**Lebanese American University**



**MCE 411-Mechatronics System Design 2**

**Self-Balancing & Self-Parking Segway**

**Instructor: Dr Noel Maalouf**

**Project Report**

**Date: May 10, 2023**

**Group Members**

**Jennifer Mitri - 202001080**

**Adnan Abdallah -202000775**

**Firas Abou Karnib - 202002827**

Table of Contents

[Table of Figures 2](#_Toc134623880)

[Table of Tables 3](#_Toc134623881)

[Abstract 4](#_Toc134623882)

[1. Introduction 5](#_Toc134623883)

[Previous Literature: 5](#_Toc134623884)

[Report Summary: 5](#_Toc134623885)

[2. System Overview 6](#_Toc134623886)

[2.1. System Design 7](#_Toc134623887)

[(i) Mechanical Components 7](#_Toc134623888)

[(ii) Electrical and Sensor Components 7](#_Toc134623889)

[Accelerometer 8](#_Toc134623890)

[What is an accelerometer? 8](#_Toc134623891)

[Procedure for using the Accelerometer 8](#_Toc134623892)

[2.2. Modelling 11](#_Toc134623893)

[Methodology and Summary 11](#_Toc134623894)

[Modeling of DC Motor Behavior 12](#_Toc134623895)

[Modeling the Plant 14](#_Toc134623896)

[Model Parameters: 18](#_Toc134623897)

[2.3. Control Strategy 19](#_Toc134623898)

[Creating our Fuzzy Logic Controller 20](#_Toc134623899)

[Methodology for creating the controller: 22](#_Toc134623900)

[2.4. MATLAB Implementation 23](#_Toc134623901)

[Overview: 23](#_Toc134623902)

[Simulation outputs: 24](#_Toc134623903)

[2.5. Self-Balancing Real Results 27](#_Toc134623904)

[2.6. Self-Parking 29](#_Toc134623905)

[Reaction Wheel 29](#_Toc134623906)

[Self-Parking-Part Two 32](#_Toc134623907)

[3. Limitations and Improvements 35](#_Toc134623908)

[Limitations 35](#_Toc134623909)

[Improvements 36](#_Toc134623910)

[4. References 37](#_Toc134623911)

# Table of Figures

[Figure 1 System Diagram – Self Balancing 6](#_Toc134623912)

[Figure 2 Model of the Robot 6](#_Toc134623913)

[Figure 3 GUI used to save the values read 9](#_Toc134623914)

[Figure 4 MPU6050 Data Collection for Different Orientation 9](#_Toc134623915)

[Figure 5 MPU6050 angles x y z values 10](#_Toc134623916)

[Figure 6 Equivalent circuit diagram of a field coil DC motor 12](#_Toc134623917)

[Figure 7 DC Motor Equivalent Circuit and Free Body Diagram of the Rotor 12](#_Toc134623918)

[Figure 8 Free Body Diagram of the Wheel 14](#_Toc134623919)

[Figure 9 Free Body Diagram of the Body 14](#_Toc134623920)

[Figure 10 Whole Free Body Diagram 14](#_Toc134623921)

[Figure 11 Finding Motor parameters 18](#_Toc134623922)

[Figure 12 Finding plant parameters 18](#_Toc134623923)

[Figure 13 Self-Balancing Principle 19](#_Toc134623924)

[Figure 14 Fuzzy Controller in Fuzzy Tool Box 20](#_Toc134623925)

[Figure 15 Membership Function for Input 1: theta 20](#_Toc134623926)

[Figure 16 Membership Function for Input 2: dtheta 21](#_Toc134623927)

[Figure 17 Membership Function for the Output: voltage 21](#_Toc134623928)

[Figure 18 Simulink Self-Balancing Model 23](#_Toc134623929)

[Figure 19 Self Balancing - Theta - Reference = 0 24](#_Toc134623930)

[Figure 20 Self Balancing - x - Reference = 0 24](#_Toc134623931)

[Figure 21 Self-balancing - Control input - Reference = 0 24](#_Toc134623932)

[Figure 22 Theta for Reference = 1 25](#_Toc134623933)

[Figure 23 Control Input for Impulse Input 25](#_Toc134623934)

[Figure 24 theta for Impulse Input 26](#_Toc134623935)

[Figure 25 dtheta for Impulse Input 26](#_Toc134623936)

[Figure 26 theta (deg) vs N 27](#_Toc134623937)

[Figure 27 dtheta (deg/s) vs N 28](#_Toc134623938)

[Figure 28 distance (cm) vs N 28](#_Toc134623939)

[Figure 29 velocity (cm/s vs N) 28](#_Toc134623940)

[Figure 30 Input and Output Membership Functions -Distance and Voltage 30](#_Toc134623941)

[Figure 31 Fuzzy Logic Toolbox and the Surface of the Output 30](#_Toc134623942)

[Figure 32 Without flywheel rotation 31](#_Toc134623943)

[Figure 33 Maximum weight flywheel rotating. 31](#_Toc134623944)

[Figure 34 Concept behind reference control - Reference = 2 32](#_Toc134623945)

[Figure 35 Input and Output Membership Functions -Distance and Reference 33](#_Toc134623946)

[Figure 36 Fuzzy Logic Toolbox and the Surface of the Output 33](#_Toc134623947)

[Figure 37 Distance behavior of the segway for the self-parking additions to our code 34](#_Toc134623948)

[Figure 38 Theta behavior of the segway for the self-parking additions to our code 34](#_Toc134623949)

# Table of Tables

[Table 1 Variables Used 11](#_Toc134623950)

[Table 2 Fuzzy Logic Controller Rule Base for Balancing Robot Stabilization 22](#_Toc134623951)

# **Abstract**

The robotics sector attracts the most creative minds in today's society. Dreams become a reality thanks to the growth in this industry. The two-wheel self-balancing and self-parking robot serves as yet another example of recent advancements in the field of robotics. A two-wheeled self-balancing robot is conceptualized using the inverted pendulum idea. This type of robot has attracted the attention of engineers and academics worldwide since it is based on a control system that is used to stabilize an unstable system by utilizing efficient microcontrollers and sensors. Smaller size and lower power requirements allow these robots to provide exceptional robustness and capabilities. Other applications for these kinds of solutions include transportation and monitoring. As a result, we decided to try and build a proof of concept for self-balancing and self-parking segways. Our methodology includes researching previous literature, modeling and simulating MATLAB, implementing the Fuzzy logic controllers and building the hardware, and then testing and evaluating our project performance. We had significant success and robustness with self-balancing, but we faced multiple obstacles and issues with self-parking. Further research and development needs to be done to improve on our shortcomings.

# **Introduction**

A two-wheeled self-balancing Segway is developed with the intention of moving people or things from one location to another. It is viewed as a potential future means of transportation and runs solely on battery-powered electric energy. The vehicle is created using mechatronics technology and includes control algorithms. As technology advances, the vehicle can be manufactured with important safety features at a low cost, making it a successful, eco-friendly transportation option that meets human needs. These kinds of implementations are used for a variety of things, including corporate environments, tourist destinations, medical facilities, and personal transportation.

## Previous Literature:

Numerous studies have recently focused on vehicle control methods. One study employed a PD controller and Kalman Filter with an accelerometer and gyroscope for sensing to achieve two-wheel electric vehicle balancing. Another study utilized a kinetic equation and the Newton Dynamics method to control a two-wheeled balancing robot. Two controllers, a pole placement state-feedback controller and a fuzzy logic controller, were used for control action. A third study employed an adaptive sliding-mode control method to achieve self-balancing and yaw control of a human transport vehicle. The study derived mathematical equations and tested the vehicle on different terrains to verify the results. Finally, a fourth study developed a self-balancing robot using an Arduino microcontroller board and implemented PI-PD control design for control action.

## Report Summary:

This report details the development of a self-parking, self-balancing 2-wheel robot. The robot is intended to go around and park itself independently in a predetermined parking space while keeping its balance. The components compromising the robot are a chassis, two wheels, a microcontroller, a microprocessor, an IMU sensor, power supply, and a motor control system. A Raspberry Pi microprocessor is used to create the robot's control system, while the Arduino controller is used to act as the sensor interface, communicating its readings with the Raspberry Pi through serial communication. The Raspberry Pi takes readings from the Arduino and uses those measurements to operate the motors and balance the system to steer the robot to the appropriate parking location.

# **System Overview**

This section will present the 2-wheel robot's hardware details. The hardware model is attached to two wheels and has three large surfaces. The motors that are coupled to the wheels by the motor driver are located on the bottom layer. The Raspberry Pi is located on top of the middle layer, while the IMU is located on the bottom (they are connected by a screw). The IMU is used for determining the vehicle orientation.

A picture containing LEGO

Description automatically generated

Figure System Diagram – Self Balancing



Figure Model of the Robot

## System Design

### Mechanical Components

* **Surfaces:** The three surfaces are made from plexiglass. Each has a dimension of 30x12 cm.
* **Wheels:** Two wheels of 65 mm radius each. The wheels support the robot chassis and give it stability while also allowing it to move. The diameter of the wheel is selected to best match the machine's torque needs.
* **Motors:** Two motors are used for the two wheels; left and right. The motor specification is 150 rpm and 12V. The decision to use a motor on each side of the robot was made to avoid any potential mechanical difficulties that could arise from using just one motor system. Additionally, the torque can be applied to the wheel shafts more efficiently when using two motors, which eliminates the need for complex calculations of frictional and rotational losses.

### Electrical and Sensor Components

* **Microcontroller:** Raspberry Pi controller is chosen. A microcontroller is needed to carry out calculations and process data by carrying out instructions. In this project, the Raspberry Pi is used to run the control software and interact with the motor driver and IMU.
* **Motor Driver:** L298N IC is used for driving the motor. This particular motor driver is well-suited for controlling medium to large DC motors and can handle a maximum current of up to 2A per channel. Two H-bridges are also included, enabling the motor to be controlled in both directions. Accordingly, depending on the polarity of the input signal, the motor can be driven in either the forward or the reverse direction.
* **IMU:** MPU6050is a 6-axis motion tracking device that acts as a 3-axis gyroscope, and 3- axis accelerometer. The MPU6050's gyroscope and accelerometer both offer accurate readings of angular velocity and tilt. To deliver accurate and consistent orientation measurements, we should combine data from the gyroscope and accelerometer.
* **Power Supply:** Total Drill Battery**.** The battery supplies 20V to the two buck converters that in return step down the voltage into 5V for Raspberry Pi and Arduino, and 12V for the motor driver.

## Accelerometer

### What is an accelerometer?

An accelerometer is a tool that measures a structure's vibration or acceleration of motion. The force produced by vibration or a change in motion (acceleration) causes the mass to squeeze the piezoelectric material, which produces an electrical charge proportionate to the force applied to it. Since the mass is a constant and the charge is proportional to the force, the charge must also be proportional to the acceleration.

### Procedure for using the Accelerometer

After conducting a thorough search, we were able to locate the "MPU6050\_tockn" library. This library offers a simple interface for reading data from an Inertial Measurement Unit. The library is based on the I2Cdevlib library, which offers low-level methods for reading and writing data to I2C devices.

This library includes a filter which is the “complementary filter”. In sensor fusion, the complementary filter is a common method for combining the advantages of sensors while making up for their shortcomings. This is achieved because the complementary filter consists of both a low-pass and high-pass filter.

In order to eliminate low-frequency noise and extract short-term changes, one sensor signal is high-pass filtered; the other sensor signal is low-pass filtered. The output of the two filtered signals is then calculated as the weighted sum of the high-pass and low-pass filtered signals.

The MPU6050\_tockn library's complementary filter is intended to deliver a reliable approximation of the device's orientation. To determine the change in orientation, the filter integrates the angular rate readings from the gyroscope. This estimate is then combined with the orientation estimate from the accelerometer using a weighted sum.

So, the filtered angle will have the following form:

The weight applied to accelerometer data is defined by a parameter known as the complementary filter coefficient. This complementary filter coefficient is either determined by trial or error, or by analyzing the mathematical model of the system being filtered. However, the default coefficients are 0.02 for the accelerometer and 0.98 for the gyroscope.

After using the library, we found out the accelerometer needs to be calibrated. This was obtained by adding the offset that was given by the library through the calcGyroOffsets () method.

Next, various data points were gathered. We tilted the chassis forward and backward to obtain different values for the angle and the gyroscope after the chassis was initially stable (0, 0).

To be able to plot the data in MATLAB, we must collect it first. By creating a serial reader in Python, a Python script was created to communicate with the Arduino board and read sensor data. Then using a GUI from a code found on the internet we were able to get the data that were saved in the form of a csv file.

