12-2020

The Implementation and Comparison of Fuzzy Logic Control Systems to Modern Control Methods on Low-Cost Hardware

Cristian A. Barron
The University of Texas Rio Grande Valley

THE IMPLEMENTATION AND COMPARISON OF FUZZY LOGIC CONTROL SYSTEMS TO MODERN CONTROL METHODS ON LOW-COST HARDWARE

A Thesis

by

CRISTIAN A. BARRON

Submitted to the Graduate College of
The University of Texas Rio Grande Valley
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING

DECEMBER 2020

Major Subject: Mechanical Engineering

THE IMPLEMENTATION AND COMPARISON OF FUZZY LOGIC CONTROL SYSTEMS TO MODERN CONTROL METHODS ON LOW-COST HARDWARE

A Thesis
by
CRISTIAN A. BARRON

COMMITTEE MEMBERS

Dr. Horacio Vasquez
Chair of Committee

Dr. Dumitru Caruntu
Committee Member

Dr. Stephen Crown
Committee Member

DECEMBER 2020

ABSTRACT

Barron, Cristian A., The Implementation and Comparison of Fuzzy Logic Control Systems to Modern Control Methods on Low-Cost Hardware. Master of Science in Engineering, December 2020, 84 pp, 8 tables, 49 figures, 20 titles.

Modern control engineering provides many options to automate systems for which a mathematical model is required. Another control does not rely on the mathematical model of the system and is known as fuzzy logic control. In this study, a literature review is conducted on existing control systems strategies such as proportional integral and derivative (PID), linear quadratic regulator (LQR), and fuzzy logic controller (FLC), the complexity of the systems they control, and their strengths and weaknesses. In addition, a series of experiments are conducted, both through simulations in MATLAB Simulink and using their implementation using the actual physical hardware to test the effectiveness of said controllers. The effect of changing fuzzy logic membership functions is also determined. The settling times of controllers are compared using a physical prototype of a mechanical arm. Lastly, dead zone correction techniques are addressed and implemented.

DEDICATION

Finishing my master's degree would have never been possible without my loving family supporting me every step of the way. My mother, Lissette Barron and my father, Lorenzo Barron, have provided me with an amazing opportunity to pursue my dreams. I would also like to extend a thank you to all the friends and colleagues that have stuck by my side throughout the years. My network of support has given me the courage to continue onward and become a better person than I ever thought possible. Thank you all for the love and support, I hope to accomplish even greater things.

ACKNOWLEDGEMENTS

I would like to extend my sincerest thanks to Dr. Horacio Vasquez, the committee chair, for granting me the knowledge and resources to create this document. From humble beginnings in mechatronics lectures, to providing the necessary equipment to conduct experiments, he has helped and inspired me to continue and finish this degree. I would also like to thank committee members Dr. Dumitru Caruntu, for helping me hone my skills as an engineer, and Dr. Stephen Crown for providing inspiration and advice over the years. It is thanks to you all that I know I stand atop the shoulders of giants.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

Control systems are prevalent in almost every part of modern-day life: automobiles, phones, power plants, and airplanes, all utilize control strategies and systems to achieve desired response and stable operating results. To achieve reliable and effective control systems, there are many variables that must be accounted for, both in modeling of the system as well as through standard engineering implementation and practice, and through the design and simulation. There are numerous methods that one could use to implement a control system into a system design; however, they are similar in terms of how they measure feedback and perform the control actions. Perhaps one of the most important parts of designing a control system is being able to measure the output of the system. A control system that cannot measure or estimate the current state of what it is controlling is like navigating without a notion of speed and direction. As such, it is necessary to implement a reliable method of observing the system state; nevertheless, this task poses its own challenges that can ripple throughout the design. Parameters and constraints such as sample rate, computational processing power, and the control system's governing equations all must work together to form a viable whole setup. Any one of these variables that could be improperly considered could cause the entire system to fail or become unstable. For example, a system with a sample rate too low will not be able to respond to rapid changes, whereas a system whose sample rate is too high may be demanding too much computational

processing power, delaying the response of the system to the point that it might be too late for adequate and safe control. In this situation, balance is important, and this topic will be discussed in later chapters. For the control system in this project, the goal is design and implement fuzzy logic control systems and compare them to traditional methods. A system that has shown to be a good standard for testing is the inverted pendulum. The inverted pendulum is a well-studied system that is governed by nonlinear differential equations. This system has many practical uses in both the medical and robotics fields to keep objects balanced during transportation. These equations can be difficult to evaluate analytically, and even more difficult for a control system to keep track of all at once while responding in real time. Therefore, it is a common technique to linearize, and discretize the equations to arrive at a quick-to-compute solution (Akhtaruzzaman & Shafie, 2010). As presented in later chapters, this simplification makes the equations much easier to use in microcontroller implementation; but they cannot be used to accurately represent the full range of motion of the system away from the operating point used for the linearization. While there has been extensive research about the inverted pendulum system, this study focuses on the ability to implement a controller using low-cost and relatively low computational power components. Ideally, if this can be achieved, the implication is that some of these systems could be controlled without the active aide of expensive computers or software. While using higher cost methods are perfectly valid for controlling a system, the goal is to make it work adequately at a low-cost implementation.

CHAPTER II

LITERATURE REVIEW

To implement a control system, there must first be a system to control. Take the example of an inverted pendulum as shown in Figure 2.1.

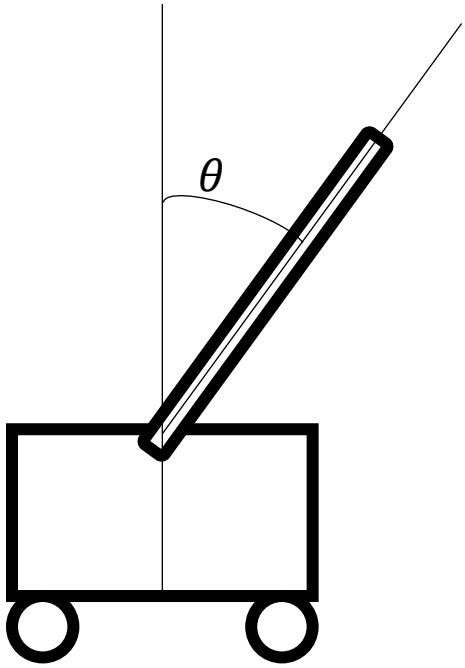

Figure 2.1: Typical Inverted Pendulum

There are several different techniques that could be used to implement a control system such as the proportional-integral-derivative controller (PID) or a linear-quadratic regulator (LQR). These controllers work with the equations that represent the system model and operate using a similar principle: changing the output based on the behavior of the error of the variable to be controlled.

The PID controller works by taking the output of the system, comparing it against a reference value, to get the error, which becomes the input to the controller to generate the controller output to act on the actuators of the system. In the case of a PID, each term has a constant which the designer can determine to change the response of the system as shown below:

$$u(t) = k_p e(t) + k_i \int_0^t e(t)dt + k_d \frac{d}{dt} e(t) \tag{2.1}$$

where k_p, k_i, and k_d are the proportional, integral, and derivative constants multiplied against error $e(t)$, its integral, and derivative, respectively. These constants could change for some control strategies at different operating points of the system. This algorithm has been shown by researchers at the University of Michigan to be capable of stabilizing the angle of an inverted pendulum (Messner & Tilbury, 2019). However, in their example, while the angle was stabilized the position of the cart would increase indefinitely. Thus, the researchers also used an LQR to demonstrate its effectiveness against a PID. An LQR functions on weighting the states of a system and prioritizing them against each other. Take the following state space system:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 5 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ 2 \\ 0 \\ 4 \end{bmatrix} [u] \tag{2.2}$$

In this system, the four states are x, $\dot{x}$, θ, and $\dot{\theta}$ (position, velocity, angular position, angular velocity). From this representation, it is possible to place priorities for the states considered the most important. This is done through a matrix that places weights on all the states of the system. The higher the number on a state variable, the more priority that state variable is given.

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 100 \end{bmatrix} \tag{2.3}$$

With the above example matrix Q, where the diagonal of the matrix represents the weights of the states of the system, respectively. In this Q matrix, the highest priorities are given to angular velocity (the fourth state) and angular position of the pendulum, while position and velocity of the cart have the lowest priority (first and second states respectively). This matrix is then used in the following integral, which is the cost function of the system to be optimized. This integral represents the performance of the system, sometimes considering fuel consumption or how far away values get from the reference. To minimize this integral is to minimize the cost of stabilizing the system. (Brunton & Kuts, 2019)

$$\int_0^\infty (x^T Q x + u^T R u)\, dt \tag{2.4}$$

In this integral, *u* represents an applied force to stabilize the system (such as the traction force of the wheels on a rail,) while R represents the cost paid to use force *u* (fuel used for an engine, limited battery life, durability of the part, etc.). In this case, the higher the R value, the more the system should be hesitant about using higher values of force *u* to stabilize the system. This integral can be minimized in MATLAB for a controllable state-space model using the command "lqr()" to create an optimal method for stabilizing and controlling the inverted pendulum and cart system. This can be very effective for multivariable control if one output needs to be controlled more strictly than another (Brunton & Kuts, 2019). Generally, this has been shown to be a rapid method due to its low computational cost and work needed to fine tune the controller (Bakaráč, Klaučo, & Fikar, 2018). In simulations using LQR, it was possible to regulate both position and angle (Messner & Tilbury, 2019). This also meant achieving the effective stabilization of both translational and angular position of the pendulum, something that could not be achieved with a single PID loop. The reason the single PID loop was insufficient was that it would balance the pendulum, but the cart's velocity was be non-zero. This meant that the pendulum would be

balanced but the cart would need to always be in motion, which is not practical. While this initially demonstrates LQR to be the superior choice for this situation, it is important to look at other factors such as the observability of the system, the speed at which both controllers can operate, and if either can be hybridized (combined with another controller to get rid of the flaws of both).

As discussed earlier, there is a need for fast computational power as a delay would cause the system to output a late response. This means that for a system that can only measure two out of the four variables, there is not only limited observability, but also there is the possibility the system response will be out-of-sync with the states of the system. In the literature, an experiment was performed testing the effectiveness of 3 different controllers on an inverted pendulum (Bakaráč, Klaučo, & Fikar, 2018). The controllers used were the PID controller, the LQR, and a model predictive controller (MPC). An MPC controller make a series of predictions for how the system behaves within a set number of time increments in the future. Then, it compares the measured results from the system with its predictions and gives an output based on how close its predictions were (Ulusoy, 2017). In the setup, the researchers imported measurements from the system into MATLAB, which would output the controller response back to the system. The results of this experiment reinforced the limitation of the PID controller to control both position and angle at once as they were unable to control both variables; however, both LQR and MPC were able to control both position and angle (Bakaráč, Klaučo, & Fikar, 2018). This demonstrates that PID alone for this application is unlikely to be practical, despite its incredible power in managing many complex systems around the world. This suggests that a PID is better suited to systems with a single input and single output; but there are ways to use PID in other cases with some additional modifications. For example, it may be possible to employ a PID in

conjunction with another control method or have multiple PID loops. A PID controller is very versatile and has many available software packages across different platforms that make it very easy to implement regardless of the hardware. This means that a PID should not be discounted in the design phase. It is possible the effectiveness of LQR may be hindered if the calculations for an observer is too much for a microcontroller to estimate the states. An MPC has potential to be a rapid system; however, there are fewer developed libraries and toolkits available to include one on a microcontroller.

Many mathematical models for an inverted pendulum rely on the system to be observed in continuous, real time; however, this cannot always be achieved, and it could be difficult, and can be expensive to accurately implement. As will be seen in Chapter IV, the system might be able to measure some of the state variables. This means it is important for the controller and model to be capable of operating with an observer for the other state variables which could be possible with present techniques in modern control engineering. Rather than measure each variable of the system in real time, it is sometimes more practical to use what can be read in the system and the mathematical model of the system to estimate the other state variables. This technique is known as "observing" the system and models or programs that handle this estimation are known as observers (Ogata, 2010). Since the inverted pendulum is a system that is dependent on the movement of the cart, it may be possible to build an observer that only requires the real input of position or angle while the velocity and angular velocity are estimated by the computer program. There are multiple ways to construct an observer system, including full-order state observers, where all the state variables in the system can be estimated, reduced-order observers for which where the number of variables estimated is less than the system total, and finally minimum-order state observers where the bare minimum variables are observed (Ogata,

2010). These types of observers require different amounts of processing power to operate because measuring a variable takes less processing power than running calculations to estimate the variable. This method of only estimating part of the system is known as constructing a reduced order observer, while a system that estimates all the states is known as a full order observer (Ogata, 2010). Using a reduced order model has been shown to be effective in other fields such as in micro electrical mechanical systems (MEMS) (Li, Ying, & Xue, 2009). By using a reduced order model, the researchers were able to run simulations much faster than if they had used a full order observer. It should also be noted, however, that not all systems can be accurately estimated using observers. In those cases, a system may have states not related or dependent on the outputs; therefore, the state would be unobservable because it cannot be accurately predicted. In the case of the inverted pendulum, it has been documented in literature to be fully observable (Messner & Tilbury, 2019). Since the system can be observed, it is important to account for any error that may occur from using this estimation. This error is based on how far away the estimated values are from the measured values. However, by careful placement of the closed-loop characteristic equation roots, it is possible to get the observer exactly in line with the real-world model (Ogata, 2010). This demonstrates the power of the method to accurately represent the system. The effectiveness of observers has been documented in experiments including an inverted pendulum on a circular track (Suphatsatienkul, Banjerdpongchai, & Wongsaisuwan, 2017). In their model, the only two states able to be measured were the pendulum angle and position on the cart. This meant that the states of velocity and angular velocity had to be estimated using an observer. Through their efforts, they discovered it was possible to get the system to reach the steady state position faster when using a reduced order observer compared to a full order observer. This proves that the state variables that

are accurately measured, the easier it is for the control system to compute the required action. Observer based control systems can be expressed in both state-space and as a series of transfer functions. This allows them to be used in low cost microcontrollers and traditional simulation software like MATLAB.

As stated previously, to use a transfer function on a microcontroller, it is necessary to convert the equations from continuous to discrete form. The discrete domain operates by taking inputs at a set interval with the microcontroller operating using a sampling rate. Discrete functions often must take the previous values in the system into account when calculating the next values. This transformation from continuous to discrete is described as discretization of the system model (Ibrahim, 2006).

Accurately modeling a system yields much insight on how to control it; however, it is also possible to use fuzzy control methods to control certain systems. Fuzzy logic is a different approach to the traditional logic in controllers that is used by most systems. For example, some digital systems have an on or off state (0 or 1), whereas fuzzy logic allows for values between 0 and 1 in order to make decisions. Fuzzy logic controllers (FLC) allow for a response tailored to the current state of the system. For example, there can be several configurations that are deemed stable, and the controller will only react strongly if the system veers out of one of those configurations. The way fuzzy logic works is through a multistep process that involves the initialization (setup of the functions and rules), fuzzification (conversion of hard, crisp data values into fuzzy values), inference (the application of rule sets and evaluation of each), and finally defuzzification (the conversion of fuzzy values back to hard data). To determine if fuzzy logic is a viable option, regardless of the success of other control options, it is important to know how it works from a mathematical perspective. Fuzzy logic uses a combination of linguistic rules

and geometric calculation to determine an output to control the system. The way this is conducted is by categorizing input values for the system. For example, a set of temperatures could be classified as too cold, too hot, or just right. These categories and all the values they contain are called membership functions. These functions can be overlaid atop each other to determine what values belong to multiple categories. For example, let us consider an incubator system where the target temperature is 30°C. Temperatures above 30°C are considered too hot, temperatures below 30°C are considered too cold, and temperatures within 10 degrees of 30°C are considered to be just right. Each of these functions have values on the y-axis that go from 0 to 1. These values represent how closely they match that category. (A temperature of 30°C has a value of 1 on the Just Right membership function and 0 on the other membership functions, while a temperature of 50°C has a value of 1 on the Too Hot membership function and a value of 0 on the others. As can be seen from the graph below, when these functions overlaid, there are values that belong to multiple categories.

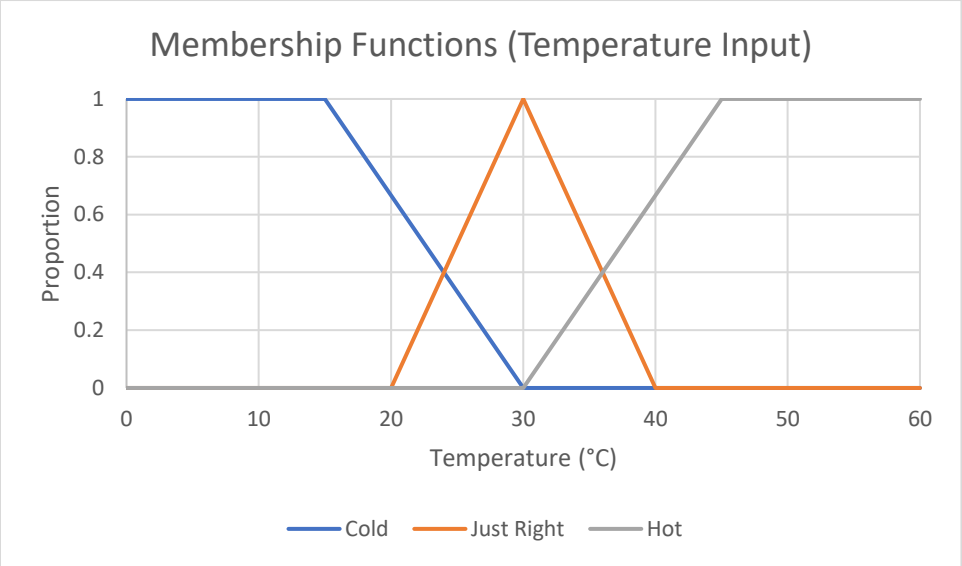

Figure 2.2: Input Membership Functions Example

As can be seen from the above graph, there are sections where the three functions intersect. If a temperature value were to have a nonzero value for multiple functions, it would belong partly to both. To see how this function is read, take the example input temperature of 28°C. At this temperature, the function is interpreted as follows:

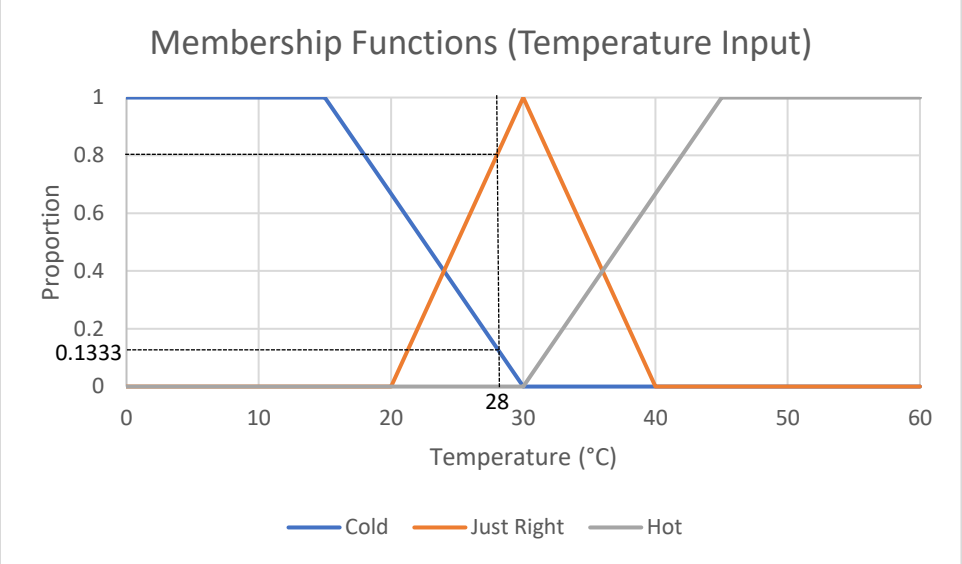

Figure 2.3: Reading Membership Functions Example

If the measured temperature were 28°C, the Cold function and the Just Right function would return values of 0.1333 and 0.8, respectively (as indicated by the dotted line), while the Hot function returns 0. Before these data can be used to create a response, there must be functions to describe the possible output. Output membership functions work similarly to input membership functions; but, instead the value of the x-axis is what needs to be solved. An example of output membership functions can be seen below in Figure 2.4:

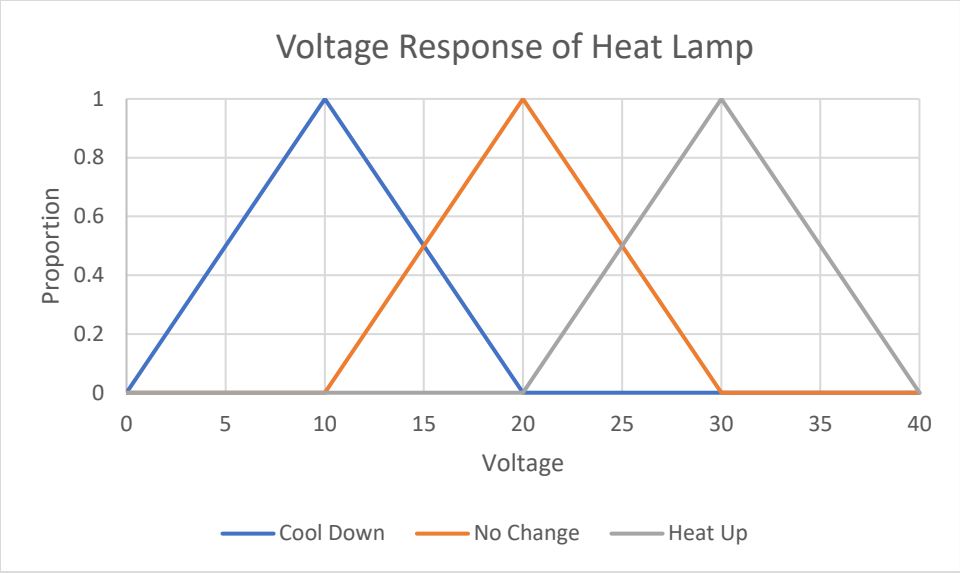

Figure 2.4: Fuzzy Logic Output Membership Functions Example

The shape of the output functions affects what values they will return. In the case for triangular membership functions, the peak of the triangle will be the ideal value for that function. For example, if the above system was told to "Heat Up", then the value returned would be 30 volts. The reason for this is because the output is based on the centroid of the shape. Before the centroid calculations can be discussed; however, it is important to discuss how membership functions interact. Fuzzy logic operates using a set of linguistic rules that dictate how the input membership functions affect the output. They can be written as a set of "and, if, and or" statements to set conditions for certain outputs. An example for the previous membership functions would be "if cold, heat up." This phrase literally tells the system to heat up if the input temperature is considered cold. A fuzzy logic controller requires rules for each combination of inputs to be assigned to an output value. In a control system with multiple inputs, this creates an n^m number of required rules where n is the number of membership functions and m is the number of inputs (assuming each input has the same number of membership functions).

