Biorobotics and Locomotion Laboratory

Andy's commented version of:

Towards a Robotically-stabilized Bicycle

Authors: Arundathi Sharma (ams692, MAE senior), Olav Imsdahl (oai2, MAE senior), Will Murphy (wsm64, MAE junior), Scott Bollt (sab453, MAE freshman), Frances Bryson (feb45, MAE senior), Kenneth Fang (kwf37, undecided freshman), Dylan Meehan (dem292, MAE freshman), Pehuen Moure (ppm44, CS junior), Kyle Fenske (kaf222, ORIE sophomore), Michelle O'Brien (mo345, CS/undecided sophomore), Hayley Sopko (hjs225, MAE senior), Haoyun Xu (hx39, MAE junior)

Supervisor: Professor Andy Ruina

Cornell University

Final Semester Report
Fall 2016

Abstract

The Autonomous Bicycle Team is developing a robotically-stabilized bicycle. Others have tried using a variety of balance strategies, including gyroscopes and reaction wheels; our bike will use only steering for balance, much like a human does. We control steering angle rate as a linear function of the instantaneous steer angle, the bike lean angle, and the bike falling rate. Ultimately, we intend to demonstrate the bike riding around Cornell campus on its own. To that end, our achievements this semester were the following: finally successfully assembling the working parts

***

Andy says:

All working parts? I saw the bike coasting, not using its drive motor, nor kick stand, nor cushions.

***

(sensors, code, mechanical/electrical hardware, etc.) to produce a working prototype, and actually seeing the bicycle balance itself, using the balance controller, when pushed at high speed. This report will review all the steps that went into making this happen. The dynamics team also made headway

in developing a new controller to keep the bicycle upright in a track-stand using a forward-velocity controller, while the rear-motor team has started working to better understand how we can control forward velocity enough to successfully track-stand.

***

Andy says:

meanwhile, did it ever go forwards at near constant speed under control?

***

The front-motor team developed a controller that would allow us to effectively control front-motor velocity. Finally, some of our newer members on the project team were given side-projects to help them build the skills necessary to lead the team in the future, and we added a business arm to take care of various Project Team administrative responsibilities.

Contents

1

Dynamics and Simulation (Frances Bryson)

***

Andy says:

Andy's comments look like this.

***

1.1 Comparison of Models

Research done in previous semesters has yielded equations of motion for the linear and non-linear point mass models of the bicycle.

***

Andy says:

References needed.

***

Manipulating these equations, we can find a state space model with one control input.

***

Andy says:

Not clear. What state space? What input?

***

However, modeling a physical bicycle as a point mass is prone to error.

***

Andy says:

Actually, I think it is good. Rather, you can use the Whipple model to check if the point mass model is good.

***

To better estimate the accuracy of our equations of motion, and our resulting controllers, we developed a simulation using the Whipple model of a bicycle.

***

Andy says:

Reference?

***

The Whipple model takes into account the moments of inertia of four main

parts of the bicycle; the handlebars, front wheel, rear wheel, and frame.
***

Andy says:

Why only comment on the moment of inertia? What about head angle, trail and locstions of three more centers of mass?
***

1.1.1 Equations of motion

***

Andy says:

The text that follows seems to largely duplicate the text above.
***

Research done in previous semesters has yielded linear and non-linear point mass models of the bicycle. Manipulating these equations, we can find a state space model with one control input.

However, modeling a physical bicycle as a point mass is prone to error. To better estimate the accuracy of our equations of motion, and our resulting controllers, I developed a second simulation using the Whipple model of a bicycle. The Whipple model takes into account the moments of inertia of four main parts of the bicycle; the handlebars, front wheel, rear wheel, and frame. By comparing the simulations of the recovery of the bicycle from a given initial lean angle, I found that the recovery of the Whipple model closely matches that of the point mass model, and conclude that the point mass model is an acceptable model to use for our simulations and controller

development.

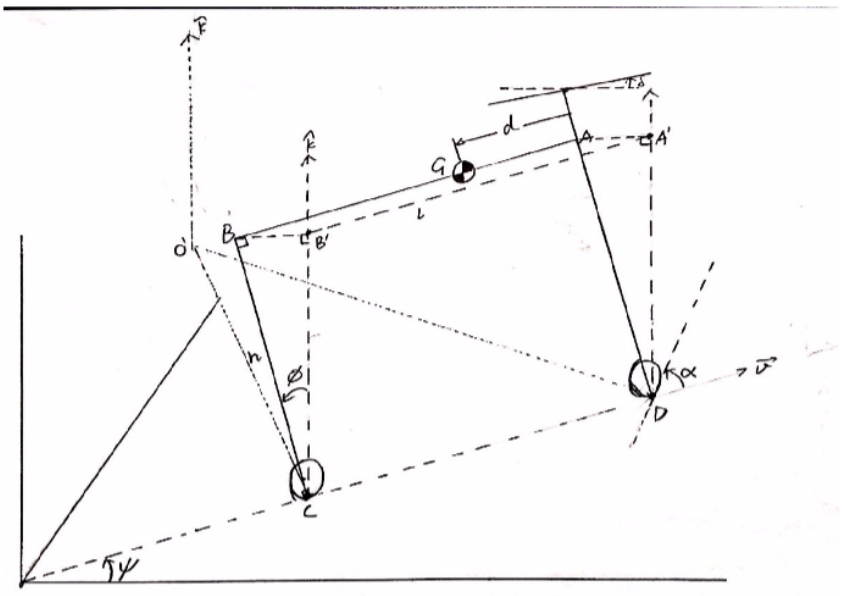

Figure 1.1: Point mass bicycle model.

***

Andy says:

Need more extensive caption. Figure needs gaps in lines that are behind other lines. Figure is original, or from where?

***

The variables in the non-linear and linear equations are defined as:

***

Andy says:

Sign convention needs to be clear for all variables.