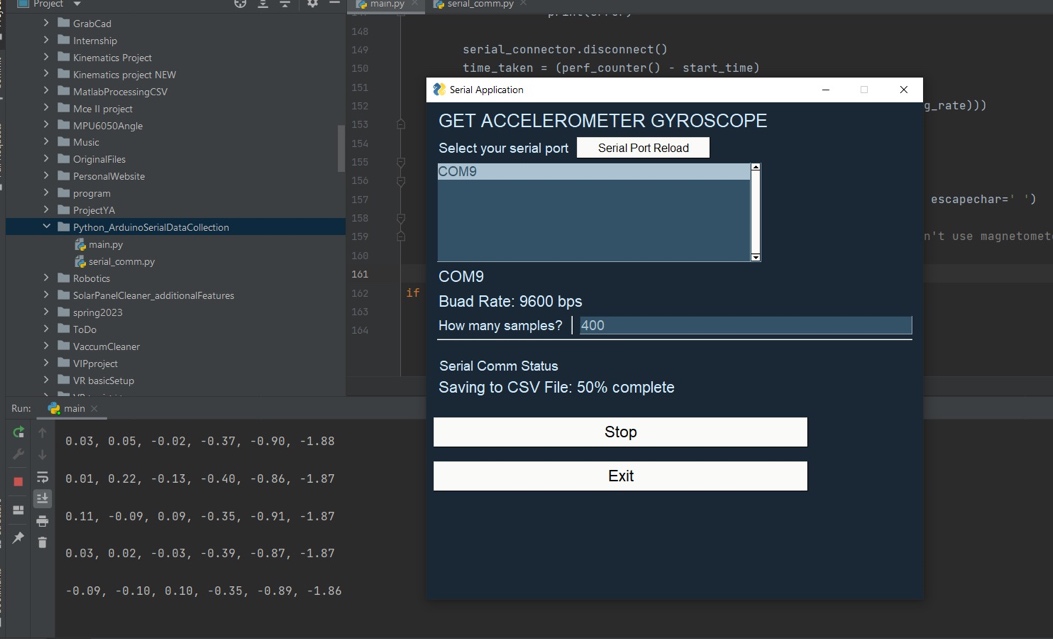


Figure GUI used to save the values read

We discovered that we had precise and logical values by evaluating the data plotted on MATLAB. This is due to the fact that we obtain zero gyroscope and angle data when the chassis is steady. Then to ensure accuracy, we measured the angle when the chassis is tilted using the phone application and compared the results to those on MATLAB. In the below figures we can observe and analyze our plotted data.

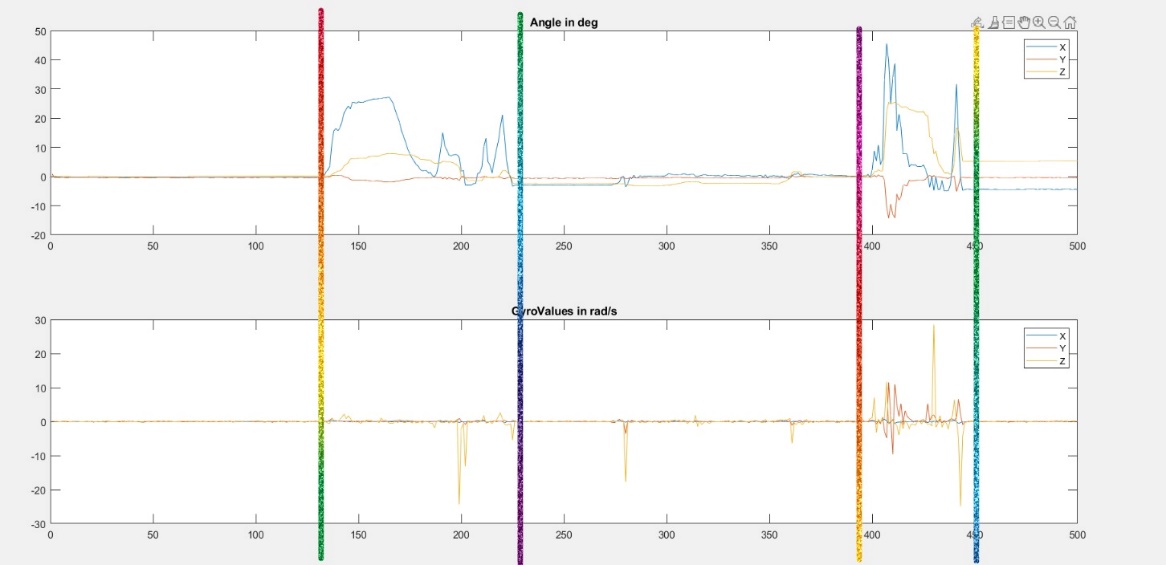


Figure MPU6050 Data Collection for Different Orientation

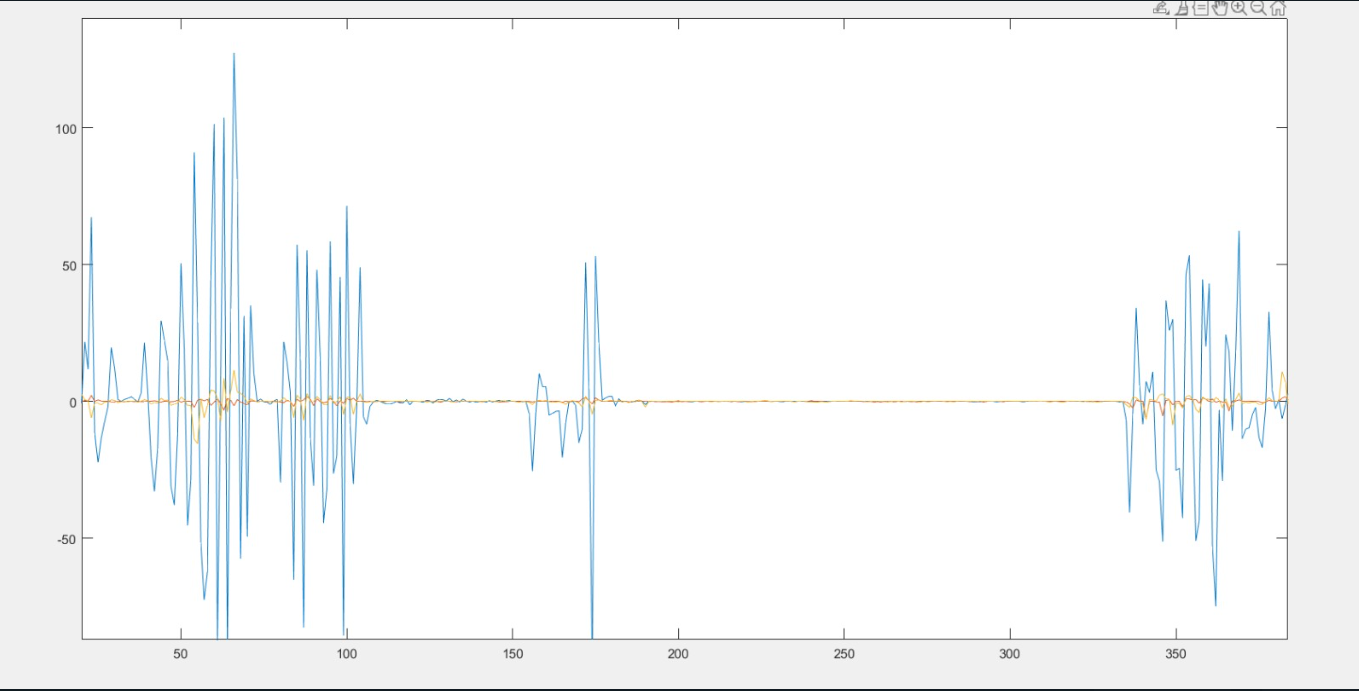


Figure MPU6050 angles x y z values

The first plot is divided into four parts. We can directly tell that the first part corresponds to the chassis being at equilibrium sin both x angle and gyro x are both zero. Moving to the second part we notice that the angle x in changing along with angle y. This is due to us tilting the chassis forward while rotating it along y axis to check the accuracy of all our readings. For the third part the values are not exactly converging to zero since we detached from the chassis and put it at rest than started to rotate it in different orientations along with different tilt speed. This is confirmed through the change of angles and gyro values depending on the force applied to rotate the MPU6050. As mentioned previously, all data were validated using calibrated tools such as phone accelerometer.

In the second plot we can clearly see how we have only one curve that is oscillating. This is the blue curve that corresponds to our angle of interest which is the angle along x axis. To be able to collect these data, we fixed the accelerometer on the chassis after we made sure that it is installed at zero angle and in the correct orientation, then we started to tilt chassis backward and forward while monitoring the actual angle change. We can tell that we have accurate measurements since we do not have angle drift and when the chassis is at rest (sample 200~325) the three angles are aligned with the horizontal axis.

## Modelling

To control the robot more accurately, we rely on modeling to better grasp its dynamic nature. We defined several parameters prior to formulating mathematical equations which include:

|  |  |
| --- | --- |
| **Parameter** | **Definition** |
|  | Distance from origin |
|  | Torque constant of the motor |
|  | Voltage constant of the motor |
|  | Mass of the wheel |
|  | Acceleration of gravity (9.81 m/s2) |
|  | Radius of the wheel |
|  | Armature Resistance |
| n | Gear Ratio of the motors |
| m | Mass of chassis |
| l | Distance between chassis COG (b) and the wheels |
| θ | Chassis tilt |
| θM | Internal motor shaft rotation (before gearbox) |
| θWheel | Absolute rotation of wheel |
| θW | Net rotation of wheel shaft |
|  | Torque applied on the wheels |

Table Variables Used

### Methodology and Summary

We modeled our system using Newton’s laws of motion, which is a challenging task considering the complexity of our system with several components interacting simultaneously:

* We first started by modeling the DC motor. Using the **Voltage and Torque Equations** and the **Electrical Model of the DC motor,** combined with Newton’s Second Law for rotation, we found a proper equation modeling the behavior of the DC motor, we then simplified our equation using some assumptions.
* We then moved to modeling our Plant behavior. We get a system of equations-of-motion, which we then simplify using MATLAB and matrix calculations to obtain equations that govern the tilt of the chassis and the system distance from origin (). The equations depended on the torque applied to the wheels as well as other state variables. We also linearized this equation using some logical assumptions.
* Most importantly, we noticed that the relative motion of the wheel and the chassis has an important impact in linking the different components of our system. This is because when the chassis tilts by θ, the motor also rotates with the chassis. For example, if our motor desires to rotate its shaft by 50 degrees clockwise, and the chassis tilted by 10 degrees clockwise. In such case, the motor shaft also “rotated” (through the tilting) by 10 degrees and the wheels will rotate a net 60 degrees clockwise.
* Therefore, we decided to incorporate our motor into the plant model to satisfy the interdependencies and allow us to achieve a more correct model.

### Modeling of DC Motor Behavior

A lot of the equations are screenshots of the of the word document and inserted as pictures since they caused huge lag and slowness.

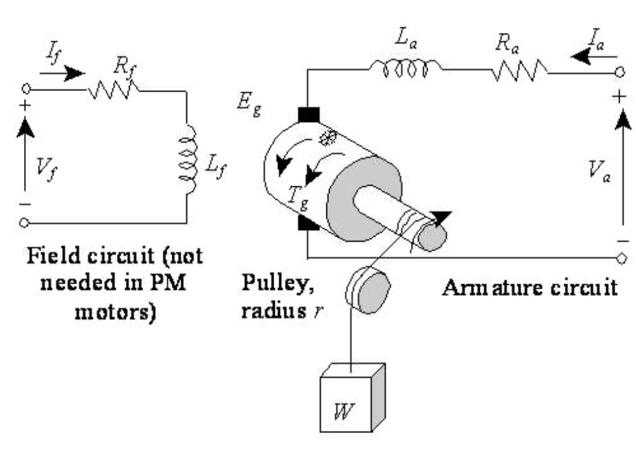


Figure Equivalent circuit diagram of a field coil DC motor

**Voltage and Torque Equations**

Due to the armature rotation, back electromotive force is induced and is proportional to the angular velocity of the rotor **ω** by the voltage constant of the motor

The motor-generated torque is linearly related to the armature current by the torque constant of the motor

**Electrical Model of a DC Motor**

However, L which is the armature inductance is neglected. So, the equation becomes:



Figure DC Motor Equivalent Circuit and Free Body Diagram of the Rotor



### Modeling the Plant

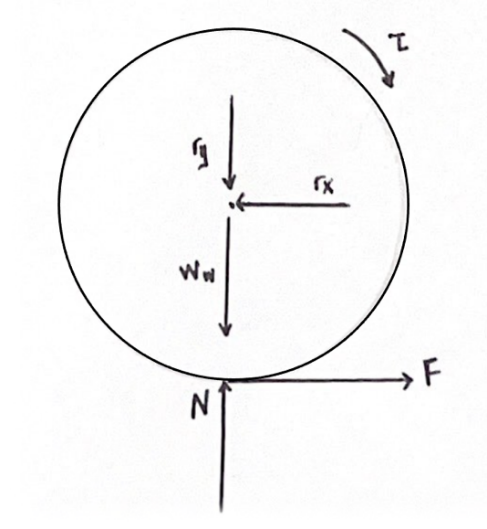
**

Figure Free Body Diagram of the Wheel

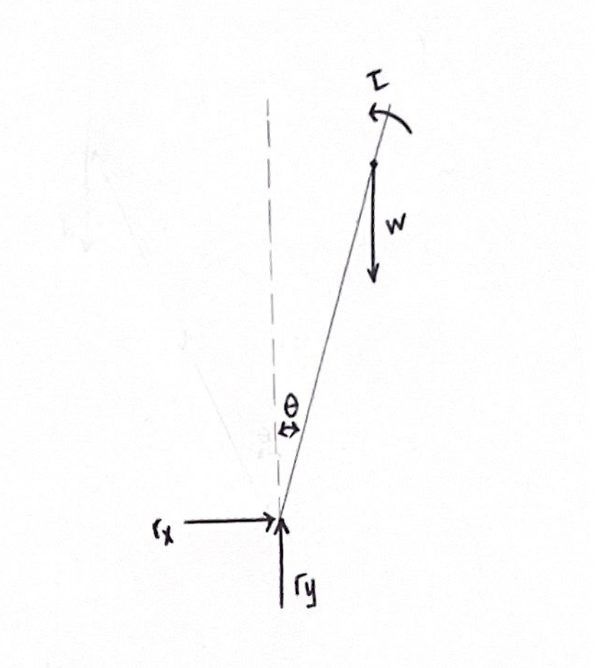
**

Figure Free Body Diagram of the Body

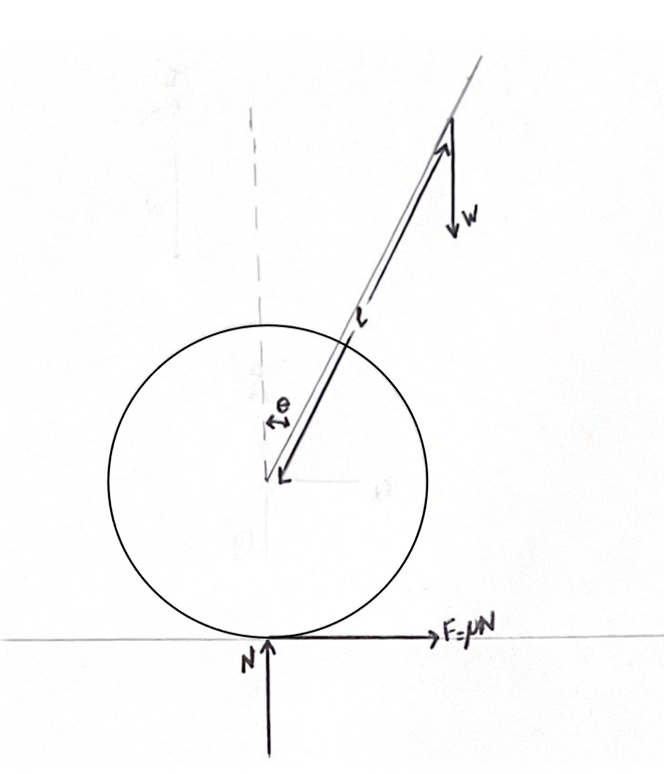
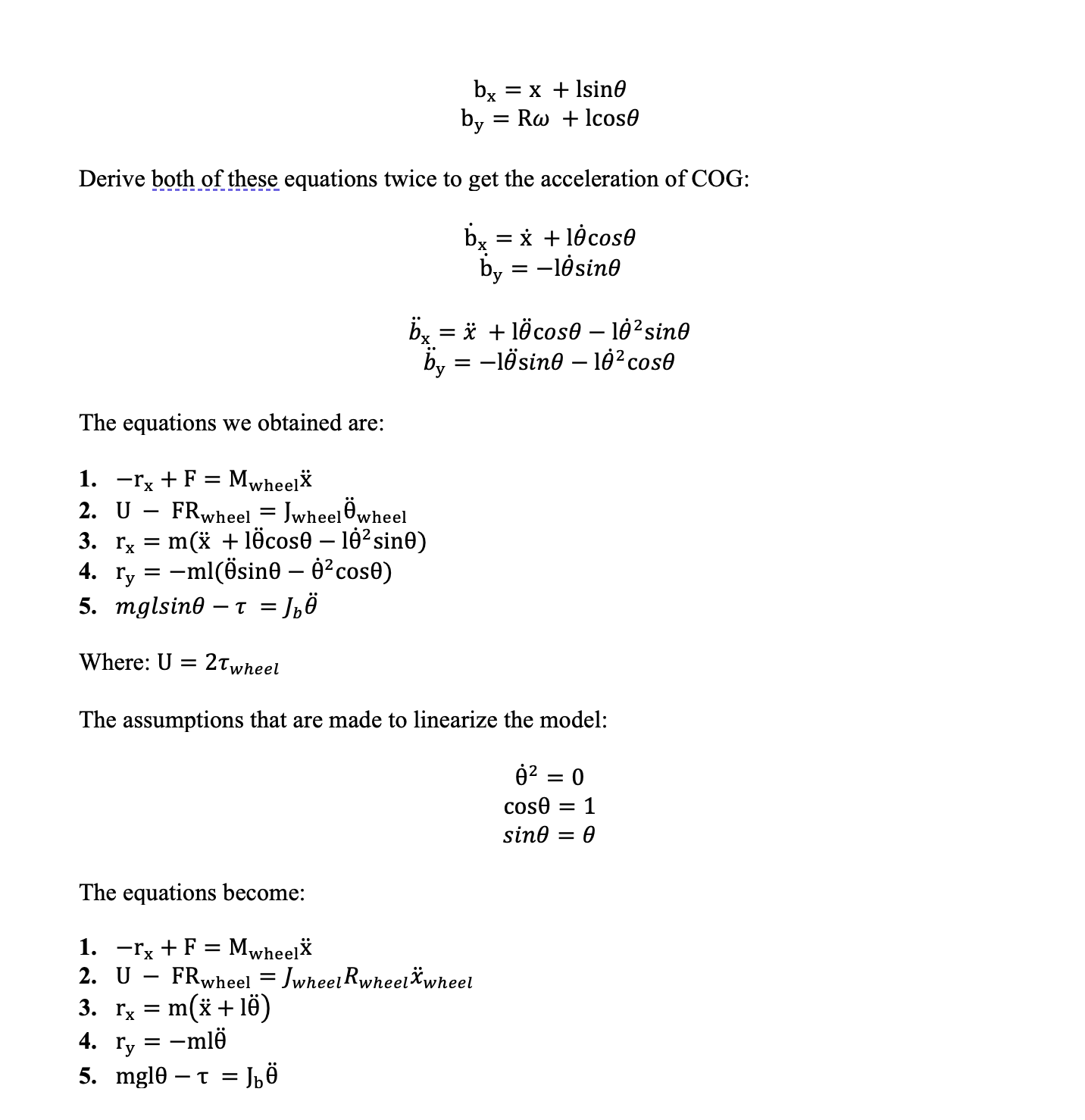
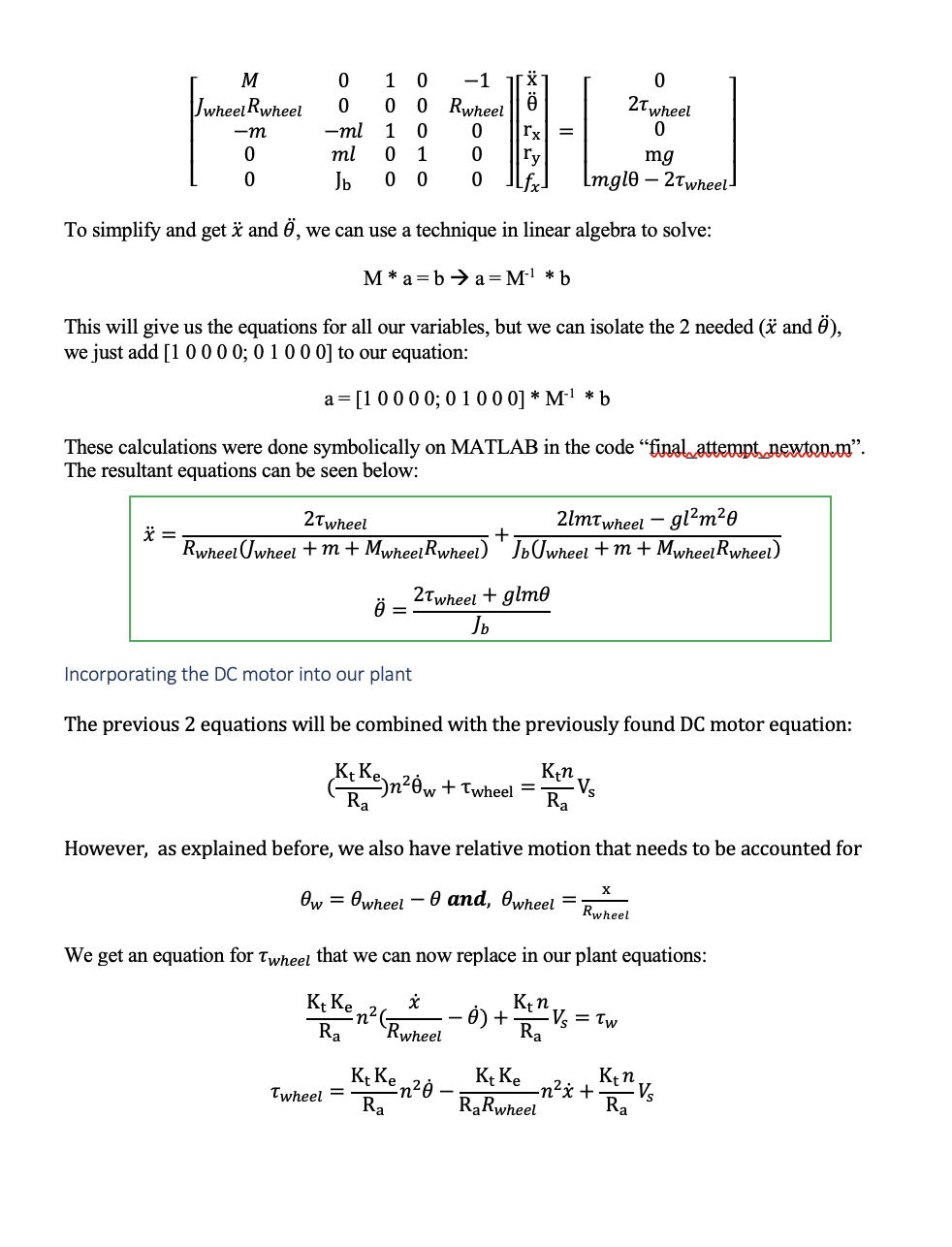
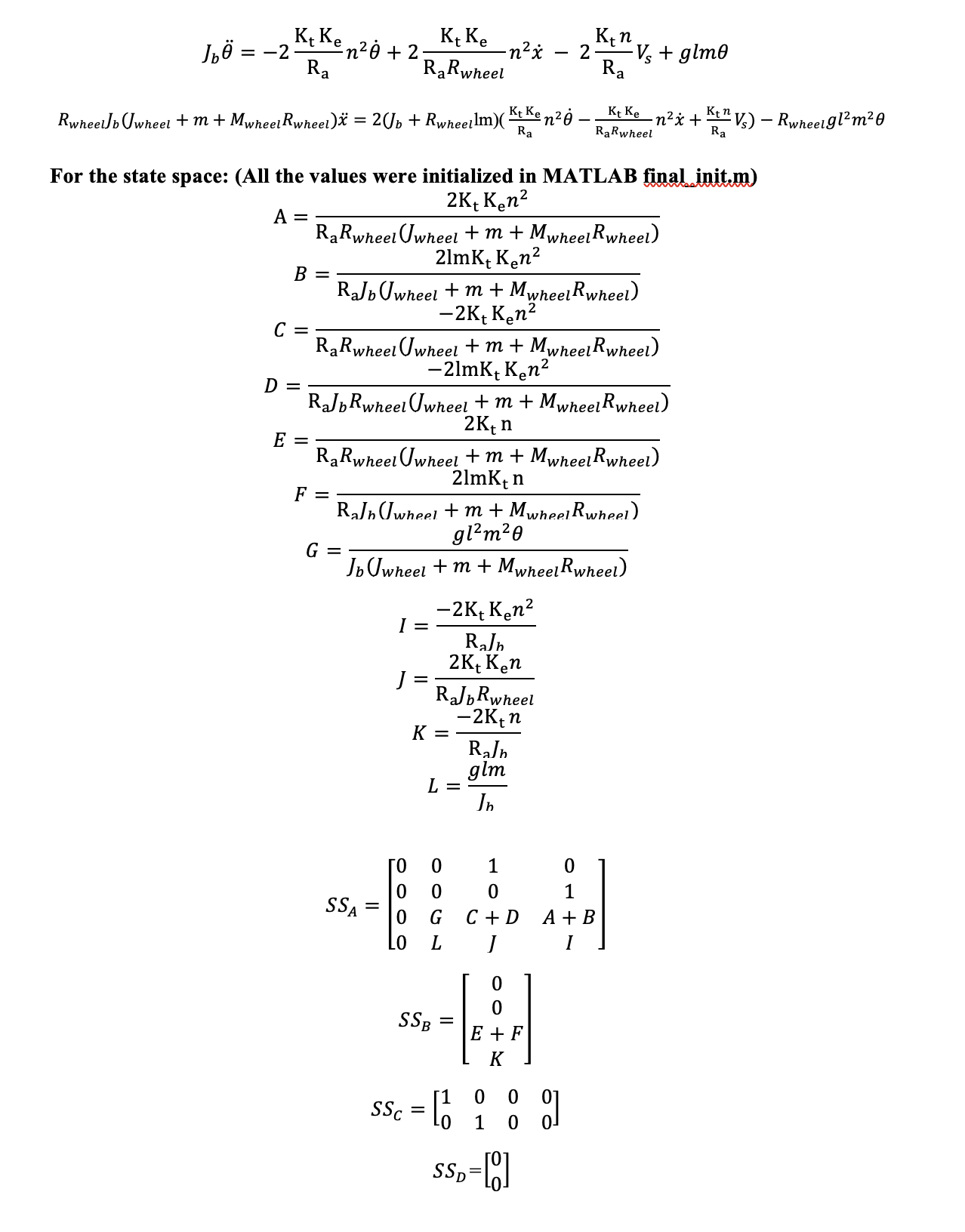
**

Figure Whole Free Body Diagram







### Model Parameters:

Our model has a significant number of constants that we need to carefully choose while balancing assumptions with realism. For the motor parameters, we first had to first find the resolution of the encoders to find the no-load rpm of our motors. This was done by a simple Arduino sketch “Find\_Resolution” that just prints out the current count of pulses (at lowest resolution). We rotate the wheel 1 revolution over multiple iterations, and we found it to be around 470 pulses/revolution. Afterwards, we just observed the motor speeds at no load while looking at the “Mce2\_Main\_2” and found an approximate no load speed of 150 rpms. Lastly, we used a datasheet that shows all the possible motor models for our motor name GM25-370. For a 12V, 150rpm DC motor there is only 1 possibility which is GM25-370CA-15360-XXX at gear reduction ratio n = 34.

