic feedback control system designed to maximally leverage the capabilities of the onboard electronics hardware. The sailing platform is a 50-inch model sloop collecting wind data from a waterproof rotary encoder equipped with a wind vane. The onboard microcontroller, a Raspberry Pi Pico, has three autonomy modes: Manual, Pilot-Assist, and Autonomous Tack, which runs a closed-loop feedback control system with full control over rudder and sail servos. The feedback controller was designed with real-world dynamics data and leverages discrete-time controller simulations to verify reliability. The highest level of autonomy reached in this project is considered to be semi-autonomous; whereas all servo commands were autonomously controlled, the platform nonetheless requires an external operator to send a command for when to execute turns. Once completed, the vessel was fully tested in real-world conditions in the presence of highly variable winds and performed admirably, demonstrating a robust and stable controller for semi-autonomous tacking.

Acknowledgements

I would like to offer my most sincere and heartfelt gratitude Jon Prevost, who selflessly worked with me through countless challenging obstacles. You are a truly talented educator and were my most important and reliable resource.

I owe a great debt of gratitude to my family, Rob, Christine, Susanna, and Ellie Carter, for their unending love and support. In particular, thank you to my father, Rob Carter, whose love of sailing inspired my own, making this senior thesis possible.

Thank you to my thesis adviser, Professor Clarence Rowley, for your technical expertise and valuable advice. Additional thanks to Mike Vocaturo, who graciously welcomed me into his lab space and to Glenn Northey and Alexander Gaillard, whose machine shop expertise proved indispensable.

Thank you to Tommy McBride '22, my best friend and loyal first mate, as well as to all of my closest friends in the MAE Department including, but not limited to, Lunch With DaBois, The COVID Vaccmemes, The MAEnhattan Project, and the MAE Cinematic Universe. To Tommy, Alfred, Miguel, Grace, Nancy, Hayden, Lauren, and Megan: Thank you for being the greatest friends I ever could have asked for.

I am incredibly grateful to the Princeton University School of Engineering for supporting this project and to the faculty and staff of the entire Department of Mechanical and Aerospace Engineering.

Thank you to William Lesh '74 for providing the supplies and instructions necessary to build my boat and for your wealth of sailing expertise. Thank you to Maggie Chamberlain for keeping my lungs clean.

Finally, thank you to The Princeton Roaring 20, both current and graduated, for giving me the joy of music throughout my Princeton career. R20 has been practically synonymous with my time at Princeton, and I have found no greater joy than in the warmth of your company and in the spirit of our music.

To Eagles Mere, where it all began.

Contents

List of Figures

List of Symbols

Chapter 1

Introduction

1.1 A Brief Overview of Sailing

1.1.1 The Dynamics of Sailboats

Sailing vessels are nearly as old as civilization itself. As long as man could build them, they did. From ancient Egypt to remote island societies to the voyages of Magellan, sailing has been a preferred, and often vital, method of transportation [1]. However, in order to make sailing versatile and compatible with flexible travel, upwind sailing is the key.

The prevailing concept at the heart of sailing against the wind is lift. Unlike an airplane, which generates lift within a single fluid, a sailboat must contend with two different fluids, water and air. Whereas the former interfaces with the vessel's sails, the latter acts primarily on the keel and the rudder. Figure 1.1 depicts the primary forces acting on a boat sailing into the wind at constant velocity (equilibrium).

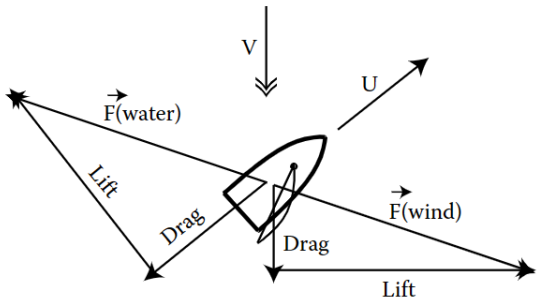


Figure 1.1: Forces on a Sailboat in Equilibrium [1]

Simply put, the sail creates a lift force perpendicular to the surface of the canvas. This force can be decomposed into Lift (perpendicular to the direction of apparent wind, V) and Drag (parallel to the direction of V). Similarly, the keel creates a force vector which is decomposed into Lift and Drag (directions relative to relative water direction rather than wind direction). Whereas the forces in Figure 1.1 are balanced (depicting a boat at cruising speed), an accelerating boat would be characterized by sail lift exceeding the drag forces created by the sail and keel, allowing the vessel to accelerate from rest. Figure 1.2 illustrates the lift and drag forces generated by a keel below the water.

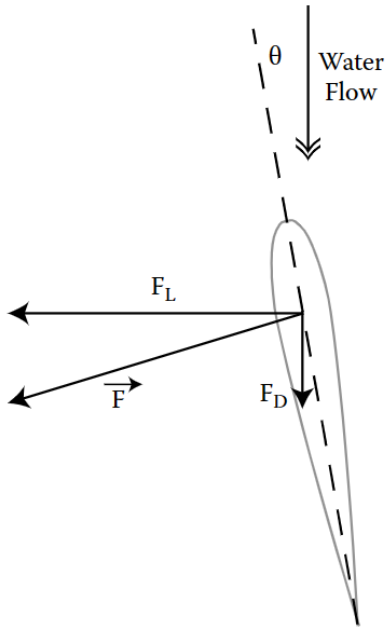


Figure 1.2: Forces on the Keel [1]

1.1.2 The Sailor's Dilemma

Once a sailor understands the forces at play, a key strategic question presents itself: What is the optimal heading to make the quickest progress upwind? Sailing close-hauled (as close into the wind as possible without luffing, often less than 45°) gives the boat a maximally upwind heading. However, doing so reduces the sail's angle of attack severely, resulting in much less overall force. Conversely, sailing in a more crosswind direction allows for significantly stronger sail forces and, consequently, a greater velocity, but the boat's heading may not be sufficiently upwind. How should a sailor weigh the benefits of a more windward heading versus a greater velocity? A

speed diagram resolves this problem.

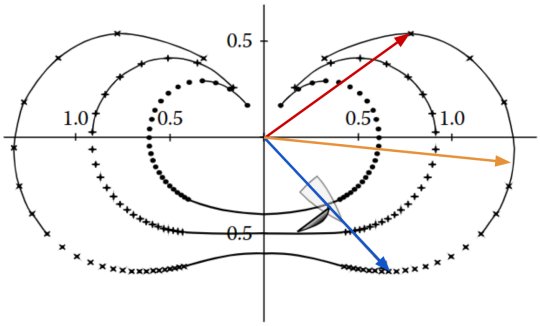


Figure 1.3: Three Speed Diagram Examples [1]

Figure 1.3 illustrates speed diagrams for three different sailboats. Each curve corresponds to a boat with a different downwind speed ratio S_0.

$$S_0 = \frac{U}{V} \tag{1.1.1}$$

where U is the true boat speed and V is the apparent wind speed. Every sailboat has a characteristic speed ratio, which can be interpreted as a measure of the boat's optimal performance sailing directly downwind. Equation 1.1.1 can be reorganized using equation 1.1.2

$$V = W - U \tag{1.1.2}$$

to relate U to the true wind speed, W:

$$U = \frac{S_0}{1 + S_0} W \tag{1.1.3}$$

A speed ratio of 1, for example, indicates that, when sailing directly downwind, the boat speed, U, is equal to the apparent wind speed, V. Solving for true wind speed using equation 1.1.2 shows that, for a boat with this characteristic S_0, the maximum downwind speed is 1/2 the true wind speed.

In the speed diagram in Figure 1.3, the distance from the origin to any point on the curve gives the ratio between the boat speed, U, and the true wind speed, W, at that heading. The three curves correspond to boats with $S_0 = 2/3$, 1, and 3/2. The arrows depict three critical directions relevant to a sailor in the $S_0 = 3/2$ boat. The red arrow is the optimal upwind sailing direction discussed previously in this section. This heading is the optimal balance between a sufficiently windward heading without

sacrificing the benefits of crosswind velocity. The orange arrow is the fastest-possible sailing speed in any direction, allowing some vessels to sail faster than the wind itself. One key observation is that the boat is not fastest when sailing directly downwind. In fact, directly downwind is often the slowest-possible heading. Instead, the quickest downwind heading is illustrated with the blue arrow. This heading (usually around 135°) takes advantage of a crosswind component to increase the boat's apparent wind speed and generate greater lift.

1.2 The Problem of Autonomous Sailing

1.2.1 Defining the Problem

The problem of navigating using the wind is a difficult one, even for experienced sailors onboard a real ship. One might consider that an electronic system of sensors is capable of better performance than a human captain, thanks to the capabilities of advanced control algorithms and hyper-accurate sensors. However, an autonomous system is actually at a significant disadvantage. A human captain has a multitude of sensors at their disposal. They can monitor the strength and direction of the wind, the velocity and angle of attack of the boat, and the physical orientation in space quite seamlessly using an array of highly-advanced sensors and an unparalleled computational architecture, known commonly as the human senses and the human brain. Whereas a sailor can rely on experience and context clues to make decisions, an autonomous sailboat must rely on the veracity of its sensors and a control algorithm. So, although the computational model of an autonomous boat may be theoretically superior to a sailor's intuition, the sailor has innumerable advantages.

The single most vital piece of information relevant to an autonomous sailboat is the wind direction. Simply knowing the direction of the wind with respect to the boat is sufficient to set the rudder heading and sail trim for an optimal upwind tack. Beyond a wind direction sensor, a range of other additions have the potential to improve the control system. A wind speed anemometer, for example, enables the vessel to determine whether the wind direction measurement is reliable since, in the absence of any wind, a wind vane will nonetheless continue to output a reading. An Inertial Measurement Unit (IMU) can describe the acceleration, orientation, rotation, and velocity of the boat. However, an IMU has the disadvantage of being purely inertial, unable to correct for systemic drift over longer time frames. Another valuable source of information would be a GPS chip, allowing the vessel to determine its location in

space, as well as its absolute velocity, depending on the precision of the measurement. Other possible sensors which may provide their own merits include a water speed sensor and a digital camera.

The scope of this project did not allow for the exploration of all of the above sensors, but they are an apt illustration of the variable complexity of the problem and provide opportunities for future work.

1.2.2 Project Objectives

The primary objective of the project was to design and implement a sailing platform with the capability to reliably and semi-autonomously tack upwind in a variety of wind conditions. The distinction between autonomous and semi-autonomous sailing examines the need, if any, for a human to participate "in the loop," even in a very limited context. There were a number of benchmark progress steps, each with an additional layer of autonomy relative to the previous step:

1. Automated sail trim with manual rudder input.
2. Automated rudder control for desired headings.
3. Autonomous rudder and sail control for optimal tacking with manual tack commands.
4. Autonomous rudder and sail control with automated tack decisions.
5. Full Autonomy - Rudder/Sail control and navigation

This thesis represents a successful implementation of the first three project objectives. Objectives 4 and 5 are left as opportunities for future work, both of which are discussed in the Future Work Section.

1.2.3 Design Considerations

The Sailing Platform

The first and most pressing decision was choosing an appropriate model sailboat for the project. The primary drivers in this decision were size, flexibility, and quality. Firstly, as far as autonomous control is concerned, bigger is often better. Water has waves, wind is irregular, and sensors are noisy and imperfect. All of these factors

tend to reduce the effectiveness of any autonomous system. By utilizing a larger vessel, the unpredictable fluctuations of water and wind were proportionally smaller. Additionally, the large size allows for the addition of larger, more stable sensors, as well as more room for a wide range of electronics. Second, selecting a vessel with the capability to be heavily modified was vital. Modifying the original build allows the vessel to meet all design requirements as needed.

Electronics

The primary design factors that influenced the selection of onboard electronics were size, power, and interoperability. Firstly, cargo space onboard the vessel was always going to be a major constraint. As a result, choosing electronics which were compact and lightweight was essential. Additionally, reducing mass improves the performance of a boat. Second, one major consideration was choosing a power system for all electronics onboard. Servo motors, for example, generally run on 5-6 Volts, whereas many electronic sensors and micro-controllers require 3.3 or 5 V. Moreover, linking the power networks of servos and computational electronics posed a potential risk to reliability due to the highly variable power draw of servos. Finally, interoperability of sensors, computers, and servos was of paramount importance.

Chapter 2

Design Overview

2.1 Sailing Platform

The boat used in this project, fondly named *Eaglespring*, is a modified version of Tippecanoe Boats' T50 Carbon Fiber Racing Sloop [4]. The boat was purchased as a disassembled kit and assembled using provided instructions.

Figure 2.1: *Eaglespring* - A Modified T50 Carbon Fiber Racing Sloop

Eaglespring is approximately 50 inches long with a 2-foot keel and a 6-foot tall mast. The hull is made from a molded carbon fiber layup comprising the lower portion of the hull. The deck, transom, rudder, and keel were all constructed using 1/16 inch okoume marine plywood. All of these components were clear-coated in

WEST System epoxy resin (105 resin and 205 hardener) [5] and treated with multiple coats of a generic spar urethane exterior wood varnish for long-term durability. The primary assembly method for the vessel is epoxy. Certain epoxy applications which required greater rigidity utilized a thickened mixture of clear epoxy resin and colloidal silica powder (406 filler), affording the joint greater structural integrity. Additionally, phenolic power (407 filler) [5] was used to increase the viscosity of resin batches to improve the quality of the application.

The overall build followed the original T50 assembly instructions fairly closely, with some notable exceptions. *Eaglespring* has an additional port opening in the deck forward of the mast used to house the primary electronics and computing suite. This port was cut using a laser cutter and is sealed using a compressed X-profile O-ring mounted on a 3-D printed port cover. Additionally, although *Eaglespring* uses many of the electronics and servo hardware supplied with the original kit, none of the control architecture from the original designs were retained.

2.2 Electronics

The brain of the electrical system onboard is a Raspberry Pi Pico [6] running Micropython. The Pico uses both of its serial UART pins. UART0 interfaces with an IP68 Waterproof Mini Absolute Encoder [7], which collects wind direction data using a large vane mounted on the bow. UART1 receives command and control data from a radio receiver (TGY-iA6B) [2] linked to a Turnigy TGY-i6S Radio Controller [8]. The Pico uses an I2C data port to interface with an MPU6050 IMU, requiring both a clock line and a data line. Finally, the Pico sends PWM control signals to the two servos, an HS-422 [9], which controls the rudder, and an HS-785HB [10], which controls the sails. Figure 2.2 shows the electrical schematic of all onboard electronics.