***

ϕ = lean angle (rad)

$\dot{\phi}$ = lean angular rate [rad/s]

δ = steering angle [rad]
$\dot{\delta}$ = steering angular rate [rad/s]
v = velocity [m/s]
b = distance from ground contact point of rear wheel (C) to center of mass (G) projected onto ground [m]
h = height of the bicycle center of mass [m]
l = distance between front and rear wheel ground contact point [m]

The full non-linear equation for the point mass model of a bicycle is:

$$\ddot{\phi} = \frac{g}{h}\sin(\phi) - \frac{v^2}{hl}\tan(\delta) - \frac{bv\dot{\delta}}{hl\cos^2(\delta)} - \frac{b\dot{v}}{hl}\tan(\delta) + \frac{v^2}{l^2}\tan^2(\delta)\tan(\phi) - \frac{bv\dot{\phi}}{hl}\tan(\delta)\tan(\phi) \tag{1.1}$$

This equation can be linearized by assuming that the lean angle is small, such that $\sin(\phi) = \phi$. Similarly, we assume that the steer angle is small such that $\cos(\delta) = 1$ and $\sin(\delta) = \delta$:

$$\ddot{\phi} = \frac{g}{h}\phi - \frac{v^2}{hl}\delta - \frac{bv}{hl}\dot{\delta} \tag{1.2}$$

The Whipple Model

Relying on Jay Jiang's paper ´Dynamics and Control of a Self-stabilizing Bicycle" from 2014, I found the equations of motion for the Whipple model, which models the bicycle as four parts: the handlebars, front wheel, rear wheel, and frame.

The linear, simplified equation is:

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\delta} \end{bmatrix} = M^{-1}(-C\begin{bmatrix} \dot{\phi} \\ \dot{\delta} \end{bmatrix} - K\begin{bmatrix} \phi \\ \delta \end{bmatrix} + \begin{bmatrix} T_\phi \\ T_\delta \end{bmatrix}) \tag{1.3}$$

These matrices, M, C, K, T are referred to as the mass matrix, damping matrix, stiffness matrix, and torque matrix. The mass, stiffness, and damping matrices are found from the moments of inertia, centers of mass, and

other physical parameters of the four parts of the bicycle.

The equation of motion can be manipulated to find a controller, which will provide the steering angular velocity; the same as the controllers previously developed for the point mass model. The following form of the equation is written for ease of use in code:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\delta} \\ \ddot{\phi} \end{bmatrix} = A \begin{bmatrix} \phi \\ \delta \\ \dot{\phi} \end{bmatrix} + B\dot{\delta} \tag{1.4}$$

in which the matrices A and B are found from the mass, stiffness, and damping matrices in the above equation of motion, and the vector $\begin{bmatrix} \dot{\delta} \end{bmatrix}$ is the rate of change of the steer angle found by a controller, such that:

$$\dot{\delta} = \begin{bmatrix} k_1 & k_2 & k_3 \end{bmatrix} \begin{bmatrix} \phi \\ \delta \\ \dot{\phi} \end{bmatrix} \tag{1.5}$$

1.1.2 Simulation

Using previously developed code for simulating the balance and navigation control of the bicycle, with navigation turned off so that the bicycle was only using balance control (with a controller of $k_1 = 71, k_2 = 21, k_3 = -20$) stabilizing itself from an initial lean angle of about 40°, I compared the Whipple and point mass models of the bicycle to find how well the point mass model of the bicycle describes the motion of the bicycle.

The recovery of the two models is very similar; both follow the same shape

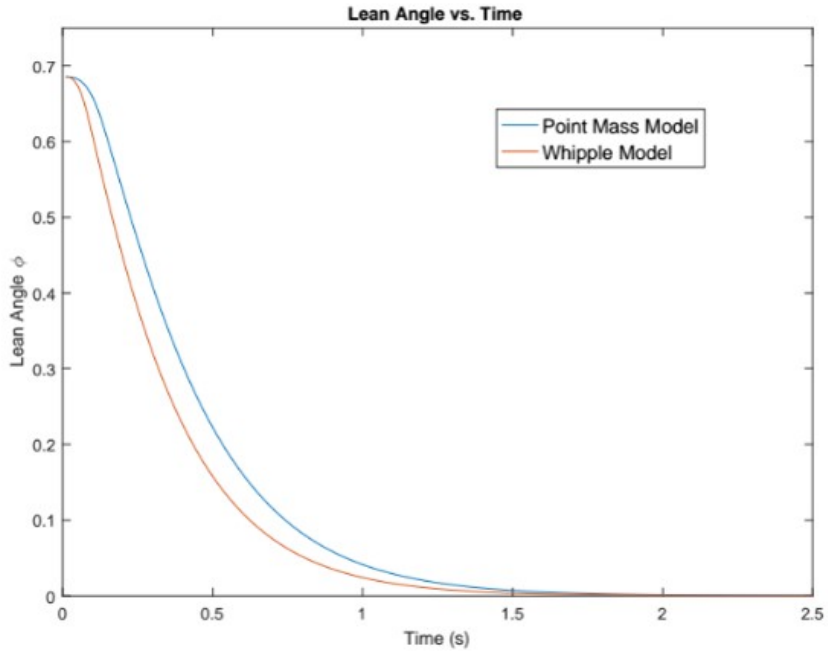

Figure 1.2: Comparison of Whipple and Point-mass Models

of plot and stabilize (lean angle goes to less than 1 degree) in a similar amount of time. The nonlinear EOM causes larger oscillations of the lean angle than the linearized equations for the same gains, but for small lean angles we can assume that the linearized equations of motion correctly describe the motion of the bicycle.

1.2 Track-standing Bicycle

A new project for the dynamics subteam was to simulate and find controllers that would allow for the bicycle to trackstand; i.e. stand upright in one place for an extended amount of time. This is made possible by maintaining a constant steer angle ($\delta = \frac{\pi}{3}$) and controlling the velocity of the bicycle,

as a trackstand is obtained by moving forwards and backwards very small amounts to change the lean angle.

1.2.1 Equations of Motion

For the model of a trackstanding bicycle, the non-linear equations can be linearizing by assuming that the velocity is very small, and thus all terms of higher order v can be assumed as approximately zero, and neglected. In addition, since the steer angle does not change when the bicycle is trackstanding, any terms dependent on $\dot{\delta}$ are equal to 0. Thus:

$$\frac{v^2}{hl}\tan(\phi) = 0; \frac{v^2}{l^2}\tan^2(\delta)\tan(\phi) = 0; \frac{bv}{hl\cos^2(\delta)}\dot{\delta} = 0$$

And the linearized point mass model for a trackstanding bike is:

$$\ddot{\phi} = \frac{g}{h}\sin(\phi) - \frac{b\dot{v}}{hl}\tan(\delta) - \frac{bv\dot{\phi}}{hl}\tan(\delta)\tan(\phi) \tag{1.6}$$

Assuming that the lean angle is small, we can approximate $\sin(\phi) = \phi$:

$$\ddot{\phi} = \frac{g}{h}\phi - \frac{b}{hl\cos^2(\delta)}v - \frac{b\tan(\delta)}{hl}\dot{v} \tag{1.7}$$

We can use these equations of motion to help us develop an optimal controller to track-standd a bicycle.