This way, we found all the remaining motor parameters through the no-load speed, no load-current, stall-torque, and stall current:

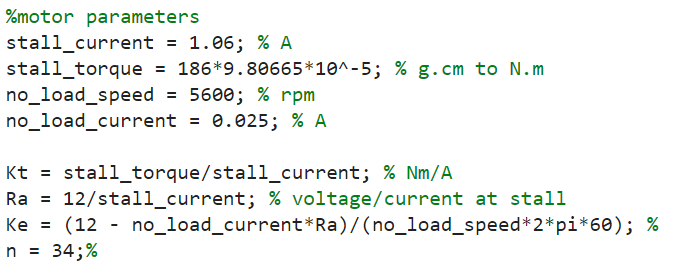


Figure Finding Motor parameters

For the plant parameters, we measured m and Mwheel to obtain their corresponding masses. And afterwards, we calculated their inertias, assuming the wheel as a disk and the chassis as a point mass at a distance l from its axis of rotation (the wheel). We made this assumption because the chassis is not an evenly distributed mass, and it wouldn’t make sense to consider it a rod.

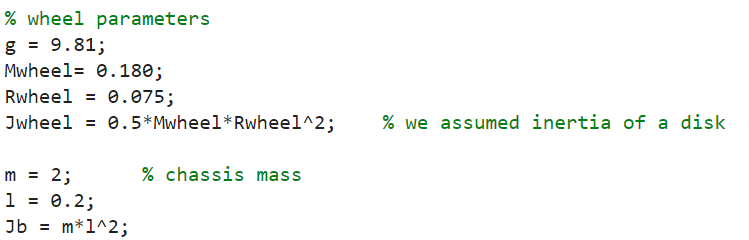


Figure Finding plant parameters

## Control Strategy

To be able to design the right control approach, we must first understand the working principle of self-balancing. The wheels of a self-balancing two-wheeled robot, such as a Segway, rotate in the direction of the tilt when the robot or vehicle tilts forward or backward because of the rider's movement. According to the tilt's direction, this causes the segway to go either forward or backward. This is only possible due to the conservation of momentum that will ensure that the wheel rotating clockwise will cause the chassis to tilt counterclockwise, and vice versa. Using this principle, we can make sure the wheel rotates in a direction that will tilt the chassis back to its unstable equilibrium.

Usually, humans are needed to provide a tilt that is both controlled and maintained by our intuition, thereby the human acting as a controller. Our challenge is to replicate that while keeping the segway autonomous.

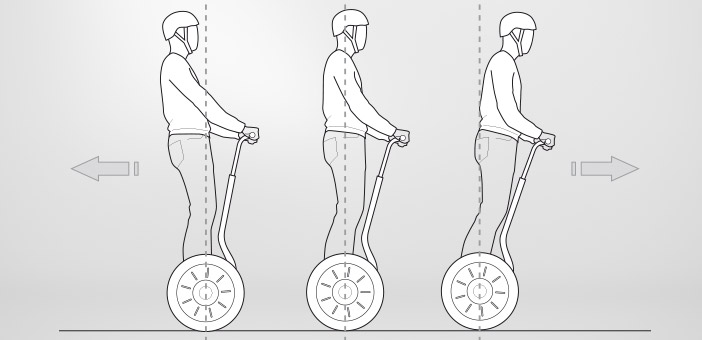


Figure Self-Balancing Principle

To be able to balance the system after modeling it, we must identify the best control approach. Strategies include Neural Network control, PID control, Fuzzy Logic control and many more. However, for this task we opted to use Fuzzy based control strategy.

Our Fuzzy Logic controller will be a classical Mamdani Max-Min fuzzy inference system. We have two input values provided by the IMU which is the tilt of the chassis “theta” (deg) and the chassis tilt speed “dtheta” (deg/s). The fuzzy logic controller must output the voltage needed to control both wheels and move the vehicle as needed. Below are the steps we should follow to develop this fuzzy logic controller:

1. **Crisp inputs:** The IMU readings theta (deg) and dtheta (deg/s)
2. **Membership Functions**: Terminology used to characterize input and output variables, must be defined. Common membership functions include Low, Medium, High… These are responsible for dictating the process of fuzzifying our crisp inputs to fuzzy inputs.
3. **Fuzzy Logic Rules:** Define the fuzzy logic rules that depict how the input and output membership functions are related.
4. **Rule Evaluation:** Apply the fuzzy logic rules to the created fuzzy sets. The implications of the rules are then aggregated to create our Fuzzy output.
5. **Defuzzification:** The final step is to use a defuzzification technique, such as the centroid method or the max membership approach, to transform the fuzzy output set into a crisp output value.

### Creating our Fuzzy Logic Controller

The fuzzy controller is created and implemented using MATLAB since it offers a fuzzy toolbox. This fuzzy logic controller is made up of 2 inputs **theta** and **dtheta** that are obtained from the state variable vectors and one output which is the **voltage**.

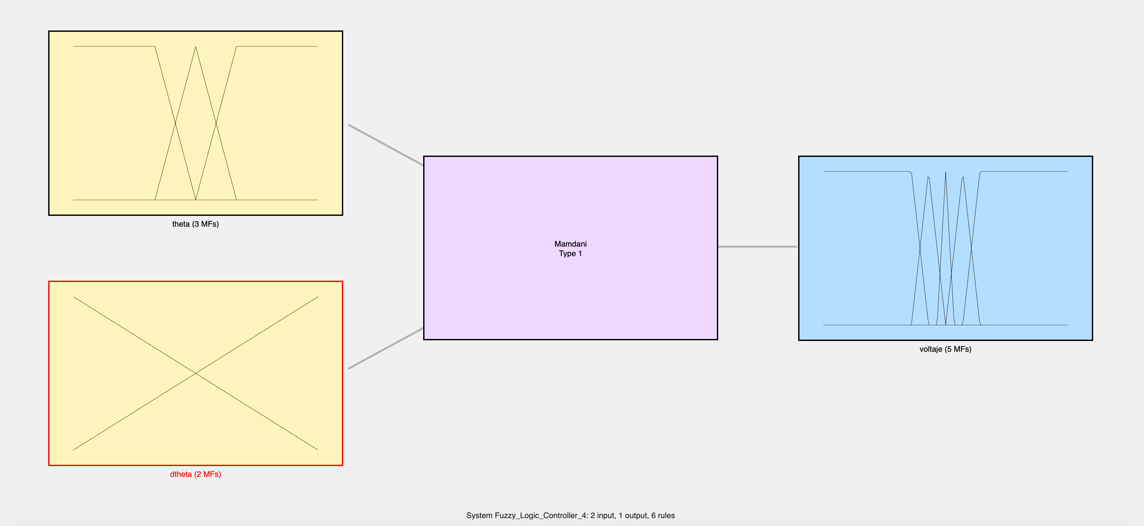


Figure Fuzzy Controller in Fuzzy Tool Box

**For our 1st input X1 = theta**

* Universe of Discourse for input U1 = [-180,180]
* Number of membership functions is N1 = 3

Linguistic values for the fuzzy input :

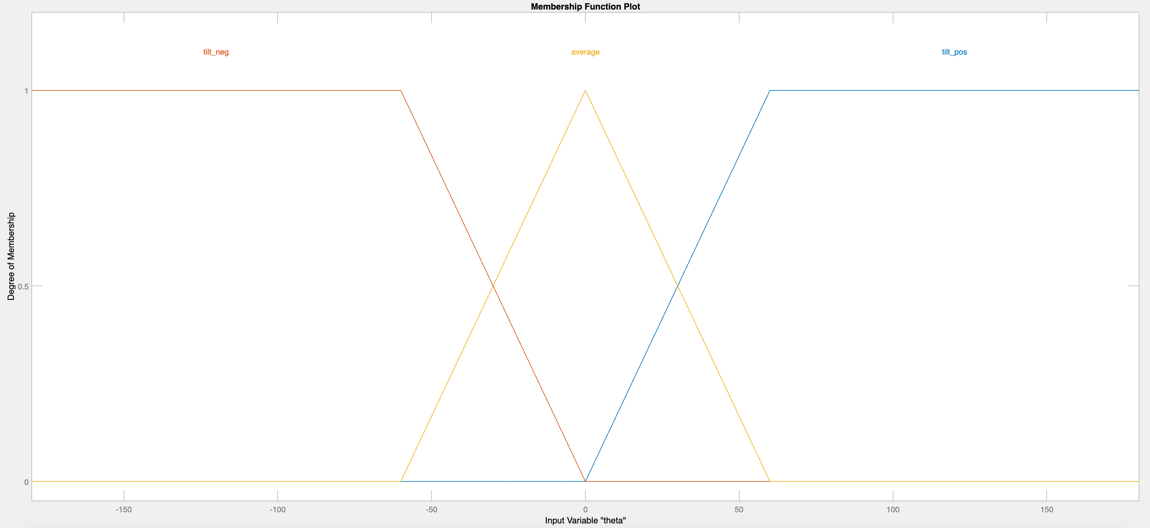


Figure Membership Function for Input 1: theta

**For our 2nd input X2 = dtheta**

* Universe of Discourse for input U2 = [-600,600]
* Number of membership functions is N2 = 2

Linguistic values for the fuzzy input :

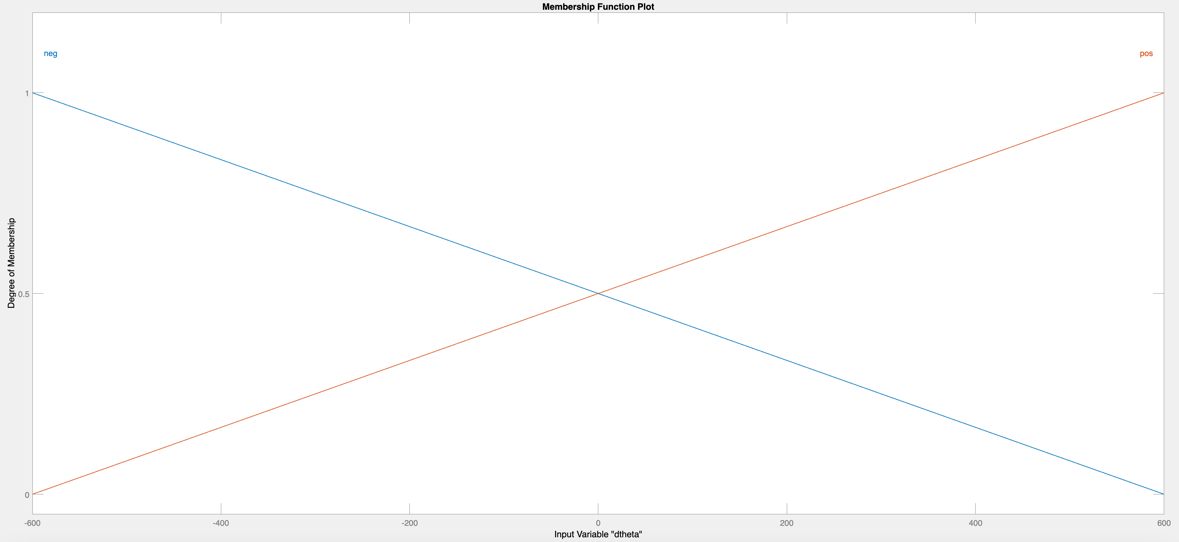


Figure Membership Function for Input 2: dtheta

**For our output: voltage**

* Universe of Discourse for output Y1 = [-2,2]
* Number of membership functions is N3 = 5

Linguistic values for the fuzzy input :

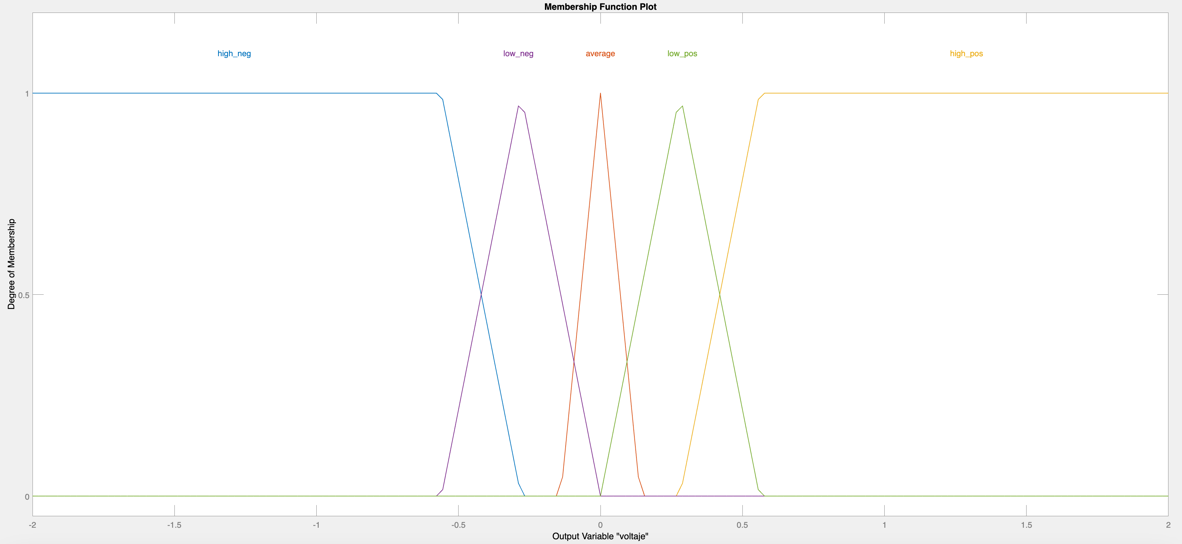
****

Figure Membership Function for the Output: voltage

Following the definition of the membership functions, an associative matrix (considering multiple conjunctive antecedents and a disjunctive set of rules) a fuzzy rule base is created as illustrated below.