Consequently, in the above example, there are 3^1 required rules. These three rules can be seen in the table below:

INPUT	OUTPUT
Cold	Heat Up
Just Right	No Change
Hot	Cool Down

Table 2.1: Fuzzy Logic Rules Example (3x1)

Taking the previous values of 0.1333 Cold and 0.8 Just Right, it is possible to apply rules to get an output voltage. Applying these rules means that our output response would be 0.1333 of the "Heat Up" function, 0 of the Cool Down function, and 0.8 of the "No Change" function. This can be visually demonstrated in Figure 2.5:

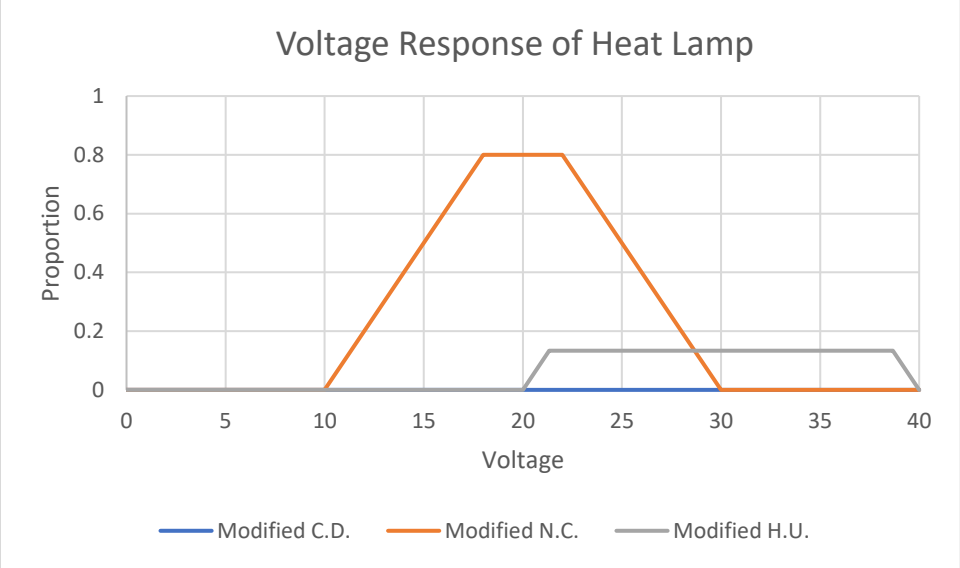

Figure 2.5: Modified Voltage Response with Rules Applied Example

This graph is the same as the previous output graph, however, the maximum value allowed to be reached for the "Cool Down," "No Change," and "Heat Up" functions are 0, 0.8, and 0.1333, respectively. The process of changing the geometry of the output membership functions is an example of fuzzification. However, to get a usable response from the control system, a defuzzification process is required next. There are many ways to defuzzifying a system, one of which is called the "centroid method" where the centroid of the intersecting areas is found, and its response given. Another, slightly faster, method is called the "weighted average method." This method defuzzifies by taking the average of each function's peak output value per input. For this example, it would look like the following equation:

$$\frac{0*10\ V+0.8*20\ V+0.1333*30\ V}{0+0.8+0.1333} = 21.42\ V \tag{2.5}$$

As can be seen, this defuzzification method is a simple calculation that gives an approximate answer but is extremely viable for symmetric output functions. The same fuzzy system can be constructed in MATLAB to return the following graphs:

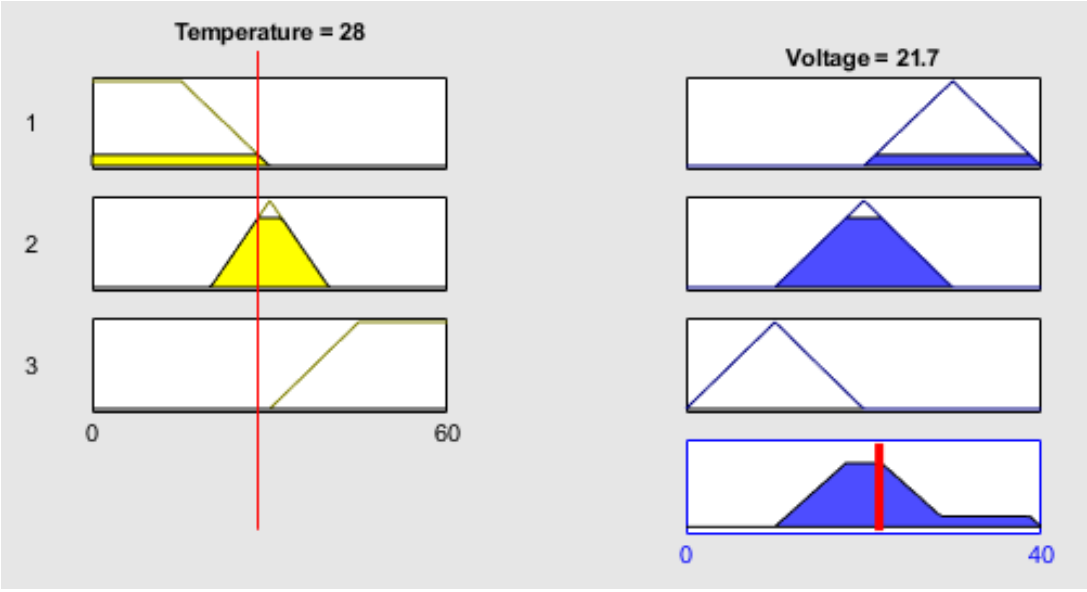

Figure 2.6: Fuzzy System MATLAB Example

These graphs are the same as the input and output graphs above; however, the voltage response is slightly higher at 21.7 V at the same input of 28°C. This is because MATLAB uses the centroid method to defuzzify, which returns a more "geometrically accurate" response. This does not mean that the weighted average method is incorrect, but rather that the defuzzification technique is different. In both cases, they are approximating a response through human linguistic rules and bounds, so a small difference from using a faster technique is inconsequential. What matters the most is that both techniques return the same values at key points such as when the temperature is 30°C (in which both techniques return 20 V). Should more precision be required, it is possible to modify the number and size of input functions to become more precise as certain values are approached, but if extreme precision is truly desired, then fuzzy logic is best avoided in place of another control technique. The use of the centroid or weighted average alongside linguistic to find the appropriate response makes fuzzy logic highly geometric and intuitive and is one of its greatest strengths. Like a PID however, it must be fine-tuned, therefore, one must take that into consideration before implementing it.

The above example is a simple case with one input and one output, but it is possible to account for more than one input. For example, the rate of change of temperature could be accounted for as an input, then, creating membership functions for it. The ruleset would then have to be updated to include every combination of inputs. The controller would then be complete, but if more precision is required, one would need to update the membership functions through a fine-tuning/debugging phase.

As an example of how to include multiple inputs, take a similar system to the temperature controller where both temperature and rate of change in temperature are taken into account. Using an example plant, the block diagram of the control system is as follows:

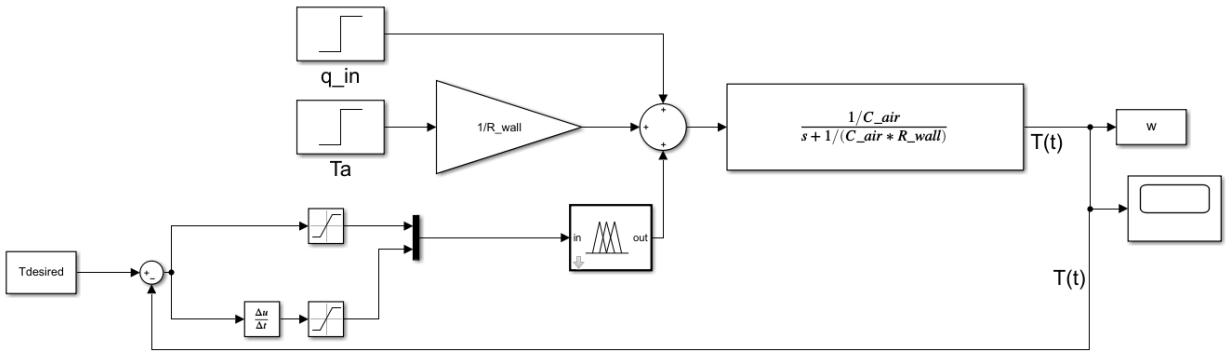

Figure 2.7: Temperature Control Example Block Diagram

Note, in the above diagram, there are saturation blocks present on the inputs to the FLC. This is to avoid any errors that may arise from temperatures winding up outside the operating range of the FLC. As will be seen above, when using trapezoidal membership functions, the intention is to designate values outside a certain range as too hot or too cold. This means the saturation blocks do not affect the fuzzy logic calculations.

Before building the controller, it is important to know what the goal is and how it will operate. For this example, the plant is controlled by three values, heat transfer into the system from external sources, the current atmospheric temperature, and the heat transfer from our “heating mechanism” that is directly controlled by the FLC. In this example, the desired temperature will be 70 +/-0.5°C. The process to create a controller for this is like the previous example:

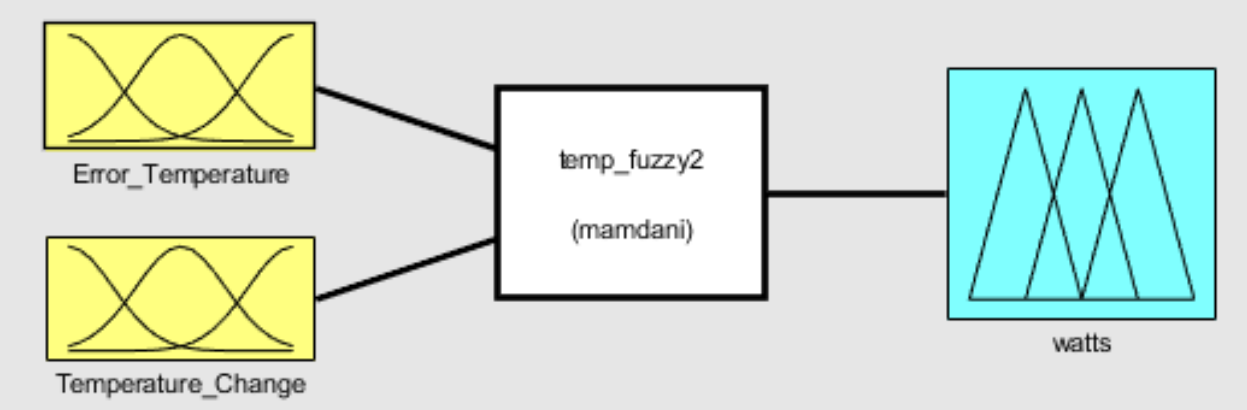

Figure 2.8: Fuzzy Logic 2-Input Example

The above diagram represents the fuzzy logic interface in MATLAB where the number of inputs is shown on the left, and the number of outputs is shown on the right. Note that the figures inside the boxes in Figure 2.8 do not represent any real values and are only used to distinguish one from the other. In this case, the two inputs will be the temperature error (how far away it is from the desired value) and the rate of change in temperature; and the output is the power (in Watts) by the heating element. The block in the middle represents the fuzzy logic rule application, converting fuzzified input values on the left into defuzzified output values on the right.

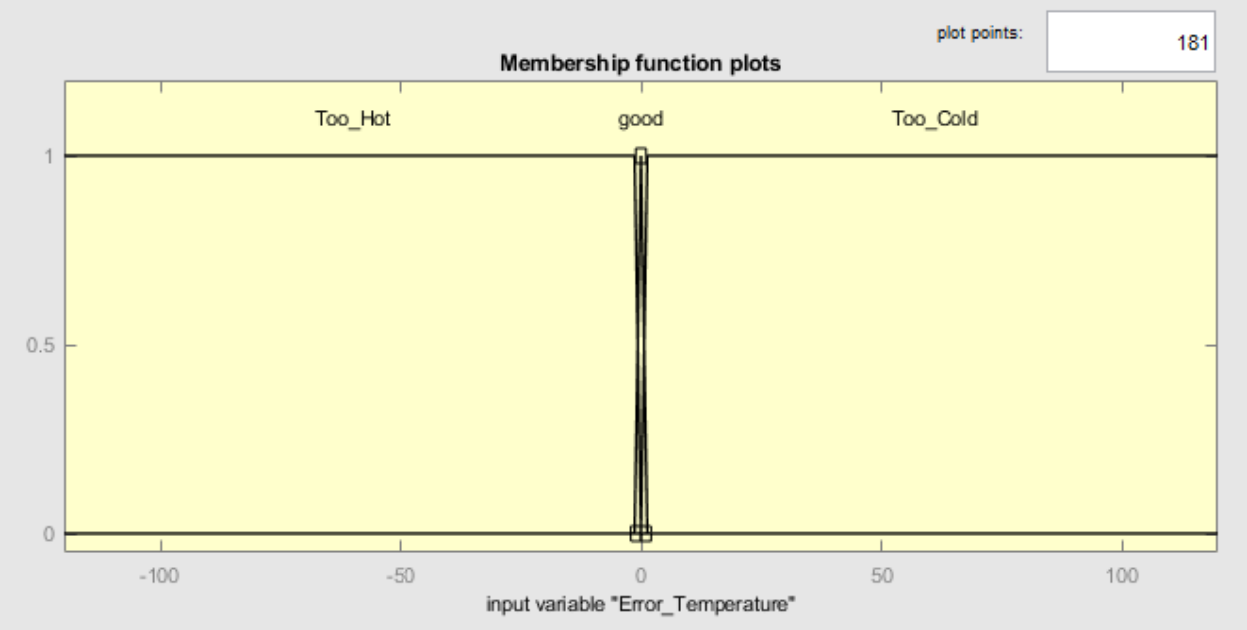

Figure 2.9 Input Membership Functions for Temperature Error

As can be seen in the figure 2.9, the triangular membership function is much narrower than in Figure 2.2. In this example, the goal was to get within half a degree of the setpoint, so the values for the membership functions are in the table below:

Membership Function Name	**Membership Function Parameters**
Too_Hot	[-120.5 -120 -0.5 0]
good	[-0.5 0 0.5]
Too_Cold	[0 0.5 120 120.5]

Table 2.2: Temperature Membership Function Parameters

The values +/-0.5 in the table represent the half-degree window to get to the desired temperature. Anything greater than 0.5°C in magnitude is considered no longer considered a temperature

classified under the “good” membership function. Likewise, the membership functions and parameters for change in temperature are as follows:

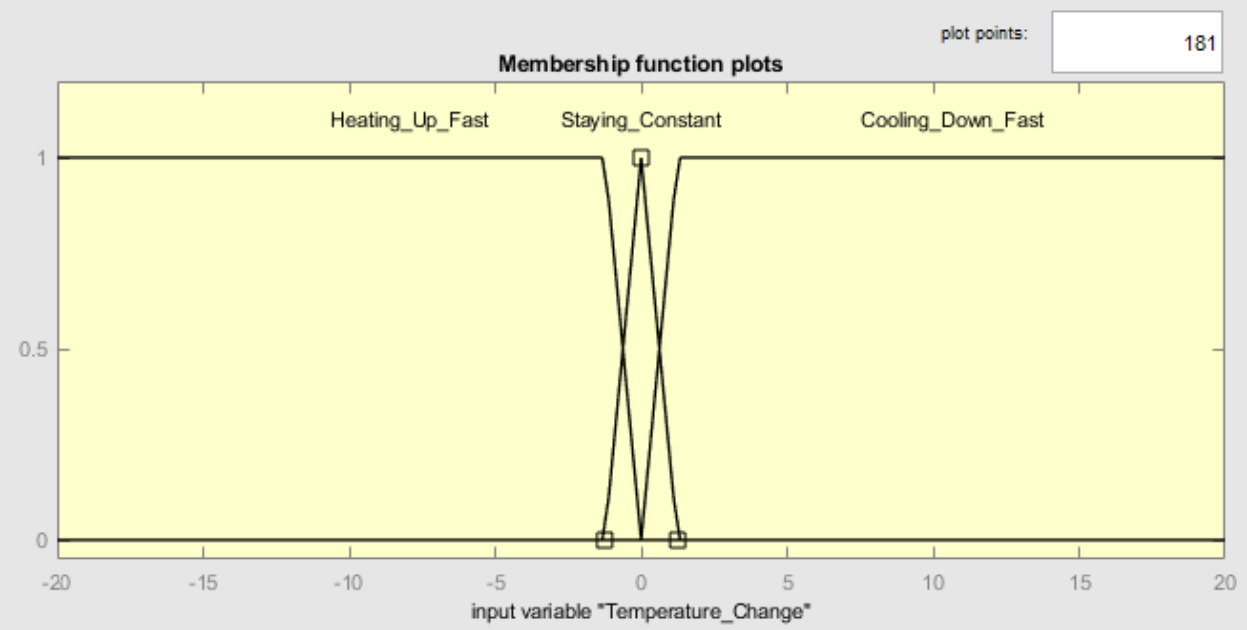

Figure 2.10 Input Membership Functions for Temperature Change

Membership Function Name	**Membership Function Parameters**
Heating_Up_Fast	[-21.25 -20 -1.25 0]
Staying_Constant	[-1.25 0 1.25]
Cooling_Down_Fast	[0 1.25 21 21.25]

Table 2.3: Output Membership Function Parameters

In this case, the value 1.25 was used after measuring the response of the plant without the controller. Anything greater in magnitude than 1.25 is considered fast.

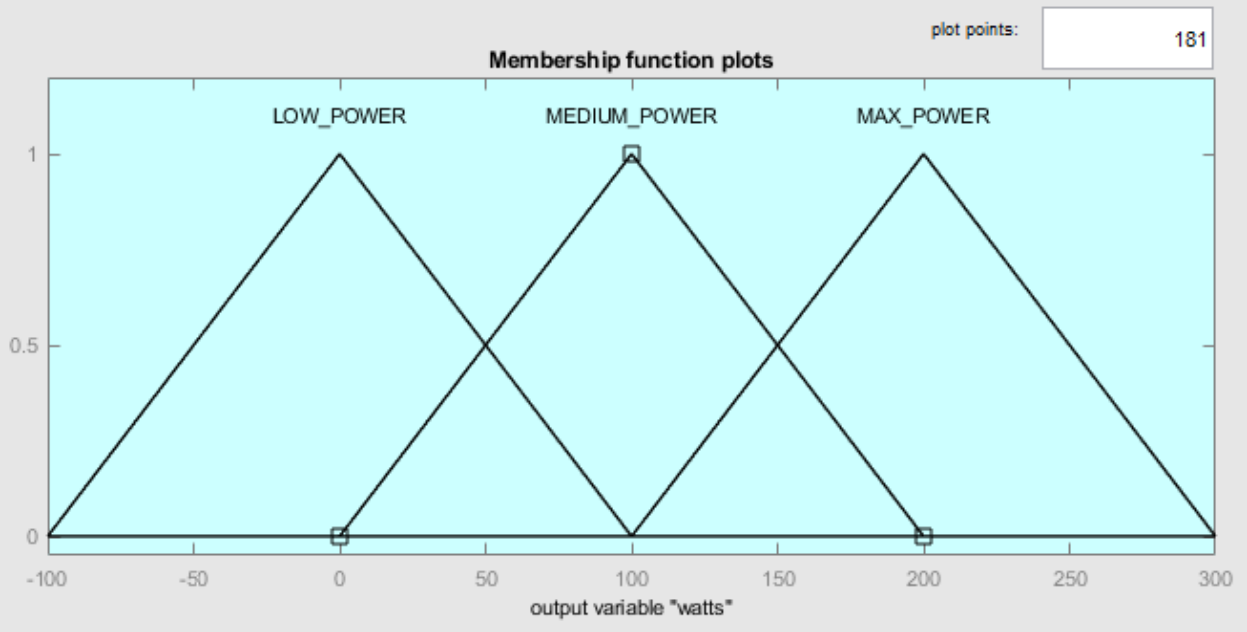

Figure 2.11: Output Membership Function Example 2

The output is very similar to the previous example. Here, the peaks of the triangular membership functions are is the output of the controller (0, 100, 200). The values of -100 and 300 are shown in the operating window to keep the graphs symmetrical for the centroid calculation. With inputs and outputs created, the controller now needs rules.

Temp. Error / Temp. Change	Too_Hot	good	Too_Cold
Heating Up Fast	LOW_POWER	LOW_POWER	MEDIUM_POWER
Staying Constant	LOW_POWER	MEDIUM_POWER	MAX_POWER
Cooling Down Fast	MEDIUM_POWER	MAX_POWER	MAX_POWER

Table 2.4: Fuzzy Logic Example Rules 2 (3x3)

With these rules in place, the FLC was complete and could be used in Simulink. The results are below:

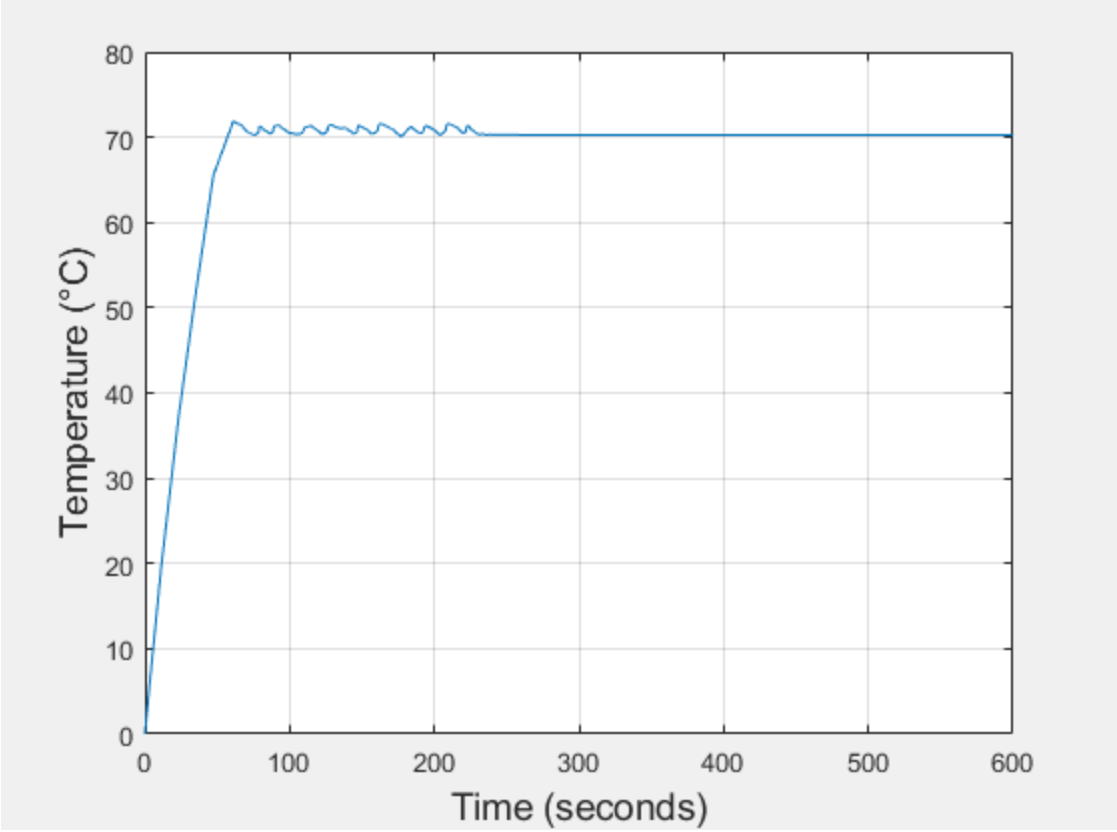

Figure 2.12: 2-Input Fuzzy Logic Example, Temperature (°C) vs Time (s)

As can be seen from the above figure, the temperature manages to stabilize to 70°C, however, there is some temperature oscillation about the setpoint occurring. This is indicative of a too extreme response close to the desired temperature. This can be somewhat alleviated by widening the membership functions to be less extreme. The graph below was made with the same system, but membership function values of +/-0.5 were replaced with +/-1.