1.2.2 Simulation

To compare the non-linear and linear models of the trackstanding bicycle, in this case relying only on the point mass models, I simulated the trackstanding bicycle for the same controller (using a simple velocity controller $[k_1 k_2 k_3]$ = [-5 30 5] such that $\dot{v} = k_1\phi + k_2\dot{\phi} + k_3 v$) and compared the lean v. time plots:

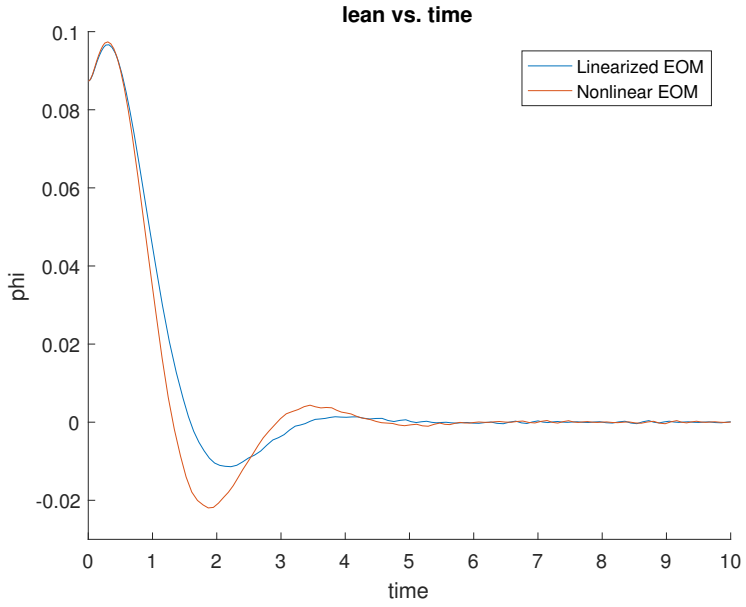

Figure 1.3: Comparison of the stabilization of the linearized and nonlinear EOMs

For small angles - $\phi < .1$ - the models agree; however, as the lean angle increases, the models begin to diverge. This is directly related to an assumption made when deriving the models; a small lean angle reduces $\sin\phi$ to ϕ. I conclude that for small lean angles, the dynamics of the trackstanding bicycle can be described by the linearized model.

1.2.3 Controller Development

For a trackstanding bicycle, the controlled variable is the acceleration of the bicycle, $\dot{v}$, which is a function of the lean angle, angular rate of change of the lean angle, and the instantaneous velocity. In this case $\dot{v} = k_1\phi + k_2\dot{\phi} + k_3 v$. The linearized equation of motion becomes (written in state-space form):

$$\begin{bmatrix} \dot{\phi} \\ \ddot{\phi} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ \frac{g}{l} & 0 & -\frac{b}{hl\cos(\delta)} \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \phi \\ \dot{\phi} \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{b\tan(\delta)}{hl} \\ 1 \end{bmatrix} \begin{bmatrix} k_1 & k_2 & k_3 \end{bmatrix} \begin{bmatrix} \phi \\ \dot{\phi} \\ v \end{bmatrix} \tag{1.8}$$

Similarly, the equation for $\dot{v}$ can be substituted into the non-linear equation of motion and integrated in MATLAB.

The controllers developed for the trackstanding bicycle were subjected to two constraints; a maximization of time before the bike falls - i.e. the lean angle increases beyond $\frac{\pi}{4}$- and a minimization of distanced travelled from the origin (to make the trackstand more friendly to physical testing). After searching through a large search space for each controller, testing each for the above constraints and scoring them based on how long the controller allowed the bicycle to remain balanced versus the distance the bike traveled from its initial position, I found the top controller found was $[k_1\ k_2\ k_3]$ = [-10 34.5 5]:

Figure 1.4 shows a comparison of the top five controllers, where the score is determined by subtracting the maximum distance the bike travels from the origin from the total time the bike remains upright in a simulation of 100 seconds.

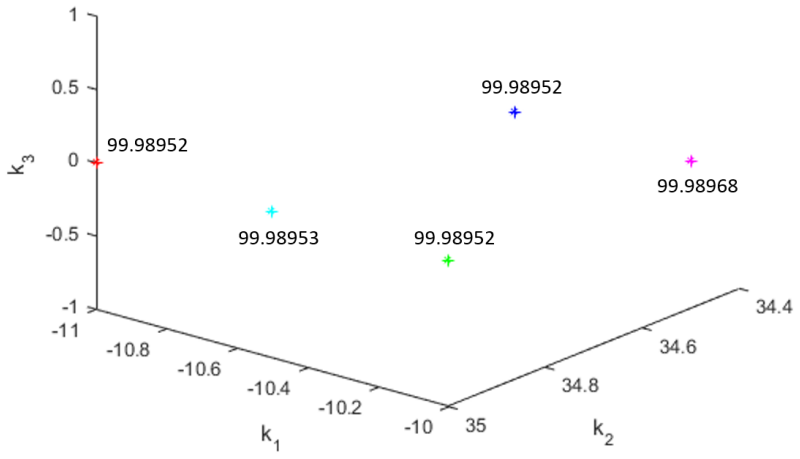

Figure 1.4: Comparison of the scores of the top five controllers

The plots in Figures 1.5,1.6,1.7 were run on the nonlinear simulation with the controller $[k_1\ k_2\ k_3]$ = [-10 34.5 5], for an initial lean of 5 degrees and constant steer angle of $\frac{\pi}{3}$. In Figure 1.5 the bike is seen to stabilize within five seconds, and with only two oscillations. The motor commands in Figure 1.6 are the acceleration $\dot{v} = k_1\phi + k_2\dot{\phi} + k_4 v$. In Figure 1.7 we see that the trackstanding bike moves from its initial position (to correct the initial lean angle) less than a meter from its initial position, with few oscillations. This is feasible motion for testing on our physical bicycle.