Table Fuzzy Logic Controller Rule Base for Balancing Robot Stabilization

|  |  |  |
| --- | --- | --- |
| **X2**  **X1** | **pos** | **neg** |
| **tilt\_neg** | low\_pos | high\_neg |
| **average** | average | average |
| **tilt\_pos** | high\_pos | low\_neg |

According to the FAM table, our rule base is as follows:

**Rule 1:** If theta is tilt\_pos and dtheta is pos then voltage is high\_pos

**Rule 2:** If theta is tilt\_pos and dtheta is neg then voltage is low\_neg

**Rule 3:** If theta is tilt\_neg and dtheta is neg then voltage is high\_neg

**Rule 4:** If theta is tilt\_neg and dtheta is pos then voltage is low\_pos

**Rule 5:** If theta is average and dtheta is neg then voltage is average

**Rule 6:** If theta is average and dtheta is pos then voltage is average

### Methodology for creating the controller:

When creating the fuzzy logic controller, it was a combination of logical reasoning, trial-and-error, and observing the surface created in the fuzzy logic toolbox. We first started with too many membership functions (5 for each input), and it made the process hard where understanding the rules needed, the output behavior, and attempts for improvements became ineffective. We then removed multiple membership functions and converged to 3 membership functions for theta and 2 for dtheta. Having 2 opposing membership functions for dtheta over the entire universe of discourse makes sense because any output produced by one of them is then opposed by the outputs produced by the other membership function. This is only possible because we are using the centroid for defuzzification. This way, one membership function always tries to cancel the opposing membership function by deviating the centroid towards the center.

The other input can also implement the same idea, but we need an equilibrium point when the theta is zero. In such case, our chassis should be doing nothing regardless of dtheta.

For our output membership function, we want the chassis to have a quick response and converge to theta = 0. We first tried having our universe of discourse as [-1,1], but because the defuzzification method is centroid, the output will always average out closer to 0 and never actually fully use the motors if needed. Thus, we made the universe of discourse [-2,2], and then we can clamp the controller output to [-1,1] before sending it to the motors. It is important to note that our output membership functions for “low” and “average” voltages are very narrow because the chassis will go out of control if we allow theta to deviate away from the stable region. However, this doesn’t eliminate the need for these three membership functions because we do need small pushes from our motors during stability, using the “high” voltage membership functions means even within stability we will get aggressive output voltages which will throw the plant out of equilibrium.

The main purpose of dtheta is for disturbance rejection, this is because any spike in dtheta must be opposed by the chassis tilting against it to make sure we don’t leave the stability region. And more importantly, dtheta will allow us to understand how the chassis is moving to “predict” and apply the voltage accordingly.

## MATLAB Implementation

### Overview:

We implemented our self-balancing system in Simulink, where:

* Our plant and actuator reside within the state-space representation. The plant takes as input the voltage Vs which is out control input, and it outputs theta and x.
* Theta, dtheta (derivative of theta), and x are then observed, fed back (if needed) and compared with thee references, if needed. Specifically, only dtheta and theta are fed back. And only theta is compared to a reference\_theta.
* The theta error, and dtheta are then transformed from rad to degrees because our controller uses degrees.
* The controller outputs a voltage which is saturated and then scaled to our DC motor rated voltage [-12,12V]
* We also created a way to add an impulse if needed for checking our model robustness. It is directly applied on the plant, and it is an impulse. We can disable it using the gain block.

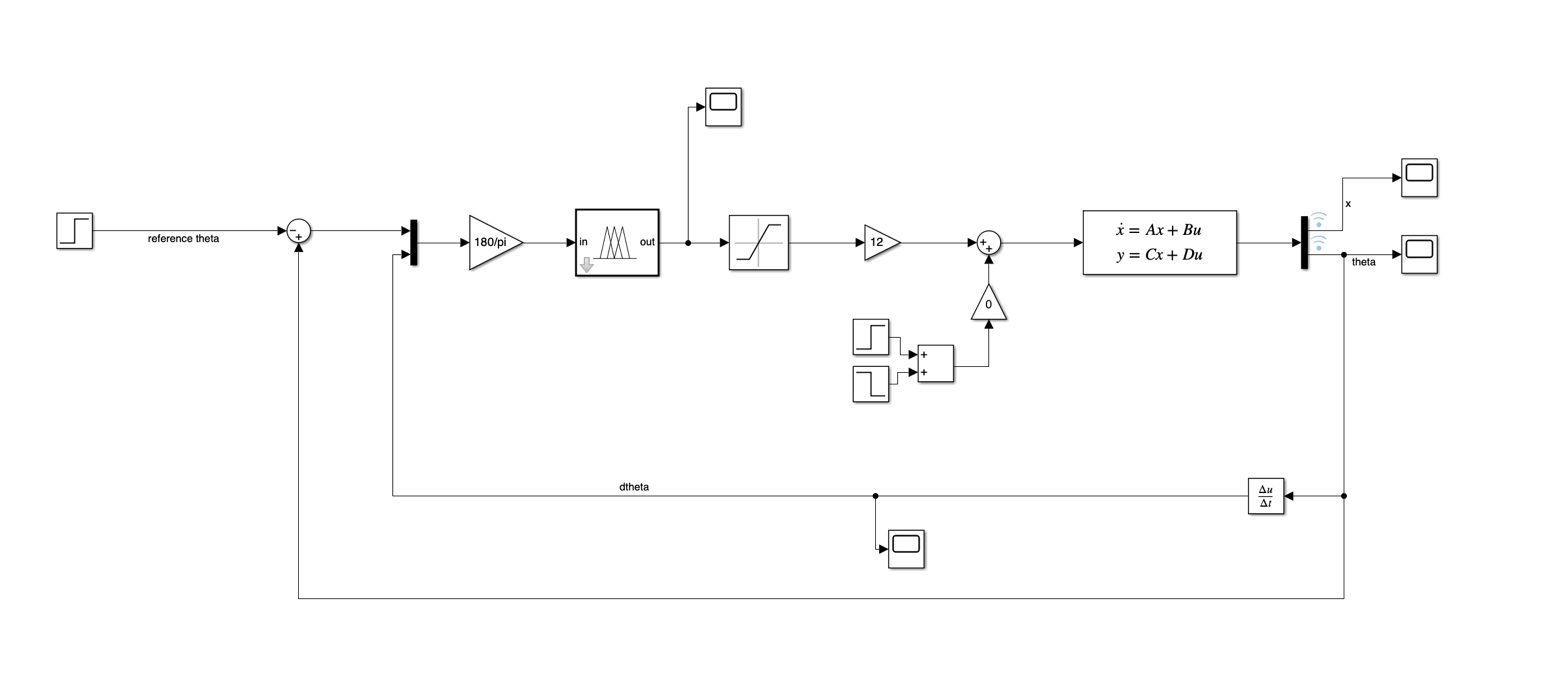


Figure Simulink Self-Balancing Model

### Simulation outputs:

* **Reference: 0**

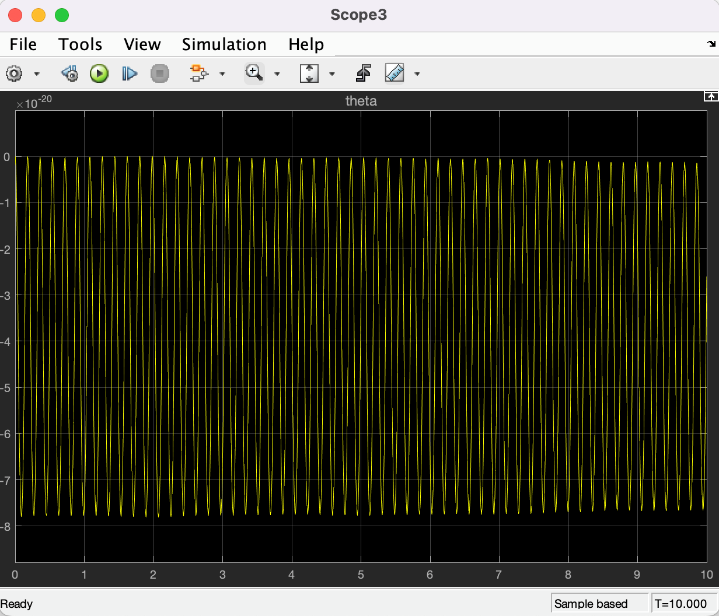


Figure Self Balancing - Theta - Reference = 0

Graphical user interface

Description automatically generated

Figure Self Balancing - x - Reference = 0

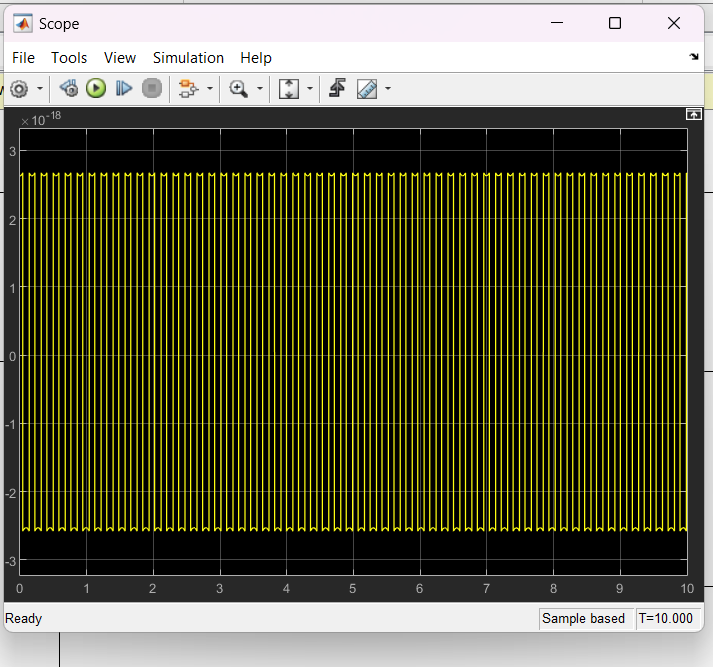


Figure Self-balancing - Control input - Reference = 0

As we can see, if the reference is at zero our model performs well, and theta and x stay at almost zero. In previous attempts, we had divergence despite the reference being zero. This was due to missing modeling steps or large step size. Our plant, being super dynamic, needs a small step size. This is especially important for later in self-parking.

However, any reference different than zero and any added impulse led to divergence. This was the case no matter what we did to the controller or the parameters. This is confusing because the real implementation performs well for a certain range of references.

For example, here is the output for reference = 1. This is the same behavior for all other cases, even disturbances, and variables (x and dtheta diverge).