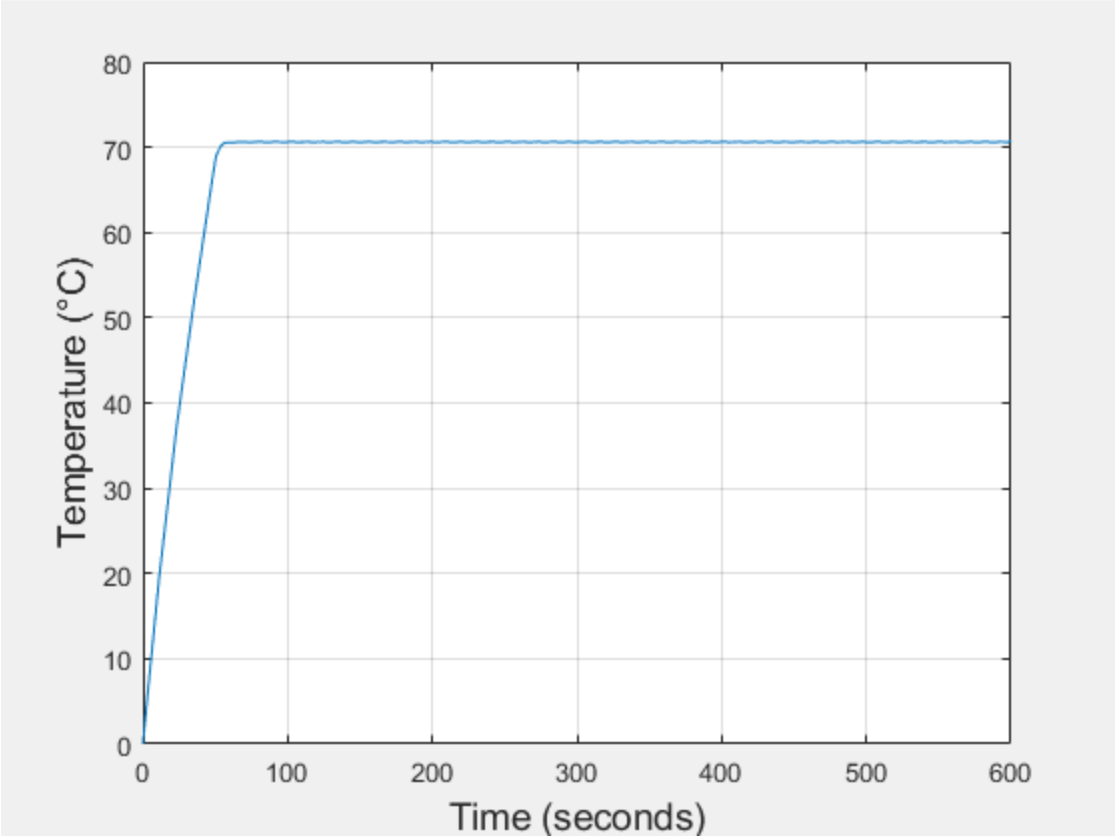

Figure 2.13: Widened Membership Functions, Temperature (°C) vs Time (s)

By widening the membership functions, the response is less extreme, and the hysteresis levels out. This example demonstrates how to incorporate fuzzy logic onto a 2-input system as well as the benefits of fine-tuning membership function parameters.

To demonstrate the effectiveness of this method, the results of using fuzzy logic and PID controllers to control an inverted pendulum system were compared (Afaq, Asghar, Abbasi, Wallam, & Saeed, 2015). Their experiment involved using three controllers, a PID, a fuzzy logic controller, and a hybrid fuzzy-PID controller. The results were compared in terms of both settling time, steady-state error, and the force applied to the system. They found that the settling time for the three controllers was similar, however, the fuzzy-PID used the least amount of force to stabilize the system. They found that the use of this method allowed for a much easier control

of the system since it did not rely on a transfer function (and therefore discrete function) to operate. This method of creating a controller based on the condition of the system rather than the model makes it especially effective at handling non-linear systems difficult to model accurately. This demonstrates the power of this method, as it does not require a full computational model, however, it is still important for the designer creating the controller to understand how the system should respond. A rule set was needed to accurately describe what the system should do in response to the different operating conditions (Afaq et al. 2015). They assembled a large table of fuzzy logic rules that described all possible configurations of angular position and angular velocity and how to respond to them. That model used the reading of both angle and angular velocity to build its logic; therefore, if it were to be implemented into the proposed design, there would be a need for a way of measuring angular velocity, or use another variable that will yield a mathematically equivalent result, like rapidly taking the derivative of the position. Using a fuzzy logic controller to control an inverted pendulum by taking the inputs to be the angular position and velocity is possible. (Peng & Wei, 2010). This account for velocity shows that sometimes more information is needed from the system for the controller to give an appropriate response; however, even when velocity cannot be directly measured, it is possible to account for similar variables like the change in position between steps (Akmal et al. 2017). It was also shown that a faster settling time was possible for an inverted pendulum if more fuzzy logic rules are used (Akmal, Jamin, & Abdul Ghani, 2017). It appears that the FLC is a reliable method for stabilizing a nonlinear system such as the inverted pendulum. Fuzzy logic can handle these systems and gets more precise the more rules are implemented. Fuzzy logic has been shown to require low computational power to run and it is simple to program. However, as stated earlier, as the number of inputs increases, so does the number of rules, exponentially. For this reason,

fuzzy logic is best used in systems with less variables to track. R.E. King et al. (1994) stated that a major drawback of fuzzy logic is the extensive debugging phase when designing a controller (King, Magoulas, & Stathaki, 1994). This has logical merit as the foundation of the controller's mechanism requires more linguistic "soft-computing" methods that are founded in human understanding of the system; therefore, the system contains human error. This makes optimizing a fuzzy controller difficult, but if it can stabilize the system correctly, the benefits may outweigh the drawbacks. R.E. King et al. (1994) demonstrated that for a multivariable fuzzy logic, the table of scenarios built by rules expands exponentially. As explained by the example before, there was a 3^1 vector of rules with one input. If there were more inputs, it could become a 3^2 table of rules with 2 inputs, or 3^3 rules with three inputs. This can lead to very complex geometry with many inputs. To smooth out the response in a complex system, R.E. King et al. (1994) suggested a "meta rule" for handling it where adjacent responses in the ruleset to have output responses close to each other. In other words, the system should not go from a low to a high response suddenly, it should transition smoothly. This helps prevent erratic behavior and unnecessary strain on the system components. From their research, they were able to essentially simulate a human operator using fuzzy controls. In this case, it was to control a hypothetical cement mixing plant where pressure, mill power, and returns (separated cement) were the input variables. This resulted in 3 inputs with 5 rules each, making a 5^3 number of rules. While the system was effective in imitating a human operator, it will take more research to determine if fuzzy logic controllers are efficient with an even greater number of rules. To determine what rule set to use, it is necessary to either build them by hand or to use a computer program. Sunu S. et al. (2016) was able to develop a fuzzy logic control scheme through the MATLAB fuzzy logic toolbox. This allowed them to construct and simulate a PWM response from a controller using

fuzzy logic for high to low values. (Babu & Pillai, 2016). In their system, they utilized a Takagi-Sugeno fuzzy logic controller, where the membership functions are constants rather than geometric shapes (Fast = 255, Medium = 150, Slow = 60, Stop = 0). This version of fuzzy logic is less visually intuitive, but functions similarly to the previous examples. In this case, the weighted average method is used to determine the output. Since the rules of fuzzy logic are linguistic, they were able to easily be transferred to a microcontroller (they used an Arduino Mega). Their controller was used to control a human operated vehicle called a Segway, which functions identically to a two-wheeled inverted pendulum. The use of an Arduino Mega should be noted as it is of similar processing power to common low-cost microcontrollers. That research showed a proof of concept by bridging MATLAB code and the language of a microcontroller through fuzzy logic. These control methods all have potential to control and stabilize a system, even by using drastically different methods. In the following chapter, systems in which these control methods can be applied will be modeled and discussed.

CHAPTER III

MATHEMATICAL MODELS

To apply control systems, it is very useful to have a model for a system one would want to control. One such example is an inverted pendulum. The inverted pendulum is a nonlinear, unstable system that is affected by both linear and angular positions and velocities. This system is very well understood and has been studied for many years; therefore, it is possible to validate results by cross-checking them with an already existing model. A typical system is like the one by the University of Michigan (Messner & Tilbury, 2019). This inverted pendulum system is akin to a vehicle such as the Segway, which uses human input to determine where to go. While this can be implemented using MATLAB and LQR as shown in the literature, implementing it on less powerful hardware poses its own unique sets of challenges. As such, the model built should take this into account. Since available hardware and software packages will influence the systems that can be controlled, the experiments in Chapter IV influence the models in this chapter. The basics of the inverted pendulum are as follows:

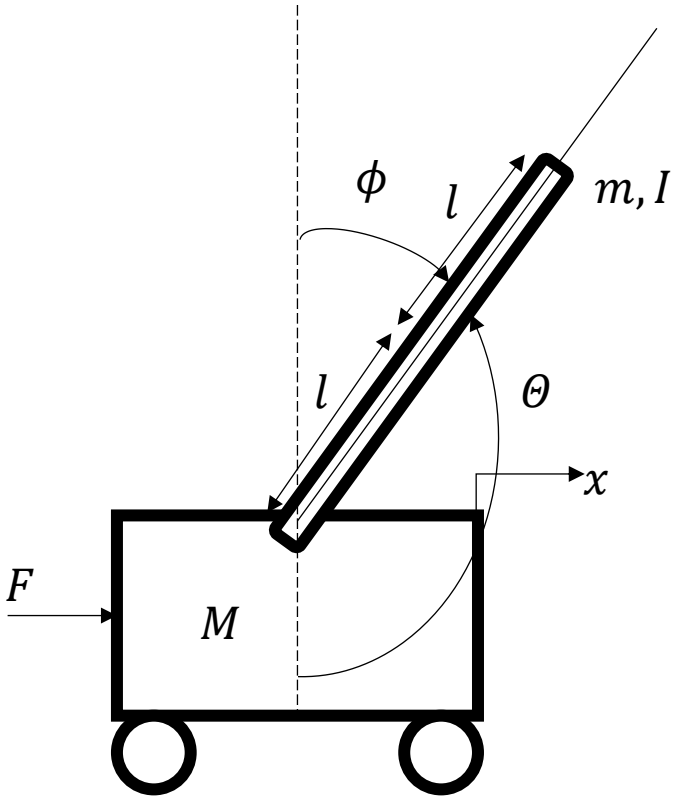

Figure 3.1: Annotated Typical Inverted Pendulum

The above figure represents the standard cart and inverted pendulum system. Because of the system's two components (the cart and the pendulum) two separate, coupled equations must be arranged to describe the motion of the system. To establish a foundation for an inverted pendulum, the method to acquire the equations of motion for a typical system by the University of Michigan is recalled in equations 3.1-3.10. (Messner & Tilbury, 2019). These equations have also been linearized and modeled in other works in the literature, but, the principle of how they operate is the same (Ogata, 2010). This system has also been broken down into the free body diagram below:

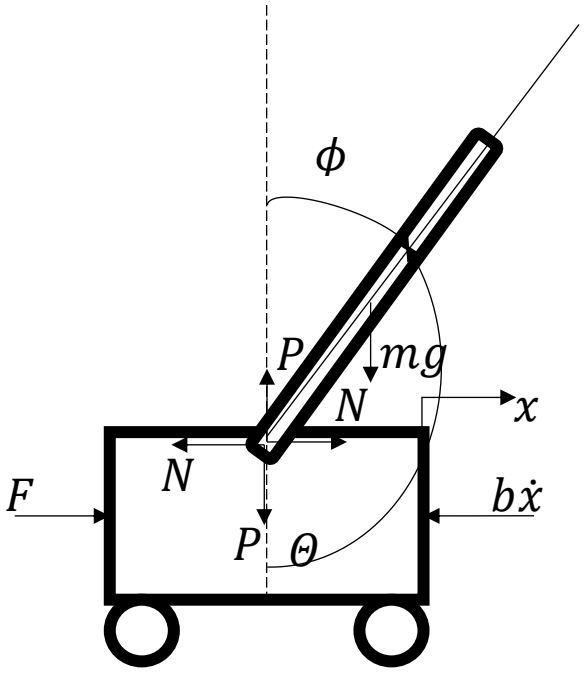

Figure 3.2: Free Body Diagram of Typical Inverted Pendulum

The equation of motion for the mass of the cart is as follows:

$$M\ddot{x} + b\dot{x} + N = F \tag{3.1}$$

Where M is the mass of the cart, m is the mass of the pendulum, b is the viscous friction coefficient in the x-direction, N and P are the vertical and horizontal forces between the cart and the pendulum, and F is the horizontal force applied to the cart (i.e. the wheel traction, or the tension of a belt). The mathematical model consists of the following equations:

$$N = m\ddot{x} + ml\ddot{\theta}cos\theta - ml\dot{\theta}^2 sin\theta \tag{3.2}$$

Combining the two equations, we arrive at:

$$(M + m)\ddot{x} + b\dot{x} + ml\ddot{\theta}cos\theta - ml\dot{\theta}^2 sin\theta = F \tag{3.3}$$

As can be seen from equation 3.3, this equation is nonlinear By assuming small angles around an operating point, the system can be linearized:

$$cos\theta = \cos(\pi + \phi) \approx -1 \tag{3.4}$$

$$sin\theta = \sin(\pi + \phi) \approx -\phi \tag{3.5}$$

$$\dot{\theta}^2 = \dot{\phi}^2 \approx 0 \tag{3.6}$$

Where ϕ represents the angle's position from equilibrium or $\theta = \pi + \phi$. This transforms equation 3.3 into:

$$(M + m)\ddot{x} + b\dot{x} - ml\ddot{\phi} = F \tag{3.7}$$

With a relationship between the cart's linear acceleration and the pendulum's angular acceleration, it is necessary to derive the equation of motion for the pendulum as well. Using the original notation with theta, the equation is:

$$Psin\theta + Ncos\theta - mgsin\theta = ml\ddot{\theta} + m\ddot{x}cos\theta \tag{3.8}$$

By summing the moments about the centroid of the pendulum arm, it is possible to eliminate P and N from the equations. The equation simplifies to:

$$(I + ml^2)\ddot{\theta} + mglsin\theta = -ml\ddot{x}cos\theta \tag{3.9}$$

Where I is the mass moment of inertia of the pendulum about the axis of the hinge. Applying equations 3.4 and 3.5, equation 3.9 becomes:

$$(I + ml^2)\ddot{\phi} - mgl\phi = ml\ddot{x} \tag{3.10}$$

Using equations 3.7 and 3.10, there is now a linear way of representing the motion of both the pendulum and cart system. This can allow for both state space or linear representations that can be controlled and simulated in MATLAB.

A second system that will be explored in Chapter IV is a simple motor arm. This simple arm has been discussed in a paper by the University of Texas-Pan American (Vasquez, Kypuros, & Villanueva, 2010). The full block diagram of the system can be seen within Chapter IV in Figure 4.21. It is a simple system with an arm attached to a motor.

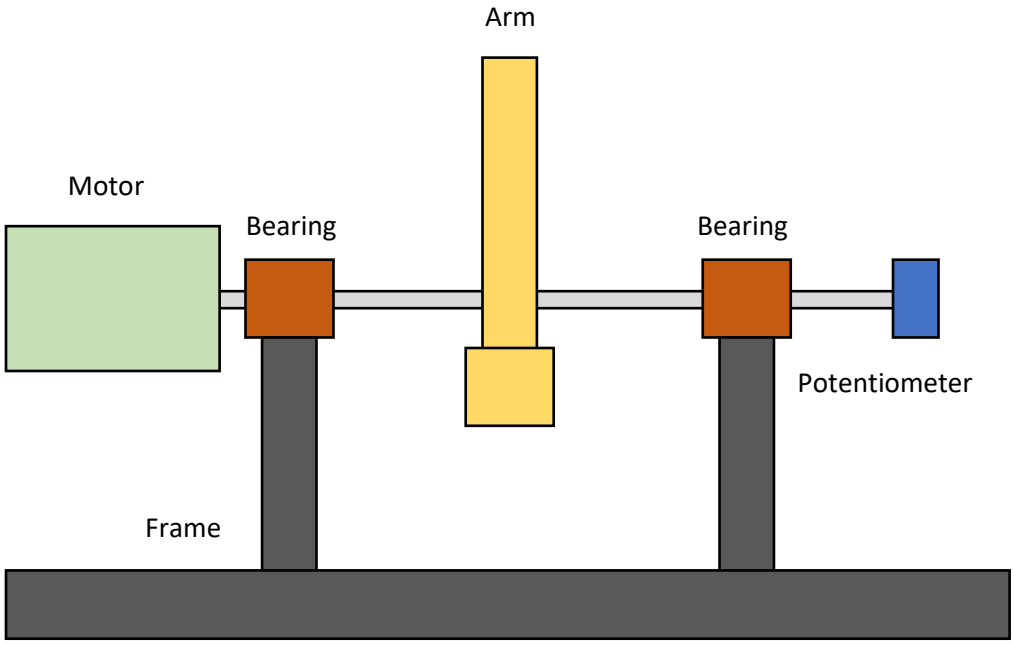

Figure 3.3: Motor Arm Diagram

The transfer function relating voltage applied to the motor to the angular position of the arm is as follows:

$$\frac{\theta(s)}{E_{in}(s)} = \frac{1}{s}\left(\frac{R_a N}{L_a(N^2 J_m + J_L)s^2 + [L_a(N^2 b_m + b_L) + R_a(N^2 J_m + J_l)]s + R_a(N^2 b_m + b_L) + k_b^2 N^2}\right) \quad (3.11)$$

Where L_a is the inductance of the motor, N is the gear box ratio, J_m and J_s are the moments of inertia for the motor and load, respectively, R_a is the motor resistance, b_m and b_L are friction coefficients due to the motor and bearings, respectively, and k_b is motor torque constant. Using this, it is possible to run simulations based on the voltage applied due to a microcontroller.

Using a micro controller such as Arduino to control the system implies several constraints. One constraint is how the Arduino approximates analog voltage input and output. To give an analog output, a pulse-width modulation (PWM) is given by the Arduino to fire the transistors in an H-bridge at a certain relatively high frequency. This type of output is a sequence of high and low values that are analogous to a square wave function whose time on and off is changed to adjust its duty cycle. This is because microcontrollers can only send signals as a mixture of low and high (0 and 1) values through the digital output pins. What this means for the system is that when a value for the motor voltage (between the maximum and minimum value) is needed, the microcontroller can approximate it with the duty cycle of the PWM signal. Take the example where a motor would need 50% of its strength to balance the system, then, the duty cycle of the PWM signal that the microcontroller needs to output is 50%. Effectively this means that the motor experiences a voltage equivalent to half the maximum voltage provided by the power supply. This method allows microcontrollers to output a wide range of responses. Another constraint of using an Arduino to control the system is the time response of the controller. As discussed in Chapter II, if the sampling rate is too low, the response of the controller will be out of sync with the system, possibly failing to stabilize toward desired setpoint. The sampling rate is largely affected by the efficiency of the code and hardware limitations. If the controller is fast enough to keep up with the system, a microcontroller can be a substitute for other means of control such as human operator. To determine if a system is being sampled fast enough, it is possible to use the natural frequency of the system. The equation to find the natural frequency is generally as follows:

$$\omega_f = \sqrt{\frac{k}{m}} \tag{3.12}$$

Using the model of the system, it is possible to find this value. So long as the sampling frequency is greater than 10 times the natural frequency, the system is sampled sufficiently fast for control purposes. (Fadali & Visioli, 2013) It should be noted, in cases with damping, the damped natural frequency is used instead.

In the next chapter, the control systems and the models discussed in the previous chapters will replicated and tested. Simulations of models from literature using MATLAB and the implementation on microcontrollers will also be tested.

CHAPTER IV

CONSTRUCTION AND TESTING

It is important to know if a low-cost microcontroller (like an Arduino UNO) can handle the multivariable model and processing power required to control an inverted pendulum. While the literature shows that an Arduino is more than capable of controlling complex systems in real-time, there is still the uncertainty of how it will handle the controlling both the angular position and the translational position of the pendulum, for example. To test if this were possible, it was important to have a basic hardware setup. An available hardware setup in the form of a ball and beam system was used to test code implementation on hardware. This setup was a motor in the center of a beam that was about 1 meter long with an ultrasonic sensor on one end. While the original setup had yet to be optimized with its own PID controller and code, it was enough to test out new control schemes like fuzzy logic and debug design flaws like motor stall due to low duty cycle output. To validate that the Arduino did have speed and computational power to handle the processing and output of data in real-time, a PID controller was tested. The PID controller used a relatively high derivative coefficient compared to the proportional and integral coefficients. The values $Kp = 3$, $Ki = 1$, and $Kd = 100$ were obtained after numerous trial-and-error tests. The ball was not completely balanced by these values due to the current hardware limitations, but these values for the PID's constants got the ball close before the system became unbalanced (more details on these limitations discussed later in the chapter). The system did show the capacity to

respond fast to changes, which was a crucial factor. Through use of the built-in serial monitor in the Arduino software, it was possible to see how fast the PID controller was responding, which was found to be approximately at 34 Hz. While this value is not indicative of the true sampling rate because the printing to the monitor itself adds some delay, it is a good indicator that the program is not too slow. With this in mind, it was time to test out a fuzzy logic controller on the hardware. To start, a simple fuzzy logic test controller built from MATLAB was constructed. It had 2 inputs, Error (ball's displacement from the desired position), and Speed (velocity of the ball) with 3 membership functions. The output consisted of 3 membership functions and would output a PWM value between -255 and 255. (In this case, a value of -255 functions the same as a positive one, but the direction in an H-Bridge is changed). The rules and output response to rotate the arm for each combination of inputs were as follows:

Position Error / Velocity Error	Negative Error	Okay Error	Positive Error
Negative Speed	Positive	Positive	Neutral
Okay Speed	Positive	Neutral	Negative
Positive Speed	Neutral	Negative	Negative

Table 4.1: Fuzzy Logic Ball-Beam Rules (3x3)

The membership functions and logic system built are as follows:

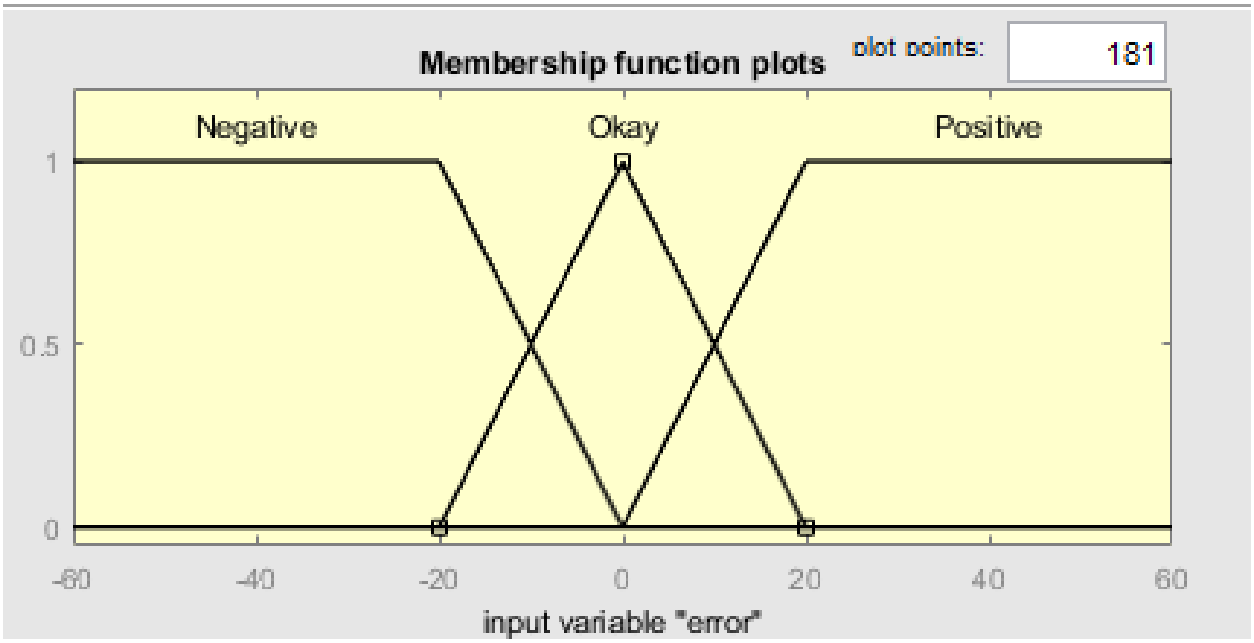

Figure 4.1: Membership Functions for Error

This figure sets up what is considered high error. Anything above 20 cm away from the target is considered positive or negative and warrants a response, however, anything within 20 cm will increasingly belong to the “okay” membership function where the response should be lessened.