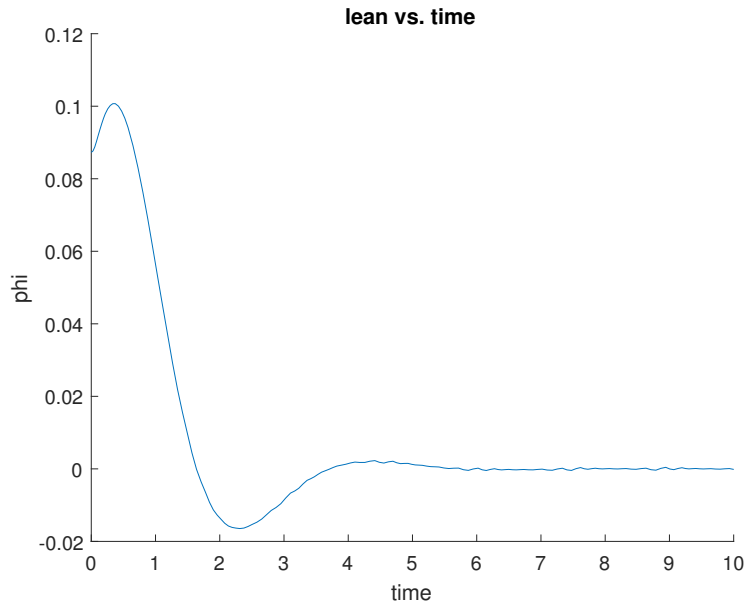

Figure 1.5: Lean angle ϕ v. time for controller [-10 34.5 5] of a trackstanding bicycle

1.3 Conclusion

When compared with the more comprehesive Whipple model of the bicycle, the point mass model is a realistic representation of the dynamics of a bicycle. In addition, the controllers for the trackstanding bicycle have been optimized for ease of physical testing and how long the controller allows the bicycle to stabilize. The next step would be to translate this to the physical bicycle, which includes determining how to run the rear wheel motor at a slow enough speed to allow for trackstanding the bicycle

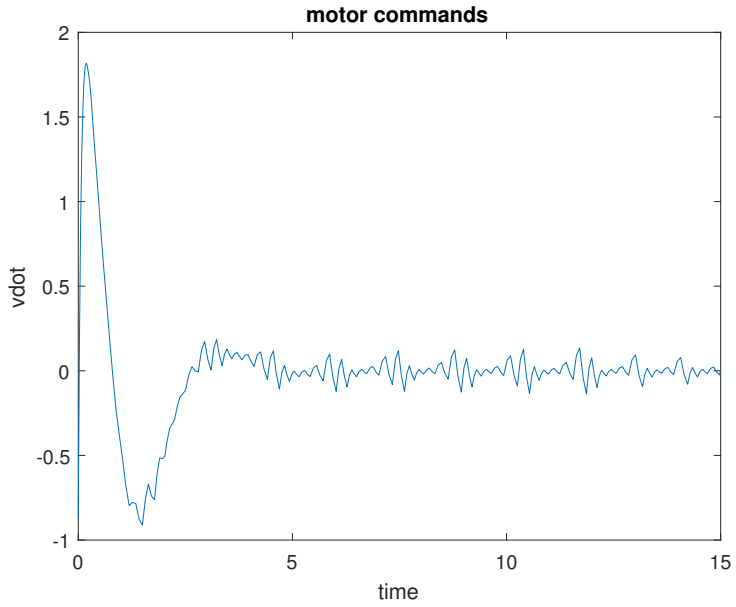

Figure 1.6: Time history of motor commands for controller [-10 34.5 5] of a trackstanding bicycle

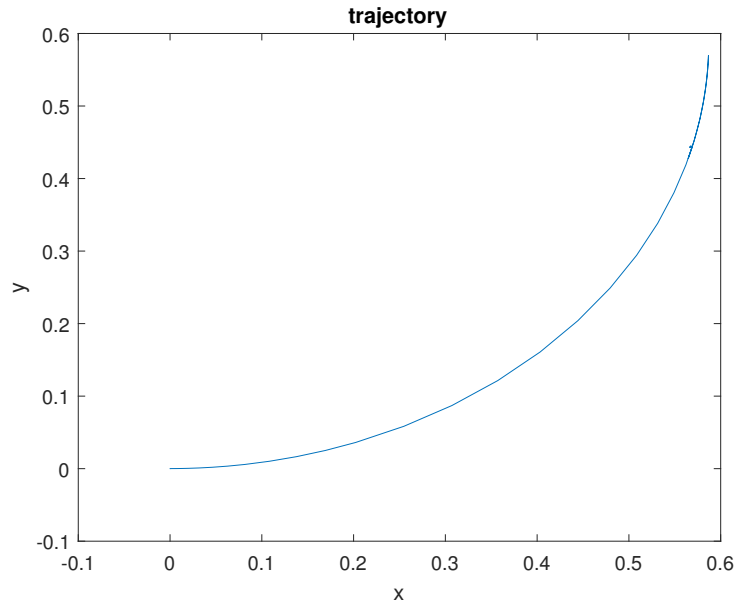

Figure 1.7: Trajectory for controller [-10 34.5 5] of a trackstanding bicycle

2

Miniature Bicycle (Scott Bollt & Olav Imsdahl)

2.1 Overview

The goal of this sub team is to create a miniature bicycle that is much more simplified compared to its larger counterpart in the hope that it will allow for quicker preliminary testing of new control schemes and other novel ideas. The bike is also small enough that it could be used as a promotional and learning tool for new members.

2.2 Mechanical Design

The mechanical design of the bike started with the size constraint of being a couple inches tall. This translated into a wheel size since it was decided that the wheel size should roughly fit the proportions of a real bike. The wheels were designed to have O-Ring tyres and thus have a notched rim.