2.2.1 Power Distribution

There are two primary power supplies onboard. While they share a common ground, they are otherwise independent of one another. First, a 4-cell AA battery box supplies 6V to both servos and to the TGY-iA6B receiver. Second, a 7.4V Li Ion battery powers the Pico using a DE-SW050 voltage regulator [11], which converts the 7.4V output into 5V. The DE-SW050 has integrated decoupling capacitors, mitigating the need for external capacitors. The Pico also contains a voltage regulator to convert the 5V input to 3.3V, which is the nominal operating voltage of the board. All remaining

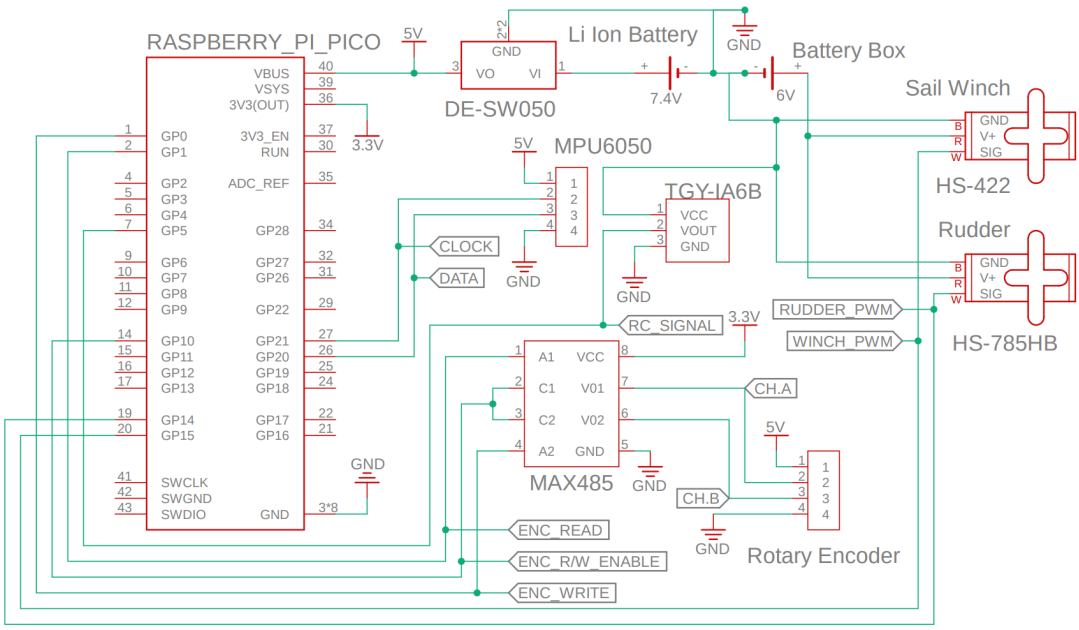


Figure 2.2: Electronics Schematic

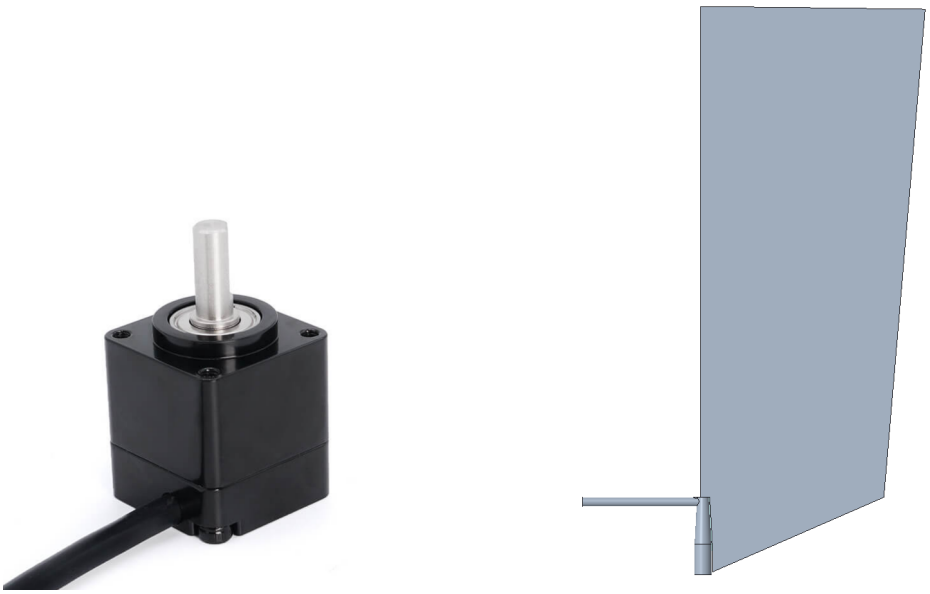
chips and electronics are powered from either the regulated 5V or 3.3V supply.

The primary motivation for splitting the power system into two separate battery supplies was two-fold. First, servos are notorious for causing large power fluctuations within a power supply. If the servos and computational electronics were linked on a single supply, the voltage could become irregular enough to cause problems for the Pico. Second, because 6V is very close to the 5V input for the Pico, a voltage regulator would struggle to make that conversion reliably, especially if the voltage of the AA battery box gradually dropped over the lifespan of the batteries.

Unlike the rest of the non-servo electronics, the radio receiver is powered by the 6V battery box supply. The drivers for this decision were similarly two-fold: First, the receiver was designed to share power supplies with servos, so potential power fluctuations pose less of a risk. Second, the signal range of the receiver is directly proportional to the input voltage. Connecting the receiver to the 6V supply instead of the regulated 5V supply allows for communication at a greater distance.

2.2.2 Wind Sensor

The wind sensor is the primary source of information for the control system. It provides a real-time description of the relative wind velocity relative to the boat, allowing the vessel to trim both the rudder and sail servos to desired angles of attack.



(a) IP68 Waterproof Encoder (b) Custom Wind Vane

Figure 2.3: Wind Vane Sensor Components

The sensor is comprised of a waterproof rotary encoder [7], shown in Figure 2.3a, and a custom 3D printed wind vane, shown in Figure 2.3b. The encoder has a low starting torque for a waterproof device, and the vane was designed to maximize rotational torque without drastically increasing the rotational inertia of the system. An additional set of small 45° fins were later installed at the top, trailing edge of the vane to improve sensitivity to small winds.

From an electronics standpoint, the wind sensor is the most intensive implementation. The Briter encoder takes a 5V input and communicates with external devices using the full-duplex RS-485 serial communication protocol in the MODBUS format. In total, the encoder has 5 wires: 5V, ground, a set-to-zero wire, and two RS-485 wires corresponding to the A and B channels. Unfortunately, traditional microcontrollers, including the Pico, do not have the capability to interpret an RS-485 signal. In order to covert to a usable signal, an RS-485 to TTL (Transistor-Transistor Logic) Converter Chip, the MAX485, was implemented between the encoder and the Pico [12]. The MAX485 enables full-duplex communication. The Pico interfaces with the MAX485 with 3 wires: two data lines (one for reading and one for writing) and a read/write enable line.

The Pico communicates with the encoder by sending periodic requests and reading the reply through a serial UART port. The encoder has a standard 8-byte data request, which is transmitted while the R/W Enable line holds a write state. It takes

25 ms for the encoder to transmit a return package containing the current encoder position, at which point the MAX485 must be set to a read state in order to transmit the package on to the RX pin of the Pico's UART. Once the serial MODBUS package is received by the UART, it is parsed in its raw form to extract the current sensor orientation. Because this data exchange occurs over a longer time interval than the timing of a nominal control loop (2-5 ms), the wind sensor is sampled less frequently than either the Radio Controller or the IMU.

2.2.3 Radio Controller

Radio communication onboard is perhaps the most important piece of electronics, apart from the microcontroller itself. Depending on the current sailing mode, the remote control has roles ranging from full dual-servo control to merely sending tack commands. Regardless, no matter the current state of control, the operator may reclaim full servo control at any time by switching to manual mode, making the radio communication protocol vital to safety and reliability.

The two devices making up the command and control link are a TGY-iA6B Radio Receiver and a TGY-i6S Radio Controller [2][8]. The receiver is powered by the 6V AA battery box in parallel with both control servos. The physical placement of the receiver is of vital importance due to the presence of a carbon fiber hull, a material which heavily obstructs radio waves [13]. To mitigate this, the controller is attached to the underside of the deck with both antennas extending out into the middle of the rear deck port. This placement keeps the antennas away from the carbon fiber walls and maximizes the control range.

Figure 2.4: TGY-iA6B Radio Receiver [2]

The receiver has the capability to transmit PWM (Pulse Width Modulation) signals channel-by-channel on individual ports. It also offers a single-port serial communication protocol known as i-Bus, which encodes all channels onto a single wire. For the purposes of data processing and building an autonomous system, the second option is far preferable. The Pico reads the continuous i-Bus data stream into the board's second UART bus, which is decoded into channel-by-channel outputs by a

custom Python library which was adapted from an existing open-source implementation [14].

2.2.4 IMU

The final source of data available to the microcontroller is from an MPU6050 chip [3], a 6-axis IMU equipped with a 3-axis Gyroscope, a 3-axis Accelerometer, and a temperature sensor. It operates using an I2C communication interface, allowing it to transmit orientation data directly to the Pico. I2C requires two wires, a clock line and a data line, between the microcontroller and the MPU6050. The I2C data is decoded using a pair of open-source custom Python libraries [15][16][17].

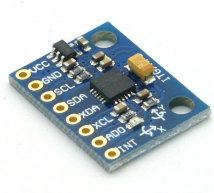


Figure 2.5: MPU6050 6-Axis IMU [3]

In order to ensure that produced IMU data is reliable, the Pico runs a calibration protocol in the first 40 seconds after power-on. This protocol determines "offsets" to subtract from subsequent chip values so that any fluctuations are inertially driven. However, even given an excellent calibration, systemic drift is a major issue for inertial observer chips like the MPU6050, making them unreliable in determining velocity, position, or orientation relative to initial conditions over long time scales. As a result, the current control implementation onboard *Eaglespring* does not utilize the MPU6050.

2.3 Custom Hardware

The two custom-designed physical components onboard are the wind vane and forward port cover. Both were printed from standard PLA filament.

2.3.1 Wind Vane

The wind vane was designed to be as lightweight as possible while maximizing the surface area. In order to achieve this, the vane was printed in two parts: the mounting

head and the thin vane.

The mounting head slides down onto the vertical shaft of the rotary encoder with a tight compression fit. The top of the head is a narrowing cone with a deep ∼2 inch hole through the center for installing the vane portion. Additionally, at the top of the mounting head is a forward-pointing arm which holds counterweights to balance the entire structure. The vane itself is quite simple. It is an extremely thin (0.5 mm) surface designed to respond to even very light winds. At the base of the vane is a narrow ∼2 inch stem which interfaces with the mounting head to join the two. Finally, in order to further improve the vane's sensitivity to light winds, a set of two fins were installed on the top, trailing edge of the vane set at a 45° angle. This addition allows the vane to detect smaller variations in wind direction when the vane is already pointed close to, but not directly into, the wind.

The two components were glued together at the joint to ensure their rigidity. Additionally, very thin PLA has very poor flexural rigidity, making the vane susceptible to deformation and bending in the presence of a side-loading wind. In order to prevent this, a 12 inch 1/16" diameter stainless steel rod was secured to the leading edge of the fin, through the counterweight arm, and along the forward surface of the mounting head. In addition to improving the mechanical lock between the two components, it successfully eliminated the majority of bending actions, even in the presence of very strong winds.

Figure 2.6: Installed Wind Vane

2.3.2 Forward Port Cover

The other custom component created for the vessel is a port cover for the circular forward opening in the deck. This opening was not part of the original T50 build, but was one of several modifications made to accommodate the additional electronics. Additionally, whereas the rear deck port could be sealed with a plastic window, the forward port necessitated an opening to accommodate the wind sensor wire bundle. In order to ensure that this interface is maximally watertight, the cover was designed as a circular plate, the underside of which is lined with an X-profile circular O-ring. There are two small holes in the port cover. The forward-most hole is sized to create a snug interference fit with the encoder's rubber wire bundle, making that interface highly water-resistant. The second hole in the center of the cover is much smaller, and is used to secure the cover in place. A brass eyelet is secured to the bottom of the hull directly below the port opening using high strength epoxy. In order to secure the port cover, a short piece of 90 lb spectra line runs from the eyelet up through the center hole in the port and wraps snugly around a protruding cleat which keeps the line held in tension. This tension pulls the port cover down, compressing the O-ring and creating a seal.

Figure 2.7: Installed Forward Port Cover

Chapter 3

Autonomous Control Design

3.1 Manual Mode

The baseline for sailing control is Manual Mode. This configuration is the simplest to implement, and technically does not even require a microcontroller to be fully functional. While in manual mode, all sensors are ignored by the Pico. The only system inputs are the rudder and winch controls transmitted from the remote control. These signals are decoded from the i-Bus protocol coming from the radio receiver and are subsequently re-scaled to the minimum and maximum allowable servo input range. These controls are written directly to the servos as PWM signals, allowing the operator to manually adjust both servos at will. Subsequent autonomous modes (Pilot-Assist and Autonomous Tack) can be activated from Manual Mode using a switch on the remote control.

3.2 Pilot-Assist Mode

Pilot-Assist is the first level of autonomy onboard the boat. In essence, it is an implementation of Project Objective #1: automated sail trim with manual rudder input. Pilot-Assist mode allows the operator to continue steering the boat manually, but handles the more complicated task of trimming the sails automatically.

3.2.1 Selecting the Sail Angle

In order to implement Pilot-Assist, the controller makes one significant assumption: There exists an optimal angle of attack of the sail with respect to the wind, regardless of the orientation of the hull. Specifically, in this assumption, "wind" refers to the relative wind direction since that is the metric observed by the onboard wind sensor. In practice this assumption is not necessarily true. Determining the optimal sail angle for a given heading requires a decomposition of the sail and keel forces into lift and drag components and solving for the angle that maximizes the forward force on the vessel.

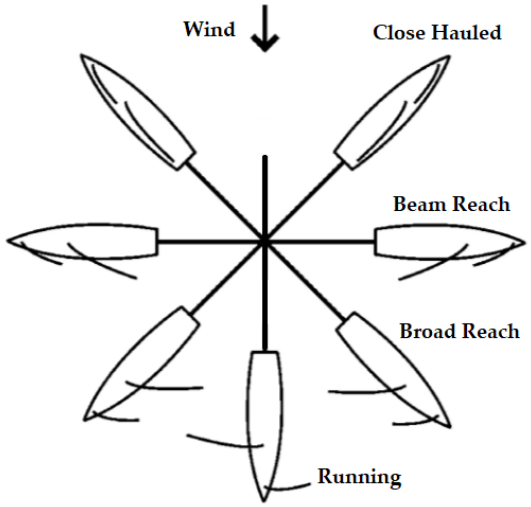


Figure 3.1: Optimal Sail Angle at Various Headings

Figure 3.1 illustrates a simplified representation of the optimal angle of attack of the sails for a wide range of headings. When sailing close hauled, the optimal angle of attack is close to the ship's heading (roughly 60-65°). As the reach becomes broader and the heading shifts downwind, the optimal sail angle of attack increases, eventually converging toward 90° while running. However, the model illustrated in Figure 3.1 shows the angle of attack with respect to *absolute* wind, whereas the onboard controller calculates the trim with respect to *relative* wind. The effect of forward velocity primarily results in a smaller relative angle of attack than the absolute angle of attack, especially at broader reaches where the difference between absolute and relative wind is most pronounced. As a result, the optimal angle of attack with respect to *relative* wind is actually very similar for most headings. Consequently, Pilot-Assist trims the sail in all configurations to match the angle of attack at the optimal upwind tack heading. Applying this angle to headings with a broader reach produces

excellent results for such a simple yet effective model.

3.2.2 Trimming the Sail

In order to set the sail to any relative angle of attack, the onboard controller only needs one source of information: the wind sensor orientation. This sensor outputs the direction of the relative wind with respect to the centerline of the boat.

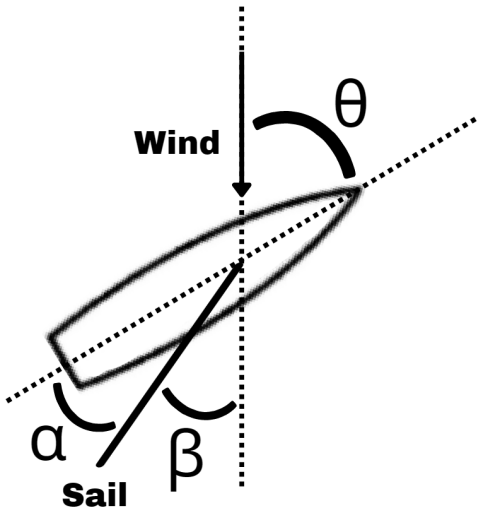


Figure 3.2: Angles Relevant to Computing Sail Trim

Figure 3.2 shows the three primary angles involved in calculating the required sail trim, where θ is the heading of the boat with respect to the wind, β is the target angle of attack for the sail, and α is the sail angle with respect to the boat. θ is continually measured by the wind sensor, and β is a pre-determined target angle of attack. Thus, the required sail trim α can be computed simply as

$$\alpha = \theta - \beta \tag{3.2.1}$$

This relation holds for all θ where

$$\beta \leq \theta \leq 90 + \beta \tag{3.2.2}$$

If θ is less than the target angle of attack β, it is impossible for the sail to achieve the desired angle of attack. In this case, the sail trim α is simply set to 0 so that $\beta = \theta$. In the other case where θ is greater than $90 + \beta$, the sail trim α would have to be greater than 90°, which is neither advisable nor possible given the geometry of

the boat. As such, in any such configuration, the sail is set to the maximum angle (90°).