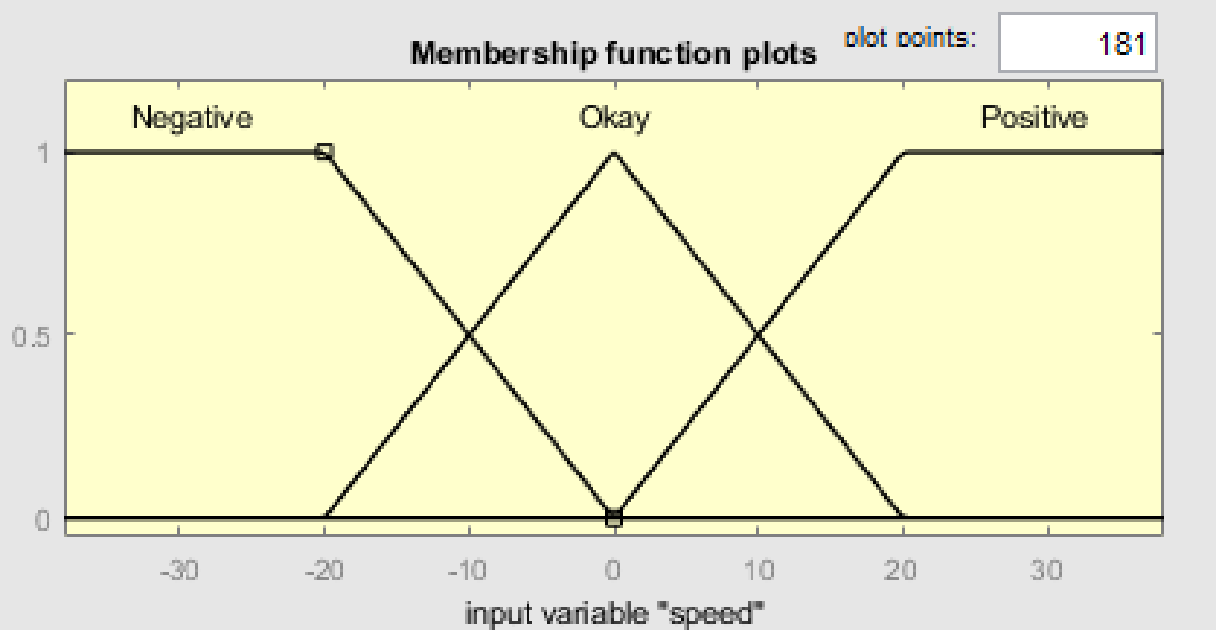

Figure 4.2: Membership Functions for Speed

Similarly, the speed variable has anything above 20 cm/s in either direction is classified as positive or negative or anything between is “okay” speed with the ideal speed being 0 cm/s for the “okay” membership function. The way the centroid calculation will work for having two inputs is through the use of proportions. Recall that each input membership function can only return a value between 0 and 1. These proportions are what dictate what proportion of the output membership function is expressed. In the case of two inputs, the lesser proportion is used. So, having a return of 0.5 Negative for position, and 0.3 Negative for speed, will result in a Positive output membership function with a height of 0.3. It should be noted that which output membership functions are expressed is dependent on what rules are currently in effect and so there may be multiple valid rules firing at any given time.

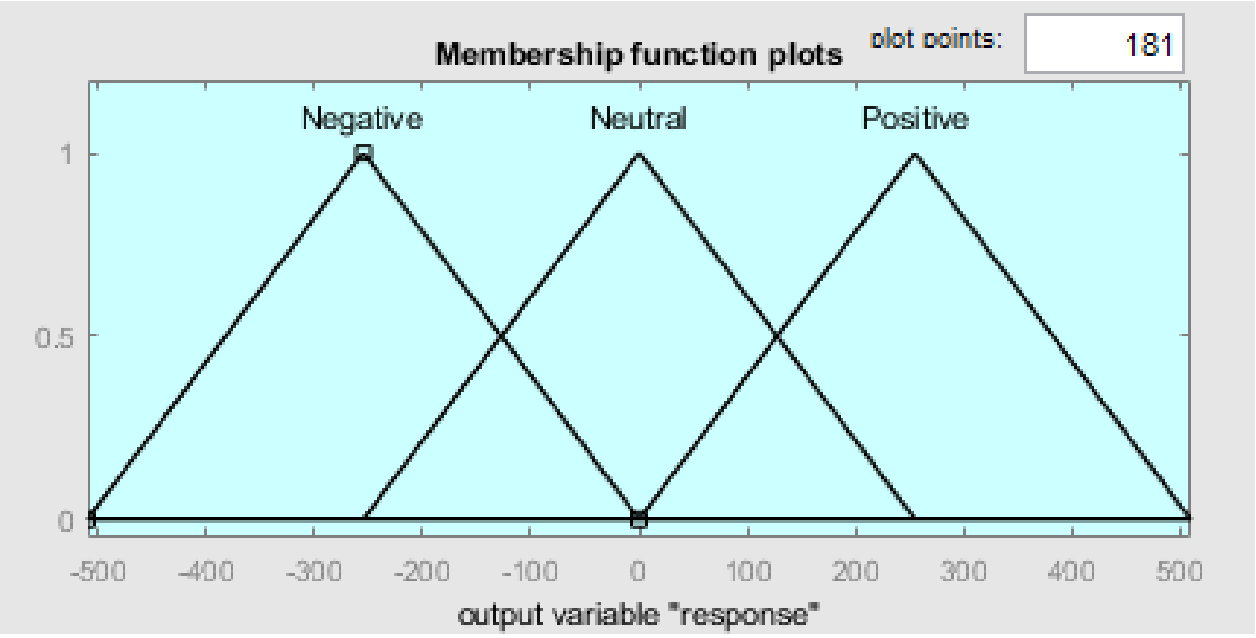

Figure 4.3: Output Membership Functions

The output membership function has 3 peaks at -255, 0, and 255 for the Negative, Neutral, and Positive membership functions, respectively. This ensures that the logic system will not exceed the range of a PWM response from the microcontroller. In this case, the peaks of the output membership functions are the centroids of those functions, so the output will never reach a value greater than 255 or less than -255.

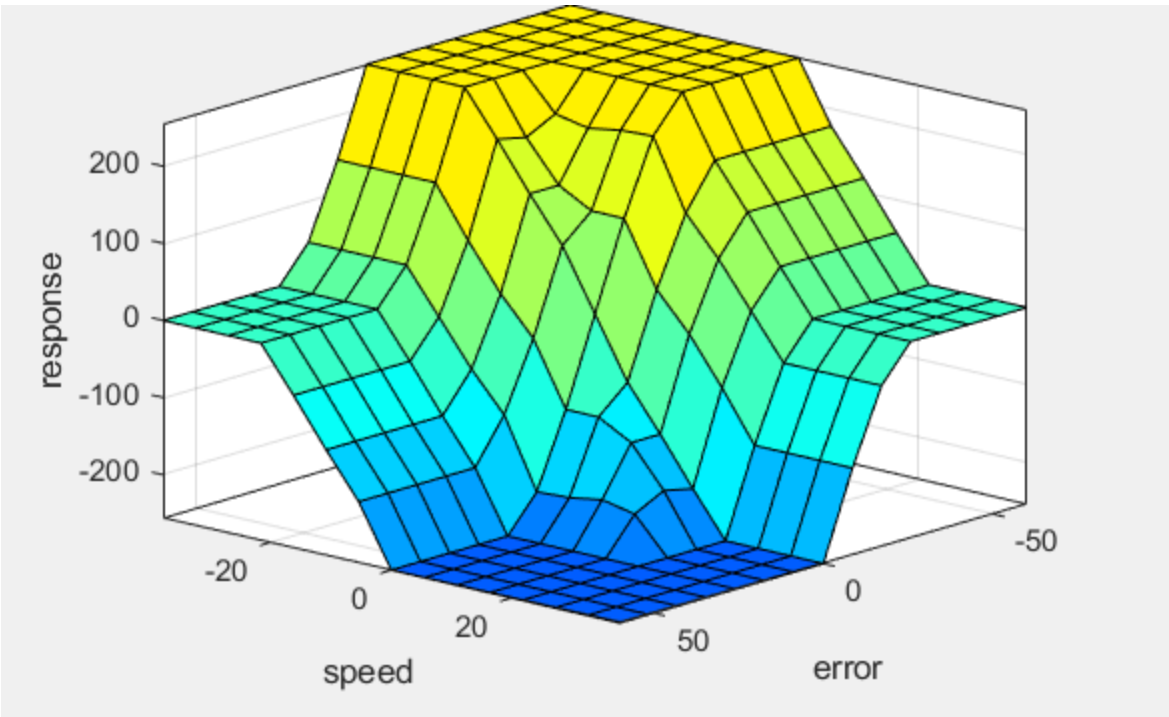

Figure 4.4: Surface Graph of the Fuzzy Logic Rules

Finally, here is a surface representation of how all 9 rules interact with each other. In summary, if the speed or distance is too negative, the system will output a positive response to counter it and vice versa. While the values selected are mostly arbitrary, the main purpose of this is to see how an Arduino Uno will handle these rulesets. Given the above graph, if two inputs were selected to be position 7 cm and speed 24 cm/s, the rules would apply as follows:

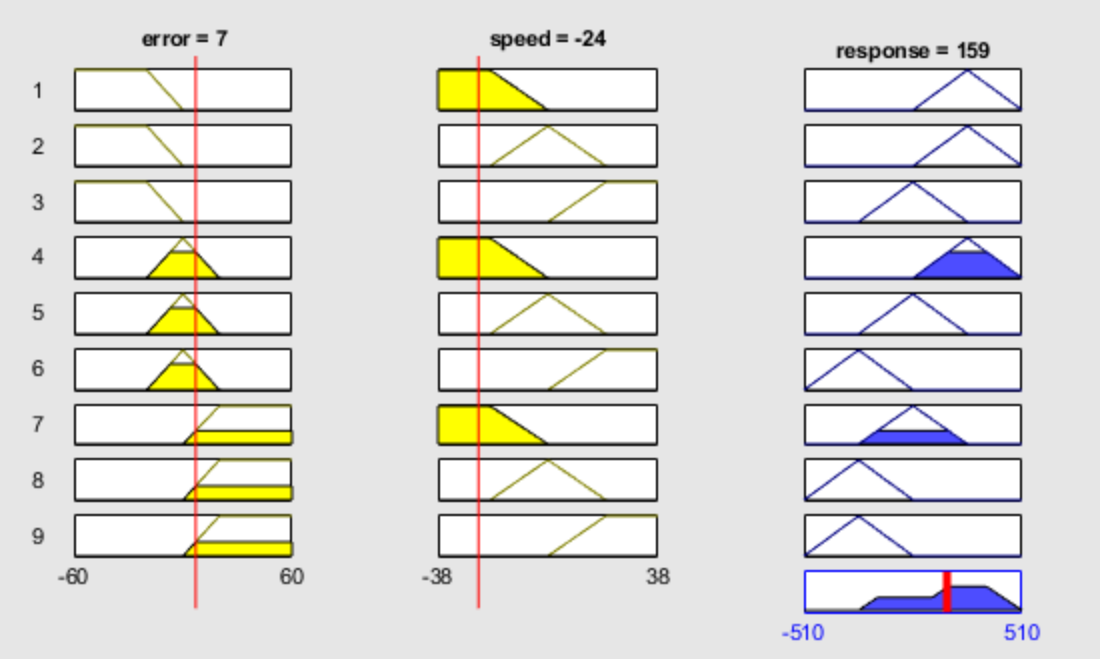

Figure 4.5: MATLAB Rules – 2 Inputs

Notice how there are 9 rows instead of the 3 in Figure 2.6, this is because there is one row per rule, as each combination of inputs needs an output. In this case, a position of 7 cm and speed of -24 cm/s results in a PWM response of 128.

With a logic system created in MATLAB, it was time to transfer it over to Arduino. The fastest method to transfer it appeared to be an online converter tool that converted MATLAB's fuzzy logic interface (which is built to be compatible with universal coding languages) into an Arduino script. This is because it meant no additional coding had to be done on the Arduino. This made it very simple to get the logic onto the board; however, the result was less than satisfactory. After merging the codes to get a working input/output system running on fuzzy logic, a look at the serial monitor revealed a disparity between sample rates. Compared to the previous PID running at 34 Hz, the system was now running at only 5 Hz on the serial monitor. This was unacceptable as the ball and beam (and more importantly, the inverted pendulum) can

go from one extreme to the other in less than half a second. With this test, it was clear that if fuzzy logic was to be implemented, it would need to be using a much more efficient coding style. After some investigation, it was discovered that a fuzzy logic library created by Dr. Lira et al. (2012) had been included in the official list of Arduino libraries. The library claimed it was extremely fast and efficient in the handling of fuzzy logic. To test this, the same fuzzy logic interface from MATLAB was recreated from the ground up in Arduino using this library. The initial tests using random values seemed to work well and fast, following the rules that had been previously established. When the fuzzy logic system was used with the sensors, the result from the serial monitor showed a response speed of 33 Hz on the serial monitor. This is akin to the PID response and it might be capable of handling fast changing conditions. Due to a combination of faulty sensors and a lack of fine-tuning, the ball and beam system was not able to be balanced by this fuzzy logic ruleset. To be able to balance complex system like this, more fine tuning and debugging would need to take place. As for the sensor, it would sometimes read distance to be maximum for a brief period, making the speed calculation inaccurate. Despite this, the test had done its job in showing that with the library, it is possible to get a fuzzy logic controller running smoothly on a low-cost microcontroller. In addition, it proved that an Arduino can handle a 3x3 ruleset FLC with 2 inputs. Further testing of the Arduino will be demonstrated on a working system later in the chapter. In order to avoid having difficulty with hardware being damaged by faulty controllers, the safe option was to build a controller in MATLAB and Simulink and build a model around that; however, this comes with the challenge of making a controller that is both viable in MATLAB and an Arduino. MATLAB can handle many variables and membership functions at the same time and calculate them without having to worry about real time use. As such, it is possible to build inputs with many membership functions meant to fine tune the

system. The goal would be to create a working model and controller in MATLAB and transfer it to Arduino. Of the papers studied, M. I. H. Nour et al. (2007) was able to construct a fuzzy logic system that based its outputs on both angular position and angular velocity for an inverted pendulum in MATLAB. This logic employed the use of Takagi-Sugeno fuzzy logic, a more efficiently computable version than the one discussed earlier. However, in this paper, Fuzzy Logic took a backseat role in fine-tuning the constants of an LQR. In this system, an LQR is used to stabilize the system, while a fuzzy logic controller fine tunes the control law parameters. While this method could be used on a microcontroller, it may not be necessary if the LQR parameters are enough to stabilize the system. Another system that used Fuzzy Logic for an inverted pendulum was documented in a paper by Yanmei Liu et al. (2009). This system operated using a hybrid PID-fuzzy logic controller that took angular position and angular velocity as inputs, and outputs the force that act on the cart (Liu, 2009). This is shown in Figure 4.6. The paper provided enough information to replicate the results in Simulink. Below are both the results provided by Yanmei Liu et al. (2009) and the simulation used to replicate their results.

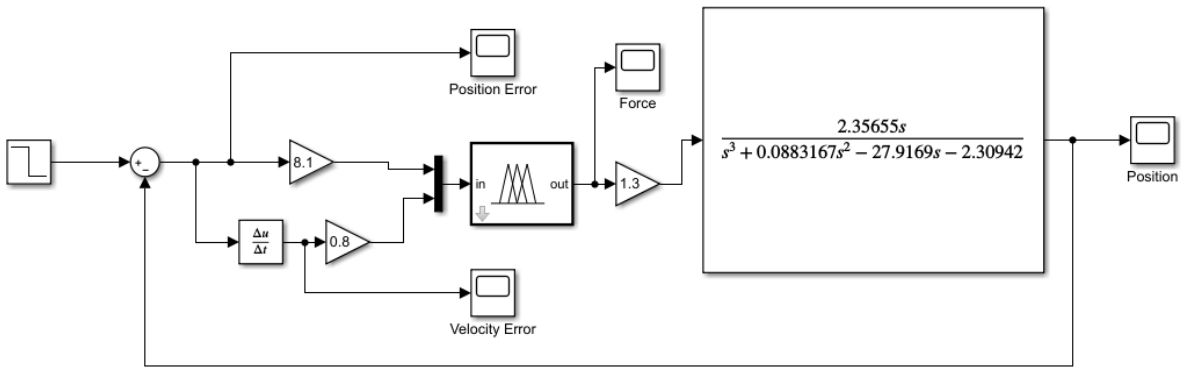

Figure 4.6: Transfer Function Block Diagram

The above diagram is a reconstruction of the block diagram by Yanmei Liu et al. (2009). It should be noted that instead of taking both angular position and velocity directly as inputs, the fuzzy logic controller takes the values of the error through a proportional gain of 8.1 and a derivative gain of 0.8, similar to a PID. In the ruleset, the FLC attempts to control both angular position and velocity by controlling the output force (the membership functions for these inputs are described further into the chapter). Their results, and the one produced by this diagram are as follows:

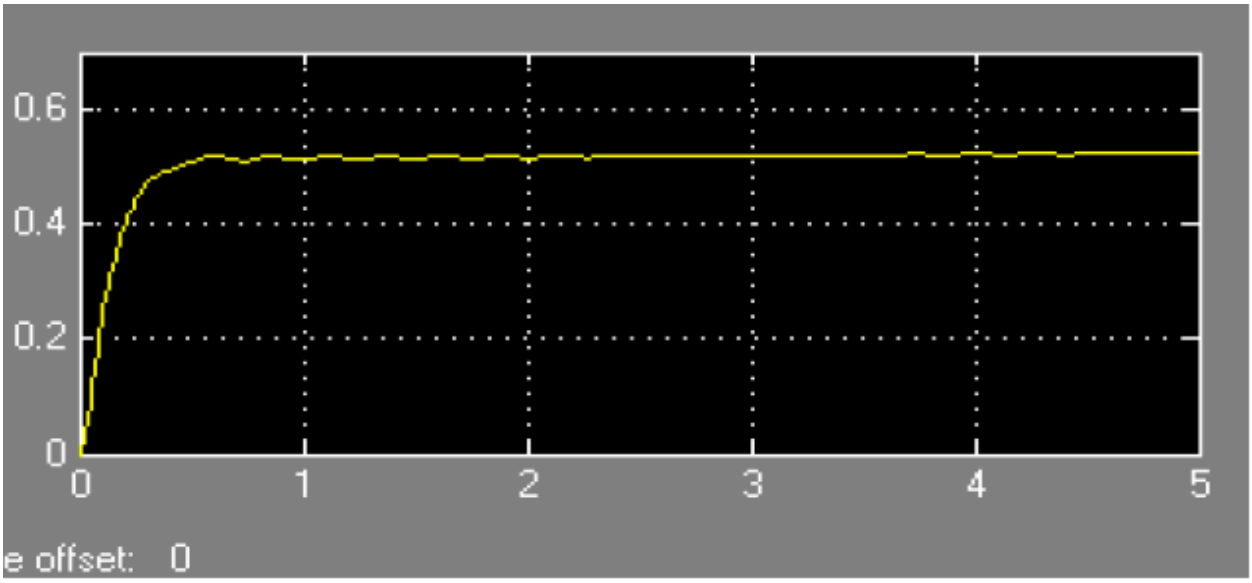

Figure 4.7: Simulation Result of Yanmei Liu et al. (2009) Angular Position (rad) vs. Time (s)

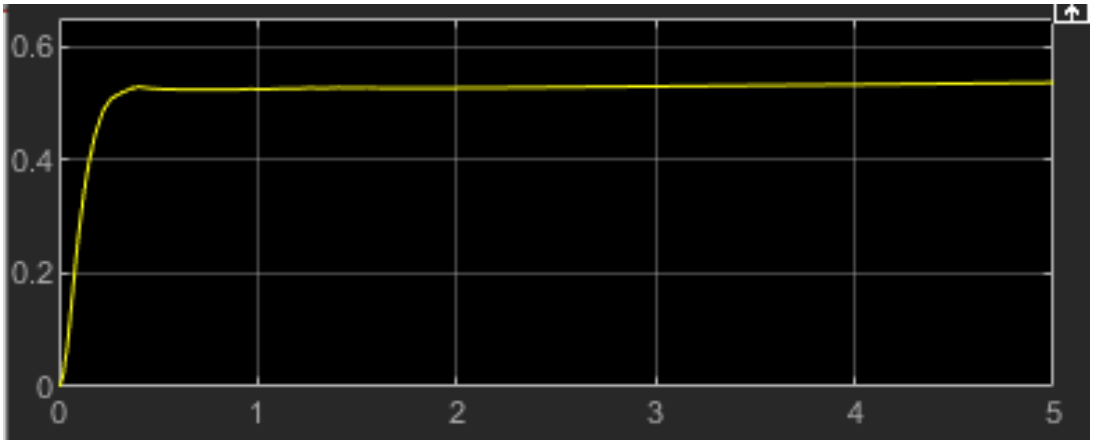

Figure 4.8: Replication of Yanmei et al. (2009) Results Angular Position (rad) vs. Time (s)

As can be seen by the above figures, the results from both Yanmei Liu et al. (2009) and the results from the attempt to replicate them are similar. With the enhanced image resolution, it is easy to see that there is a slight upward trend with the angular position over time. This is because in the paper, they set their target angular position to be 0.5 radians, and this position is an unstable point, and the system must constantly input work and accelerate for the pendulum to stay there. The figure demonstrates what begins to happen with time.