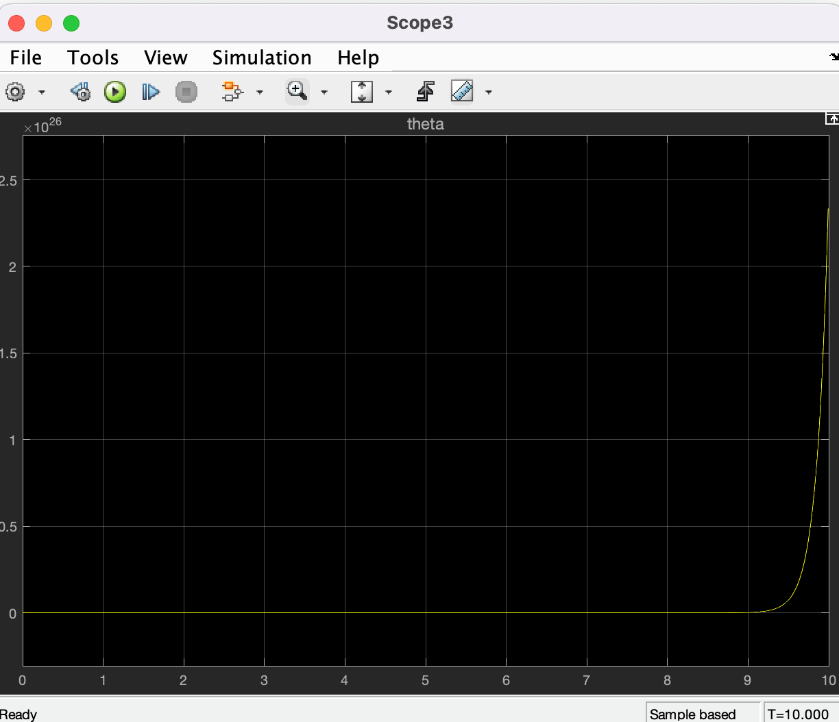


Figure Theta for Reference = 1

**Disturbance as Impulse**

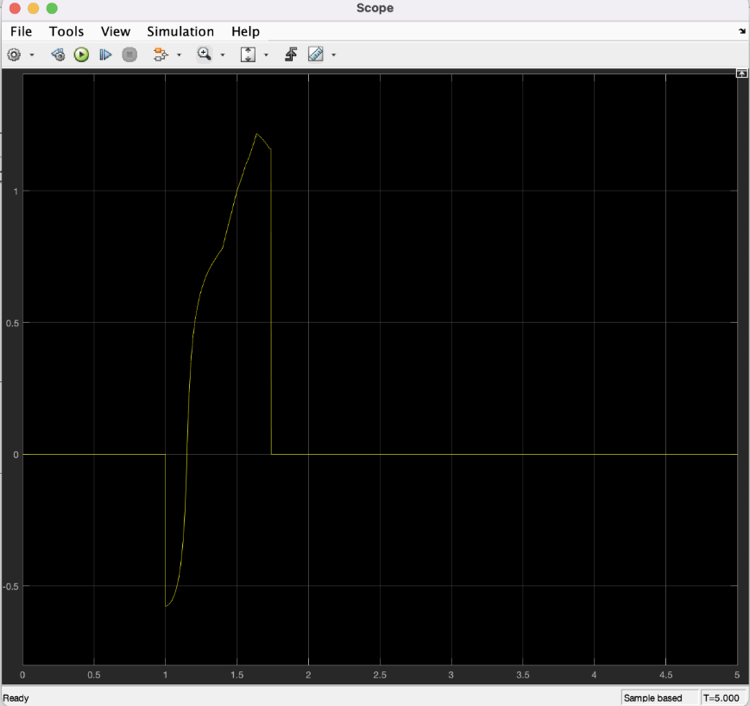


Figure Control Input for Impulse Input

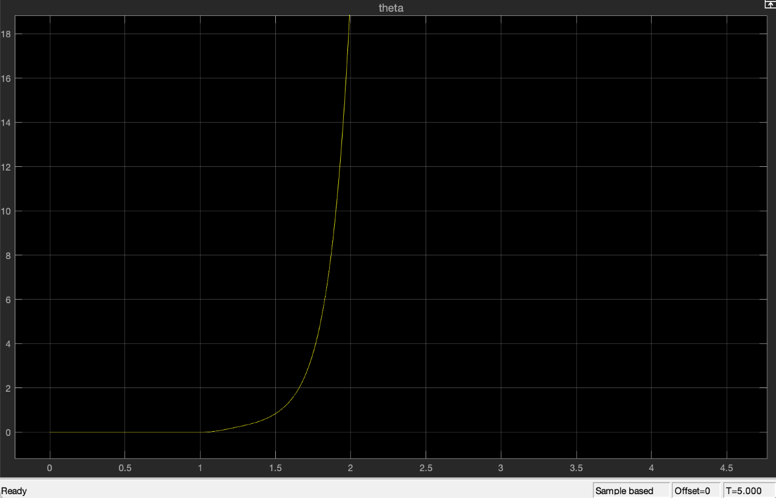


Figure theta for Impulse Input

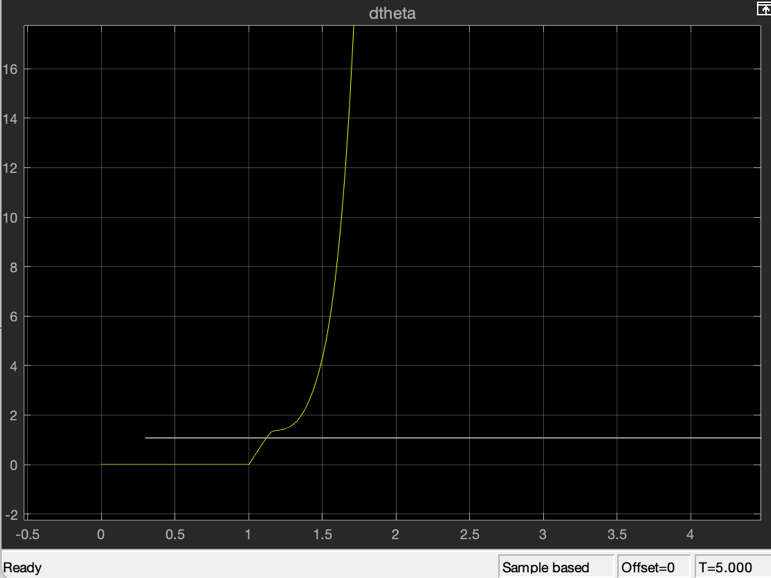


Figure dtheta for Impulse Input

In all cases whenever we have divergence of dtheta or theta, our control input goes to zero because either dtheta or theta leave the universe of discourse. As a result, no rules are triggered making the output zero.

On of our theories for the reason why MATLAB modeling isn’t working is due to our heavy linearization. For example, in the cases of an impulse and divergence. Our dtheta is huge, and thus dtheta2 is very large. This contradicts our assumption that .

Further research is required to improve our model and make it work.

## Self-Balancing Real Results

After experimentation, we noticed that we have a constant calibration error that is approximately 1 degree and changing out reference to 1 enhances our hardware performance greatly and consistently. Here is our output where we observe very adequate performance and disturbance rejection with little bias.

**Reference = 1**

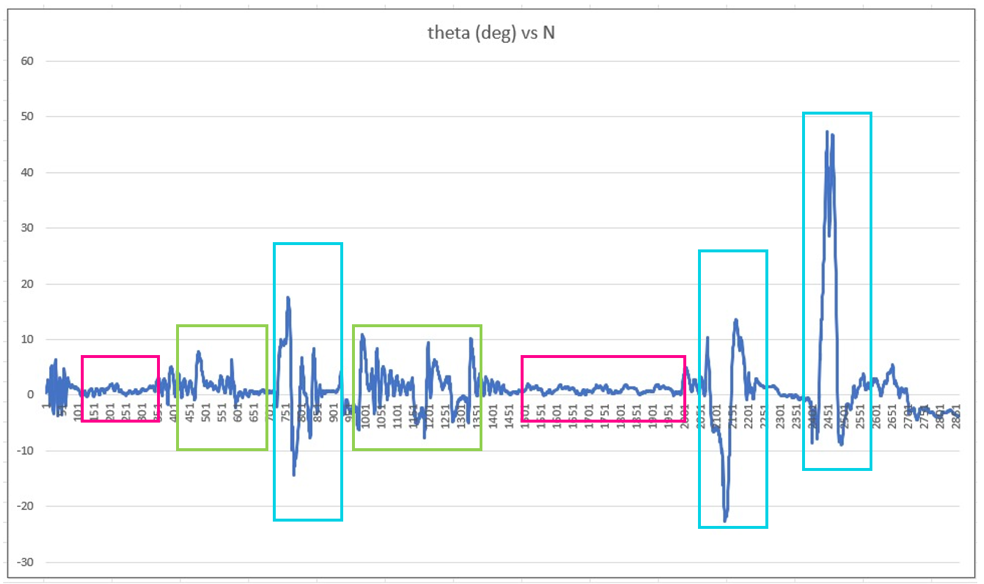


Figure theta (deg) vs N

The pink region is the stable region, the blue is the failure (it was going to fall, and we held it), and the green are impulses that converged to stability. Moreover, these plots allow us to see the range of stability of our Segway where around 10 degrees is the maximum allowable tilt that would not lead to divergence.

The plots for dtheta, distance, and velocity of our chassis is also included below. For the distance curve, we can see that the distance oscillates around the current position of the chassis, which proves that our system is very robust and has almost no bias at reference = 1. Whenever disturbances are added, the controller rejects it by moving the whole system to avoid falling. These results make sense and agree with our targets set at the beginning of our project.

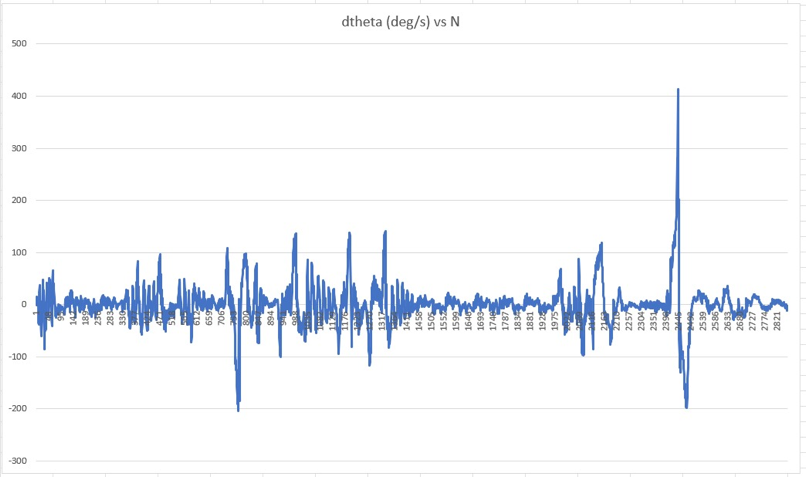


Figure dtheta (deg/s) vs N

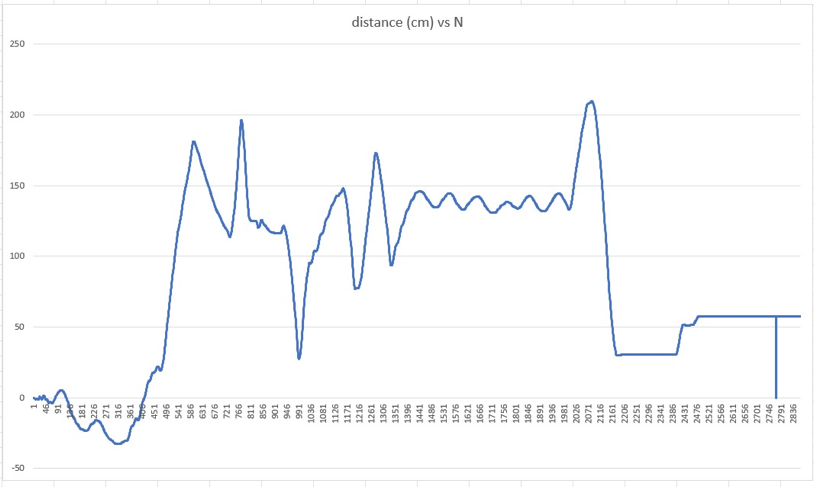


Figure distance (cm) vs N

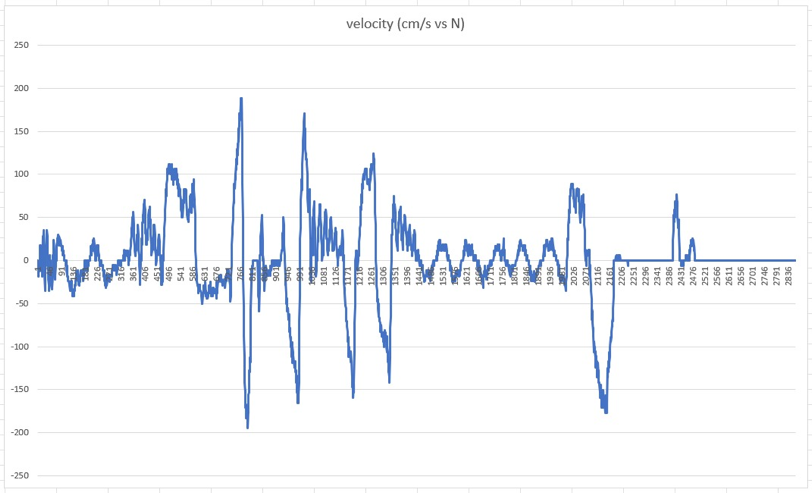


Figure velocity (cm/s vs N)

**Note:** The encoders have weird behavior whereas you can see in the end we have constant distance(no displacement) despite the chassis moving in the last part of the plot.

## Self-Parking

Our aim for self-parking is to have the Segway move a specific distance in a straight line and then stop moving around that point. Since we have the self-balancing working already, we just needed a disturbance to be added by something in our system to cause the chassis to tilt in a desired direction and consequently move in that direction.

However, since there is no research or previous literature on self-parking segways, we decided to explore two ways of achieving self-parking in our chassis: reaction wheels and reference control. Our choices were based on both logical observations we saw when working on self-balancing, as well as literature that suggests the use of reaction wheels for modifying the tilt of objects.

### Reaction Wheel

The working principle behind the reaction wheel is the like the principle used in our self-balancing. The torque applied by the motor to rotate a flywheel will apply an opposite torque on our chassis. According to previous literature, we found that maximizing the kinetic energy expenditure of the flywheel is the easiest method of achieving effective control with the reaction wheel. Thus, we used a high rpm DC motor along with a 3D printed disk whose mass distribution is concentrated at the radius and weighted with added bolts.