3.3 Autonomous Tack Mode

Autonomous Tack Mode is the highest level of autonomy achieved by this project. It is a successful implementation of Project Objective #3: Autonomous rudder and sail control for optimal tacking with manual tack commands. This mode implements a heading feedback control system using the rudder and leverages the automatic sail trim from the previous mode to fully automate a tack in either direction. Autonomous Tack mode still requires a human operator to send periodic commands for *when* to come about, but the actions themselves and all servo control are handled autonomously.

3.3.1 Feedback Control Introduction

Figure 3.3 illustrates the block diagram for the feedback system used to model the heading of the boat:

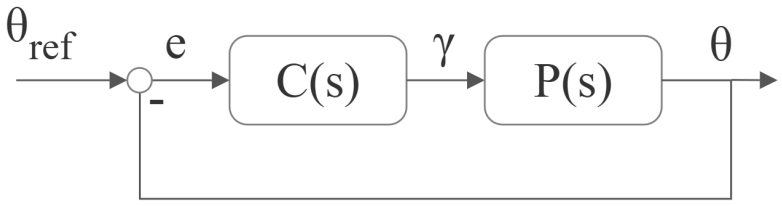


Figure 3.3: Feedback Control Loop Block Diagram

θ_{ref} is the desired reference heading, θ is the actual heading, γ is the rudder deflection angle, and e is the error, the difference between the θ_{ref} and θ. The P(s) block is the "Plant," which represents the physical dynamics of the boat given an input rudder angle γ. The C(s) block is the "Controller," which computes a rudder angle γ in order to reduce the error e. Whereas P(s) is defined by the physical characteristics of the boat, C(s) can have many forms and is designed to stabilize the system.

3.3.2 Determining the Boat Dynamics

The first step in designing an appropriate controller is creating a reliable model for the plant, P(s). A dynamics model that describes the turning behavior of the boat given a rudder deflection is critical to creating a controller with reliable and stable behavior. Without a dynamics model, the controller would have no context for what rudder angle is appropriate for a given configuration.

The Mathematical Framework

In order to mathematically describe the physical behavior of the boat, it is useful to consider the forces created by the primary turning instrument: the rudder. If the rudder is assumed to be a flat plate in an ideal flow, the coefficient of lift, C_L, produced as it moves through the water can be approximated as

$$C_L = 2\pi\gamma \tag{3.3.1}$$

where γ is the angle of rudder in the flow. Similarly, the lift, L, generated by the rudder in the flow can be expressed as

$$L = C * C_L U^2 \tag{3.3.2}$$

where U is the velocity of the boat and C is some constant related to the density of the flow and the surface area of the rudder. Thus, combining equations 3.3.1 and 3.3.2 shows that the lift generated by the rudder is proportional to γU^2.

$$L = C * 2\pi\gamma U^2 \tag{3.3.3}$$

The next step is to determine how the lift generated by the rudder impacts the orientation of the boat as a function of time. Figure 3.4 shows the approximate orientation of the lift force and its position relative to the center of mass of the boat.

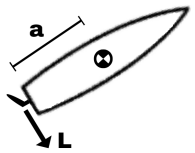


Figure 3.4: Rotation about the Center of Mass Induced by the Rudder

Applying Newton's Second Law of Motion to a torque balance about the center of mass of the boat yields

$$aL = J\theta'' + b\theta' \tag{3.3.4}$$

where a is the moment arm shown in Figure 3.4, J is the rotational inertia, and b is a damping coefficient. Taking the Laplace Transform of the above ODE gives

$$s^2 J\theta + sb\theta = aL \tag{3.3.5}$$

which can be reorganized as

$$\theta = \frac{a}{Js^2 + bs} L \tag{3.3.6}$$

where $s = i\omega$. Equation 3.3.6 is a critical step. It shows that, given the moment arm, damping coefficient, and rotational inertia, the boat heading θ can be computed for all time given an input L. Since L can be approximated by γ and U^2 (Equation 3.3.3), the rudder angle and boat velocity should, at least in theory, be sufficient to determine the heading of the boat.

Once this mathematical model has been determined, the values of each physical parameter must be determined in order to fully describe the system. Although an idealized mathematical solution would be optimal, the computation of rotational inertia, damping coefficient, and true coefficient of lift presents significant challenges. The complexity arises from various factors, such as accounting for the viscosity of the water, developing accurate models for the size and shape of the hull, keel, and keel ballast, and overcoming the idealized assumption that the rudder behaves as a flat plate in an inviscid flow. Additionally, the use of Computational Fluid Dynamics (CFD) may be required to model the problem comprehensively.

Building a Computational Model

An alternative method of solving the dynamics is to collect real-world dynamics data with known initial conditions and subsequently fit the idealized hypothetical model to this data. This strategy has the benefit of being a true-to-reality representation of the physical system, although it is more prone to noise, error, and uncertainty. In order to mitigate these challenges, six independent trials were conducted at different velocities and different rudder deflections. The following steps use the above mathematical structure in conjunction with several data collection trials to build a computational

model for the dynamics of the boat.

To start, substituting the coefficients in Equation 3.3.6 with

$$K = \frac{a}{b}, \tau = \frac{J}{b} \tag{3.3.7}$$

gives

$$\theta = \frac{K}{s(\tau s + 1)} L \tag{3.3.8}$$

In essence, Equation 3.3.6 has been expressed as a time-domain transfer function between L and θ, where K is a gain and τ is a time constant. This has the benefit of describing the plant in terms that are familiar to feedback control. The transfer function between L and θ is simply

$$P(s) = \frac{K}{s(\tau s + 1)} \tag{3.3.9}$$

The next step is to collect real-world dynamics data for standard speed and standard deflection turns, then fit the values of K and τ to best fit the experimental data.

3.3.3 Dynamics Data

Orientation and accelerometer data were collected for six separate turning trials, split over 3 boat speeds and 2 rudder angles. The rudder angles, 27° and 52°, were chosen in advance, as they represent a half deflection and full deflection, respectively, of the rudder servo. The speeds, however, had to be calculated from the accelerometer data as a consequence of pushing the boat by hand. Data was collected on the boat's forward, horizontal, and vertical acceleration as well as the angles of its roll, pitch, and yaw. However, the two primary data sets required for processing were forward acceleration and yaw (also known as heading).

Computing Velocity

Since Lift is proportional to the square of the velocity, knowing the speed of each trial is vital. Figure 3.5 shows an example of the 3-axis accelerometer data collected during a trial with the yaw data overlaid to indicate when the turn was initiated.

Since the boat began at rest in every trial, the forward velocity can be calculated numerically as the trapezoidal integral of the forward acceleration data. Figure 3.6

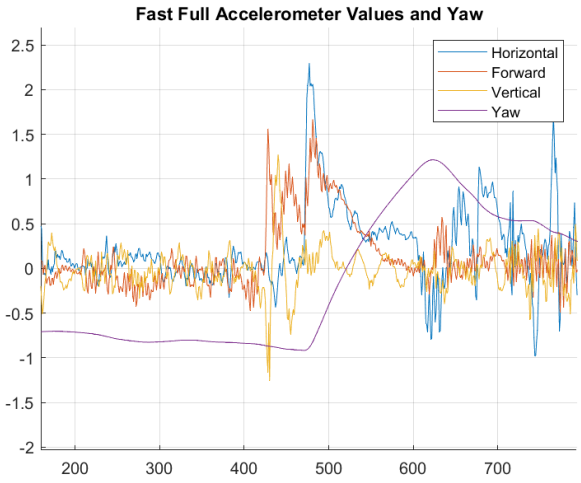


Figure 3.5: Accelerometer and Yaw Data - Fast Speed / Full Deflection

shows the forward acceleration and yaw with the computed velocity overlaid for all six trials. For the purposes of modeling the turning dynamics, the relevant value is the speed attained by the boat *prior* to the start of the turn.

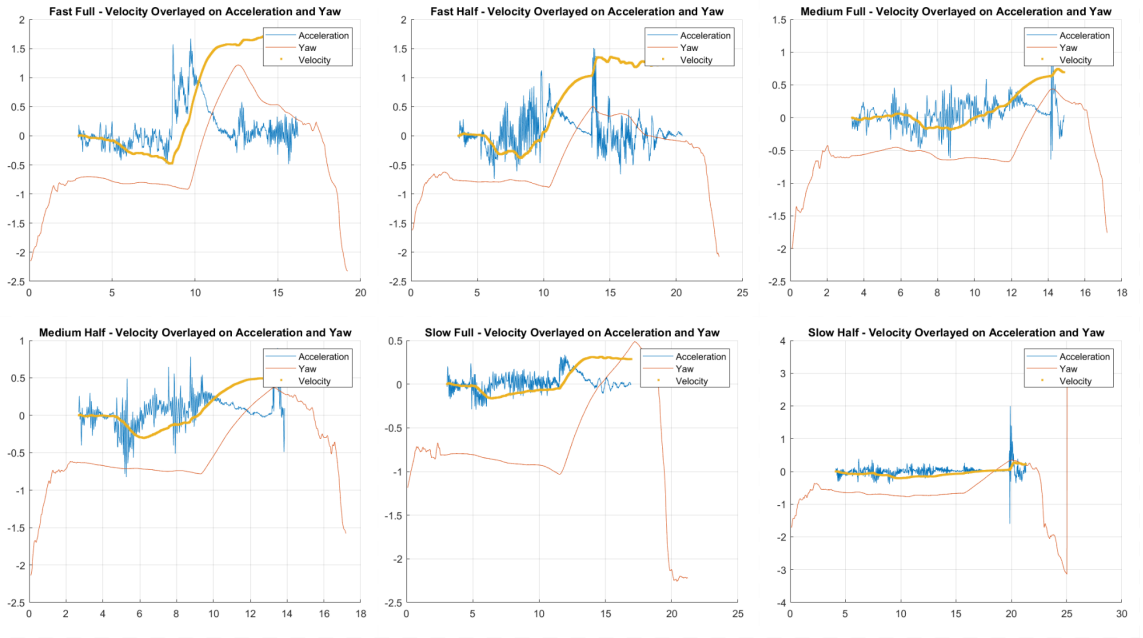


Figure 3.6: Numerical Velocity Calculation for all Six Trials

Drawing from the above data, estimates for the boat speed of each trial are organized in the table below.

Boat Speeds by Trial			
Rudder Deflection	Fast	Medium	Slow
Half	0.38 m/s	0.30 m/s	0.21 m/s
Full	0.48 m/s	0.25 m/s	0.16 m/s

Processing Yaw Data

Figure 3.7 shows the yaw data from all six turns. They are cropped so that t = 0 corresponds to the start of each turn, and the heading at which the turn began is shifted to zero. These translations allow for the trials to be compared side by side. It can clearly be seen that the rate of turn is positively correlated with both boat speed and rudder deflection, as predicted by the mathematical model.

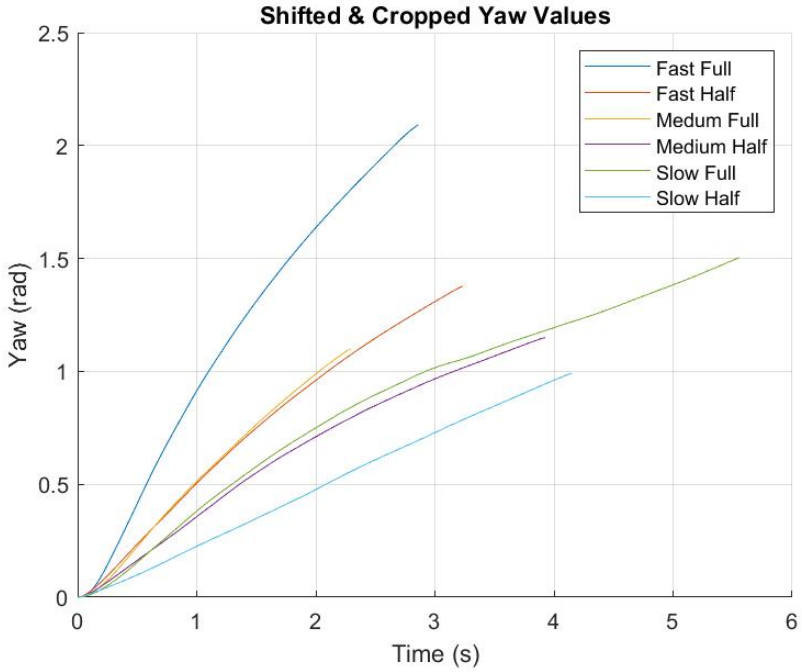


Figure 3.7: Yaw Data from all Six Turning Trials

With the addition of boat velocities in the previous calculations, it is possible to examine the relationship between rate of turn, which can be computed from the slope of each curve in Figure 3.7, rudder angle, and boat speed.

Figure 3.8 shows the turn rate produced by each rudder deflection at each of the three different boat speeds. One key observation is that, whereas the idealized mathematical model assumes a linear relationship between rudder angle and rate of turn (Equation 3.3.3), the true behavior has characteristics of diminishing returns

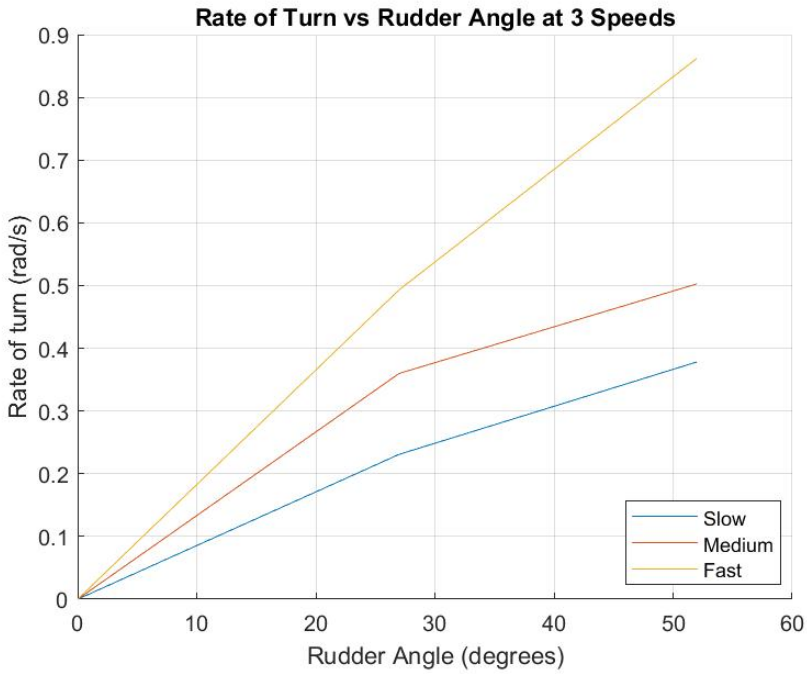


Figure 3.8: Rate of Turn vs Rudder Angle for Each Speed

at large angles. Indeed, this is the behavior which would be expected from a more comprehensive fluid dynamics model. However, Figure 3.8 illustrates that a linear assumption for this relationship is still a fair approximation.