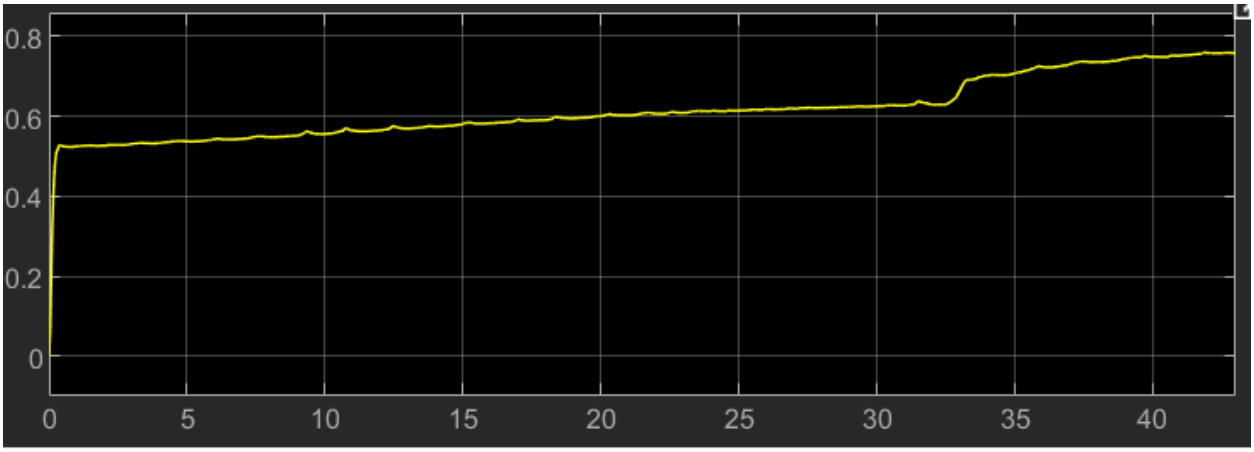

Figure 4.9: Extension of a 0.5 Radian Position Graph

Here the upward trend continues to rise, and further simulation reveals that it will continue to do so until it sharply jumps toward infinity when outside the operating window of the fuzzy logic. At approximately 32 seconds, there is a sharp upward turn in position, which is likely due to a rule change. While the other parts of the graph have a rule change with a smooth transition, this rule change at the edge acts differently. This is because of the geometry of the membership functions and how MATLAB uses the centroid method in calculation.

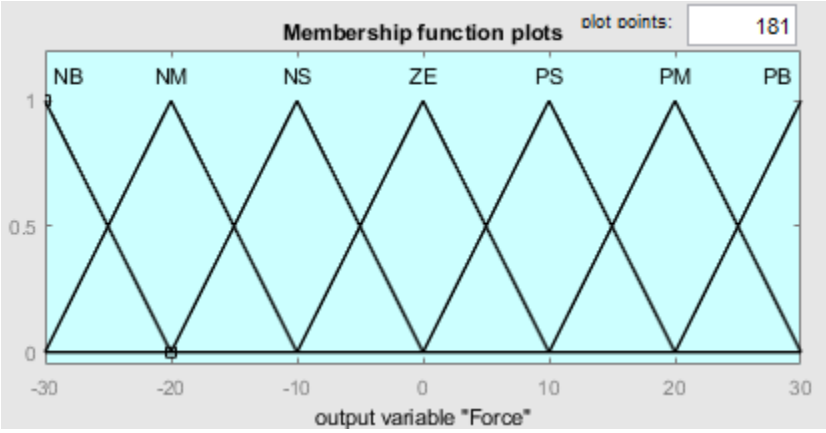

Figure 4.10: Replication of Output Membership Functions

The membership functions replicated keep the same notation where the first letter describes the sign (N = Negative, P = Positive) and the second letter describes the magnitude (S = Small, M = Medium, B = Big), ZE is used to represent 0. At 32 seconds, the force was measured to be approximately -20 N which is a peak of the NM membership function. This would mean that the most extreme response (NB) would be the next output in line.

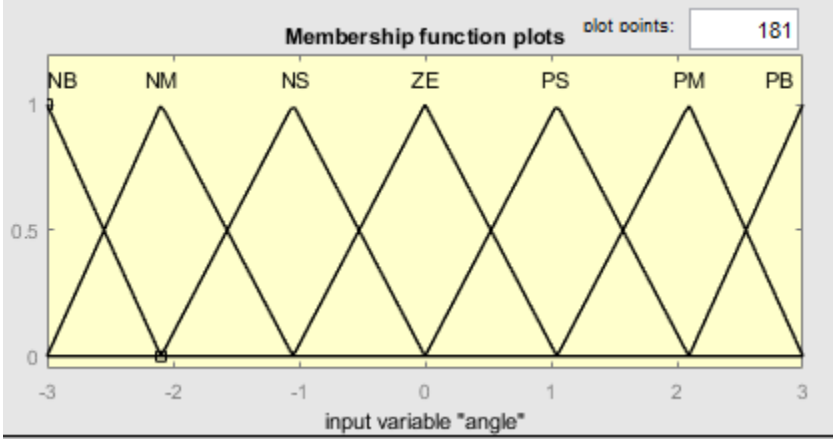

Figure 4.11: Replication of Input Membership Functions

However, the NB input membership function is displayed as a half-triangle, In MATLAB, the geometric calculations are based on what is displayed in the operating window; therefore, the NB input would have less geometric "weight" in the centroid calculation. In other words, the system would try to transition from the NM to the NB input membership function, but NM has more influence than NB. This would be the case for all inputs with a magnitude greater than 2/3 pi radians. One way to fix this issue is to use trapezoidal membership functions at the end to lump all values outside a certain range to the most extreme responses. This was the response for a system at 0.5 radians, an unstable point. To test the capability of the PID-Fuzzy Hybrid to stabilize the system, the target was set to 0 radians. To simulate initial conditions for a transfer function, the target angle was set to 0.5 for 0.01 seconds. This was enough to put the pendulum in an unstable position to test the controller.

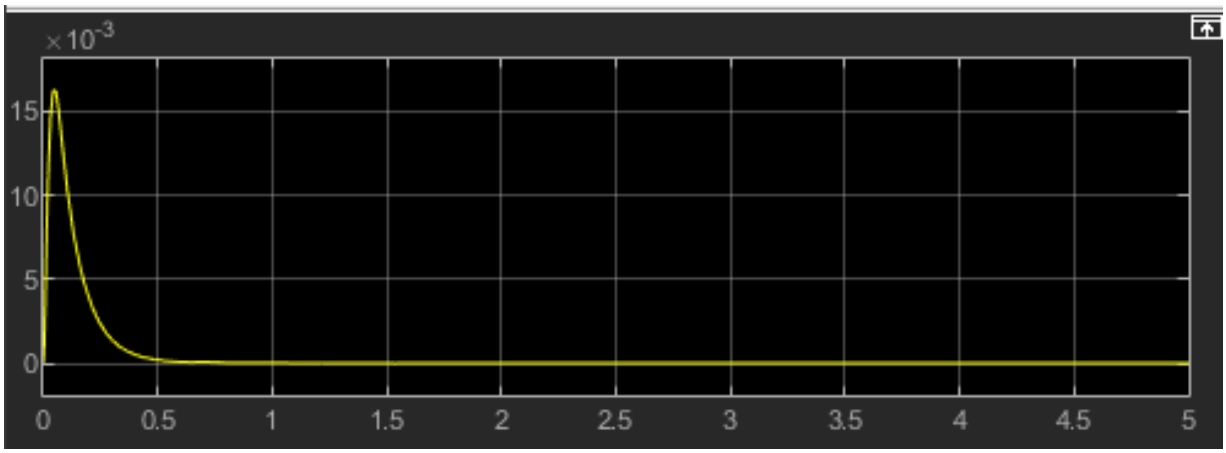

Figure 4.12: Angular Stabilization of the PID-Fuzzy controller, Angle (rad) vs Time (s)

From the above figure, we see that the system reaches a maximum displacement of approximately 0.015 radians before returning to 0. This shows that the hybrid controller can keep the pendulum stable at the vertical up position. To test the viability of the fuzzy logic controller on its own, the proportional and derivative multipliers were set to 1, meaning the controller was only taking the direct measurement of those variables into account.

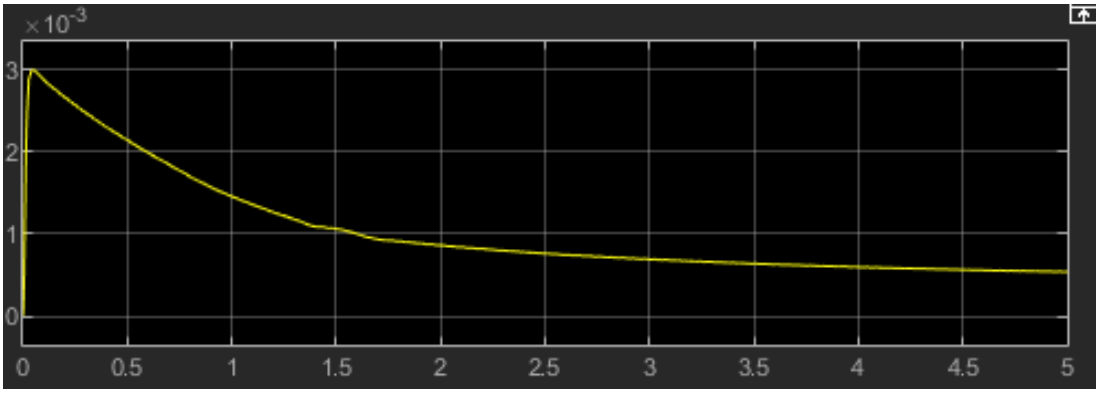

Figure 4.13: Angular Stabilization of Fuzzy Controller, Angle (rad) vs Time (s)

While there is less overshoot in this case, it appears to be taking longer to reach the 0 radian position. Extending this graph reveals that it does not reach 0 and begins to increase again to infinity. It would appear, for this setup, such fuzzy logic controller was not enough to stabilize

the system. This is further shown by the state space representation where the same problems arise if the fuzzy logic controller is left to stabilize a displacement of 0.2 radians on its own:

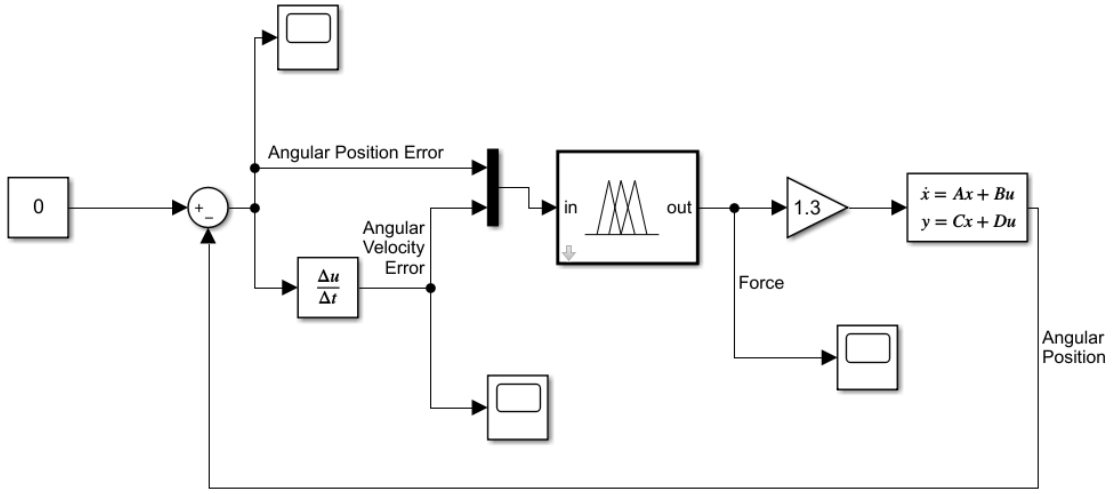

Figure 4.14: State-Space Fuzzy Logic Test

To test the ability of the fuzzy logic controller on its own, a state space model was used and connected to the fuzzy logic controller without a PID. The initial condition was set to be a 0.2 rad displacement.

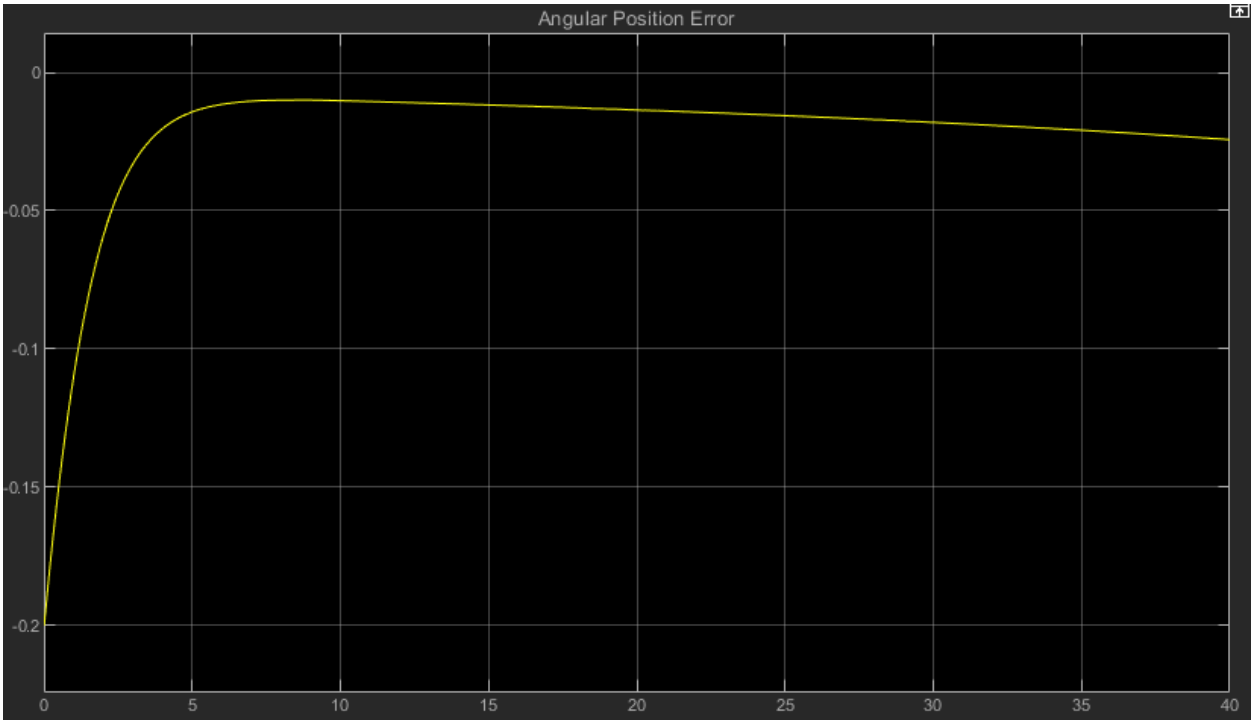

Figure 4.15: State-Space Results for 0.2 rad displacement, Error (rad) vs. Time (s)

This shows that the system follows the same turnaround trend as the previous example, and it will continue to do so until it is outside the operating window. The reason this happens is because this fuzzy logic controller was tuned to function in tandem with a PID. The PID would apply a gain to the angular position and angular velocity, which would result in a stronger response from the controller. Without this, the controller is unable to bring the system to a stable position and it diverges from it. If a gain is applied to the fuzzy logic controller to apply more force to the system, the problem resolves:

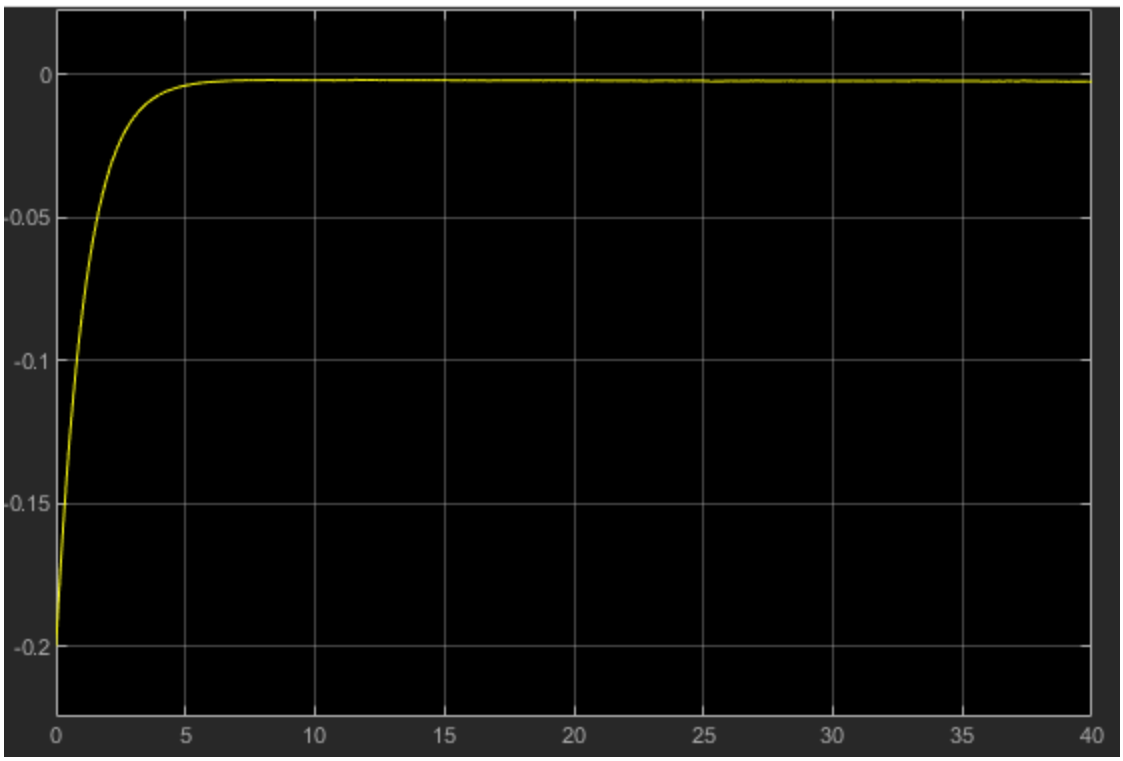

Figure 4.16: State-Space Results for 0.2 rad Displacement with Gain of 4, Error (rad) vs. Time (s)

The gain after the fuzzy logic in Figure 4.14 was adjusted from 1.3 to 4, and as a result, the system behaved in a more stable way as shown in Figure 4.16. There is still a slight offset from 0 radians, indicating there needs to be more force in areas near the stable point. In other words, the angular position is not prioritized as highly without the PID, as a result, the force applied is insufficient to bring the system to 0. This could be remedied by fine-tuning the membership functions or rules to generate a stronger response in the area near 0 radians.

In another paper, it was possible to hybridize both an LQR and FLC to control a two-wheeled inverted pendulum (Wu & Zhang, 2011). For this, it is possible to replicate the results of Jufeng Wu et al. (2011) to test if there is a bridge that can be built between the two methods. Unlike the previous example, the two wheeled system used a hybridization of LQR and fuzzy

logic. This pattern of hybridization is an indication of the role fuzzy logic can play in modern control systems. Using the control law values provided in the literature, it was possible to replicate the model in MATLAB using the same values for the cart's parameters. Unfortunately, without knowing the exact weights of their matrix Q, rebuilding the k matrix from scratch was not likely to yield the same results. Below are the results provided in the literature for the states of a 2 wheeled system with an initial velocity of 0.1 m/s.

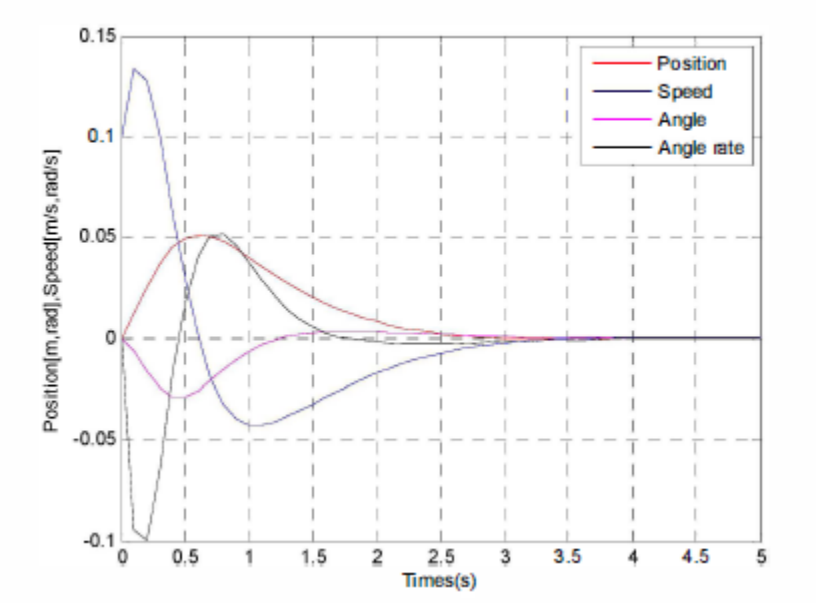

Figure 4.17: LQR Controller of Jufeng Wu

These results can be compared with the results gained from using the same k matrix:

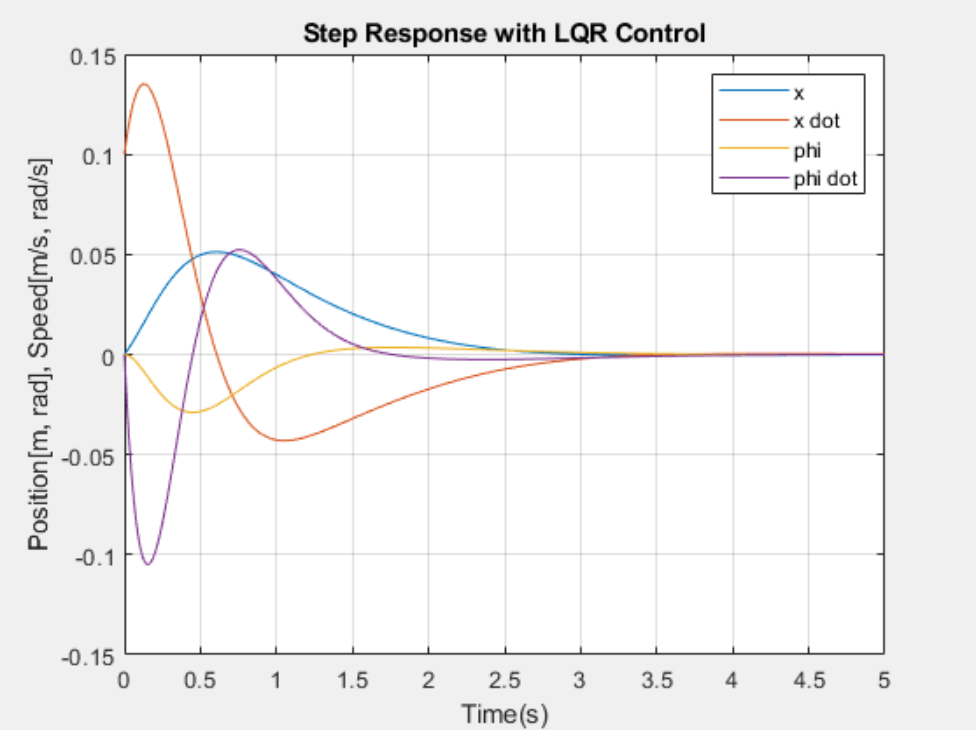

Figure 4.18: Replication of LQR Controller

As can be seen, the results are near identical. Note how in both graphs, the response of all variables is stabilized at 0. Using this, it is possible to control multiple variables like distance and angle at the same time. A controller could set the initial condition of position to be a variable *x* distance from the destination and the system would attempt to go back to 0. This showcases the power of LQR; however, this setup requires the system to be fully observable. Reducing the observability of the system is one of the primary challenges in which fuzzy logic may be a solution. While a reduced order system allows for the system to have less sensors, it requires more calculations to be done on the software side of the controller. This has the possibility of reducing the speed of the controller. As such, building an observable system may have its own costs that are not easily remedied in a microcontroller implementation. This makes it important to test the speed of a new system by checking its response rate in the serial monitor. While LQR

is a fast method (Messner & Tilbury, 2019), whether it retains that speed when hybridized requires testing.