#### Fuzzy Control

Our control strategy for self-parking is very simple. Rotating the wheel in one direction should cause a bias in the segway that will make move forwards, and the case is the same for the opposite direction. Our fuzzy logic controller is a Mamdani Max-Min fuzzy inference system, with the input “distance” which is the distance that the segway must travel.

distance error = reference distance – position of chassis

The position of the chassis is provided to our controller by the encoders. The controller will then output the voltage needed to control our reaction wheel. To design the fuzzy logic controller, we decided to start simple with two membership functions for the input and the output. After gaining a deeper understanding of fuzzy logic controller design in our self-balancing, we used the same principles for designing this controller. We used the “surface” provided to us by MATLAB to check whether our controller output and performance is satisfactory, and we extended the universe of discourse of our output so that our surface reaches the maximum output voltage for our reaction wheel (Universe of discourse for output is [-2,2] so that our output voltage ranges from [-1, 1] as seen in the figure below). We must note that to avoid repetition, our self-parking fuzzy controller for our reference control employs the same idea and design methods, but it has a different output. Therefore, we will explain our last fuzzy controller in the next section with less detail.

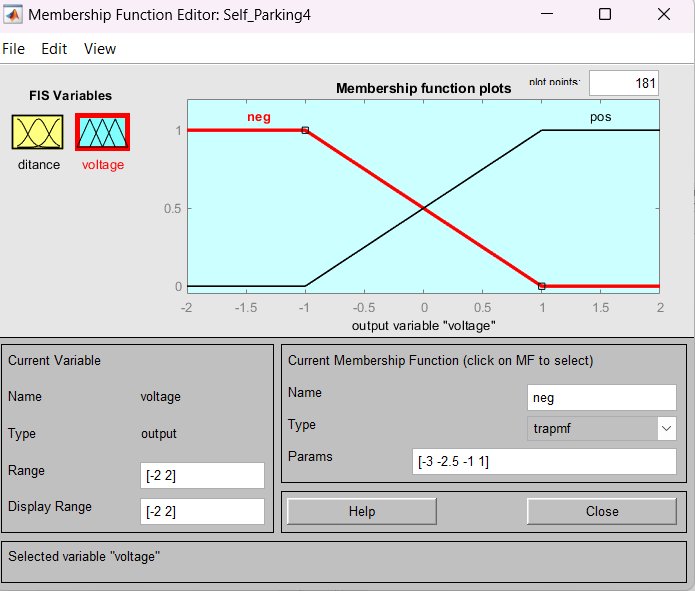
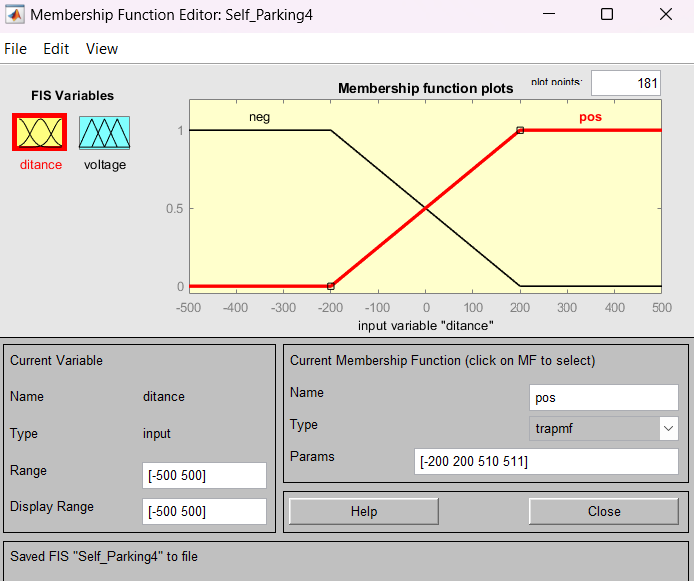


Figure Input and Output Membership Functions -Distance and Voltage

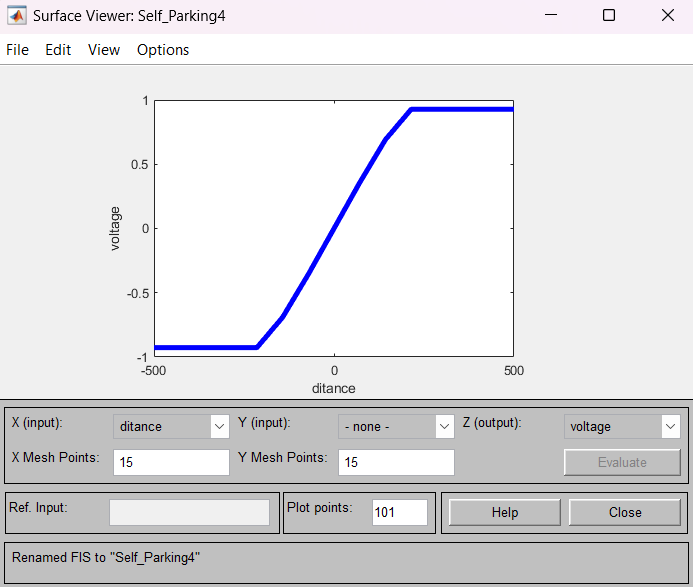
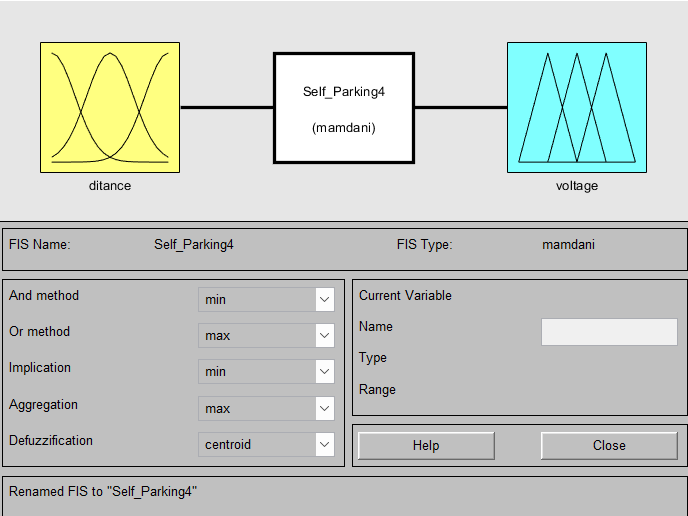


Figure Fuzzy Logic Toolbox and the Surface of the Output

#### Results

Fortunately, adding the reaction wheel and motor and the extra H-bridge to our system did not affect the self-balancing of the system. However, because the reaction wheel is 100-150g with the added nuts and bolts, while the chassis is around 2.5 kg total with high torque self-balancing action. The effect of the reaction wheel was negligible if not minute at best.

As you can see, we tried adding weights while trying to avoid bad weight distributions, and as a result, vibrations. For our current hardware this is the plot for the maximum weight vs without the flywheel rotation.

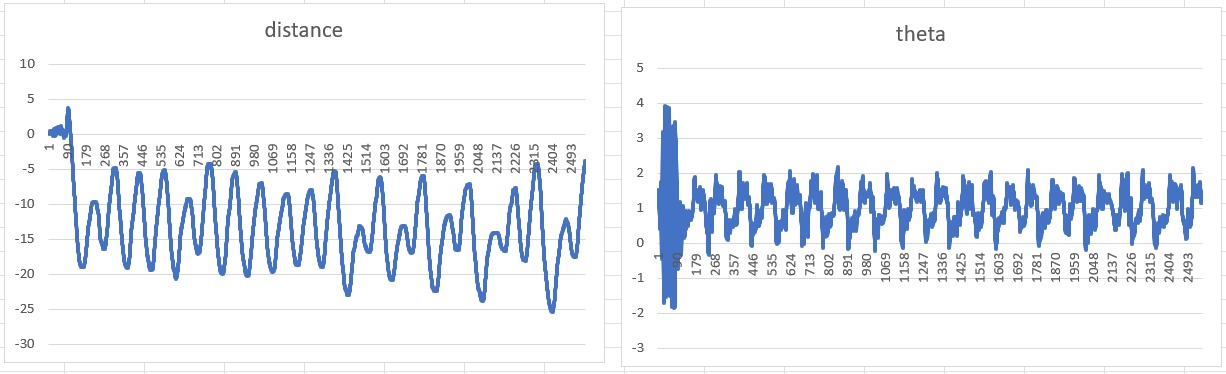


Figure Without flywheel rotation

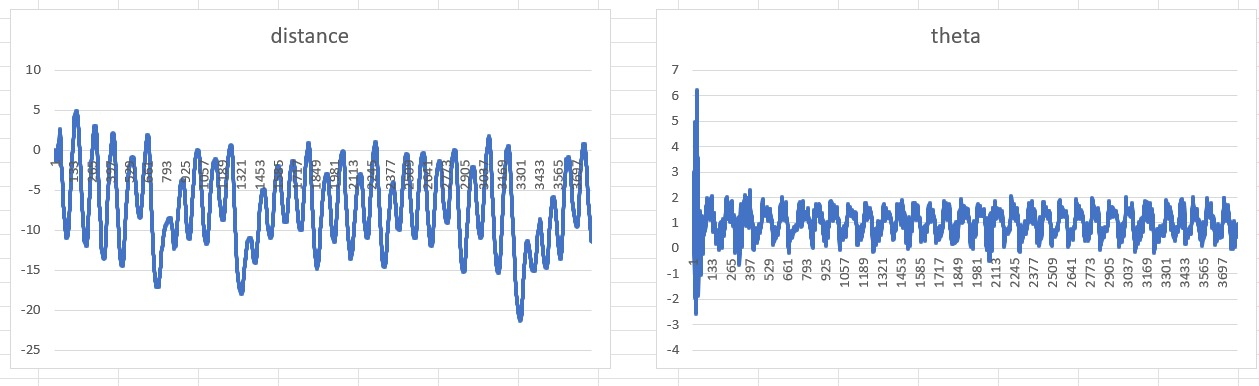


Figure Maximum weight flywheel rotating.

We can see in the figures that when the flywheel is not rotating, our chassis is oscillating consistently with no bias towards 1 direction. On the other hand, with our reaction wheel employed, we can see that the chassis swings in the negative distance more aggressively sometimes. From our experiment, we believe that the reaction wheel might be feasible, but its size and energy expenditure required might make it impractical for a human sized segway, if this concept were to be scaled.

We are aware that calibration errors and small variations with the tests might be the reason for our difference in values, so we tested the system multiple times to make sure that our difference with and without the wheel is consistent. Besides, calibration, if done carefully and with the help of our phone sensors for reference, rarely produces a vast difference unless the calibration is very invalid (our calibration variation is usually within ±0.2 degrees).

### Self-Parking-Part Two

We chose reference control because we believed it would be effective since by adjusting the reference away from one, we can get constant movement in one direction which we desired. This phenomenon can be seen in the figure below:

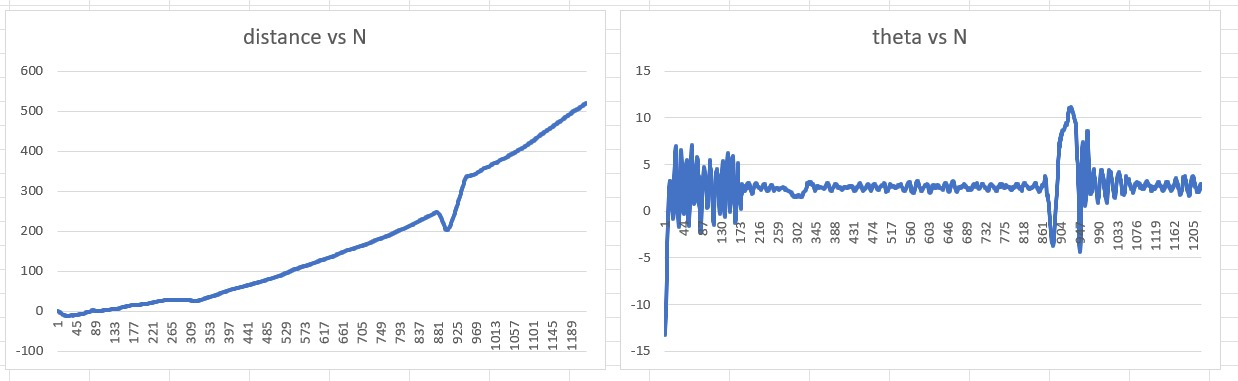


Figure Concept behind reference control - Reference = 2

When setting the reference to be farther from our equilibrium reference of 1, we obtain consistent movement in one direction. This is because the segway tries to balance at a theta that is not fully upright, thus allowing gravity to tilt the chassis in one direction consistently, and when the self-balancing tries to fight the effect of gravity, what we end up with is net movement forward.

Therefore, we decided to explore the range of references that might lead us to a stable and effective self-parking experience. We tested out multiple references and found that the ranges:

* [0, 2]: slow, steady, and stable segways movement in forwards or backwards direction
* : accelerating, risky, and fast movement. This region can be used with advanced control to unlock higher speeds for the chassis, but we must not stay in such references for too long as the segway will eventually go into instability.
* Beyond these references, the chassis is unable to stand up for more than 2 seconds and the references must not move into such ranges.