Developing a Transfer Function

The yaw data illustrated in Figure 3.7 is, in essence, six individual step responses to a fixed-input rudder deflection and fixed-value velocity. Each curve can be interpreted as the output of a step into a transfer function for heading. However, because the speeds and rudder deflections were different for each trial, fitting a transfer function to each curve would produce six different sets of coefficients. Revisiting the transfer function format derived earlier in Equations 3.3.8 and 3.3.9:

$$\theta = \frac{K}{s(\tau s + 1)} L \tag{3.3.8}$$

$$P(s) = \frac{K}{s(\tau s + 1)} \tag{3.3.9}$$

The yaw data in Figure 3.7 is a reflection of this transfer function. However, the Lift is embedded in the data in the form of rudder angle and velocity; it is never

expressed explicitly as an input in itself. Since Lift is directly proportional to rudder angle and the square of velocity (Equation 3.3.3), normalizing each yaw curve by its respective rudder angle and squared velocity should cause all six to converge to a single transfer function. The following is a mathematical demonstration of this logic:

$$L = constant * \gamma U^2 \tag{3.3.10}$$

As previously approximated, Lift is proportional to the rudder angle and the square of velocity. Combining this relation with Equation 3.3.8 yields:

$$\theta = \frac{K}{s(\tau s + 1)} * constant * \gamma U^2 \tag{3.3.11}$$

Dividing both sides by γU^2 produces a transfer function for θ normalized by γU^2. Note that the arbitrary constant remaining from the lift computation can be absorbed into the gain K:

$$\frac{\theta}{\gamma U^2} = \frac{K}{s(\tau s + 1)} \tag{3.3.12}$$

The only two variable parameters influencing the turning behavior of the boat are rudder angle and velocity. Thus, if the heading for each trial is normalized by γU^2, all six trials should result in the same coefficients K and τ.

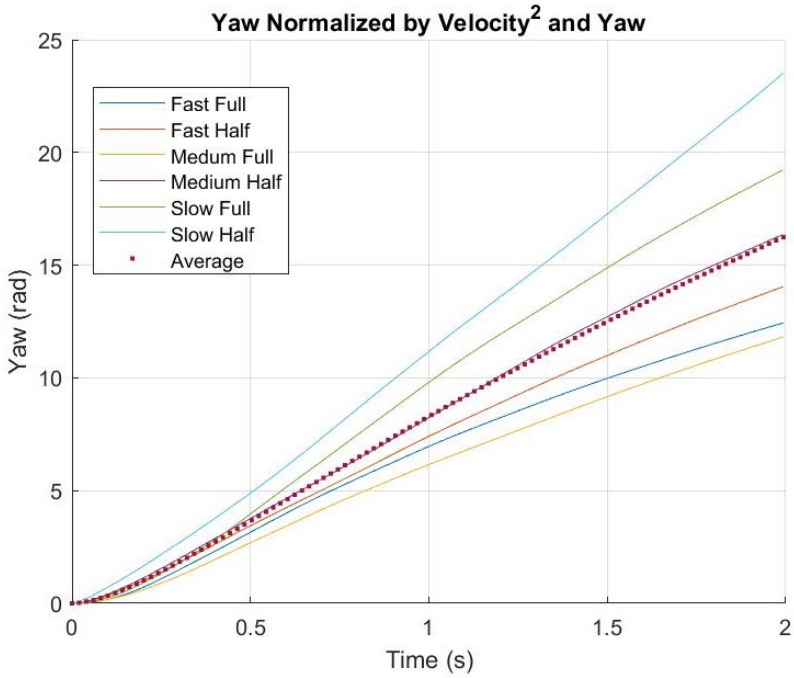


Figure 3.9: Yaw Normalized by Rudder Angle and Velocity Squared

This framework is sufficient to determine the transfer function for the dynamics of the boat. Figure 3.9 shows the yaw values for all six trials, normalized by γU^2. It also shows a curve plotting the average of the six trials, which is the response used to compute the ultimate transfer function coefficients.

Fitting a transfer function of the desired form to the average normalized curve in Figure 3.9 yields the transfer function for the plant:

$$P(s) = \frac{K}{s(\tau s + 1)} \tag{3.3.9}$$

$$K = 8.698$$
$$\tau = 0.059$$

3.3.4 Designing the Controller

A Lead Compensator was used as the controller for the system. There are a number of reasons for this controller selection. First, there is no need to include an integrator in the controller to obtain zero steady-state error. This is because the plant already contains an integrator in the transfer function from rudder angle to the boat's heading. Second, a proportional-only controller, although possible, would likely have smaller phase margin, causing greater oscillations and poor robustness. Third, phase margin could theoretically be improved through the addition of a derivative term (PD controller), but such a controller is not practically realizable and would be inclined to amplify high-frequency noise. A lead compensator, by contrast, adds phase lead without amplifying high-frequency noise, making it a logical choice. The form of a lead compensator is given by:

$$C(s) = K_C \frac{s + z}{s + p} \tag{3.3.13}$$

where K_C is a gain, z is a zero, and p is a pole. These three parameters can be chosen and adjusted as desired to achieve desired performance parameters such as rise time, phase margin, and settling time. After much trial and error, the controller coefficients were chosen to be:

$$K_C = 0.6$$
$$z = 5$$
$$p = 7$$

Figure 3.10 shows a step response of this controller in feedback with the plant model for a set reference heading of 30°. The step response reflects one of the key design decisions made in selecting the controller coefficients: conservative control. The rise time of the step response is on the order of 6 to 8 seconds, but extremely rapid course corrections is not a design driver. Instead, slow and conservative course corrections ensure that *Eaglespring* sails smoothly and with stability.

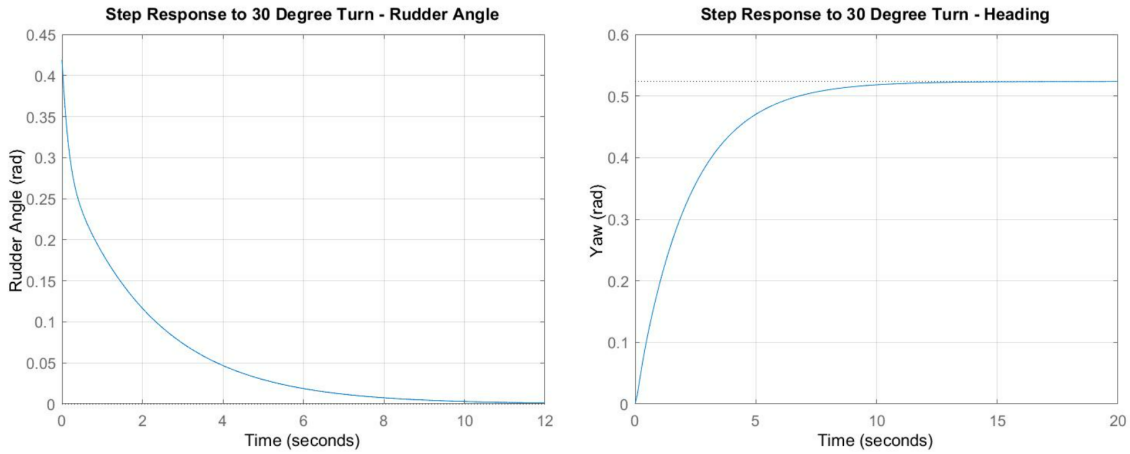


Figure 3.10: Step Response to a 30° Reference Heading

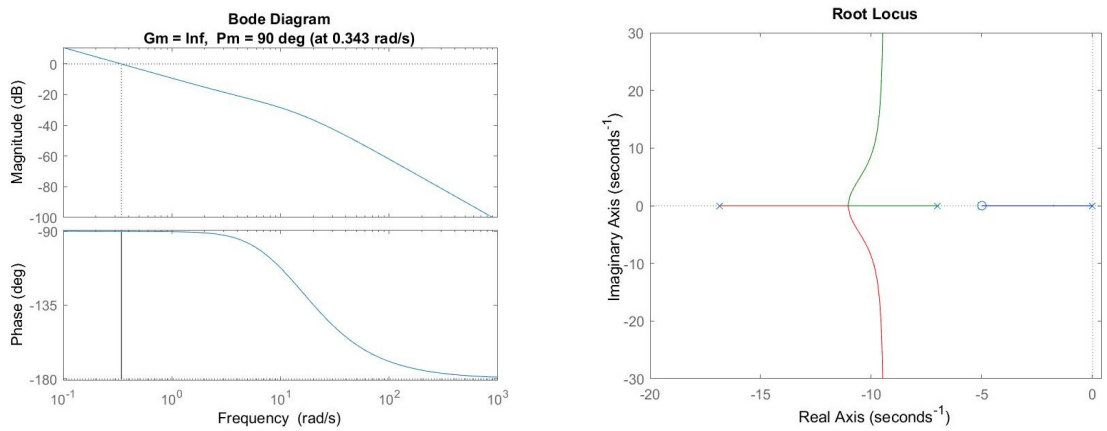
The Bode Diagram and Root Locus Diagram (Figure 3.11) were also used to verify the stability and reliability of the control loop. Specifically, the Bode Diagram shows a Phase Margin of 89.9° located at a crossover frequency of 0.457 rad/s. The crossover frequency is indicative of the types of timescales which the controller is sensitive to. Specifically, fluctuations on the order of ∼2 seconds are well-attenuated, which is ideal for a wind-based system like *Eaglespring*.

3.3.5 Implementing the Controller

Continuous-Time Solution

In order to implement the controller, it was first converted into a continuous-time state space system:

$$\begin{aligned} \dot{x} &= Ax + Be \\ u &= Cx + De \end{aligned} \tag{3.3.14}$$



(a) Loop Gain Bode Diagram (b) Loop Gain Root Locus Diagram

Figure 3.11: Loop Gain Characteristic Plots

where x is an internal state vector, e is the error in the heading (the controller input), u is rudder angle γ (the controller output), and A, B, C, and D are the state space matrices. Because the controller is a one-dimensional model, both the state vector and the state space matrices are simply scalar values. Computing A, B, C, and D from the Laplace-Space controller (Equation 3.3.13) yields:

$$\text{A} = \text{-7}$$
$$\text{B} = 1$$
$$\text{C} = \text{-1.2}$$
$$\text{D} = 0.6$$

Discrete-Time Solution

A continuous-time system, though useful for designing a controller, is not practical for implementation. Real-world computers have discrete time steps and complete operations at regular, or in some cases, irregular intervals. The Pico is one such device with a discrete-time computing architecture. As a result, the state space system must be discretized in order to implement the controller.

To convert a continuous-time state space system into a discrete-time system, the continuous derivative represented by $\dot{x}$ must be converted into a version which is sampled at discrete time intervals Δt. Letting $t_n = n\Delta t$, $x_n = x(t_n)$, $e_n = e(t_n)$, and $u_n = u(t_n)$, the time derivative $\dot{x}$ can be approximated by

$$\dot{x} = \frac{dx}{dt} \approx \frac{x_{n+1} - x_n}{\Delta t} \tag{3.3.15}$$

where Δt is the sampling time between the state at step n and step $n+1$. The state equation for $\dot{x}$ then becomes

$$\frac{x_{n+1} - x_n}{\Delta t} = Ax_n + Be_n \tag{3.3.16}$$

which can be rewritten as

$$x_{n+1} = (I + \Delta tA)x_n + \Delta tBe_n \tag{3.3.17}$$

Consequently, letting $A_d = I + A\Delta t$ and $B_d = B\Delta t$, the continuous-time state space system (Equation 3.3.14) can be written in the approximate discrete form

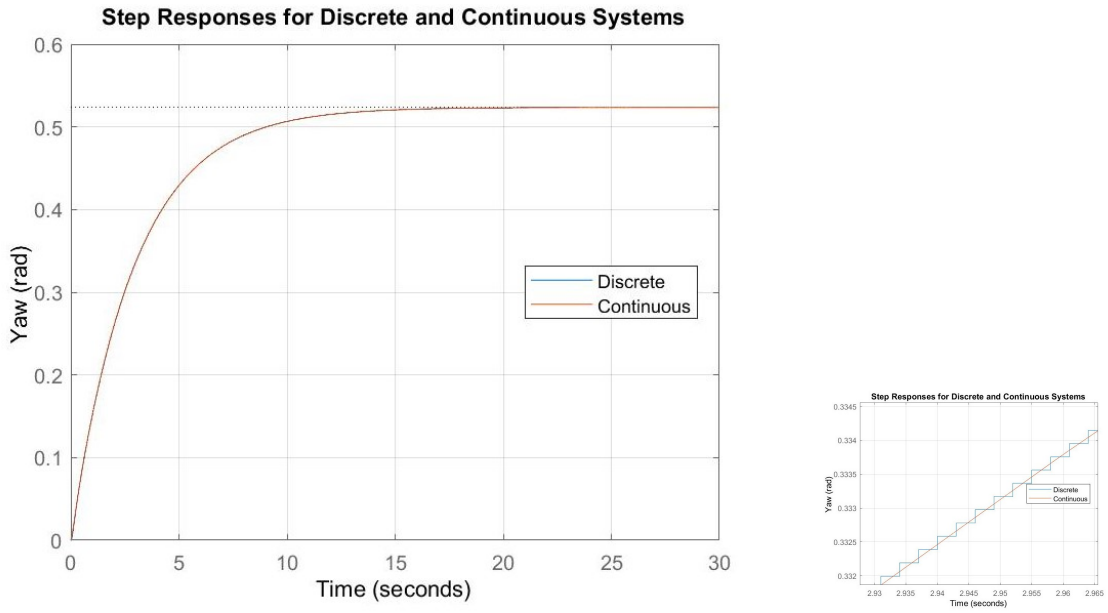
$$\begin{aligned} x_{n+1} &= A_d x_n + B_d e_n \\ u_n &= Cx_n + De_n \end{aligned} \tag{3.3.18}$$

The final step in implementing the discrete system is determining Δt. For certain chips and microcontrollers, especially those designed for discrete time computations, this value is predetermined and constant. However, because the Pico is processing the RC signal, encoder and IMU inputs, and servo outputs simultaneously, the characteristic time step Δt is variable, ranging from 2 to 6 milliseconds. Consequently, for each loop in the controller, the Pico re-computes the values of A_d and B_b with the true Δt from the most recent iteration.

Depending on the nature of the system and variability of the time step, the discretized form of the transfer function may not perform as ideally as the continuous-time case. In order to determine whether the variable time step would allow the system to maintain desired behaviors, several simulations were conducted to evaluate the behavior of the discretized model.

Simulations of the Discrete-Time System

The first simulation tested the constant-Δt discrete state space system. A step response of a 30° turn at medium speed was simulated for the continuous-time (Equation 3.3.14) and discrete-time (Equation 3.3.18) systems. Figure 3.12a shows the two step responses overlaid. Figure 3.12b shows a close-up of one portion of the curve, where the discretization can be seen clearly. Overall, the step responses show that, for the constant-Δt case, the discrete time system is very reliable. This simulation takes the first step toward validating the conversion of the continuous system to a discrete form.



(a) Step Response (b) Close-Up

Figure 3.12: Continuous and Discrete Time Step Responses

However, the Pico is not a fixed-Δt microcontroller, instead running with a variable-Δt discretization. Depending on the robustness of the controller, this can introduce potential instabilities, necessitating further simulations to verify its stability.

To fully test the viability of the controller on the Pico, a further simulation testing a variable-Δt case was conducted. To accomplish this, each state-space system was simulated manually with a randomized variable for Δt at each loop iteration. First, the constant-Δt system was simulated as a control. For this simulation, the input error, e, was kept constant to observe how the controller would converge to a single output u. Next, the same simulation was conducted with a randomized Δt at each step. Finally, the same two simulation trials were conducted (one variable Δt and one constant Δt) with a continuously changing input e.