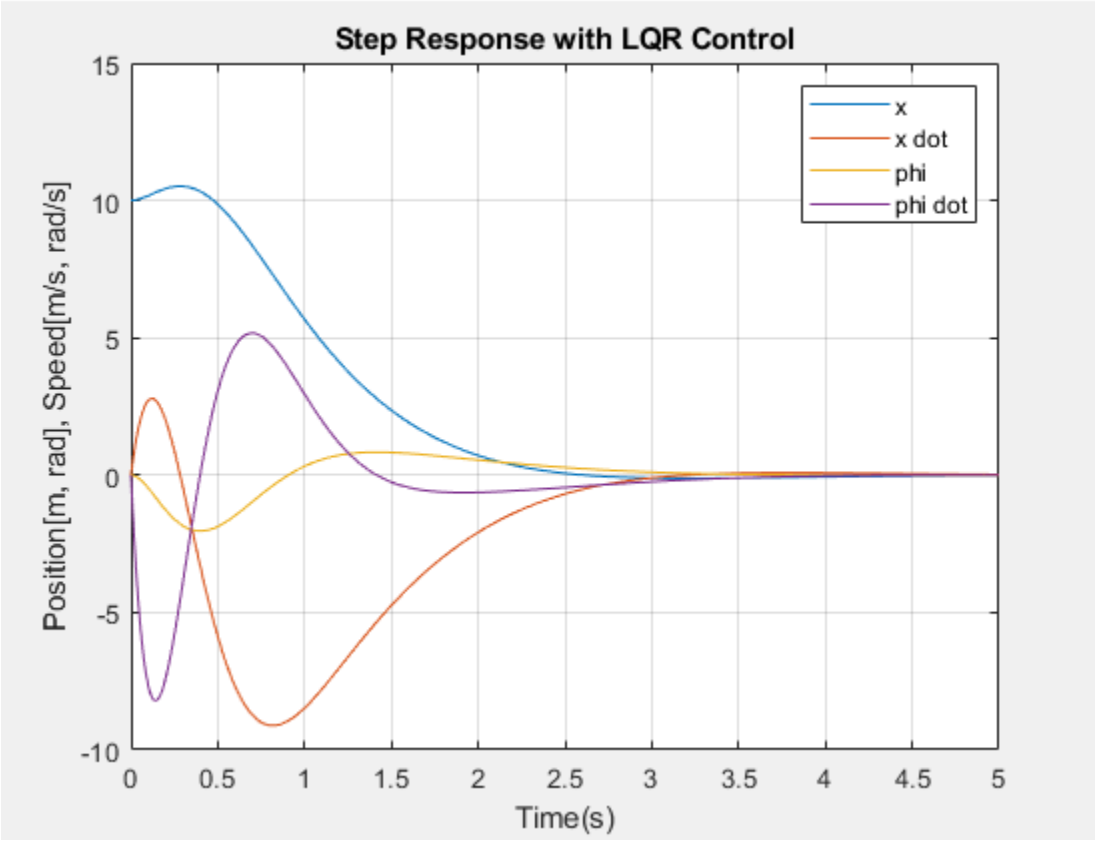

Figure 4.19: LQR Controller with Set Distance 10 m

As can be seen in the above figure, when the initial distance is set to 10 m from the offset, the LQR attempts to return the position to 0. This means that some weight must have been put on the position state during the controller design. This large number is meant to test how this set of control law values can get the cart to travel a large distance while still stabilizing the pendulum. While the graph shows all variables stabilizing, it should be noted the velocity and the angular velocity both reach elevated levels of over 5 m/s and 5 rad/s, respectively. This would cause the system to rely heavily on motor selection: a strong, reactive motor would be required. Since part of the system's objective is to be low cost for implementation on a microcontroller,

this is unacceptable. To remedy this, it could be possible to use coding to travel this distance incrementally, using the LQR to stabilize and using either fuzzy logic or a standard coding loop to keep track of the remaining distance. However, rather than using another's existing control law, it would be more efficient to construct one weighted accordingly to the requirements of the designer.

As previously stated, the viability of these methods hinges on the ability for them to be run on microcontrollers. Normally, a state space representation is much more easily handled in a program like MATLAB, but this would be impractical for a system that is not always going to be at the side of a PC. To transfer the model over to a microcontroller would normally require discretizing the transfer function; however, using Arduino libraries, it is now possible to implement a state space model in matrix form directly into the microcontroller. Furthermore, this state space model allows for the implementation of LQR, with estimator (observer), and integral control. With the system able to handle these calculations, it was important to check the speed of the system. An example system of the one constructed by the University of Michigan was included on the library's page. Acquiring this code, and uploading, it was possible to see how fast the program was on actual hardware. Using the serial monitor, the program that included a state space inverted pendulum with LQR, observers, and integral control was able to run at a steady 84 Hz. This speed is good considering all the calculations that are being performed.

It may be possible to resolve the position problem using fuzzy logic or coding to increase distance incrementally. As stated before, an LQR controller allows for the control of multiple variables; therefore, controlling of distance should be possible. However, it is possible that the distance is too great for the system to match the expectations of the controller.

To test if it was theoretically possible to use both an FLC and LQR on the same microcontroller, the model in the Arduino was combined with the fuzzy logic code. This was to test if the codes could run simultaneously before implementing a model onto the system. By including a full fuzzy logic ruleset into the state-space model code and having them feed into each other, it was possible to test the viability of them on the hardware. The result on the serial monitor was a rate between 81-84 Hz. This is also acceptable and should be more than capable of handling a real model if implemented.

To implement and validate the above fuzzy logic concepts, it was important to have a physical model to test and implement them. Due to limited availability of facilities and equipment, a motor arm system was procured by the University of Texas Rio Grande Valley for use. This motor arm system consisted of a motor attached to an arm atop bearings with a potentiometer at the end to record the arm's position. This is illustrated in the figure below:

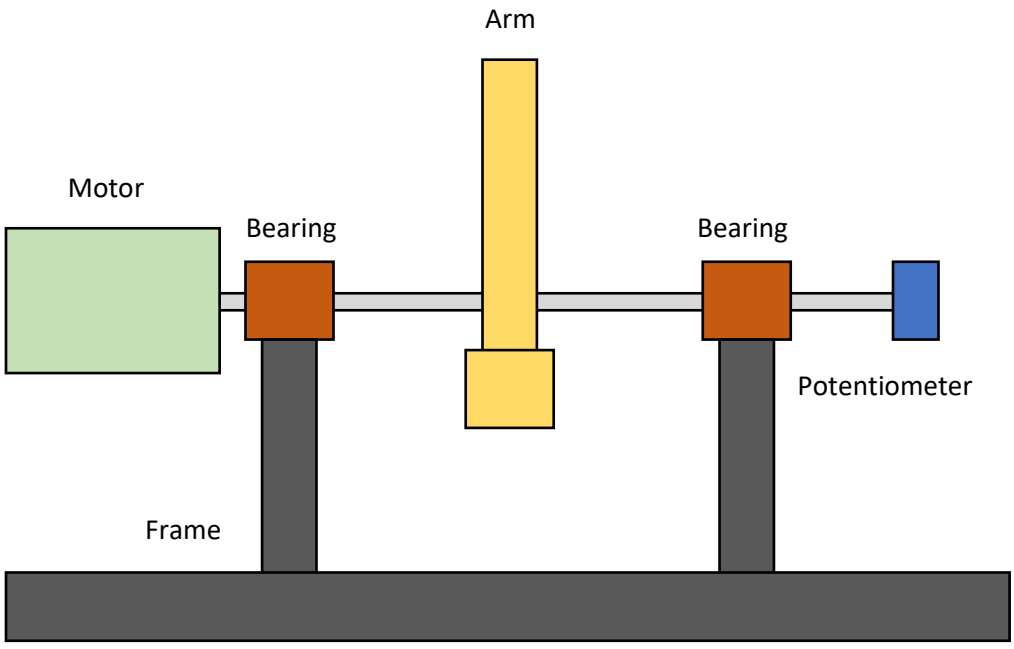

Figure 4.20: Motor Arm Photo

The transfer function for the arm's movement in relation to the input voltage had been previously documented. This allowed for the rapid prototyping of controller systems in Simulink. Below is the block diagram of the system. (Vasquez, Kypuros, & Villanueva, 2010)

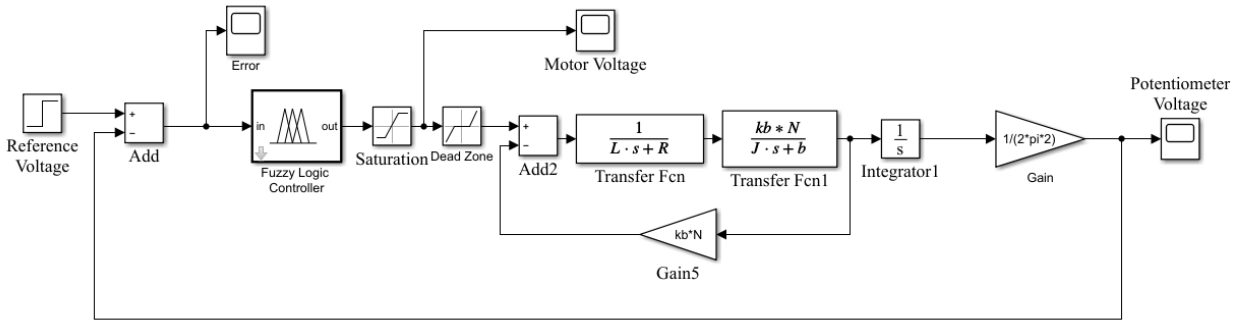

Figure 4.21: Motor Arm Block Diagram with Fuzzy Logic Controller

To account for the stick-slip friction of the bearings, a dead zone of +/- 1.8 V was added. This value came from previous testing of the system as documented in literature. For the Fuzzy Logic Controller to work in Simulink, it must be based on a constructed controller with inputs, outputs, and rules set up. This controller was created in MATLAB. The membership functions for the inputs and outputs, as well as rule table are below. The following image is the Mamdani FLC structure for the controller. This system only requires one input and one output, so the structure is as follows:

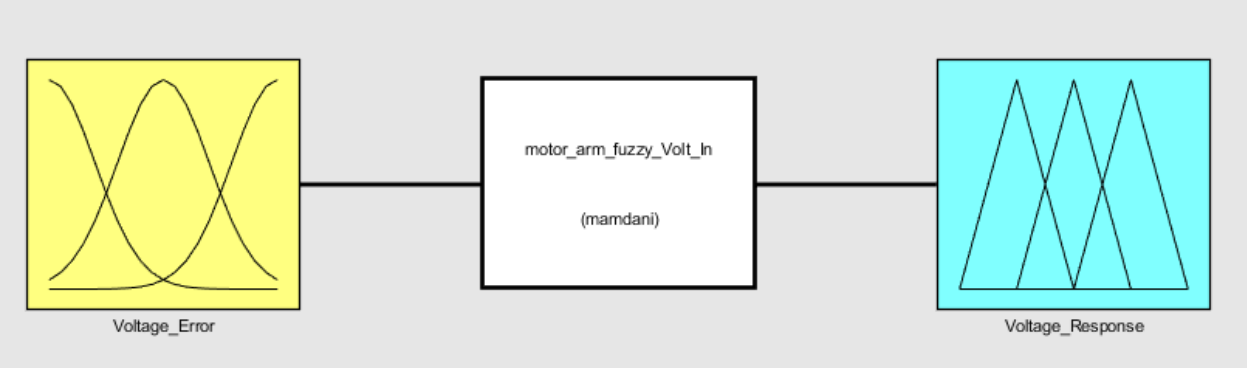

Figure 4.22: Input and Output of Fuzzy Logic Controller

The fuzzy logic controller runs using a single input, “Voltage Error” which is how much the voltage of the potentiometer differs from the reference voltage. In this case, each 0.5V corresponds to one revolution of the arm. Using this, it is possible to correspond voltage to position. The single output is the voltage delivered to the motor.

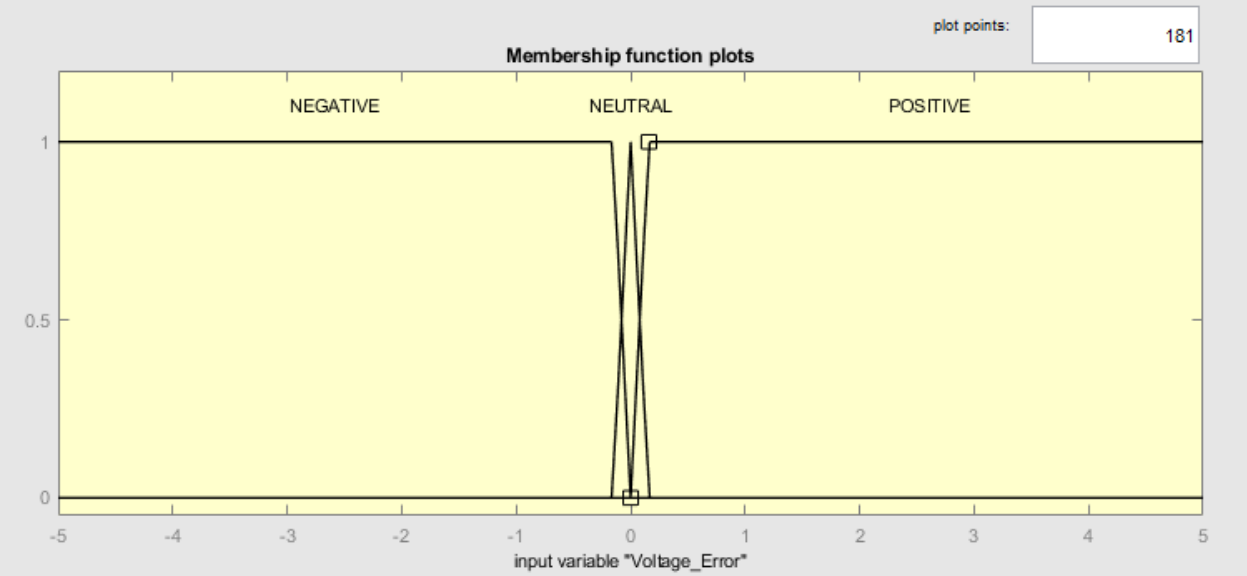

Figure 4.23: Input Membership Function of Motor Arm

The input has three membership functions: NEGATIVE, NEUTRAL, and POSITIVE. The NEUTRAL membership function is triangular with a peak at 0 and the base being the voltage

equivalent of 2 radians in either direction: (-0.1592, 0, 0.1592). The POSITIVE membership function is a trapezoidal membership function with anything above 2 radians of error being considered wholly part of the POSITIVE function. This value of 2 radians was obtained through running experiments in Simulink and was chosen because of its comparable performance to a PID controller. Details of these experiments will be discussed later in the chapter. This continues up to 5V of error, the maximum value that can be interpreted by an Arduino. Likewise, the NEGATIVE membership function is the mirror image of the positive with anything smaller than -2 radians of error being wholly part of the NEGATIVE function.

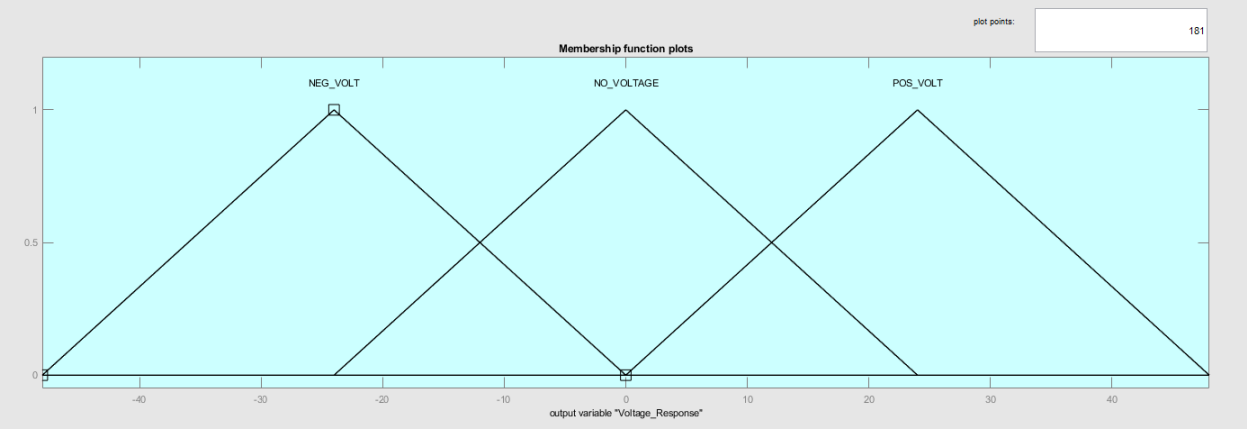

Figure 4.24: Output Membership Function of Motor Arm

The output membership function consists of 3 triangular membership functions: NEG VOLT, NO VOLTAGE, and POS VOLT. These membership functions have peaks at -24, 0, and 24 volts, respectively. This voltage is based off the power supply in the original model of the motor arm system. As will be seen later in this chapter, the voltage value of each function will not matter when transferred to a microcontroller.

INPUT MEMBERSHIP FUNCTION	OUTPUT MEMBERSHIP FUNCTION
NEGATIVE	NEG VOLT
NEUTRAL	NO VOLTAGE
POSITIVE	POS VOLT

Table 4.2: Fuzzy Logic Rules for Motor Arm

The above table shows the output for each corresponding input. Since there is only one input, there are only as many output responses as there are membership functions. With the rules set up, it is possible to graph the response versus the input error.

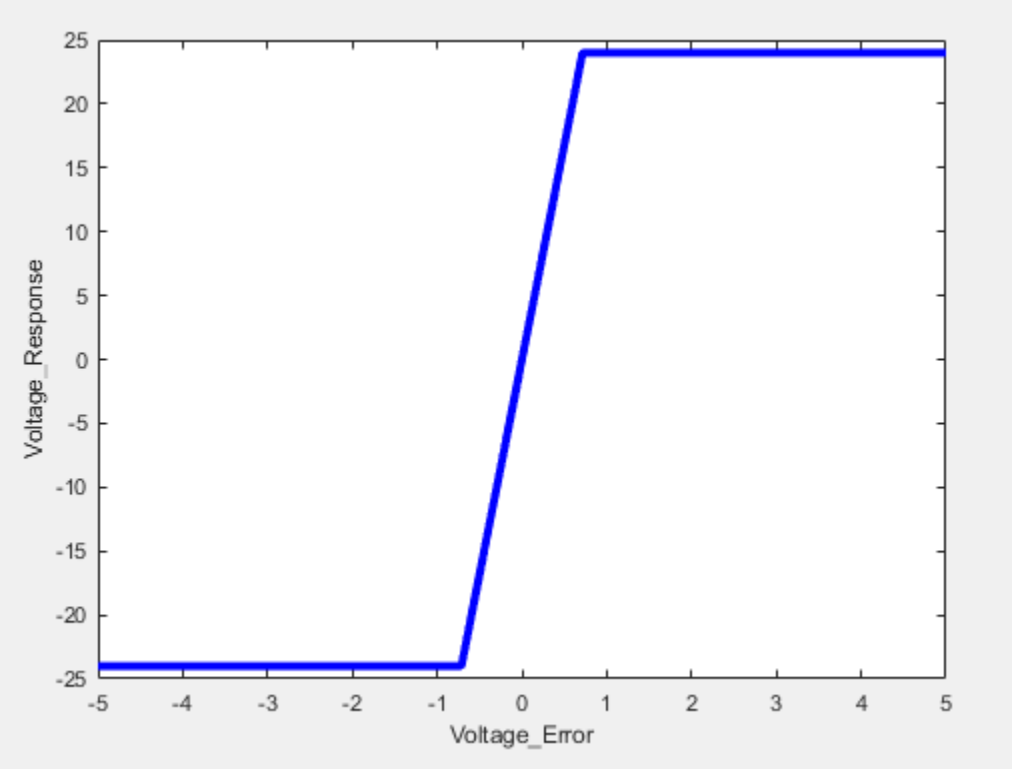

Figure 4.25: Voltage Response versus Voltage Error

The above graph represents all possible responses of the output by the fuzzy logic controller. With the controller constructed, it was now possible to conduct simulations.