Note that the weird behavior of the chassis indicated with an arrow in our plot is just the effect of us manually rotating the segway as to avoid hitting a wall. It can be ignored.

#### Fuzzy

As mentioned before, we will not dive deep into the design of the controller as it is very similar to the already explained controllers. Our output in this case is the reference of our self-balancing control system.

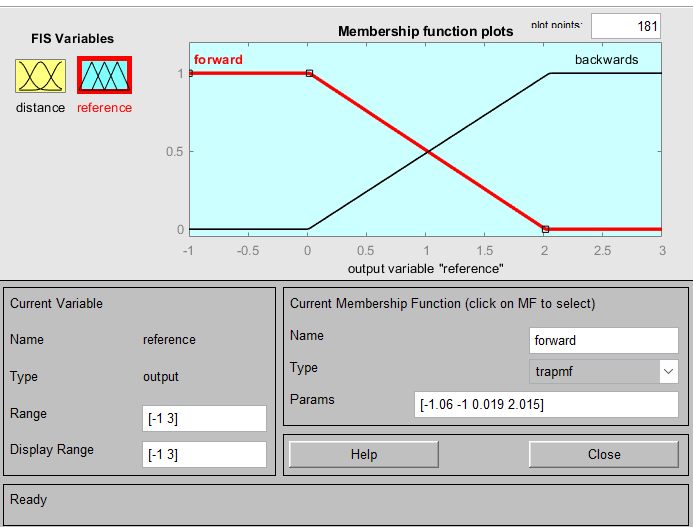
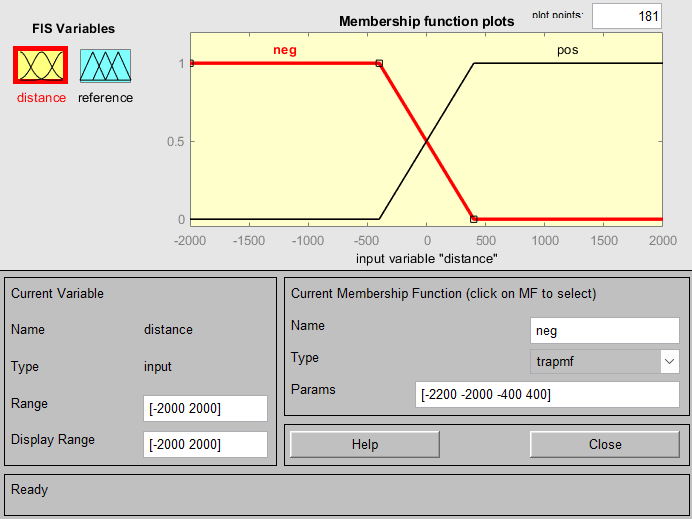


Figure Input and Output Membership Functions -Distance and Reference

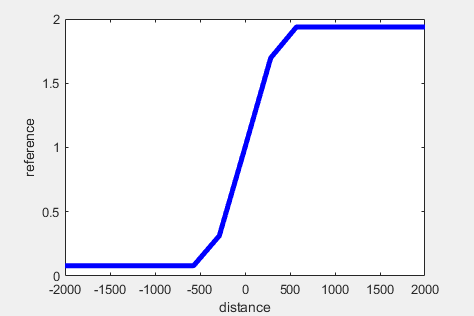
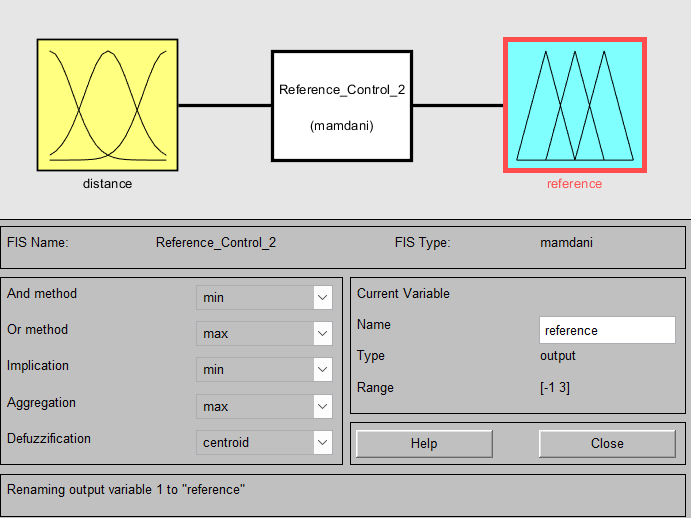


Figure Fuzzy Logic Toolbox and the Surface of the Output

#### Results

Despite the reference control being a very fool-proof idea in theory. We encountered an obstacle that we didn’t expect. Our loop execution time in our python was too “large”, and, effectively, the sampling time of our controller is too large for it to be able to converge our system. We have noticed this issue before when we were working on the self-balancing segway. When our self-balancing code was unoptimized for the purpose of debugging and experimentation, we had a noticeable effect on the performance of the controller, but after optimizing the code and giving more tasks to our Arduino, which is very fast and optimized due to using C++, we reached our best sampling time of 16ms.

Therefore, when we added a few lines of code for the self-parking controller to our infinite loop. It was enough to turn our loop execution time to 20+ ms, which is apparently too slow for our system to stay in stability. Below you can see the performance of the segway for a constant reference of 1 but with the added lines of code necessary for self-parking. The segway always fails at staying in stability and ultimately tends to fall over:

Graphical user interface, chart, application, line chart

Description automatically generated

Figure Distance behavior of the segway for the self-parking additions to our code

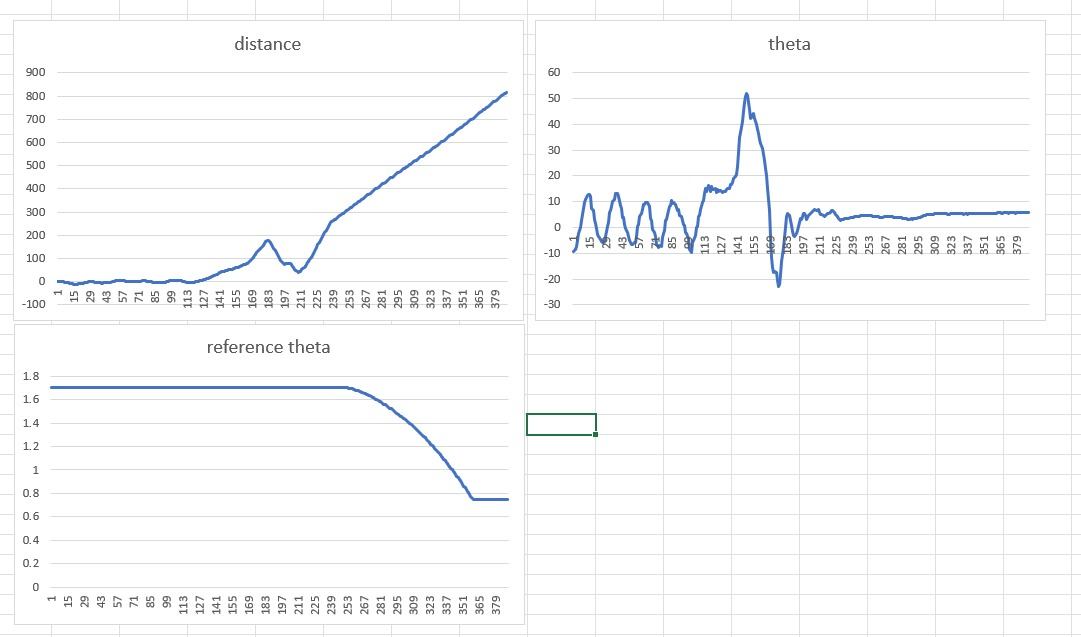


Figure Theta behavior of the segway for the self-parking additions to our code

This obstacle can be avoided by implementing our system with C++ in the raspberry pi or by replacing the raspberry pi with another Arduino. However, with the time restrictions, budget restrictions, and skill restrictions (we only know basic C++ and Cmake), we are unfortunately unable to meet our goals of having a success at self-parking. This was a very important lesson for valuing speed performance of controllers.

# **Limitations and Improvements**

## Limitations

1. The encoders behave strangely:

Despite the encoders being embedded with our motors, they do not portray the most consistent and accurate behaviors. Mainly, one of the encoders works in phase B and not phase A thus it can’t be used to see the direction of the motor. As a result, in our hardware implementation we only used one motor to track the distance the chassis traveled. Moreover, this led to other limitations where we can’t track the rotation of the chassis because we need the speed or rotation of both motors. Additionally, we can’t do speed matching for the motors because we can’t find the velocity for the motor with the correct encoder. However, we were lucky that our motors portray the same velocity for the same voltages, but if that was not the case it would have been a huge issue.

Another weird behavior for the encoders is the fact that despite both motors being isolated from each other except for the power supply, rotating motor 1 at high speeds will cause motor 2 phase A and phase B to oscillate randomly. Thus, both encoders can’t be used even if the second encoder was functioning normally. We tried isolating the encoder pins and the motors completely, but nothing was stopping the effect of motor 1 on motor 2.

Lastly, since our encoder is magnetic rotary with alternating north and south poles on the disc that the sensor has to read, if the encoder plate lands perfectly north and south (intermediate) in front of the sensors, we would obtain oscillating (0 to 1) phase A and phase B which would cause a discrepancy in out distance measurements in some uncommon cases.

1. Calibration issues with IMU

As we mentioned in our results, our IMU calibration is not always perfect, especially since we have to calibrate every time we turn on the Segway. This both affects the repeatability and the accuracy negatively. A future improvement might include an automatic calibration without the need for human intervention.

More importantly, we always had a 1-degree calibration error that we overcame by changing the reference. This is not too much of a limitation, however it added complications to our control system.

1. Bias in the Segway’s movement preferences

This is mainly due to the 1-degree calibration error. If we set the reference to zero, it would favor the movement of the Segway to one side.

1. Possibility of the Center of Gravity deviating from the Center of the Chassis

Because the constant errors in measurement and weight distribution of objects we might have deviation in the center of gravity from the central axis of the chassis which is also causing a bias in the tilt.

1. No implementation of motor speed-matching for general cases

As we mentioned before, due to our encoder malfunction motor speed matching could not be implemented. However, it is important to have such a feature since even motors having the same speed behavior might portray different speeds after being used for a while.

1. Hardware Issues

Other than the encoders malfunctioning, the motor quality we received, despite their price, was not optimal. Mainly, the wheel on one of the motors kept falling off no matter how much we tightened the grub screw. Moreover, we had a lot of problems with the wires disconnecting randomly and having connection issues.

## Improvements

1- Modeling

The value in having a model that is accurate and representative of the model is that it would allow the process of creating the controllers and testing out additions to the system such as the self-parking much easier. It is unfortunate that Newton’s second law did not work with us perfectly, and now we find value in Lagrangian method as it avoids all the complex dynamics of the system by looking at it from an energy standpoint.

2- Faster loop execution time for our controllers

Our project is a clear portrayal of the importance of having smaller sampling times for the stability of the system. Unfortunately, one of python’s main limitations is the fact that it performs poorly from a runtime standpoint. Our self-parking control loop was very simple and optimal, but the loop execution time (around 25ms compared to our 16ms self-balancing code) was enough to cause our system to exit stability. A future improvement would be to adopt 2 C-based microcontrollers, or to implement the Raspberry Pi code in C++. Both solutions are out of our budget and scope.

3- Using a magnetometer to improve our IMU performance

Our project depends heavily on the IMU as it is necessary to have very good calibration and sensor performance to get good results. We believe that adding a magnetometer by substituting our MPU6050 with an MPU9250, for instance, might yield better results in for our system.

# **References**

Modeling of two-wheeled self-balancing robot driven by DC Gearmotors. (n.d.).

<https://www.researchgate.net/publication/319618827_Modeling_of_Two-Wheeled_Self-Balancing_Robot_Driven_by_DC_Gearmotors>

Design and implementation of two-wheeled self-balancing vehicle using ... (n.d.-b). <https://www.researchgate.net/publication/300467701_Design_and_Implementation_of_Two-Wheeled_Self-Balancing_Vehicle_Using_Accelerometer_and_Fuzzy_Logic>

Two-wheel self-balanced electric vehicle based on fuzzy PD controller ... (n.d.-d). <https://www.researchgate.net/publication/329489132_Two-wheel_self-balanced_electric_vehicle_based_on_fuzzy_PD_controller_and_kalman_filter>

Design and control of a two-wheel self-balancing robot using the ... (n.d.-b). <https://www.researchgate.net/publication/261126156_Design_and_control_of_a_two-wheel_self-balancing_robot_using_the_arduino_microcontroller_board>

An adaptive sliding mode control system and its application to real-time hybrid simulation. . (n.d.-b). <https://www.semanticscholar.org/paper/An-adaptive-sliding-mode-control-system-and-its-to-Li-Maghareh/d7b15ef587eeb0a990988c1c4ec883ab44e7168a>

GM25-370CA - Global Sources Magazines. (n.d.-f). <https://p.globalsources.com/IMAGES/PDT/SPEC/301/K1123392301.pdf>

Two-axis reaction wheel inverted pendulum - NTNU. (n.d.-j). <https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2451060/12762_FULLTEXT.pdf?sequence=1>

<https://youtu.be/n63ZmNSlSA4>