The constant error simulation represents the case where the heading error does not decrease despite attempts by the controller to turn. One potential cause for this scenario would be a lack of wind resulting in zero boat velocity. Another possibility is turning within a changing wind, in which case the boat may be required to continue turning until a suitable tack can be established. The second case, varying error, simulates a successful turn in which the controller must reduce the rudder angle as the boat approaches the target heading. These two scenarios are contrasting sailing conditions, each of which is designed to reveal the behavior of the controller. All four simulations (Figure 3.13) demonstrate that, although the variable-Δt simulations

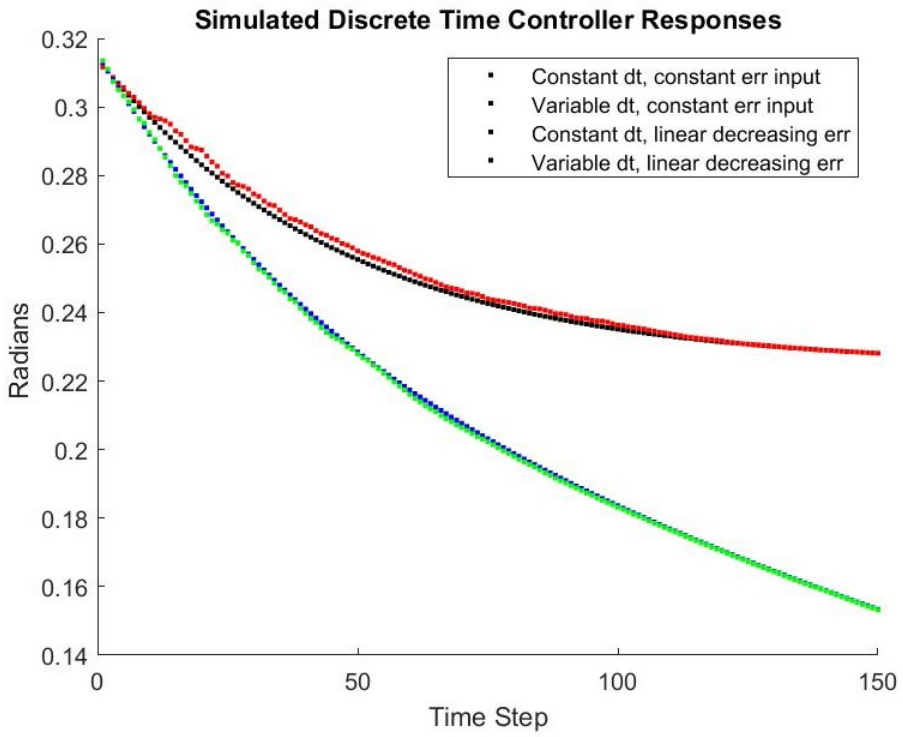


Figure 3.13: Simulated Discrete-Time Controller Responses

have more fluctuations in the curve, the presence of a non-uniform time step does not negatively impact the performance of the system.

Following these simulations, *Eaglespring* was tested on the water in real-world conditions. The vessel performed as expected, handling a wide range of wind conditions reliably. The simulations laid the foundations of confidence in the capability and robustness of the physical system.

Chapter 4

Conclusion

4.1 Summary

In total, this project saw the design, development, construction, programming, and testing of a semi-autonomous sailing platform called *Eaglespring*. There was a significant design component, as demonstrated by the custom physical components, the electrical architecture, and the feedback control system.

Starting with an existing assembly manual for a model sailboat, the T50MOD, a complete vessel with several notable design modifications was constructed. Subsequently, the boat was outfitted with a range of additional components, several custom designed, in order to facilitate the continuous collection of wind data onboard. A new electrical system was implemented to support a wide variety of chips and components. A Raspberry Pi Pico was installed with a custom control algorithm, enabling the boat to sail in three independent levels of autonomy: Manual Control, Pilot-Assist, and Full Autonomy. The control algorithm's design was informed by an array of real-world dynamics data coupled with a mathematical framework in order to accurately predict the dynamics of the boat. This control system was subsequently tested through a series of computational simulations to evaluate the capacity of the controller to manage variable-time step discretization.

Eaglespring ultimately conducted real-world testing on Lake Carnegie in Princeton, NJ. The vessel demonstrated consistent reliability in all three sailing modes, even in highly variable wind conditions. The project is considered to be a resounding success, while leaving open a number of opportunities for future additions, improvements, and applications.

4.2 Future Work

Eaglespring is a highly flexible platform that is well-suited to future addition and modification. As such, the opportunities for future work are deep and wide-ranging.

4.2.1 Onboard Computing

One of the most potent opportunities for future work is an improvement in onboard computing capabilities. When this project began, the original objective was to implement all onboard computation and control using a BeagleBone Blue computer [18]. The BeagleBone is a fully integrated Linux computing environment. It has greater capabilities for sensing, computation, and logging than the Raspberry Pi Pico by orders of magnitude. Unfortunately, the implementation of the BeagleBone Blue faced significant challenges, and the board was never used successfully in this project. The existing sensing and control system, while manageable for the Pico, brings the microcontroller to its functional limit. In order to make significant additions to the onboard sensing or computation capabilities, a more powerful microcontroller or computer would be necessary. Alternatively, the Pico could be retained in future iterations to leverage its existing integration with some onboard sensors, while offloading the feedback control computation to a separate device.

4.2.2 Additional Sensors

Sensing capabilities offer the richest opportunities for future improvements. There are few limits to the number of possible onboard sensors, each one offering its own potential advantages.

The addition of a wind speed anemometer to complement the existing wind direction sensor would drastically improve the possibilities for autonomous control. Because the current implementation has no sense for wind speed, it lacks a great deal of valuable context which would allow it to make better control decisions. Another option would be to remove the wind vane style sensor altogether, opting for an Ultrasonic Wind Sensor. Such a system would be capable of collecting both speed and direction measurements, likely with greater precision than analogue alternatives. This option was explored by this project, but was ultimately not pursued for time and feasibility reasons.

Another potential improvement in sensing would be the introduction of an onboard GPS (Global Positioning System). This would allow the vessel to achieve Project

Objective #5: Full Autonomy - Rudder/Sail control and navigation. Specifically, the navigation component of control can only be achieved if the boat has an input which provides the current spatial position. This could also be achieved through relative positioning to a fixed shore base station. Additionally, depending on the level of precision, a GPS would enable approximate velocity measurements through differential GPS.

Another sensor which couples nicely with a GPS is an IMU (Inertial Measurement Unit). *Eaglespring* does have an implemented IMU that is continuously observable by the Pico. However, due to the limitations of a strictly inertial system without the ability to eliminate or mitigate drift over time, the data from the IMU is not used in the existing control system. Finding algorithmic implementations of this IMU, perhaps as a supplement to other positional sensors such as GPS, could provide greater capabilities and is left open as future work.

A small digital camera installed on the bow could be another sensing improvement. Data from this camera could be used to determine the pitch, roll, and yaw of the boat by tracking key features such as shoreline landmarks or the horizon. A camera could also inform navigation decisions or facilitate obstacle avoidance.

Finally, addition of a below-deck water sensor and alarm would be in the interest of protecting all other electronics. In the event of water ingress, preventing sensitive electronics from serious damage is vital. Although this improvement would not impact autonomous sailing capabilities, it would be highly advisable.

4.2.3 Improvements on Existing Features

Eaglespring has many successful features. However, there are some notable areas for improvements on existing systems. The most significant of these improvements is the forward port cover. In spite of a compression fit of a rubber O-ring onto the deck, the interface never achieved a truly watertight fit. The primary challenge in sealing this opening is the need to accommodate the wire bundle running from the rotary encoder above deck. Some potential methods for improving this interface against water ingress include installation of a forward splash guard or replacement of the 3D printed cover with a plastic sliding cover to mimic the aft port cover.

Another potential improvement upon the existing platform is a better management of the center of mass. Specifically, organizing components farther aft in the vessel. With the addition of sensing and computing equipment through a forward port opening, the current balance of the vessel is slightly front heavy. To whatever

extent this ballast may be shifted farther aft, this adjustment would reduce the tendency for the boat to pitch forward during sharp turns and reduce the volume of water flowing up over to top of the deck.

4.3 Personal Reflection

This project has been a fantastic conclusion to an incredible four years in the Princeton School of Engineering. It was an opportunity to apply a wide range of skills and knowledge accumulated over four years of education including fluid dynamics, circuitry, CAD design, data processing, robotics, and programming. It was also a somewhat poetic tribute to a lifelong passion for sailing.

Bibliography

[1] J. Kimball, *Physics of Sailing*. Taylor and Francis Group, LLC, 2010.

[2] HobbyKing, "Turnigy tgy-ia6b receiver." Online, 2023. Accessed on November 1, 2022. https://hobbyking.com/en_us/turnigy-ia6b-v2-receiver-6ch-2-4g-afhds-2a-telemetry-receiver-w-sbus.html.

[3] ElectronicWings, "Mpu6050 accelerometer and gyroscope sensor guide with arduino programming." Online. Accessed on April 1, 2023. https://www.electronicwings.com/sensors-modules/mpu6050-gyroscope-accelerometer-temperature-sensor-module.

[4] Tippecanoe Boats, "T50 carbon fiber racing sloop." Online, 2023. Accessed on November 1, 2022. https://tippecanoeboats.com/t50-carbon-fiber.

[5] WEST System, "West system epoxy." Online, 2023. Accessed on November 1, 2022. https://www.westsystem.com/.

[6] Raspberry Pi, "Raspberry pi pico." Online, 2023. Accessed on November 1, 2022. https://www.raspberrypi.com/products/raspberry-pi-pico/.

[7] Briter, "Ip68 waterproof mini absolute encoder." Online, 2023. Accessed on November 1, 2022. https://briterencoder.com/product/ip68-waterproof-mini-absolute-encoder/.

[8] HobbyKing, "Turnigy tgy-i6s digital proportional radio control system." Online, 2023. Accessed on November 1, 2022. https://hobbyking.com/en_us/turnigy-tgy-i6s-mode-1-digital-proportional-radio-control-system-black.html.

[9] Hitec, "Hs-422 deluxe standard servo." Online, 2023. Accessed on November 1, 2022. https://hitecrcd.com/products/servos/analog/sport-2/hs-422/product.

[10] Hitec, "Hs-785hb karbonite, 3.5 turn winch servo." Online, 2023. Accessed on November 1, 2022. https://hitecrcd.com/products/servos/analog/boat-analog/hs-785hb-3.5/product.

[11] Dimension Engineering, "5v 1a switching voltage regulator." Online, 2023. Accessed on April 1, 2023. https://www.dimensionengineering.com/products/de-sw050.

[12] Open Impulse, "Max485 rs485 to ttl converter." Online, 2023. Accessed on November 1, 2022. https://www.openimpulse.com/blog/products-page/product-category/max485-rs-485-to-ttl-converter-module-3/.

[13] D. Munalli, G. Dimitrakis, D. Chronopoulos, S. Greedy, and A. Long, "Electromagnetic shielding effectiveness of carbon fibre reinforced composites," *Composites Part B: Engineering*, vol. 173, p. 106906, 2019.

[14] Penguin Tutor, "Radio control for the raspberry pi pico - using i-bus." Online, January 8, 2022. Accessed on November 1, 2022. http://www.penguintutor.com/news/electronics/rc-pico-ibus.

[15] M. Shilleh, "Connect mpu 6050 to raspberry pi pico w." Online, January 24, 2023. Accessed on April 1, 2023. https://www.hackster.io/shilleh/connect-mpu-6050-to-raspberry-pi-pico-w-7f3345.

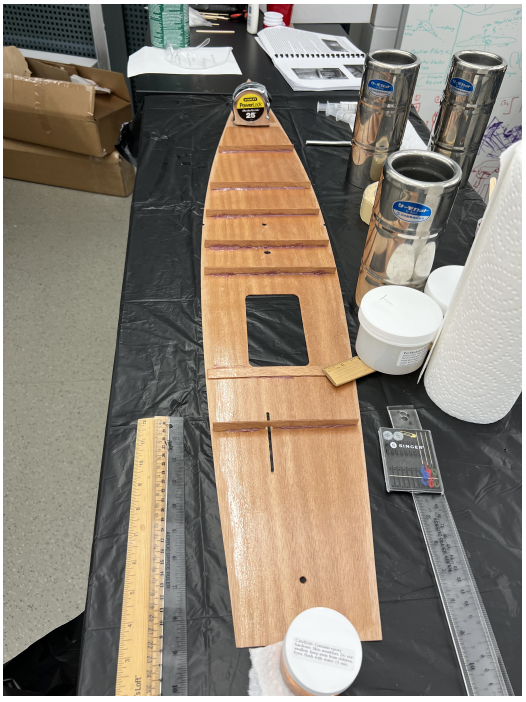
[16] P. Hinch and S. Plamauer, "vector3d.py 3d vector class for use in inertial measurement unit drivers." Online. Accessed on April 1, 2023. https://github.com/shillehbean/youtube-channel/blob/main/vector3d.py.

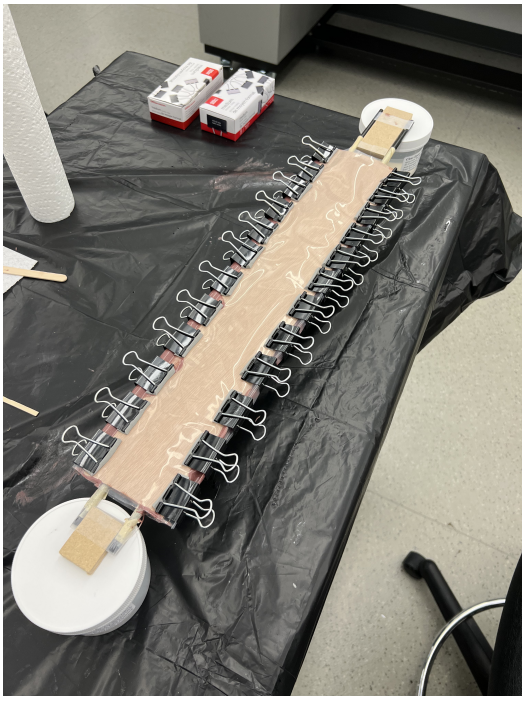
[17] P. Hinch and S. Plamauer, "imu.py micropython driver for the invensense inertial measurement units." Online. Accessed on April 1, 2023. https://github.com/shillehbean/youtube-channel/blob/main/imu.py.

[18] Beagleboard, "Beaglebone blue." Online. Accessed on November 1, 2022. https://beagleboard.org/blue.