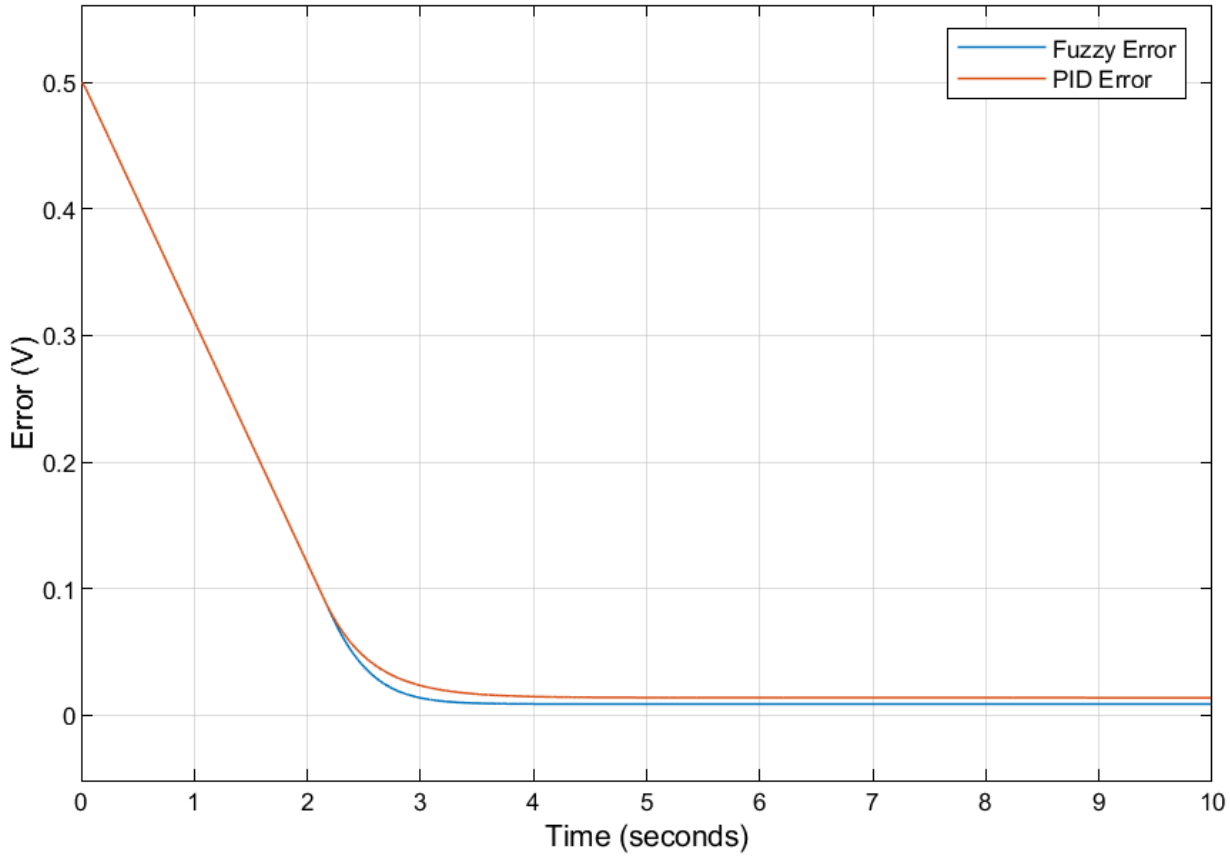

Figure 4.26: Error versus Time Simulink Graph (PID/FLC)

Above is the graph of the error between the desired position of 1 revolution and the actual position versus time. The error of the system quickly decreases to near zero with the fuzzy logic controller. The settling time was approximately 3.36 seconds. The steady state value was 8.5mV. In comparison, a PI controller with a kp of 10 and ki of 0.01 had a settling time of approximately 3.6 seconds. The steady state value for the error was 13.6mV.

Note, in the above graph, the PI controller used originally took radian error as an input. Since the new block diagram for the FLC uses voltage error, it is necessary to convert volts back

to radians for the PID controller. As a result, in the block diagram, the fuzzy logic controller is replaced with the following:

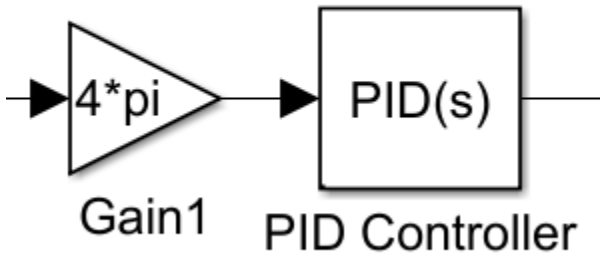

Figure 4.27: PI Controller in Block Diagram

The gain of 4pi is derived from the following:

$$0.5\left(\frac{V}{rev}\right) * \frac{1}{2\pi}\left(\frac{rev}{rad}\right) = \frac{1}{4\pi}\left(\frac{V}{rad}\right) \quad (4.1)$$

Converting back to volts is as follows:

$$x(V) * 4\pi\left(\frac{rad}{V}\right) = 4\pi x\ (rad) \quad (4.2)$$

These results from the FLC are possible because of how close the values composing the NEUTRAL membership function are to each other (-0.1592, 0, 0.1592). When membership functions are narrow, like this one, it allows for a stronger response near specified operating points. If this window is widened, the response may be more gradual.

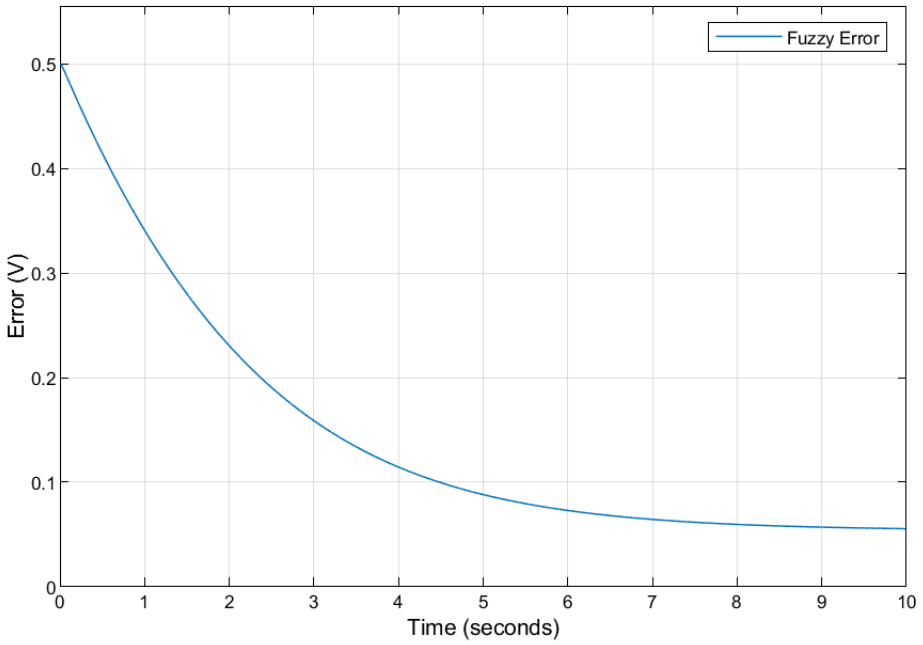

Figure 4.28: Error Graph with Wide NEUTRAL Function, Error (V) vs Time (s)

To show this, the NEUTRAL membership function was adjusted to be [-1 0 1] with the POSITIVE and NEGATIVE membership functions also adjusted accordingly. As can be seen, the response is slower and appears to be settling further away from the desired value. This shows how narrower membership functions can give more precise results. With a working model in Simulink, it was time to transfer this to an Arduino.

To transfer the above fuzzy logic onto an Arduino, it was convenient to change the membership functions of the Fuzzy Logic Controller into a range of values that can be interpreted by an Arduino. An Arduino reads voltage between 0-5V and interprets it as an integer between 0-1023. To find the equivalent membership functions, it is possible to change the "Range" value in the MATLAB fuzzy logic toolbox and the values will automatically scale up or down.

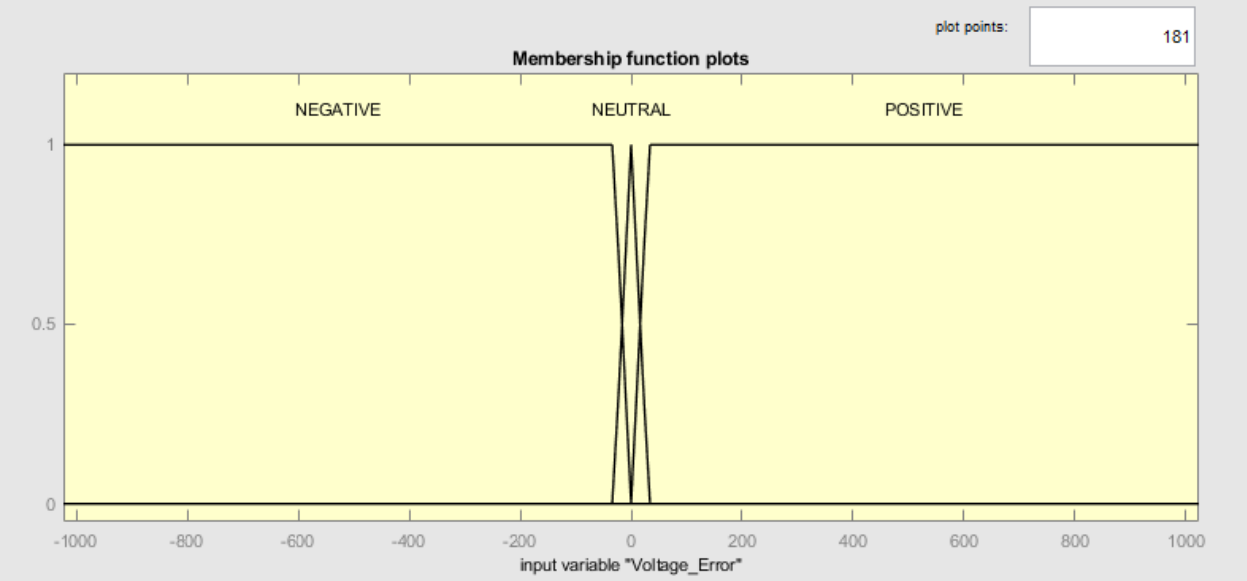

Figure 4.29: Motor Arm Scaled Input Membership Functions

In the above graph, the range was changed from [-5 5] to [-1023 1023]. This caused the NEUTRAL membership function to change to [-32.57 0 32.57]. However, because the read from the Arduino is an integer value, the error must also be an integer. To accommodate for this, the NEUTRAL membership function was changed to [-33 0 33] when transferred to the Arduino. The POSITIVE and NEGATIVE membership functions were adjusted in the same way accordingly. Like the Ball and Beam FLC, the output membership functions were adjusted to have the peaks at -255, 0, and 255, respectively.

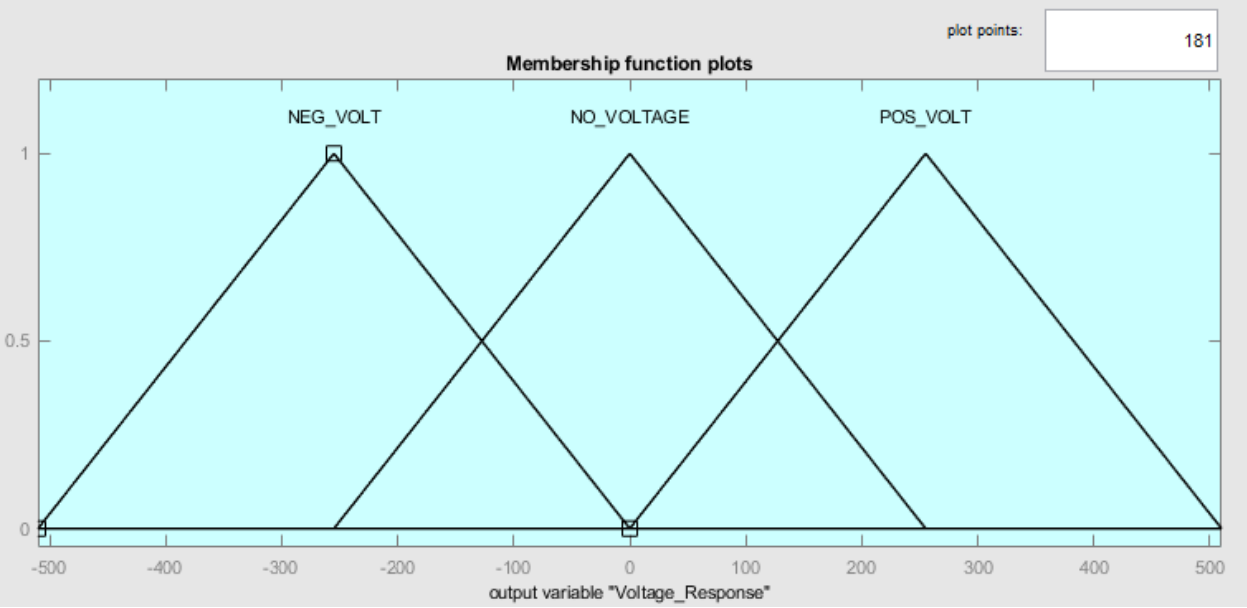

Figure 4.30: Motor Arm Output Membership Functions for Arduino

With the membership functions converted, it was now possible to compile the code onto an Arduino. Using the embedded Fuzzy Logic Library (eFLL), it was possible to directly implement the membership functions. (Lira et al. 2012) It should be noted, however, that eFLL only allows for trapezoidal membership functions, meaning that custom and sinusoidal membership functions cannot be used in it. This does not pose a problem for triangular membership functions, for a triangle can be expressed as a trapezoid with a base of width 0. For example, the triangular membership function NO VOLTAGE can be expressed as (-255, 0, 0, 255) in the Arduino coding language. To transfer the FLC onto an Arduino, all the input and output membership functions, as well as the rulesets must be transferred to an Arduino. To do this using eFLL, each of the membership function parameters must be first converted to trapezoidal format. Notation for MATLAB and Arduino functions will be different due to formatting reasons. The input will be referred to as “radianerror” and the output will be referred

to as “response.” The tables below will show the new notation for individual membership functions as well as their updated parameters.

MATLAB Name	Arduino Name	Arduino Parameters
NEGATIVE	neg	(-1023, -1023, -33 0)
NEUTRAL	neutral	(-33, 0, 0, 33)
POSITIVE	pos	(0, 33, 1023, 1023)

Table 4.3 Input Membership Functions, MATLAB to Arduino

The input membership functions can be seen in Table 4.3. It should be noted that the value 1023 and -1023 are repeated twice. This is because eFLL interprets this to include all values beyond the range of [-1023, 1023] to belong to either the positive or negative neutral membership function. Since an Arduino will never read a value beyond 1023, however, this will not come into play.

MATLAB Name	Arduino Name	Arduino Parameters
NEG_VOLT	negvolt	(-510, -255, -255, 0)
NO_VOLTAGE	novolt	(-255, 0, 0, 255)
POS_VOLT	posvolt	(0, 255, 255, 510)

Table 4.4 Output Membership Functions, MATLAB to Arduino

Similarly, the output membership functions are represented as triangular, with -255, 0, and 255 being the peaks. With these values, the Arduino will never attempt to output a value greater in magnitude than 255. With these membership functions in place, the fuzzy logic rules had to be converted over as well. The eFLL has a format for inputting rules, the exact structure can be seen in Appendix Figure A.1. In general, the rules in the Arduino follow an IF x THEN y structure.

For example, the rule for the negative response is "IF radianerror = neg THEN response = negvolt" with each rule from Table 4.2 also following this format. For more details on how this structure is formatted, one can refer to the tutorials written by the library's creator. Once all the membership functions and rules are converted to the Arduino's language, it is time to build the code to control the motor that moves the arm.

Since the FLC does most of the calculations, the remaining code is simple and mostly deals with the movement of the arm. In general, the code loops the following steps. Step 1 is to read the input voltage from the potentiometer. Step 2 is to use the voltage to calculate the error. Step 3 is to input the error into the built FLC. Step 4 is to determine whether the output from the FLC is positive or negative for changing the direction of the H-Bridge. Step 5 is to send the output to the H-Bridge to turn the motor. Step 6 is to print the response for monitoring purposes.

Based on the above steps, a value of -255 results in the H-Bridge turning the opposite direction with a PWM output of 255. The way this works is by sending a digitally written LOW voltage to one part of the H-Bridge and sending the PWM output from the FLC to the other part of the H-bridge. Depending on which side receives PWM output, the motor will turn one way or the other. The full code for how this is set up can be found in the Appendix Figure A.1 of this thesis.

As for how the input error is determined, the potentiometer is a 10k Ω resistor with 5 volts across it. This means that as the motor turns, the wiper voltage will be anywhere between 0 and 5 volts (0 and 1023 as a 10-bit value) with one full turn of the potentiometer equating to 0.5 V (102 as a 10-bit value). This value is referred to as "realposition" in the Arduino's code. The value for the desired position could either be an input from a second potentiometer, or a set value. To get consistent results, the starting position was written in the code itself to be at the

midpoint 10-bit value of 512. This desired value would then be adjusted by one revolution to 614 is referred to as “desiredposition” in the code. The error is then calculated by the simple line of code: “error = desiredposition - realposition;” which is then sent to the FLC in step 3. Through this process, the controller can be tested to see how close it gets and how long it takes to reach 0 error.

To ensure the arm did not turn too much and break the sensitive potentiometers, voltages were recorded in relation to the arm’s position to ensure the microcontroller would output the appropriate response. The initial reference voltage was set using another potentiometer so that the arm would attempt to match a manual input by turning the knob. However, this can also be altered to allow for other means of setting reference, such as a digital remote control. When tested physically, the arm would turn until the error read produced a response too weak to turn the motor. At this point, the steady state error would have a magnitude of 2 (in the 1023 scale, or 9mV or 0.12 rad) or less. This meant the controller translated well to the system with most of the error due to the stick-slip friction on the arm. To test the results of the FLC in comparison with a PID on Arduino hardware, the code was modified to include the same PID controller that was tested in Simulink. To include this PID in an Arduino, a PID library called PID_v1 was used. This library allows for the implementation of a PID and reduces code size by eliminating the need to manually code the calculations for the proportional, integral, and derivative functions. To use the same PID as in Simulink, the analog input had to be converted into a form that was mathematically equivalent to the radian error. The values for the input were converted from the analog input to radians by first converting the input to volts and then to radians using the same process used in equations 4.1 and 4.2. The input to the controller was in radians and an output in Volts. Unlike the FLC membership functions that were specifically written to have outputs of -

255 to 255, a mathematical equivalent for the PID's output in terms of a duty cycle would have to be calculated in the Arduino itself. The way this was accomplished was by setting a saturation voltage value to be equal to the power supply voltage. If the voltage output of the PID exceeded the saturation voltage in either direction, the output of the controller would be +/- 255, depending if the error was positive or negative, respectively. All remaining values of voltage would be mapped between -255 and 255 using the Arduino's built in map() function. The full details of the code can be found in the Appendix Figure A.2. With the control system uploaded to the Arduino board, it was now possible to test it. Once again, the steady state values of the error had a magnitude of 3 (in the 1023 scale, or 15.0mV or 0.184 rad), or less. In terms of steady-state results, the FLC got the error closer to 0 more consistently. To test for more significant differences, the speed of the controllers had to be considered. This was accomplished by setting the initial position to be at the halfway point and then having the controller set the desired position to be one revolution forward. The time it would take would then be recorded by the Arduino. A series of tests were conducted, tracking the settling time of the motor arm. The settling time was recorded using the milis() function in the Arduino, recording when the error reached 0 (Since there was no overshoot, recording time when the error reached 0 was an accurate representation). On average, the Fuzzy Logic Controller had a settling time of 3.266 seconds. The PID controller settled at an average of 3.200 seconds.

Both controller types while on an Arduino were still a considerable distance away from the target destination, sometimes almost by 10 degrees. The reason why this happened is the dead zone that was discussed earlier does not allow small PWM responses to turn the motor. Take for the example if the voltage error returned was the 10-bit value of 3 to the Arduino. The fuzzy logic calculation would be as follows:

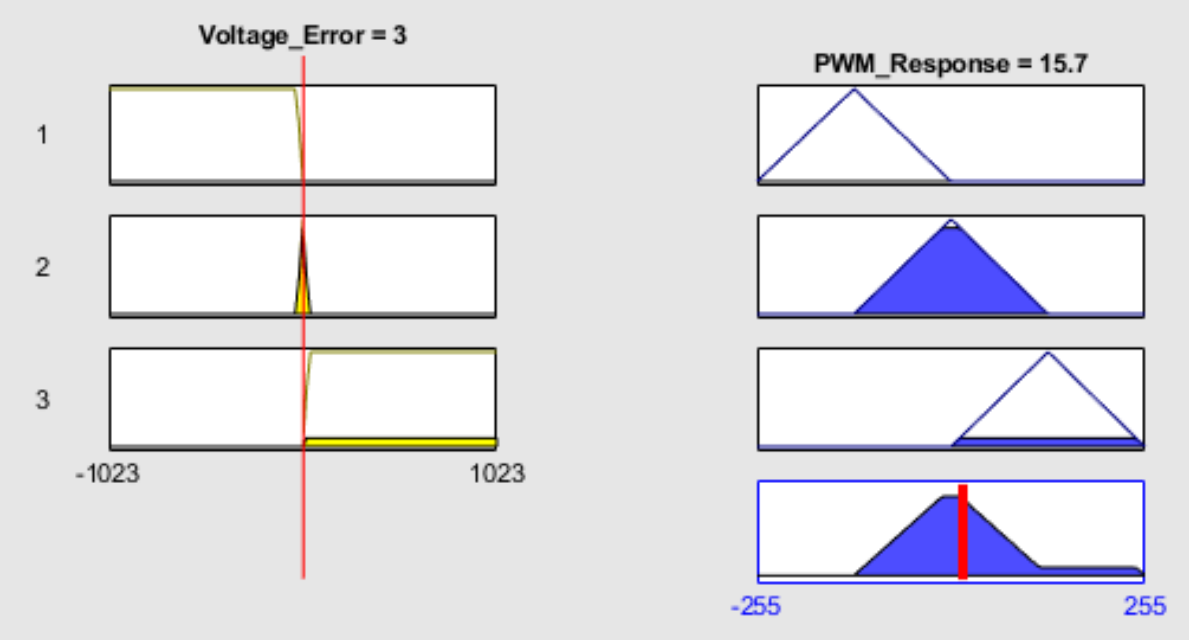

Figure 4.31: Fuzzy Logic 10-Bit Error

This PWM response of 15.7 results in an output of about 0.75 V. This is not enough to turn the motor and it will stall before it reaches 0 error. To remedy this, there are a few options that could be taken. One would be to rewrite the membership functions of the fuzzy logic code to accommodate for the dead zone and restructure the rulesets appropriately. The other is to use the coding aspect of the microcontroller to increase output near the target range. This was done by multiplying the output by 4 whenever the error voltage was within 15 mV. This segment of code is found in within Figure A.1 and is designated as the variable “highoutput.” The result of this correction was a steady-state error of 0 mV.

CHAPTER V

DISCUSSION AND ANALYSIS OF RESULTS

Through the testing of multiple control systems, it is clear that many have their own strengths and weaknesses. In the replication of the hybrid PID-FLC controller provided by Yanmei Liu et al. (2009), it was reaffirmed that a fuzzy logic controller requires an extent of fine tuning for it to control a system on its own. With the adjustment of gains, the output can be made to be as responsive as a PID controller. This was further validated in the creation of an FLC to control a motor arm. Simulations demonstrated that the tightening of the membership functions around the stable point allowed for a faster, and more accurate response. This demonstrates that through the adjustment of membership functions, either by making them more numerous or narrow, it is possible to achieve a result closer to the desired value. Fuzzy logic has been described as able to substitute a human operator, and, likewise, fine-tuning the controller requires a very "human" approach in identifying what areas should require more precision and what can be lumped together as a trapezoidal function. (King, Magoulas, & Stathaki, 1994) The simulations have also shown the advantages of using a Fuzzy Logic Controller as compared to an LQR in some scenarios. An LQR can control multiple variables and optimize settling time, however, its cost function is based on the parameters of the system (transfer function, state-space representation) to achieve optimal results. In cases where the system properties are not wholly known, or when the system cannot be fully observed, a fuzzy logic controller can be

implemented without prior knowledge of the system model. This is demonstrated by the fact that the construction of the fuzzy logic membership functions and rule sets did not require the equations of motion of the system to be implemented in any way. This makes FLCs very useful in being able to control nonlinear systems as they do not require linearization or a transfer function to operate. However, it is still very useful to have those equations to run simulations and to safely fine-tune the controller, but the actual "logic" of the controller can function wholly without it. In terms of ease of implementation, it is like a PID in that one can easily change the constants of the controller to get a different response out of the system. Both also have toolboxes across many simulation or controller software packages, making them very easy to implement. This may be a reason that PID-fuzzy hybrid controllers are so prevalent in literature. Meanwhile, in LQR-fuzzy hybrid controllers, the FLC takes more of a backseat role, fine tuning the parameters to lower inaccuracies from linearization. From experimental results with a physical motor arm, it was clear that Fuzzy Logic could be easily implemented onto an Arduino Uno and produce results comparable to the simulations. The speed of the controller was verified both with a single input and with multiple input FLCs, meaning that a controller can be complex and remain viable for systems such as the inverted pendulum or ball and beam system that require it. When it comes to physical comparisons of performance, the Fuzzy Logic and PID controllers were both able to correct the error on a motor arm within a similar level of accuracy. In addition, the physical tests coincided with the simulations, with Simulink predicting a settling time of approximately 3.36 seconds for the FLC and 3.6 for the PID, while the real model tests resulted in values of 3.27 and 3.20 seconds, respectively. In addition to this, the steady state voltage values are similar, with 8.5mV in Simulink vs 9 mV in Arduino for the Fuzzy Logic compared to 13.6 for and 15mV for the PID, respectively. In terms of settling time, both controllers

performed similarly, with the PID being faster on an average of 0.066 seconds in physical tests. This contrasts with the Simulink simulations where the FLC was slightly faster. This result was mirrored in the Arduino tests with the steady state error being slightly greater for the PID controller. This shows that the physical model and simulation coincide for both controllers mutually validating each other.