Figure 2.1: Detail of bicycle wheel rim for O-ring

The bearings for the wheels were chosen purely based on the abundance of a specific type of bearing in the lab. The design used to have two bearings per wheel, but that over constrained the motion of the wheels so high resistance was encountered since two bearings will only work well if they are aligned perfectly. As a result, the flexible and inadequate two sided front fork was phased out by a far stiffer one sided front fork. The design of the frame came

next. It is designed to be simplistic in shape, but to give the bike a familiar appearance in terms of its dimensions. With this in mind, the frame was made such that the edges and mounting locations were all horizontal to the ground with the exception of the servo mount. Aesthetics and standard bike design were also why the front servo is angled at 15 degrees. The decision to add a forward sweep was also based on stability, but the exact number for the forward sweep was mostly an aesthetic choice. The frame also includes a holder for a continuous servo which drives the rear motor. It also has many mounting holes for possible future additions to the bike. The servo attaches directly to the fork using the servo-arm as the anchor point. The servo was originally a high torque servo with a speed of 0.20s/60 degrees, however it was replaced by a far faster and smaller servo with a speed of 0.05s/60 degrees. Despite the great reduction in size and the great increase in speed, the servo is capable of a third of the torque of the much larger and slower servo. The switch to the faster servo was based on recognition of the fact that the bike will fall over in just a little over 0.20 seconds, and although the high torque servo can move shorter distances in less than 0.20 seconds the servo appeared to be having trouble keeping up with the demands of the job. Recognition of this further facilitated the choice of the faster servo.

The rear wheel drive system uses a continuous servo and an O-Ring belt drive in order to transfer force into the ground through the wheel. The sizes of the front and back belt wheels are significantly different, with the back one being much larger in order to achieve a reasonable speed for the bike to maintain stability. If the large mechanical advantage were not present, it would have been difficult to achieve sufficient forward velocity. Both the front and rear belt wheels have a grove around their rim in order to hold the O-Ring, but the aluminium machined rear belt wheel also must attach to the rear wheel in such a way that it drives it without interfering with the bearing. To do this a press fit against the inner surface of the rear bearing

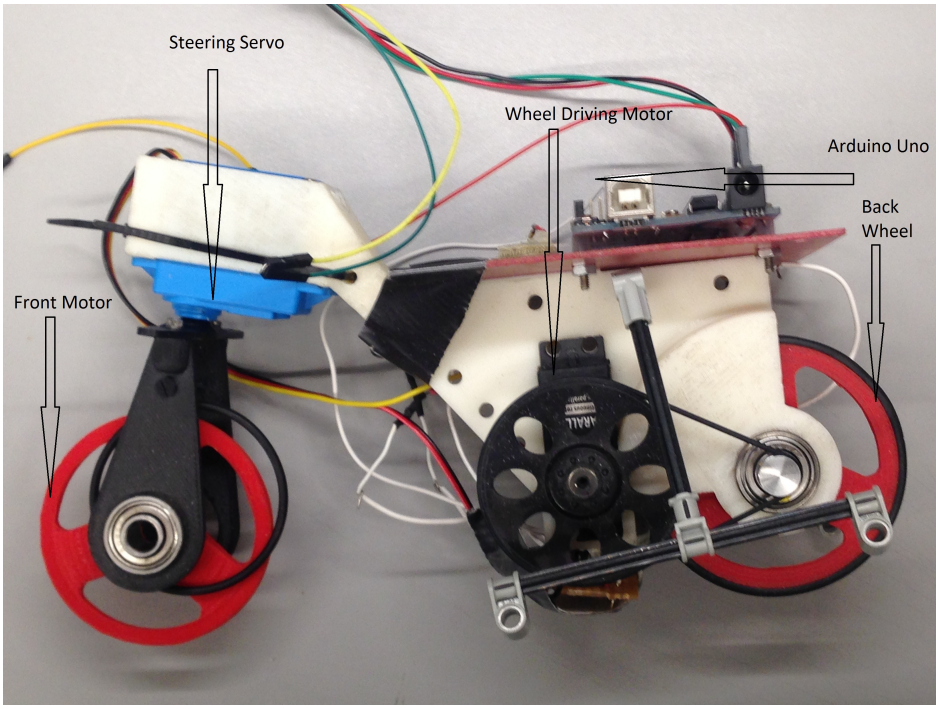

Figure 2.2: Overview of bicycle

was used.

2.3 Electrical Design

The electrical design has three very simple parts: control, power, and sensing. Control comes from an Arduino Uno. It was chosen since the processing requirements are not great. It also comes with many libraries that allow for easy communication with all of the electrical components. The power comes from a 9v battery. A 9v battery is small, light, and is within the 7v-12v range that the Arduino requires. Power for the other components is supplied through the Arduino, thus eliminating the need for regulating components. The sensing originally came from a two axis accelerometer, but now comes

from a 9 axis IMU. They both communicate directly with the Arduino. In the future it would be good to use a small rechargeable 11.1v battery since the 9v battery dies quickly.

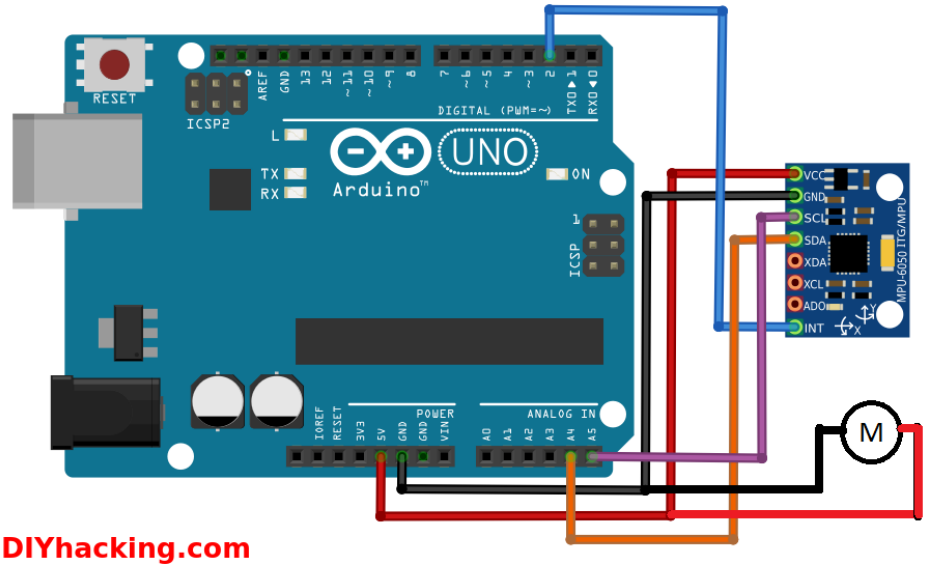

Figure 2.3: Bike Wiring Diagram (From DIYhacking.com)