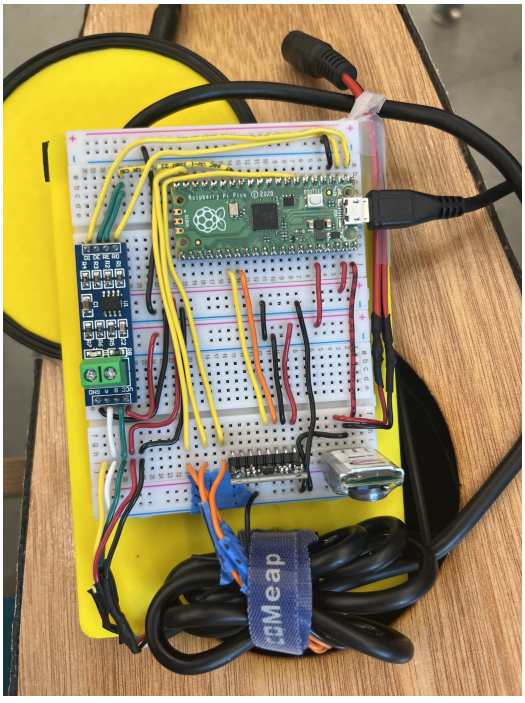
Appendix A

Photos






PRINCETON
UNIVERSITY



Appendix B

Python Code - Main Program

```
# Overall Architecture for Control

# Include libraries ########
# Standard Libraries
import time
from machine import Pin, PWM, I2C
import machine
import utime
# Custom Libraries
from ibus import IBus
from imu import MPU6050

# Initialize servo output ########
rud = PWM(Pin(15))
rud.freq(50)
win = PWM(Pin(14))
win.freq(50)
# Initialize min and max values
win_min = 3680
win_max = 6590
```

```
rud_min = 3050
rud_max = 6950
# Initial servo values
rud.duty_u16(5000)
win.duty_u16(int((win_max-win_min)/4) + 5000)

# Initialize I-BUS Input ########
ibus_in = IBus(1)

# Initialize Encoder Input ########
# GPIO pin 10 configuration
gpio10 = machine.Pin(10, machine.Pin.OUT)
# UART0 configuration
uart0 = machine.UART(0, baudrate=9600, bits=8,
   parity=None, stop=1, tx=machine.Pin(0),
   rx=machine.Pin(1), timeout=3)
# Modbus RTU request message format
# Address (1 byte) + Function (1 byte) + Starting address
   (2 bytes) + Quantity of registers (2 bytes) + CRC (2
   bytes)
request = bytearray([0x01, 0x03, 0x00, 0x00, 0x00, 0x01,
   0x84, 0x0A])
# Configure wait duration for encoder
sent_time = time.ticks_ms()
fin_sent = time.ticks_ms()
wait_time = 25
send_time = 10
enc_mode = 0 # 0 is sending, 1 is receiving
enc_sent = 0 # has the message been sent already?
enc_dir = 0 # Current encoder direction
```

```
response = b""

# Target sail angle of attack
target_aoa = 60
# Target boat heading with respect to wind
target_heading = 70
# Initialize control loop state variable
state_var = 0
# Initialize control loop state space
A = -7
B = 1
C = -1.2 # -1.6
D = 0.6 # 0.8
prev_control_time = time.ticks_ms()

# Initialize IMU Chip ########
i2c = I2C(0, sda=Pin(20), scl=Pin(21), freq=400000)
imu = MPU6050(i2c)

# Use LED to indicate calibration
LED = machine.Pin("LED", machine.Pin.OUT)
#LED.on()

pre_cal_start = time.ticks_ms()
pre_cal_time = 1000 # Give user 10 seconds to bring boat
   to rest
# Wait 10 seconds after power on before beginning
   calibration
while time.ticks_ms() - pre_cal_start < pre_cal_time:
    LED.on() # turn on LED
```

```
74      utime.sleep(1)
75      LED.off()
76      utime.sleep(1)
77
78  cal_start = time.ticks_ms()
79  cal_time = 500 # Calibrate for 40 seconds
80
81  LED.on() # Signal start of calibration
82
83  count_loops = 0
84  imu_offsets = [0 for x in range(6)]
85
86  while time.ticks_ms() - cal_start < cal_time:
87      imu_offsets[0] += imu.accel.x
88      imu_offsets[1] += imu.accel.y
89      imu_offsets[2] += imu.accel.z
90      imu_offsets[3] += imu.gyro.x
91      imu_offsets[4] += imu.gyro.y
92      imu_offsets[5] += imu.gyro.z
93
94      count_loops += 1
95
96  for i in range(6):
97      imu_offsets[i] = imu_offsets[i] / count_loops
98
99  LED.off() # Calibration complete
100 #last_imu_time = time.ticks_ms()
101
102 # Trim Sail function ########
103 def trim_sail(enc_dir):
```

```
104     # Compute desired sail orientation from encoder
105     # If 181-360: convert to 0-180
106     #print(enc_dir)
107     temp_enc_dir = enc_dir
108     if temp_enc_dir > 180:
109         temp_enc_dir = 360 - temp_enc_dir
110     # If less than target_aoa, set to 0
111     if temp_enc_dir <= target_aoa:
112         win_angle = 0
113     # Above target_aoa: enc_dir - target_aoa
114     elif temp_enc_dir > target_aoa:
115         win_angle = temp_enc_dir - target_aoa
116 
117     if win_angle > 90: win_angle = 90
118 
119     #print(enc_dir, win_angle)
120 
121     # Convert winch angle to winch input
122     # 0 angle -> win_max, 90 angle -> win_min
123     slope = (win_min - win_max) / 90
124     # yintercept = win_max
125     win_val = round(win_max + (slope * win_angle))
126     return win_val
127 
128 # Main Method ########
129 while True:
130 
131     # Sample Sensors ########
132     # I-BUS
133     res = ibus_in.read()
```

```
134     #print ("CH 1 {} Ch 2 {} Ch 3 {} Ch 4 {} Ch 5 {} Ch 6
            {} Ch 7 {} Ch 8 {}".format(
135     #     #res[0],    # Status
136     #     IBus.normalize(res[1]),
137     #     IBus.normalize(res[2]),
138     #     IBus.normalize(res[3]),
139     #     IBus.normalize(res[4]),
140     #     IBus.normalize(res[5], type="switch"),
141     #     IBus.normalize(res[6], type="switch"),
142     #     IBus.normalize(res[7], type="switch"),
143     #     IBus.normalize(res[8], type="switch")),
144     #     end="")
145     #print (" - {}".format(time.ticks_ms()))
146     rud_val = IBus.normalize(res[1], type="rudder")
147     win_val = IBus.normalize(res[3], type="winch")
148     s1_val = IBus.normalize(res[5], type="switch")
149     s2_val = IBus.normalize(res[6], type="switch")
150     s3_val = IBus.normalize(res[7], type="switch")
151
152     # Encoder
153     # Start of sending mode. Haven't send request
154     if enc_mode == 0 and enc_sent == 0:
155         gpio10.high()  # Set GPIO pin 10 high for writing
156         uart0.write(request)
157         enc_sent = 1
158         sent_time = time.ticks_ms()
159     # In sending mode after request is sent
160     elif enc_mode == 0 and enc_sent == 1:
161         if time.ticks_ms() - sent_time >= 10:
```

```
162             gpio10.low()  # Set GPIO pin 10 low for
                    reading
163             enc_mode = 1
164             fin_sent = time.ticks_ms()
165     # In receiving mode before reading the response
166     elif enc_mode == 1:
167         if time.ticks_ms() - fin_sent >= 25:
168             if uart0.any():
169                 response = uart0.read(uart0.any())
170
171             # Print the received data
172             #print("Received data:", response.hex())
173             #print(fin_sent - sent_time, time.ticks_ms()
                    - sent_time)
174             if len(response) == 7 and
                    response.startswith(b'\x01\x03\x02'):
175                 enc_dir = round(int("0x" +
                        response.hex()[7] + response.hex()[8]
                        + response.hex()[9]) * (359/1024))
176                 #print(enc_dir)
177
178             enc_mode = 0
179             enc_sent = 0
180
181     #print(enc_dir)
182
183     # IMU
184     #imu_accel = imu.accel.z - imu_offsets[2]
185     #imu_vel +=
            imu_accel*(time.ticks_ms()-last_imu_time)/1000
```

```
186     #last_imu_time = time.ticks_ms()
187     #print(imu_vel)
188     #imu_roll = imu.gyro.z - imu_offsets[5]
189     #print(round(imu_accel,3), round(imu_roll,3))
190
191     # Branch system mode ########
192
193     # Mode = Manual Control ########
194     if s2_val < 0.1:
195
196         # Verify valid inputs (rud 3050-6950, win
              3680-6590)
197         if rud_val >= rud_min and rud_val <= rud_max and
              win_val >= win_min and win_val <= win_max:
198             # Fixed Rudder angle testing
199             #if s3_val > 0.6: rud_val = int(5000 +
                  ((rud_max-rud_min)/4))
200             #elif s3_val > 0.1: rud_val = rud_max
201             # Direct write controls to servo
202             rud.duty_u16(rud_val)
203             win.duty_u16(win_val)
204
205     # Mode = Pilot Assist
206     elif s2_val  < 0.6:
207         # Verify valid rudder input (3050-6950)
208         if rud_val >= rud_min and rud_val <= rud_max:
209             # Direct write rudder to servo
210             rud.duty_u16(rud_val)
211
212         # Verify valid encoder input
```

```
213             if enc_dir >= 0 and enc_dir < 360:
214                 win_val = trim_sail(enc_dir)
215
216                 # Verify valid winch input
217                 if win_val >= win_min and win_val <= win_max:
218                     # Direct write winch to servo
219                     win.duty_u16(win_val)
220
221         # Mode = Autonomous Tack ########################
222         elif s2_val  < 1.1:
223             # Check if tack mode matches current tack
224             # Desired Tack = Port
225             if s1_val > 0.5 and enc_dir > 180:
226                 # Update state variable and start time for
                        new control on next tack
227                 state_var = 0
228                 prev_control_time = time.ticks_ms()
229
230                 # Winch and rudder to come-about-left
231                 rud_val = rud_min
232                 rud.duty_u16(rud_val)
233                 # Trim the sail
234                 # Verify valid encoder input
235                 if enc_dir >= 0 and enc_dir < 360:
236                     win_val = trim_sail(enc_dir)
237
238                     # Verify valid winch input
239                     if win_val >= win_min and win_val <=
                            win_max:
240                         # Direct write winch to servo
```

```
241                     win.duty_u16(win_val)
242             # Continue
243             continue
244
245         # Desired Tack = Starboard
246         if s1_val <= 0.5 and enc_dir <= 180:
247             #print(enc_dir)
248             # Update state variable and start time for
                    new control on next tack
249             state_var = 0
250             prev_control_time = time.ticks_ms()
251
252             # Winch and rudder to come-about-left
253             rud_val = rud_max
254             rud.duty_u16(rud_val)
255             # Trim the sail
256             # Verify valid encoder input
257             if enc_dir >= 0 and enc_dir < 360:
258                 win_val = trim_sail(enc_dir)
259
260                 # Verify valid winch input
261                 if win_val >= win_min and win_val <=
                        win_max:
262                     # Direct write winch to servo
263                     win.duty_u16(win_val)
264             # Continue
265             continue
266
267         # If encoder 181-360: convert to 0-180
268         #print(enc_dir)
```

```
269         temp_enc_dir = enc_dir
270         if enc_dir > 180:
271             temp_enc_dir = 360 - temp_enc_dir
272
273         #print(temp_enc_dir)
274         # Compute heading error from target encoder angle
275         err = (target_heading -
               temp_enc_dir)*(2*3.1415/360)
276
277         # Compute rudder input
278         # Compute discretized state space coefficients
279         dt = (time.ticks_ms() - prev_control_time) / 1000
280         if dt > 1: dt = 0.005
281         prev_control_time = time.ticks_ms()
282         Ad = 1 + A*dt
283         Bd = B*dt
284
285         # Update the state variable
286         state_var = Ad * state_var + Bd * err
287         # Compute the output
288         rud_angle = C * state_var + D * err
289
290         #print(rud_angle)
291
292         rud_angle = rud_angle * (360 / (2*3.1415))
293         #print(rud_angle)
294         if s1_val > 0.5: rud_angle = -rud_angle
295         if rud_angle > 52: rud_angle = 52
296         elif rud_angle < -52: rud_angle = -52
297
```

```
298             # Conver rudder angle to rudder servo input (52
                   deg -> max deflection)
299             slope = (rud_max - rud_min) / 104
300             # yintercept = average rud_max and rud_min
301             rud_val = round((rud_max + rud_min)/2 + (slope *
                   rud_angle))
302             #print(rud_val)
303
304             # Verify valid rudder input
305             if rud_val >= rud_min and rud_val <= rud_max:
306                 # Direct write rudder to servo
307                 rud.duty_u16(rud_val)
308             # Trim the sail
309             # Verify valid encoder input
310             if enc_dir >= 0 and enc_dir < 360:
311                 win_val = trim_sail(enc_dir)
312
313                 # Verify valid winch input
314                 if win_val >= win_min and win_val <= win_max:
315                     # Direct write winch to servo
316                     win.duty_u16(win_val)
```

Appendix C

Python Code - IBus Library

This library is a modified version of an existing architecture [14].

```
1  from machine import UART
2
3  # Works by connecting to uart, transferring data and then
       disconnecting
4  # Allows ibus to be polled regularly without creating a
       block
5
6  # returns raw values
7  # To make meaningful there is a normalize static method
8  # approx (-100 to +100) for default (standard) controls
9  # 0 is centre. The zero point can be adjusted on the
       controller
10 # actual value of min and maximum may differ
11 # approx (0 to 100) for dials
12
13
14 # Select appropriate uart pin (following are defaults)
15 # For ibus receive then only RX pin needs to be connected
16 # UART 0: TX pin 0 GP0 RX pin 1 GP1
```

```
17 # UART 1: TX pin 6 GP4 RX pin 7 GP5
18 # Connect appropriate RX pin to rightmost pin on FS-iA6B
19
20 # returns list of channel values. First value (pseudo
       channel 0) is status
21 # 0 = initial values
22 # 1 = new values
23 # -1 = failed to receive data old values sent
24 # -2 = checksum error
25
26
27 class IBus ():
28
29     # Number of channels (FS-iA6B has 6)
30     def __init__ (self, uart_num, baud=115200,
           num_channels=8):
31         self.uart_num = uart_num
32         self.baud = baud
33         self.uart = UART(self.uart_num, self.baud)
34         self.num_channels = num_channels
35         # ch is channel value
36         self.ch = []
37         # Set channel values to 0
38         for i in range (self.num_channels+1):
39             self.ch.append(0)
40
41
42     # Returns list with raw data
43     def read(self):
44         # Max 10 attempts to read
```

```
45          for z in range(10):
46              buffer = bytearray (31)
47              char = self.uart.read(1)
48              # check for 0x20
49              if char == b'\x20':
50                  # read reset of string into buffer
51                  self.uart.readinto(buffer)
52                  checksum = 0xffdf # 0xffff - 0x20
53                  # check checksum
54                  for i in range(29):
55                      checksum -= buffer[i]
56                  if checksum == (buffer[30] << 8) |
                       buffer[29]:
57                      # buffer[0] = 0x40
58                      self.ch[0] = 1 # status 1 = success
59                      for i in range (1, self.num_channels
                           + 1):
60                          self.ch[i] = (buffer[(i*2)-1] +
                               (buffer[i*2] << 8))
61                      return self.ch
62                  else:
63                      # Checksum error
64                      self.ch[0] = -2
65              else:
66                  self.ch[0] = -1
67
68          # Reach here then timed out
69          self.ch[0] = -1
70          return self.ch
71
```

```
72
73      # Convert to meaningful values - eg. -100 to 100
74      # Typical use for FS-iA6B
75      # channel 1 to 4 use type="default" provides result
           from -100 to +100 (0 in centre)
76      # channel 5 & 6 are dials type="dial" provides result
           from 0 to 100
77      # Note approx depends upon calibration etc.
78      @staticmethod
79      def normalize (value, type="default"):
80          if (type == "switch"):
81              return ((value - 1000) / 1000)
82          elif (type == "rudder"):
83              return int(((value - 1000) * 3.9 + 3050)) #
                   Range 3050-6950
84          elif (type == "winch"):
85              return int(((value - 1000) * 2.91 + 3680)) #
                   Range 3680-6590
86          else:
87              return int(((value - 1000) * 6 + 2000)) #
                   Values range 2000-8000
```