CHAPTER VI

SUMMARY AND CONCLUSION

Modern control engineering has many avenues to automate systems. Among these methods is the use of fuzzy logic controllers. These controllers can function as substitutes for human operators and allow for many options of control depending on the membership functions. In comparison to more traditional methods such as the PID, a FLC can achieve similar results. Both require their own kind of fine-tuning to avoid problems such as overshoot or slow settling times, but the fuzzy logic controller can function on an easy to comprehend linguistic level. This makes it a very beginner-friendly tool for someone designing a controller without knowledge of models or the math used by a PID. This easy-to-interpret, linguistic approach is the controller's greatest strength. In addition, the controller does not require a model to be built to acquire a control law unlike other methods like LQR. This can make it a very powerful tool for nonlinear systems, as fuzzy logic rules can accommodate for multiple operating windows depending on how the rules are set up without losing accuracy due to linearization. In addition, a FLC can be easily implemented on to a low-cost microcontroller using either existing libraries on an Arduino, or manually coding one in the style of a Takagi-Sugeno FLC. As for the implementation on a microcontroller, the existing libraries allow for easy implementation of fuzzy logic or PID control strategies. The construction and safe simulation of the FLC can be done in MATLAB if needed, but all the membership functions can be transferred over to an

Arduino through the eFLL's coding structure. In addition, since a model does not need to be constructed for a fuzzy logic controller to work, nonlinear systems can feasibly be controlled with an Arduino at multiple operating points without the need to build complex sets of observers with a control law. In future works, it should be possible to implement a FLC onto a more complex system such as an inverted pendulum or another nonlinear system. Through this, using a higher dimensional ruleset for the fuzzy logic rules or attempts of hybridizing the FLC for use on a microcontroller can be explored to find effective control solutions.

REFERENCES

Afaq, N., Asghar, S., Abbasi, A. R., Wallam, F., & Saeed, Q. (2015). *Low-Cost & Control Design of an Inverted Pendulum using Conventional, Fuzzy and Hybrid Techniques.* Karachi: IEEE.

Akhtaruzzaman, M., & Shafie, A. A. (2010). Modeling and control of a rotary inverted pendulum using various methods, comparative assessment and result analysis". *2010 IEEE International Conference on Mechatronics and Automation*, 1342-1347.

Akmal, M. A., Jamin, N. F., & Abdul Ghani, N. M. (2017). Fuzzy Logic Controller fo Two Wheeled EV3 LEGO Robot. *2017 IEEE Conference on Systems, Process and Control* (pp. 134-139). Melaka: IEEE.

Alves, A., Lira, R., Lemos, M., Kridi, D. S., & Leal, K. (2012, September 28). *eFLL - A Fuzzy Library for Arduino and Embedded Systems* . Retrieved from Blog ZeRoKoL: https://blog.zerokol.com/2012/09/arduinofuzzy-fuzzy-library-for-arduino.html

Babu, S. S., & Pillai, A. S. (2016). Design and Implementation of Two-Wheeled Self-Balancing Vehicle Using Accelerometer and Fuzzy Logic. *Second International Conference on Computer and Communication Technologies* (pp. 45-53). New Delhi: Springer India.

Bakaráč, P., Klaučo, M., & Fikar, M. (2018). "Comparison of inverted pendulum stabilization with PID, LQ, and MPC control". *2018 Cybernetics & Informatics (K&I)*, 1-6.

Brunton, S., & Kuts, J. (2019). *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control.* Cambridge: Cambridge University Press.

Fadali, M. S., & Visioli, A. (2013). *Digital Control Engineering (Second Edition).* Cambridge: Academic Press.

Ibrahim, D. (2006). *Microcontroller Based Applied Digital Control.* Cyprus: John Wiley & Sons.

King, R. E., Magoulas, G. D., & Stathaki, A. A. (1994). Multivariable Fuzzy Controller Design. *Control Eng. Practice, Vol. 2, No. 3*, 431-437.

Li, J., Ying, J., & Xue, L. (2009). Macromodeling of the electrostatically actuated circular plate based on mode superposition method. *2009 9th Internation Conference on Electronic Measurement & Instruments* (pp. 4-59-4-64). Beijing: IEEE.

Liu, Y. (2009). Real-Time Controlling of Inverted Pendulum by Fuzzy Logic. *IEEE International Conference on Automation and Logistics* (pp. 1180-1183). Shenyang: IEEE.

Messner, B., & Tilbury, D. (2019). *Inverted Pendulum: System Modeling*. Retrieved from University of Michigan: http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=SystemModeling

Nour, M. I., Ooi, J., & Chan, K. Y. (2007). Fuzzy Logic control vs. conventional PID control of an inverted pendulum robot. *2007 International Conference on Intelligent and Advanced Systems* (pp. 209-214). Kuala Lumpur: IEEE.

Ogata, K. (2010). *Modern Control Engineering Fifth Edition.* New Jersey: Prentice Hall.

Peng, X., & Wei, W. (2010). Analysis and Research of Inverted Pendulum Time-Varying Systems Based on Fuzzy Controlling Means. *2010 International Forum on Information Technology and Applications* (pp. 439-442). Kunming: IEEE.

Suphatsatienkul, V., Banjerdpongchai, D., & Wongsaisuwan, M. (2017). Design of reduced-order observer and linear quadratic regulator for inverted pendulum on cart. *2017 56th Annual Conference of the Society of Intrument and Control Engineers of Japan (SICE)* (pp. 1039-1044). Kanazawa: IEEE.

Ulusoy, M. (2017). *Understanding Model Predictive Control, Part 2: What Is MPC?* Retrieved from Mathworks: https://www.mathworks.com/videos/understanding-model-predictive-control-part-2-what-is-mpc--1528106359076.html

Vasquez, H., Kypuros, J., & Villanueva, R. (2010). Implementaing and Validating Analog and Digital Controllers. *2010 GSW Conference* (pp. 1-12). Lake Charles: ASEEGSW.

Wu, J., & Zhang, W. (2011). Design of fuzzy logic controller for two-wheeled self-balancing robot. *6th International Forum on Strategic Technology* (pp. 1266-1270). Harbin: IEEE.

APPENDIX

APPENDIX

motor_arm_fuzzy

```
#include <Fuzzy.h>

// Instantiating a Fuzzy object
Fuzzy *fuzzy = new Fuzzy();
int A1PIN = 3;
int A2PIN = 6;

int positionPin = A3;
int targetPin = A9;

int realposition = 0;
int desiredposition = 0;
boolean count = 0;

int error = 0;

unsigned long Stopwatch;

void setup()
{
  //HBRIDGE SETUP
  pinMode(A1PIN, OUTPUT);
  pinMode(A2PIN, OUTPUT);

  // Set the Serial output
  Serial.begin(9600);

  // Instantiating a FuzzyInput object
  FuzzyInput *radianerror = new FuzzyInput(1);
  // Instantiating a FuzzySet object
  FuzzySet *neg = new FuzzySet(-1791, -1108, -33, 0);
  // Including the FuzzySet into FuzzyInput
  radianerror->addFuzzySet(neg);
  // Instantiating a FuzzySet object
  FuzzySet *neutral = new FuzzySet(-33, 0, 0, 33);
  // Including the FuzzySet into FuzzyInput
  radianerror->addFuzzySet(neutral);
  // Instantiating a FuzzySet object
  FuzzySet *pos = new FuzzySet(0, 33, 1103, 1785);
  // Including the FuzzySet into FuzzyInput
  radianerror->addFuzzySet(pos);
  // Including the FuzzyInput into Fuzzy
  fuzzy->addFuzzyInput(radianerror);

  // Instantiating a FuzzyOutput objects
  FuzzyOutput *response = new FuzzyOutput(1);
```

```
  // Instantiating a FuzzySet object
  FuzzySet *negvolt = new FuzzySet(-510, -255, -255, 0);
  // Including the FuzzySet into FuzzyOutput
  response->addFuzzySet(negvolt);
  // Instantiating a FuzzySet object
  FuzzySet *novolt = new FuzzySet(-255, 0, 0, 255);
  // Including the FuzzySet into FuzzyOutput
  response->addFuzzySet(novolt);
  // Instantiating a FuzzySet object
  FuzzySet *posvolt = new FuzzySet(0, 255, 255, 510);
  // Including the FuzzySet into FuzzyOutput
  response->addFuzzySet(posvolt);
  // Including the FuzzyOutput into Fuzzy
  fuzzy->addFuzzyOutput(response);

  // Building FuzzyRule "IF radianerror = neg THEN response = negvolt"
  // Instantiating a FuzzyRuleAntecedent objects
  FuzzyRuleAntecedent *ifRadianerrorNeg = new FuzzyRuleAntecedent();
  // Creating a FuzzyRuleAntecedent with just a single FuzzySet
  ifRadianerrorNeg->joinSingle(neg);
  // Instantiating a FuzzyRuleConsequent objects
  FuzzyRuleConsequent *thenResponseNegvolt = new FuzzyRuleConsequent();
  // Including a FuzzySet to this FuzzyRuleConsequent
  thenResponseNegvolt->addOutput(negvolt);
  // Instantiating a FuzzyRule objects
  FuzzyRule *fuzzyRule01 = new FuzzyRule(1, ifRadianerrorNeg, thenResponseNegvolt);
  // Including the FuzzyRule into Fuzzy
  fuzzy->addFuzzyRule(fuzzyRule01);

  // Building FuzzyRule "IF radianerror = neutral THEN response = novolt"
  // Instantiating a FuzzyRuleAntecedent objects
  FuzzyRuleAntecedent *ifRadianerrorNeutral = new FuzzyRuleAntecedent();
  // Creating a FuzzyRuleAntecedent with just a single FuzzySet
  ifRadianerrorNeutral->joinSingle(neutral);
  // Instantiating a FuzzyRuleConsequent objects
  FuzzyRuleConsequent *thenResponseNovolt = new FuzzyRuleConsequent();
  // Including a FuzzySet to this FuzzyRuleConsequent
  thenResponseNovolt->addOutput(novolt);
  // Instantiating a FuzzyRule objects
  FuzzyRule *fuzzyRule02 = new FuzzyRule(2, ifRadianerrorNeutral, thenResponseNovolt);
  // Including the FuzzyRule into Fuzzy
  fuzzy->addFuzzyRule(fuzzyRule02);

  // Building FuzzyRule "IF Radianerror = pos THEN response = posvolt"
  // Instantiating a FuzzyRuleAntecedent objects
```

```
  FuzzyRuleAntecedent *ifRadianerrorPos = new FuzzyRuleAntecedent();
  // Creating a FuzzyRuleAntecedent with just a single FuzzySet
  ifRadianerrorPos->joinSingle(pos);
  // Instantiating a FuzzyRuleConsequent objects
  FuzzyRuleConsequent *thenResponsePosvolt = new FuzzyRuleConsequent();
  // Including a FuzzySet to this FuzzyRuleConsequent
  thenResponsePosvolt->addOutput(posvolt);
  // Instantiating a FuzzyRule objects
  FuzzyRule *fuzzyRule03 = new FuzzyRule(3, ifRadianerrorPos, thenResponsePosvolt);
  // Including the FuzzyRule into Fuzzy
  fuzzy->addFuzzyRule(fuzzyRule03);
}
```

```
void loop()
{
  // Getting a random value
  int input = random(0, 1023) - random(0, 1023);
  //READING PINS
  realposition = analogRead(positionPin);
  //desiredposition = analogRead(targetPin);
  desiredposition = 512; //Difference per revolution 102:  Setpoints 512, 614
  error = desiredposition - realposition;

  // Printing something
  //  Serial.println("\n\n\nInput: ");
  //  Serial.print("\t\t\tError: ");
  //  Serial.println(error);
  // Set the random value as an input
  fuzzy->setInput(1, error);
  // Running the Fuzzification
  fuzzy->fuzzify();
  // Running the Defuzzification
  float output = fuzzy->defuzzify(1);
  highoutput = output * 4;
  //WRITING PINS
  if (output <= 0 &&  error < -3) {
    //IF ERROR < 0 NEGATIVE RESPONSE A1PIN LOW A2PIN OUTPUT
    digitalWrite(A1PIN, LOW);
    analogWrite(A2PIN, abs(output));
  }
  else if (output <= 0 && error >= -3) {
    digitalWrite(A1PIN, LOW);
    analogWrite(A2PIN, abs(highoutput));
  }
  else if (output > 0 && error > 3) {
    //H BRIDGE PINS (REVERSED)
    //IF ERROR > 0 POSITIVE RESPONSE A1PIN OUTPUT A2PIN LOW
    analogWrite(A1PIN, abs(output));
    digitalWrite(A2PIN, LOW);
  }
  else if (output > 0 && error <= 3) {
    analogWrite(A1PIN, abs(highoutput));
    digitalWrite(A2PIN, LOW);
  }



  // Printing something

//  Serial.println("Result: ");
//  Serial.print("\t\t\tResponse: ");
//  Serial.println(output);

if (error == 0 && count == 0){
Stopwatch = millis();
Serial.print("Time: ");
Serial.println(Stopwatch);
count = 1;
}
}
```

Figure A.1: Arduino Fuzzy Logic Code

The above code is what was used to control the Arduino in Chapter IV. Note that leftover notations indicating how the Fuzzy Logic Rules function are from a guide composed by Dr. Lira et al. (2012) to keep track of how the code functions.

motor_arm_fuzzy_with_PID

```
#include <Fuzzy.h>
#include <PID_v1.h>
const float Pi = 3.14159; //DEFINING Pi

// Instantiating a Fuzzy object
Fuzzy *fuzzy = new Fuzzy();
int A1PIN = 3;
int A2PIN = 6;

int positionPin = A3;
int targetPin = A9;

int realposition = 0;
int desiredposition = 0;

double error = 0;
double Setpoint = 0;
double Coutput = 0;
double output = 0;
double Saturation = 12;
double Cinput = 0;

boolean count = 0;
unsigned long Stopwatch;


//PID PARAMETER SETUP
double Kp = 10, Ki = 0.01, Kd = 0;


//create PID instance
PID myPID(&Cinput, &Coutput, &Setpoint, Kp, Ki, Kd, DIRECT);

void setup()
{
  //PID SETPOINT
  Setpoint = 0;
  //CHANGE PID LIMITS TO OUTPUT NEGATIVE VALUES
  myPID.SetOutputLimits(-24, 24);
  myPID.SetMode(AUTOMATIC);

  //HBRIDGE SETUP
  pinMode(A1PIN, OUTPUT);
  pinMode(A2PIN, OUTPUT);

  // Set the Serial output
```

```
Serial.begin(9600);
//////////////////// FUZZY LOGIC CONTROLLER BEGIN//////////////////
//  // Instantiating a FuzzyInput object
//  FuzzyInput *radianerror = new FuzzyInput(1);
//  // Instantiating a FuzzySet object
//  FuzzySet *neg = new FuzzySet(-1791, -1108, -33, 0);
//  // Including the FuzzySet into FuzzyInput
//  radianerror->addFuzzySet(neg);
//  // Instantiating a FuzzySet object
//  FuzzySet *neutral = new FuzzySet(-33, 0, 0, 33);
//  // Including the FuzzySet into FuzzyInput
//  radianerror->addFuzzySet(neutral);
//  // Instantiating a FuzzySet object
//  FuzzySet *pos = new FuzzySet(0, 33, 1103, 1785);
//  // Including the FuzzySet into FuzzyInput
//  radianerror->addFuzzySet(pos);
//  // Including the FuzzyInput into Fuzzy
//  fuzzy->addFuzzyInput(radianerror);
//
//  // Instantiating a FuzzyOutput objects
//  FuzzyOutput *response = new FuzzyOutput(1);
//  // Instantiating a FuzzySet object
//  FuzzySet *negvolt = new FuzzySet(-510, -255, -255, 0);
//  // Including the FuzzySet into FuzzyOutput
//  response->addFuzzySet(negvolt);
//  // Instantiating a FuzzySet object
//  FuzzySet *novolt = new FuzzySet(-255, 0, 0, 255);
//  // Including the FuzzySet into FuzzyOutput
//  response->addFuzzySet(novolt);
//  // Instantiating a FuzzySet object
//  FuzzySet *posvolt = new FuzzySet(0, 255, 255, 510);
//  // Including the FuzzySet into FuzzyOutput
//  response->addFuzzySet(posvolt);
//  // Including the FuzzyOutput into Fuzzy
//  fuzzy->addFuzzyOutput(response);
//
//  // Building FuzzyRule "IF radianerror = neg THEN response = negvolt"
//  // Instantiating a FuzzyRuleAntecedent objects
//  FuzzyRuleAntecedent *ifRadianerrorNeg = new FuzzyRuleAntecedent();
//  // Creating a FuzzyRuleAntecedent with just a single FuzzySet
//  ifRadianerrorNeg->joinSingle(neg);
//  // Instantiating a FuzzyRuleConsequent objects
//  FuzzyRuleConsequent *thenResponseNegvolt = new FuzzyRuleConsequent();
//  // Including a FuzzySet to this FuzzyRuleConsequent
//  thenResponseNegvolt->addOutput(negvolt);
//  // Instantiating a FuzzyRule objects
```

```
  //  FuzzyRule *fuzzyRule01 = new FuzzyRule(1, ifRadianerrorNeg, thenResponseNegvolt);
  //  // Including the FuzzyRule into Fuzzy
  //  fuzzy->addFuzzyRule(fuzzyRule01);
  //
  //  // Building FuzzyRule "IF radianerror = neutral THEN response = novolt"
  //  // Instantiating a FuzzyRuleAntecedent objects
  //  FuzzyRuleAntecedent *ifRadianerrorNeutral = new FuzzyRuleAntecedent();
  //  // Creating a FuzzyRuleAntecedent with just a single FuzzySet
  //  ifRadianerrorNeutral->joinSingle(neutral);
  //  // Instantiating a FuzzyRuleConsequent objects
  //  FuzzyRuleConsequent *thenResponseNovolt = new FuzzyRuleConsequent();
  //  // Including a FuzzySet to this FuzzyRuleConsequent
  //  thenResponseNovolt->addOutput(novolt);
  //  // Instantiating a FuzzyRule objects
  //  FuzzyRule *fuzzyRule02 = new FuzzyRule(2, ifRadianerrorNeutral, thenResponseNovolt);
  //  // Including the FuzzyRule into Fuzzy
  //  fuzzy->addFuzzyRule(fuzzyRule02);
  //
  //  // Building FuzzyRule "IF Radianerror = pos THEN response = posvolt"
  //  // Instantiating a FuzzyRuleAntecedent objects
  //  FuzzyRuleAntecedent *ifRadianerrorPos = new FuzzyRuleAntecedent();
  //  // Creating a FuzzyRuleAntecedent with just a single FuzzySet
  //  ifRadianerrorPos->joinSingle(pos);
  //  // Instantiating a FuzzyRuleConsequent objects
  //  FuzzyRuleConsequent *thenResponsePosvolt = new FuzzyRuleConsequent();
  //  // Including a FuzzySet to this FuzzyRuleConsequent
  //  thenResponsePosvolt->addOutput(posvolt);
  //  // Instantiating a FuzzyRule objects
  //  FuzzyRule *fuzzyRule03 = new FuzzyRule(3, ifRadianerrorPos, thenResponsePosvolt);
  //  // Including the FuzzyRule into Fuzzy
  //  fuzzy->addFuzzyRule(fuzzyRule03);
}

void loop()
{
  //READING PINS
  realposition = analogRead(positionPin);
//  desiredposition = analogRead(targetPin);
desiredposition = 614;
  error = desiredposition - realposition;
  //THE REASON THIS IS NEGATIVE IS THAT THE INPUT OF PID ALREADY SUBTRACTS FROM SETPOINT, FLIPPING THE SIGN
  Cinput = -error*5/1023*4*Pi;


  // Printing something
//  Serial.println("\n\n\nInput: ");
//  Serial.print("\t\t\tError: ");
//  Serial.println(error);
  //////////////FUZZY LOGIC USAGE///////////////
```

```
  //  fuzzy->setInput(1, error);
  //  // Running the Fuzzification
  //  fuzzy->fuzzify();
  //  // Running the Defuzzification
  //  float output = fuzzy->defuzzify(1);
  myPID.Compute();
  //interpreting PID output
  if (Coutput < -Saturation) {
    output = -255;
  }
  else if (Coutput > Saturation) {
    output = 255;
  }
  else {
    output = map(Coutput, -Saturation, Saturation, -255, 255);
  }


  //WRITING PINS
  if (output < 0) {
    //IF ERROR < 0 NEGATIVE RESPONSE A1PIN LOW A2PIN OUTPUT
    digitalWrite(A1PIN, LOW);
    analogWrite(A2PIN, abs(output));
  }
  else if (output > 0) {
    //H BRIDGE PINS (REVERSED)
    //IF ERROR > 0 POSITIVE RESPONSE A1PIN OUTPUT A2PIN LOW
    analogWrite(A1PIN, abs(output));
    digitalWrite(A2PIN, LOW);
  }
  // Printing something

  Serial.println("Result: ");
  Serial.print("\t\t\tWiper: ");
  Serial.println(realposition);
//if (error == 0 && count == 0){
//Stopwatch = millis();
//Serial.print("Time: ");
//Serial.println(Stopwatch);
//count = 1;
//}
}
```

Figure A.2: PID Arduino Code

This code is identical to the code in Figure A.1, however, the Fuzzy Logic portion of the code is commented out, and replaced with a simple PID controller using the PID toolbox. The PID inputs and outputs are also converted to appropriate values by converting to radians and using the map() function respectively.

BIOGRAPHICAL SKETCH

Cristian A. Barron earned his Master of Science in Engineering in December 2020 from the University of Texas Rio Grande Valley. He has been a member of The Tau Beta Pi Association since 2018. Throughout his years attending UTRGV, he acquired an interest in robotics and control systems. It is this same interest that motivated him to pursue this degree and learn more of the subject through this thesis. He can be contacted at CRISTIAN.A.BARRON@GMAIL.COM.