2.4 The Control Scheme

The basic control scheme is to continually set the front servo steer rate proportional to the lean rate of the bike. If that does not work, other more intricate control methods can be tried. Either way, this is simple to code. The real difficulty lies in obtaining lean rate data quickly and accurately. For this reason the two axis accelerometer was insufficient for our purposes. The reason is that when the bike is undergoing angular acceleration, the accelerometer cannot separate the wanted component due to gravity and the unwanted component due to the bike's angular acceleration. The problem

further compounds when the bike starts to turn. Thus it was necessary to move to a small IMU. The IMU can measure lean and lean rate accurately because it has a 3 axis magnetometer which is inaccurate over short intervals but accurate over great lengths of time, a 3 axis accelerometer which is accurate over medium to long periods but not when the bike is in a transient state, and finally a 3 axis gyro which is accurate over short periods of time, but has drift issues when calculating lean angle over any great length of time. As you can see, these sensors can make up for the others' weaknesses in order to create a robust lean and lean rate sensor.

2.5 Conclusions and Future Work

While the bike does not currently self balance, there is every indication that the new design will when everything is put together. Next semester it would be good to try and get the bike working, to increase the bike's presentably to the public, replace the non-rechargeable 9v battery with a rechargeable battery, and get the bike to do tricks like move in a circle, do a figure eight, and possibly track-stand.

3

Hardware and Wiring (Olav Imsdahl and Dylan Meehan)

3.1 Organization of Electrical Components

Dylan and Olav worked on organizing the components in the main box. Previously, various components were simply randomly thrown into the box. This created a concern that two wires or components could accidentally touch creating a short circuit (possibly on the Printed Circuit Board). This could cause the Printed Circuit Board (PCB) to fail. Olav installed wooden rails along the side of the box and attached wooden supports to the PCB so that it

could rest on the wooden supports and sit above the box. (See Picture below.) (Note: Page 40 of Spring 2016 report shows what each of the connections on the PCB corresponds to.)

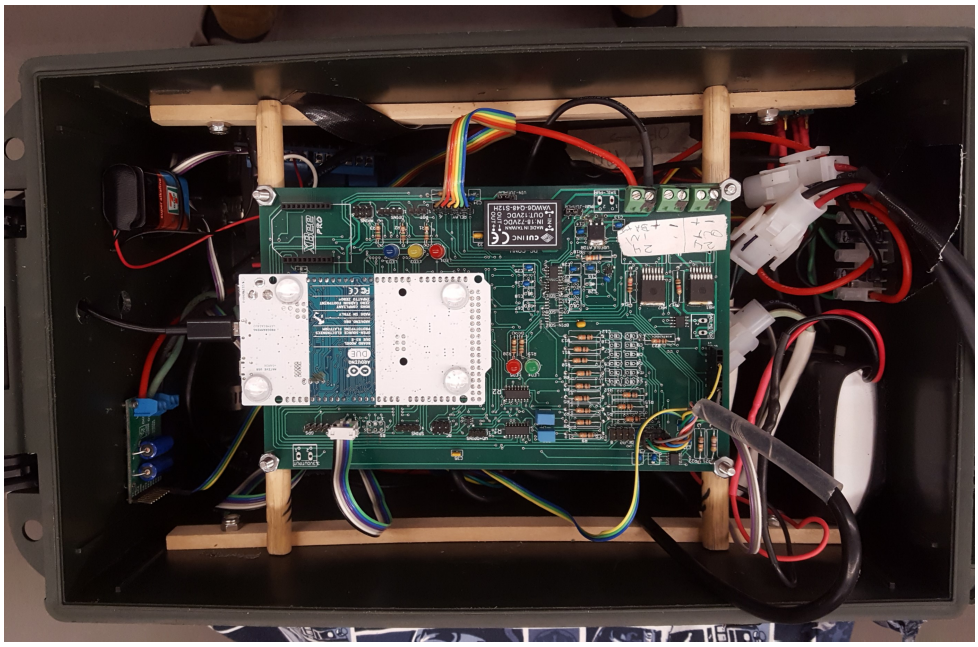

Figure 3.1: Printed Circuit Board on wooden rails in electronics box

This ensures that the PCB does not accidentally come into contact with any other components. This also places the PCB near the top of the box so that it can be easily accessed for programming.

Other pieces of hardware were also loosely thrown into the box. We screwed and zip-tied other large pieces of hardware (such as the rear motor controller and relay switch) to the bottom and sides of the box to keep them secure. See Picture below.

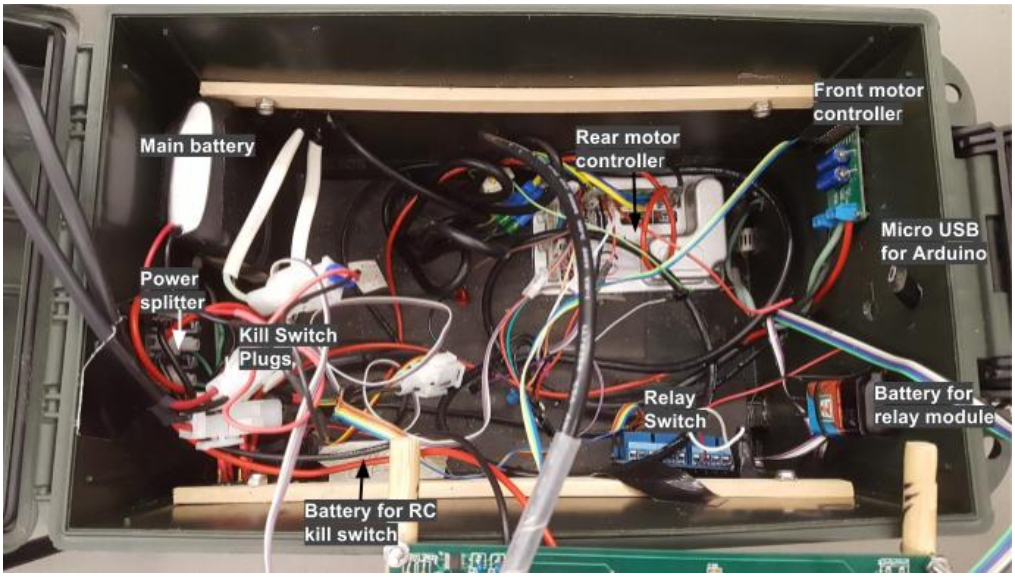

Figure 3.2: Electronics Components in Box

We attached switches to the side of the box to control the power to the front and rear motors. See picture below. The power from the battery is split three ways to give power to the front motor, rear motor, and PCB. Before we installed these switches, we could only turn on and off the main power supply which would turn on and off the front and rear motors as well as the Arduino. This created a problem because if someone wanted to test the front motor they would have to turn on the full power which would cause the rear motor to spin as well. The only way to prevent this previously was to disconnect the circuit in the box which was burdensome. Now, the individual switches allow us to control the front and rear motors separately. There is no separate switch for the Arduino because if someone wants to test code for the front or rear motors they would also presumably need the Arduino to be on. To turn everything off there is still the master kill switches.