Appendix D

Matlab Code - Data Processing, Control Design, and Simulation

```
%% Jeb Carter - MAE Senior Thesis - Dynamics Data Analysis
% DynamicsAnalysis.m

%% Import csv values
fastfullAccelerometer = readtable('fast full
    Accelerometer.csv');
fasthalfAccelerometer = readtable('fast half
    Accelerometer.csv');
mediumfullAccelerometer = readtable('medium full
    Accelerometer.csv');
mediumhalfAccelerometer = readtable('medium half
    Accelerometer.csv');
slowfullAccelerometer = readtable('slow full
    Accelerometer.csv');
slowhalfAccelerometer = readtable('slow half
    Accelerometer.csv');

```

```
fastfullOrientation = readtable('fast full Orientation.csv');
fasthalfOrientation = readtable('fast half Orientation.csv');
mediumfullOrientation = readtable('medium full Orientation.csv');
mediumhalfOrientation = readtable('medium half Orientation.csv');
slowfullOrientation = readtable('slow full Orientation.csv');
slowhalfOrientation = readtable('slow half Orientation.csv');

% Define Rudder Angles
fangle = 52;
hangle = 27;

%% Display original data

% Graph all 6 orientation yaws
figure(1), hold on, grid on
xlabel('Time (s)')
ylabel('Yaw (radians)')
title('Raw Yaw Values')
plot(fastfullOrientation.(3))
plot(fasthalfOrientation.(3))
plot(mediumfullOrientation.(3))
plot(mediumhalfOrientation.(3))
plot(slowfullOrientation.(3))
plot(slowhalfOrientation.(3))
```

```
36 legend('Fast Full','Fast Half','Medum Full','Medium
      Half', 'Slow Full','Slow Half')
37
38 %% Crop to relevant data
39 ffo = table2array(fastfullOrientation(471:613,:));
40 fho = table2array(fasthalfOrientation(513:674,:));
41 mfo = table2array(mediumfullOrientation(584:698,:));
42 mho = table2array(mediumhalfOrientation(459:654,:));
43 sfo = table2array(slowfullOrientation(570:846,:));
44 sho = table2array(slowhalfOrientation(775:981,:));
45 % figure(2), hold on, grid on
46 % plot(ffo(:,3))
47 % plot(fho(:,3))
48 % plot(mfo(:,3))
49 % plot(mho(:,3))
50 % plot(sfo(:,3))
51 % plot(sho(:,3))
52 % legend('Fast Full','Fast Half','Medum Full','Medium
      Half', 'Slow Full','Slow Half')
53
54 % Shift yaw to zero point
55 ffo(:,3) = ffo(:,3) - ffo(1,3);
56 fho(:,3) = fho(:,3) - fho(1,3);
57 mfo(:,3) = mfo(:,3) - mfo(1,3);
58 mho(:,3) = mho(:,3) - mho(1,3);
59 sfo(:,3) = sfo(:,3) - sfo(1,3);
60 sho(:,3) = sho(:,3) - sho(1,3);
61 % Shift time to zero point
62 ffo(:,2) = ffo(:,2) - ffo(1,2);
63 fho(:,2) = fho(:,2) - fho(1,2);
```

```
mfo(:,2) = mfo(:,2) - mfo(1,2);
mho(:,2) = mho(:,2) - mho(1,2);
sfo(:,2) = sfo(:,2) - sfo(1,2);
sho(:,2) = sho(:,2) - sho(1,2);
% Plot
figure(3), hold on, grid on
plot(ffo(:,2),ffo(:,3))
plot(fho(:,2),fho(:,3))
plot(mfo(:,2),mfo(:,3))
plot(mho(:,2),mho(:,3))
plot(sfo(:,2),sfo(:,3))
plot(sho(:,2),sho(:,3))
legend('Fast Full','Fast Half','Medum Full','Medium
    Half', 'Slow Full','Slow Half')
xlabel('Time (s)')
ylabel('Yaw (rad)')
title('Shifted & Cropped Yaw Values')

%% Extract only the first 2 seconds of turn data (100
    samples)
trial_dur = 2;

ffo = ffo(1:100,:);
fho = fho(1:100,:);
mfo = mfo(1:100,:);
mho = mho(1:100,:);
sfo = sfo(1:100,:);
sho = sho(1:100,:);

%% Examine Accelerometer Values
```

```
figure(4), hold on, grid on
plot(fastfullAccelerometer.(5))
plot(fastfullAccelerometer.(4))
plot(fastfullAccelerometer.(3))
plot(fastfullOrientation.(3))
legend('Horizontal','Forward','Vertical', 'Yaw')
title('Fast Full Accelerometer Values and Yaw')

%% Determine logic for velocity calculations
figure(5), hold on, grid on
testarray = table2array(fastfullAccelerometer);
% Time range for velocity integration
test = cumtrapz(testarray(140:725,2),
   testarray(140:725,4));
plot(testarray(140:725,2), test)
plot(fastfullOrientation.(2), fastfullOrientation.(3))
legend('Integrated Velocity','Yaw')
title('Testing Integration for Velocity alongside Yaw')

%% Velocity Plots for all
figure(6), hold on, grid on
% These two get the data range (Comment out after
   acquired)
%plot(slowfullAccelerometer.(4))
%plot(slowfullOrientation.(3))
sfdatarange = 149:839;
% Actual display
sfa = table2array(slowfullAccelerometer);
plot(sfa(sfdatarange,2),sfa(sfdatarange,4))
plot(slowfullOrientation.(2),slowfullOrientation.(3))
```

```
120 sfvel = cumtrapz(sfa(sfdatarange,2), sfa(sfdatarange,4));
121 plot(sfa(sfdatarange,2), sfvel, '.')
122 sfvel_max = abs(min(sfvel));
123 title('Slow Full - Velocity Overlayed on Acceleration and
      Yaw')
124 legend('Acceleration','Yaw','Velocity')
125
126 figure(7), hold on, grid on
127 % These two get the index values (Comment out after
      acquired)
128 %plot(slowhalfAccelerometer.(4))
129 %plot(slowhalfOrientation.(3))
130 shdatarange = 201:1055;
131 % Actual display
132 sha = table2array(slowhalfAccelerometer);
133 plot(sha(shdatarange,2),sha(shdatarange,4))
134 plot(slowhalfOrientation.(2),slowhalfOrientation.(3))
135 shvel = cumtrapz(sha(shdatarange,2), sha(shdatarange,4));
136 plot(sha(shdatarange,2), shvel, '.')
137 shvel_max = abs(min(shvel));
138 title('Slow Half - Velocity Overlayed on Acceleration and
      Yaw')
139 legend('Acceleration','Yaw','Velocity')
140
141 figure(8), hold on, grid on
142 % These two get the index values (Comment out after
      acquired)
143 %plot(mediumfullAccelerometer.(4))
144 %plot(mediumfullOrientation.(3))
145 mfdatarange = 163:735;
```

```
146 % Actual display
147 mfa = table2array(mediumfullAccelerometer);
148 plot(mfa(mfdatarange,2),mfa(mfdatarange,4))
149 plot(mediumfullOrientation.(2),mediumfullOrientation.(3))
150 mfvel = cumtrapz(mfa(mfdatarange,2), mfa(mfdatarange,4));
151 plot(mfa(mfdatarange,2), mfvel, '.')
152 %mfvel_max = abs(min(mfvel));
153 % Artificially set mfvel_max because data doesn't look
      correct
154 mfvel_max = 0.25;
155 title('Medium Full - Velocity Overlayed on Acceleration
      and Yaw')
156 legend('Acceleration','Yaw','Velocity')
157
158 figure(9), hold on, grid on
159 % These two get the index values (Comment out after
      acquired)
160 %plot(mediumhalfAccelerometer.(4))
161 %plot(mediumhalfOrientation.(3))
162 mhdatarange = 131:686;
163 % Actual display
164 mha = table2array(mediumhalfAccelerometer);
165 plot(mha(mhdatarange,2),mha(mhdatarange,4))
166 plot(mediumhalfOrientation.(2),mediumhalfOrientation.(3))
167 mhvel = cumtrapz(mha(mhdatarange,2), mha(mhdatarange,4));
168 plot(mha(mhdatarange,2), mhvel, '.')
169 mhvel_max = abs(min(mhvel));
170 title('Medium Half - Velocity Overlayed on Acceleration
      and Yaw')
171 legend('Acceleration','Yaw','Velocity')
```

```
172
173 figure(10), hold on, grid on
174 % These two get the index values (Comment out after
      acquired)
175 %plot(fastfullAccelerometer.(4))
176 %plot(fastfullOrientation.(3))
177 ffdatarange = 145:803;
178 % Actual display
179 ffa = table2array(fastfullAccelerometer);
180 plot(ffa(ffdatarange,2),ffa(ffdatarange,4))
181 plot(fastfullOrientation.(2),fastfullOrientation.(3))
182 ffvel = cumtrapz(ffa(ffdatarange,2), ffa(ffdatarange,4));
183 plot(ffa(ffdatarange,2), ffvel, '.')
184 ffvel_max = abs(min(ffvel));
185 title('Fast Full - Velocity Overlayed on Acceleration and
      Yaw')
186 legend('Acceleration','Yaw','Velocity')
187
188 figure(11), hold on, grid on
189 % These two get the index values (Comment out after
      acquired)
190 %plot(fasthalfAccelerometer.(4))
191 %plot(fasthalfOrientation.(3))
192 fhdatarange = 174:1013;
193 % Actual display
194 fha = table2array(fasthalfAccelerometer);
195 plot(fha(fhdatarange,2),fha(fhdatarange,4))
196 plot(fasthalfOrientation.(2),fasthalfOrientation.(3))
197 fhvel = cumtrapz(fha(fhdatarange,2), fha(fhdatarange,4));
198 plot(fha(fhdatarange,2), fhvel, '.')
```

```
199 fhvel_max = abs(min(fhvel));
200 title('Fast Half - Velocity Overlayed on Acceleration and
      Yaw')
201 legend('Acceleration','Yaw','Velocity')
202
203 %% Determine Slope of each yaw curve
204 figure(12), hold on, grid on
205
206 fffit = fitlm(ffo(:,2), ffo(:,3),'Intercept',false);
207 ffrate = fffit.Coefficients.Estimate;
208 temp = ffo(:,2) .* ffrate;
209 plot(ffo(:,2), ffo(:,3),'Color','black')
210 plot(ffo(:,2), temp,'--','Color','black')
211
212 fhfit = fitlm(fho(:,2), fho(:,3),'Intercept',false);
213 fhrate = fhfit.Coefficients.Estimate;
214 temp = fho(:,2) .* fhrate;
215 plot(fho(:,2), fho(:,3),'Color','blue')
216 plot(fho(:,2), temp,'--','Color','blue')
217
218 mffit = fitlm(mfo(:,2), mfo(:,3),'Intercept',false);
219 mfrate = mffit.Coefficients.Estimate;
220 temp = mfo(:,2) .* mfrate;
221 plot(mfo(:,2), mfo(:,3),'Color','red')
222 plot(mfo(:,2), temp,'--','Color','red')
223
224 mhfit = fitlm(mho(:,2), mho(:,3),'Intercept',false);
225 mhrate = mhfit.Coefficients.Estimate;
226 temp = mho(:,2) .* mhrate;
227 plot(mho(:,2), mho(:,3),'Color','green')
```

```
228 plot(mho(:,2), temp,'--','Color','green')
229
230 sffit = fitlm(sfo(:,2), sfo(:,3),'Intercept',false);
231 sfrate = sffit.Coefficients.Estimate;
232 temp = sfo(:,2) .* sfrate;
233 plot(sfo(:,2), sfo(:,3),'Color','magenta')
234 plot(sfo(:,2), temp,'--','Color','magenta')
235
236 shfit = fitlm(sho(:,2), sho(:,3),'Intercept',false);
237 shrate = shfit.Coefficients.Estimate;
238 temp = sho(:,2) .* shrate;
239 plot(sho(:,2), sho(:,3),'Color','cyan')
240 plot(sho(:,2), temp,'--','Color','cyan')
241
242 legend('Fast Full Data','Fast Full Regression', ...
243     'Fast Half Data','Fast Half Regression', ...
244     'Medium Full Data','Medium Full Regression', ...
245     'Medium Half Data','Medium Half Regression', ...
246     'Slow Full Data','Slow Full Regression', ...
247     'Slow Half Data','Slow Half
        Regression','Location','eastoutside')
248 xlabel('Time (s)')
249 ylabel('Yaw (rad)')
250 title('Yaw Data and Regression Lines for all 6 Trials')
251
252 %% Turn Rate vs Rudder Angle for all 3 speeds
253
254 ruddervals = [0, hangle, fangle];
255
256 slowrates = [0, shrate, sfrate];
```

```
257 mediumrates = [0, mhrate, mfrate];
258 fastrates = [0, fhrate, ffrate];
259
260 figure(13), hold on, grid on
261 plot(ruddervals, slowrates)
262 plot(ruddervals, mediumrates)
263 plot(ruddervals, fastrates)
264
265 legend('Slow','Medium','Fast','Location','best')
266 title('Rate of Turn vs Rudder Angle at 3 Speeds')
267 xlabel('Rudder Angle (degrees)')
268 ylabel('Rate of turn (rad/s)')
269
270 %% Turn Rate vs Boat Speed for both rudder deflections
271
272 halfspeeds = [0, shvel_max, mhvel_max, fhvel_max];
273 fullspeeds = [0, sfvel_max, mfvel_max, ffvel_max];
274 halfrates = [0, shrate, mhrate, fhrate];
275 fullrates = [0, sfrate, mfrate, ffrate];
276
277 figure(14), hold on, grid on
278 plot(halfspeeds, halfrates)
279 plot(fullspeeds, fullrates)
280
281 legend('Half Deflection','Full
      Deflection','Location','best')
282 title('Rate of Turn vs Boat Speed for both Rudder
      Deflections')
283 xlabel('Boat Speed')
284 ylabel('Rate of turn (rad/s)')
```

```
285
286 %% Develop trial-by-trial plant
287
288 % Define tf format
289 num_format = NaN(1,1);
290 den_format = [1,NaN(1,1),0];
291 tf_format = idtf(num_format,den_format);
292 tf_format.Structure.num.Free=(1);
293 tf_format.Structure.den.Free=[0 1 0];
294 s = tf('s');
295
296 % Fast Full
297 sampletime = 0.02;
298 ffyaw = [zeros(10,1); ffo(:,3)];
299 ffinput = [zeros(10,1); zeros(length(ffo),1) + fangle];
300 % Organize data into a single object
301 ffdata = iddata(ffyaw, ffinput, sampletime);
302 % Calculate coefficients
303 ff_est = tfest(ffdata, tf_format);
304 % Calulate K and Tau from the estimate
305 ff_tau = 1/ff_est.denominator(2);
306 ff_K = ff_est.numerator(1)*ff_tau;
307 ffplant = ff_K / (s * (ff_tau * s + 1));
308 % Plot
309 figure(15)
310 compare(ffdata, ff_est)
311 grid on
312 title('Fast Full Yaw and TF Estimate')
313 legend('Yaw Data','Transfer Function')
314
```

```
315 % Fast Half
316 sampletime = 0.02;
317 fhyaw = [zeros(10,1); fho(:,3)];
318 fhinput = [zeros(10,1); zeros(length(fho),1) + hangle];
319 % Organize data into a single object
320 fhdata = iddata(fhyaw, fhinput, sampletime);
321 % Calculate coefficients
322 fh_est = tfest(fhdata, tf_format);
323 % Calulate K and Tau from the estimate
324 fh_tau = 1/fh_est.denominator(2);
325 fh_K = fh_est.numerator(1)*fh_tau;
326 fhplant = fh_K / (s * (fh_tau * s + 1));
327 % Plot
328 figure(16)
329 compare(fhdata, fh_est)
330 grid on
331 title('Fast Half Yaw and TF Estimate')
332 legend('Yaw Data','Transfer Function')
333
334 % Medium Full
335 sampletime = 0.02;
336 mfyaw = [zeros(10,1); mfo(:,3)];
337 mfinput = [zeros(10,1); zeros(length(mfo),1) + fangle];
338 % Organize data into a single object
339 mfdata = iddata(mfyaw, mfinput, sampletime);
340 % Calculate coefficients
341 mf_est = tfest(mfdata, tf_format);
342 % Calulate K and Tau from the estimate
343 mf_tau = 1/mf_est.denominator(2);
344 mf_K = mf_est.numerator(1)*mf_tau;
```

```
345 mfplant = mf_K / (s * (mf_tau * s + 1));
346 % Plot
347 figure(17)
348 compare(mfdata, mf_est)
349 grid on
350 title('Medium Full Yaw and TF Estimate')
351 legend('Yaw Data','Transfer Function')
352
353 % Medium Half
354 sampletime = 0.02;
355 mhyaw = [zeros(10,1); mho(:,3)];
356 mhinput = [zeros(10,1); zeros(length(mho),1) + hangle];
357 % Organize data into a single object
358 mhdata = iddata(mhyaw, mhinput, sampletime);
359 % Calculate coefficients
360 mh_est = tfest(mhdata, tf_format);
361 % Calulate K and Tau from the estimate
362 mh_tau = 1/mh_est.denominator(2);
363 mh_K = mh_est.numerator(1)*mh_tau;
364 mhplant = mh_K / (s * (mh_tau * s + 1));
365 % Plot
366 figure(18)
367 compare(mhdata, mh_est)
368 grid on
369 title('Medium Half Yaw and TF Estimate')
370 legend('Yaw Data','Transfer Function')
371
372 % Slow Full
373 sampletime = 0.02;
374 sfyaw = [zeros(10,1); sfo(:,3)];
```

```
375 sfinput = [zeros(10,1); zeros(length(sfo),1) + fangle];
376 % Organize data into a single object
377 sfdata = iddata(sfyaw, sfinput, sampletime);
378 % Calculate coefficients
379 sf_est = tfest(sfdata, tf_format);
380 % Calulate K and Tau from the estimate
381 sf_tau = 1/sf_est.denominator(2);
382 sf_K = sf_est.numerator(1)*sf_tau;
383 sfplant = sf_K / (s * (sf_tau * s + 1));
384 % Plot
385 figure(19)
386 compare(sfdata, sf_est)
387 grid on
388 title('Slow Full Yaw and TF Estimate')
389 legend('Yaw Data','Transfer Function')
390
391 % Slow Half
392 sampletime = 0.02;
393 shyaw = [zeros(10,1); sho(:,3)];
394 shinput = [zeros(10,1); zeros(length(sho),1) + hangle];
395 % Organize data into a single object
396 shdata = iddata(shyaw, shinput, sampletime);
397 % Calculate coefficients
398 sh_est = tfest(shdata, tf_format);
399 % Calulate K and Tau from the estimate
400 sh_tau = 1/sh_est.denominator(2);
401 sh_K = sh_est.numerator(1)*sh_tau;
402 shplant = sh_K / (s * (sh_tau * s + 1));
403 % Plot
404 figure(20)
```

```
405 compare(shdata, sh_est)
406 grid on
407 title('Slow Half Yaw and TF Estimate')
408 legend('Yaw Data','Transfer Function')
409
410 %% All estimate Plant TF's
411
412 figure(21), hold on
413 plot(ffo(:,2), ffo(:,3), 'Color', 'r')
414 step(ffplant*fangle, trial_dur)
415 plot(fho(:,2), fho(:,3), 'Color', 'r')
416 step(fhplant*hangle)
417 plot(mfo(:,2), mfo(:,3), 'Color', 'r')
418 step(mfplant*fangle)
419 plot(mho(:,2), mho(:,3), 'Color', 'r')
420 step(mhplant*hangle)
421 plot(sho(:,2), sho(:,3), 'Color', 'r')
422 step(shplant*hangle)
423 plot(sfo(:,2), sfo(:,3), 'Color', 'r')
424 step(sfplant*fangle)
425 grid on
426 legend('Fast Full Data','Fast Full TF', ...
427     'Fast Half Data','Fast Half TF', ...
428     'Medium Full Data','Medium Full TF', ...
429     'Medium Half Data','Medium Half TF', ...
430     'Slow Full Data','Slow Full TF', ...
431     'Slow Half Data','Slow Half
        TF','Location','eastoutside')
432 xlabel('Time (s)')
433 ylabel('Yaw (rad)')
```

```
434 title('Yaw Data and Transfer Function Estimates for all 6
       Trials')
435
436 %% Determine average coefficients for each speed and
       average speed
437
438 fmean_K = mean([fh_K ff_K]);
439 mmean_K = mean([mh_K mf_K]);
440 smean_K = mean([sh_K sf_K]);
441 fmean_tau = mean([fh_tau ff_tau]);
442 mmean_tau = mean([mh_tau mf_tau]);
443 smean_tau = mean([sh_tau sf_tau]);
444
445 % Average boat speed based on estimates
446
447 fmean_vel = mean([fhvel_max fhvel_max]); % m/s
448 mmean_vel = mean([mhvel_max mhvel_max]); % m/s
449 smean_vel = mean([shvel_max shvel_max]); % m/s
450
451 %% Build new transfer functions using average values
452
453 slowplant = smean_K / (s * (smean_tau * s + 1));
454 mediumplant = mmean_K / (s * (mmean_tau * s + 1));
455 fastplant = fmean_K / (s * (fmean_tau * s + 1));
456
457 figure(22), hold on
458 plot(ffo(:,2), ffo(:,3),'Color','blue')
459 step(fastplant*fangle, trial_dur)
460 plot(fho(:,2), fho(:,3),'Color','black')
461 step(fastplant*hangle)
```

```
462 grid on
463 title('Fast - Yaw Data vs Final Transfer Function')
464 legend('Data - Full', 'Model - Full', 'Data - Half',
       'Model - Half')
465
466 figure(23), hold on
467 plot(mfo(:,2), mfo(:,3),'Color','blue')
468 step(mediumplant*fangle, trial_dur)
469 plot(mho(:,2), mho(:,3),'Color','black')
470 step(mediumplant*hangle)
471 grid on
472 title('Medium - Yaw Data vs Final Transfer Function')
473 legend('Data - Full', 'Model - Full', 'Data - Half',
       'Model - Half')
474
475 figure(24), hold on
476 plot(sfo(:,2), sfo(:,3),'Color','blue')
477 step(slowplant*fangle, trial_dur)
478 plot(sho(:,2), sho(:,3),'Color','black')
479 step(slowplant*hangle)
480 grid on
481 title('Slow - Yaw Data vs Final Transfer Function')
482 legend('Data - Full', 'Model - Full', 'Data - Half',
       'Model - Half')
483
484 %% Plot velocity vs K and tau
485
486 mean_vels = [smean_vel, mmean_vel, fmean_vel];
487 mean_Ks = [smean_K, mmean_K, fmean_K];
488 mean_taus = [smean_tau, mmean_tau, fmean_tau];
```

```
489
490 % Calculate regression lines
491 K_vel_reg = fitlm(mean_vels, mean_Ks);
492 tau_vel_reg = fitlm(mean_vels, mean_taus);
493 K_vel_slope = K_vel_reg.Coefficients.Estimate(2);
494 tau_vel_slope = tau_vel_reg.Coefficients.Estimate(2);
495 K_vel_int = K_vel_reg.Coefficients.Estimate(1);
496 tau_vel_int = tau_vel_reg.Coefficients.Estimate(1);
497
498 figure(25), hold on, grid on
499 plot(mean_vels, mean_Ks)
500 plot(mean_vels, mean_taus)
501 % Regression plots
502 temp = mean_vels*K_vel_slope + K_vel_int;
503 plot(mean_vels, temp)
504 temp = mean_vels*tau_vel_slope + tau_vel_int;
505 plot(mean_vels, temp)
506 title('K and tau Coefficients vs Boat Speed')
507 legend('Mean K Values','Mean tau Values','K Regression',
      'tau Regression')
508 xlabel('Speed (m/s)')
509
510 %% Normalize yaw values by U^2 and alpha
511
512 ffnorm = ffo(:,3)./fmean_vel^2;
513 fhnorm = fho(:,3)./fmean_vel^2;
514 mfnorm = mfo(:,3)./mmean_vel^2;
515 mhnorm = mho(:,3)./mmean_vel^2;
516 sfnorm = sfo(:,3)./smean_vel^2;
517 shnorm = sho(:,3)./smean_vel^2;
```

```
518
519 figure(26), hold on, grid on
520 plot(ffo(:,2), ffnorm)
521 plot(fho(:,2), fhnorm)
522 plot(mfo(:,2), mfnorm)
523 plot(mho(:,2), mhnorm)
524 plot(sfo(:,2), sfnorm)
525 plot(sho(:,2), shnorm)
526 legend('Fast Full','Fast Half','Medum Full','Medium
      Half', 'Slow Full','Slow Half')
527 xlabel('Time (s)')
528 ylabel('Yaw (rad)')
529 title('Yaw Normalized by Velocity^2')
530
531 % Normalize by alpha
532 fangle_rad = deg2rad(fangle);
533 hangle_rad = deg2rad(hangle);
534
535 ffnorm = ffnorm./fangle_rad;
536 fhnorm = fhnorm./hangle_rad;
537 mfnorm = mfnorm./fangle_rad;
538 mhnorm = mhnorm./hangle_rad;
539 sfnorm = sfnorm./fangle_rad;
540 shnorm = shnorm./hangle_rad;
541
542 % Calculate average curve
543 figure(27), hold on, grid on
544 plot(ffo(:,2), ffnorm)
545 plot(fho(:,2), fhnorm)
546 plot(mfo(:,2), mfnorm)
```

```
547 plot(mho(:,2), mhnorm)
548 plot(sfo(:,2), sfnorm)
549 plot(sho(:,2), shnorm)
550 % Calculate average
551 norm_avg = (ffnorm+fhnorm+mfnorm+mhnorm+sfnorm+shnorm)/6 ;
552 plot(ffo(:,2), norm_avg,'.')
553
554 legend('Fast Full','Fast Half','Medum Full','Medium
      Half', ...
555     'Slow Full','Slow Half','Average','Location','best')
556 xlabel('Time (s)')
557 ylabel('Yaw (rad)')
558 title('Yaw Normalized by Velocity^2 and Yaw')
559
560
561 %% Calculate transfer function for norm_avg
562
563 yaw = [zeros(10,1); norm_avg];
564 input = [zeros(10,1); ones(length(norm_avg),1)];
565 % Organize data into a single object
566 normdata = iddata(yaw, input, sampletime);
567 % Calculate coefficients
568 norm_est = tfest(normdata, tf_format);
569 % Calulate K and Tau from the estimate
570 norm_tau = 1/norm_est.denominator(2);
571 norm_K = norm_est.numerator(1)*norm_tau;
572 normplant = norm_K / (s * (norm_tau * s + 1));
573
574 figure(28)
575 compare(norm_est, normdata)
```

```
576 figure(29), hold on
577 step(normplant*fangle_rad*fmean_vel^2, trial_dur)
578 plot(ffo(:,2), ffo(:,3), 'Color','b')
579 step(normplant*hangle_rad*smean_vel^2, trial_dur)
580 plot(sho(:,2), sho(:,3), 'Color','g')
581 step(normplant*hangle_rad*mmean_vel^2, trial_dur)
582 plot(mho(:,2), mho(:,3), 'Color','r')
583
584 grid on
585 title('Normalized Model Fit Comparison for Example Data')
586 xlabel('time (s)')
587 ylabel('Yaw (rad)')
588 legend('Fast Full Model','Fast Full Data','Slow Half
      Model','Slow Half Data', ...
589     'Medium Half Model','Medium Half Data')
590
591 % Note: The model over-estimates yaw at high speeds and
      under-estimates at
592 % low speeds. As shown, medium-half is almost perfectly
      modeled by the
593 % normalized example, as shown in the figure
594
595 %% Control Analysis
596
597 % Create controller
598 z = 5;
599 p = 7;
600 Kc = 0.6;
601 controller = Kc * (s + z) / (s + p);
602 sys = feedback(controller*mmean_vel^2*normplant,1);
```

```
603 loop = mmean_vel^2*normplant*controller;
604 figure(31)
605 grid on
606 margin(loop)
607
608 figure(32)
609 rlocus(loop)
610
611 figure(33)
612 step(sys*deg2rad(30))
613 title('Step Response to 30 Degree Turn - Heading')
614 ylabel('Yaw (rad)')
615 grid on
616
617 %% Convert to state space
618
619 [num,den] = tfdata(controller,'v');
620 [A, B, C, D] = tf2ss(num,den);
621
622 %% Test discrete conversion
623
624 dt = 0.003;
625
626 Ad = 1 + A*dt;
627 Bd = B*dt;
628 %plant_disc = tf(norm_K, [norm_tau, 1, 0], dt);
629 plant_disc = c2d(normplant, dt);
630
631 con_disc = ss(Ad, Bd, C, D, dt);
632 disc_sys = feedback(con_disc*mmean_vel^2*plant_disc,1);
```

```
633 figure(34), hold on
634 step(disc_sys*deg2rad(30))
635 step(sys*deg2rad(30))
636 grid on
637 title('Step Responses for Discrete and Continuous
      Systems')
638 ylabel('Yaw (rad)')
639 legend('Discrete','Continuous','Location','best')
640
641 %% Test loop manually
642
643 figure(35), hold on
644 err = deg2rad(30);
645 state_var = 0;
646 dt = 0.003;
647 Ad = 1 + A*dt;
648 Bd = B*dt;
649 sim_len = 150;
650 % Constant dt case
651 for i = 1:sim_len
652     state_var = Ad * state_var + Bd * err;
653     rud_angle = C * state_var + D * err;
654     plot(i, rud_angle, 'black.')
655 end
656
657 state_var = 0;
658 % Variable dt
659 for i = 1:sim_len
660     % Simulate variable dt (between 0.001-0.005)
661     dt = round(rand()/250,3) + 0.001;
```

```
662     Ad = 1 + A*dt;
663     Bd = B*dt;
664     state_var = Ad * state_var + Bd * err;
665     rud_angle = C * state_var + D * err;
666     plot(i, rud_angle, 'red.')
667 end
668 
669 state_var = 0;
670 dt = 0.003;
671 % Constant dt and decreasing err
672 for i = 1:sim_len
673     Ad = 1 + A*dt;
674     Bd = B*dt;
675     err = deg2rad((500-i)/16.6);
676     state_var = Ad * state_var + Bd * err;
677     rud_angle = C * state_var + D * err;
678     plot(i, rud_angle, 'blue.')
679 end
680 
681 state_var = 0;
682 % Variable dt and decreasing err
683 for i = 1:sim_len
684     % Simulate variable dt (between 0.001-0.005)
685     dt = round(rand()/250,3) + 0.001;
686     Ad = 1 + A*dt;
687     Bd = B*dt;
688     err = deg2rad((500-i)/16.6);
689     state_var = Ad * state_var + Bd * err;
690     rud_angle = C * state_var + D * err;
691     plot(i, rud_angle, 'green.')
```

```
692     %plot(i, err, 'b.')
693 end
694
695 title('Simulated Discrete Time Controller Responses')
696 legend('Constant dt, constant err input', ...
697     'Variable dt, constant err input', ...
698     'Constant dt, linear decreasing err', ...
699     'Variable dt, linear decreasing err')
700 xlabel('Time Step')
701 ylabel('Radians')
```