Figure 3.3: Drawing of Switches on Outside of Box

3.2 Safety Features

There are three safety switches that form a chain that connect the battery to the splitter (which divides the voltage between the Arduino, front motor, and rear motor. There is a physical wired switch, a emergency shutoff button, and a remote controlled switch. The switches are connected in a chain so that tripping any of the switches will kill the power to both motors and the Arduino. Each switch is connected with a plastic clip so they could be connected or removed independently of the others. For example, during later testing we may not want a wired switch so we could disconnect that switch and leave the other ones. (See diagram and picture of switches).

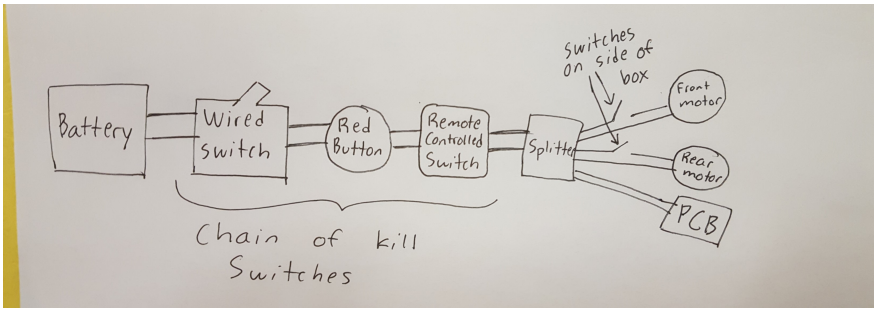

Figure 3.4: Drawing showing chain of shutoff switches

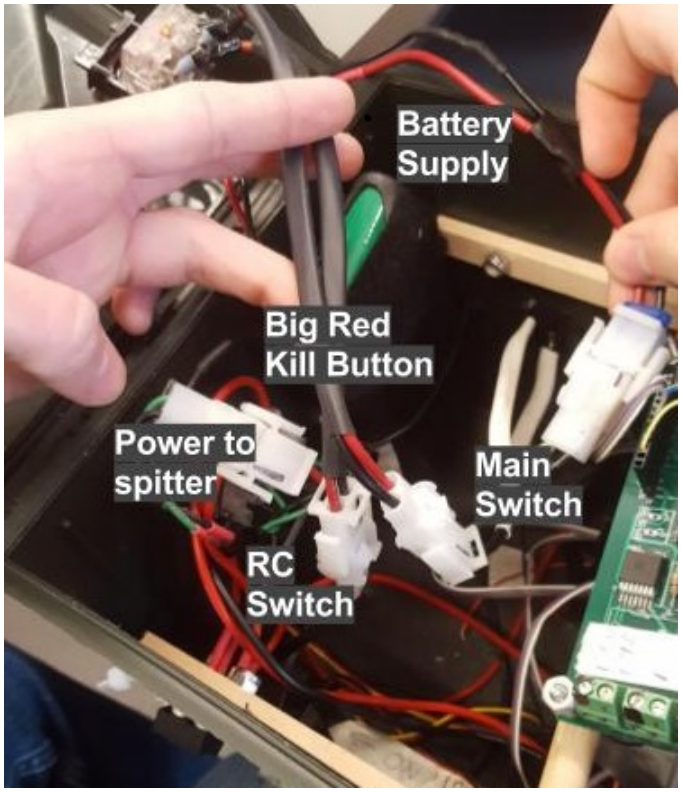

Figure 3.5: Picture Showing Clip Plugs for Shuttoff Switches

All three switches need to be in their ON positions so that the bike will be powered. The wired switch and RC switch both have ON/OFF written

on the face of the switch (See figure 3.6). To turn the red button ON, twist it so that it pops into the up position (See figure 3.3).

When making one of the clip connectors to connect plugs together it is important to check the orientation of the of the existing plugs and replicate this on the new plug. In other words, double check to ensure that the power and ground of the new plug match that of the old plug. The connectors cannot be flipped over so it is important the correct orientation before assembling the clip.

We made the wired kill switch longer so that people can more easily hold the switch while walking with the bike. This is the switch that is primarily used to control power to the bike during testing. We put a large, red button on top of the box. (See figure 3.2) This button can be used as an emergency stop whenever we are testing the bike. Slamming the button kills power to the bike. Twisting the button releases it and allows power to flow.

We installed an remote control switch so that the bike can be shut off from a distance. (See figure 3.2 regarding the location of the RC switch). The RC controller we use is the Tactic TTX610. To control the kill switch we use Channel 5 on the receiver which corresponds to a binary switch on the remote. The switch on the remote is marked with masking tape and is on the top left of the remote. This channel is designed to control a servo motor so toggling the switch changes the output signal between about .20 and .35 volts. This is a relatively specific voltage range. We use this signal to control a servo, as it is intended to do, and then use that servo to hit a physical switch which kills the power to the bike. (See Figure 3.6 below of servo and switch). We considered using the receiver signal to switch a relay, but we did not have a relay that switched within this range. We considered using one of the relays we did have, but this would require modifying the signal into something the relay would accept. This would be difficult and complicated

which is why we decided to use the servo instead.

Figure 3.6: Picture RC Switch and Wired Swtich

The servo kill switch does not allow us to turn ON the bike with the RC controller. After turning off the bike with the remote control, the physical switch on the bike must be set to ON so that the bike will start. The RC receiver currently runs on 4 AA batteries (ie, seperate from the main bike battery), so it may be possible for the RC transmitter to die without the bike dying which is why the RC switch should be checked before doing testing and the batteries replaced periodically. The 4 AA batteries are placed in the main box of the bike. We tested the range of the RC transmitter and believe that it is large enough so that the bike should not go out of range, however we need to be vigilant not to let the bike get excessively far from the transmitter.

3.3 Mount for IMU

The IMU is mounted to the bike frame below the seat of the bike. See Figure 3.7 below.

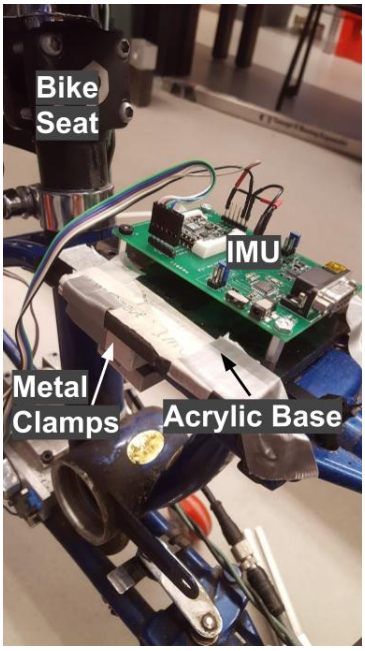

Figure 3.7: Labled Picture of IMU attached to bike. The IMU is attached to the Acrylic Base by 4 standoffs. The IMU must be placed on the standoffs while the standoffs are not tight to the arcylic. When the standoffs are tightened, they push slightly against the board for the IMU. This means that the IMU cannot be removed while the standoffs are tight. To remove the IMU, first remove the nuts on each standoff and then loosen each standoff. The acrylic base attaches to a metal bar with two he screws. The acrylic is hard enough so that it does not bend about the line of these screws. There are two metal bars that clamp around the circular tubes of the bike frame. These clamps are screwed tight to each other. The acrylic base for the IMU screws into the top piece of metal. The IMU is relatively flat and aligned with the axis of the bike.

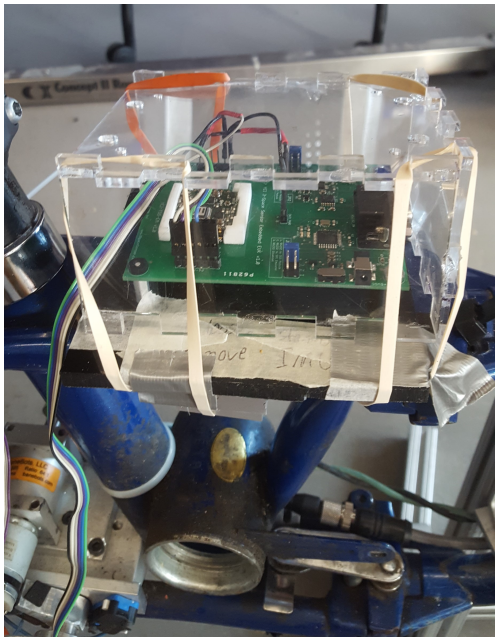

Figure 3.8: Picture IMU with plexiglass case around it. The case protects the IMU from people hitting it when we service the bike. Elastic bands are used to hold the plexiglass case to the mount.

The mount and case for the IMU does not provide any collision protection for the IMU. This is acceptable because the IMU should never hit the ground: if the bike falls the cushions attached to the box and front wheel should hit the ground - not the IMU. Also, the mount is designed to be rigid to the bike. We considered placing rubber between the bike and the IMU to lessen high frequency vibrations, however we decided against this idea. Due to problems with the IMU, we decided to have the IMU connected as rigidly as possible to the bike in order to try to get the best data about the bike frame.

3.3.1 IMU Troubleshooting

When we encountered IMU problems, one concern was that electromagnetic interferance from the power wires to the front motor could be causing the IMU to give poor data. To test this, we printed IMU values in the serial port both with the front motor off (no current in wires) and the front motor on (current in wires). We also tested moving the location of the wires from one side of the IMU to the other: if the wires were causing problems, switching which side of the IMU they were on might have switched which direction the error on the IMU. These tests showed no noticeable change in IMU data between having the wires on and off or changing their position. We concluded that the wires were not the cause of the IMU issue and furthermore it was not necessary to put shielding on the wires.

3.4 Landing Gear

To support the bike when it is not moving we have a set of training wheels called landing gear. (See page 43 of Spring 2016 Report for a detailed view of the landing gear hardware.) Note that for the landing gear to support the weight of the bike, it must be perpendicular to the ground. In other words, if the landing gear makes an angle of less than 90° if the bike starts to fall, it may push the landing gear UP allowing the bike to fall. This would be bad.

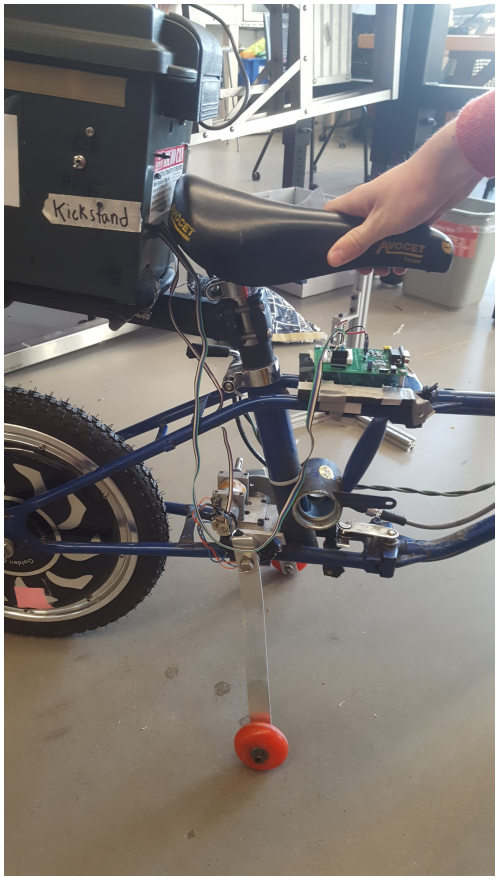

Figure 3.9: Picture of Landing Gear in Down Position

The landing gear on the bike is controlled by a relay switch which is controlled by the Arduino. The circuit designed by Olav is shown below.

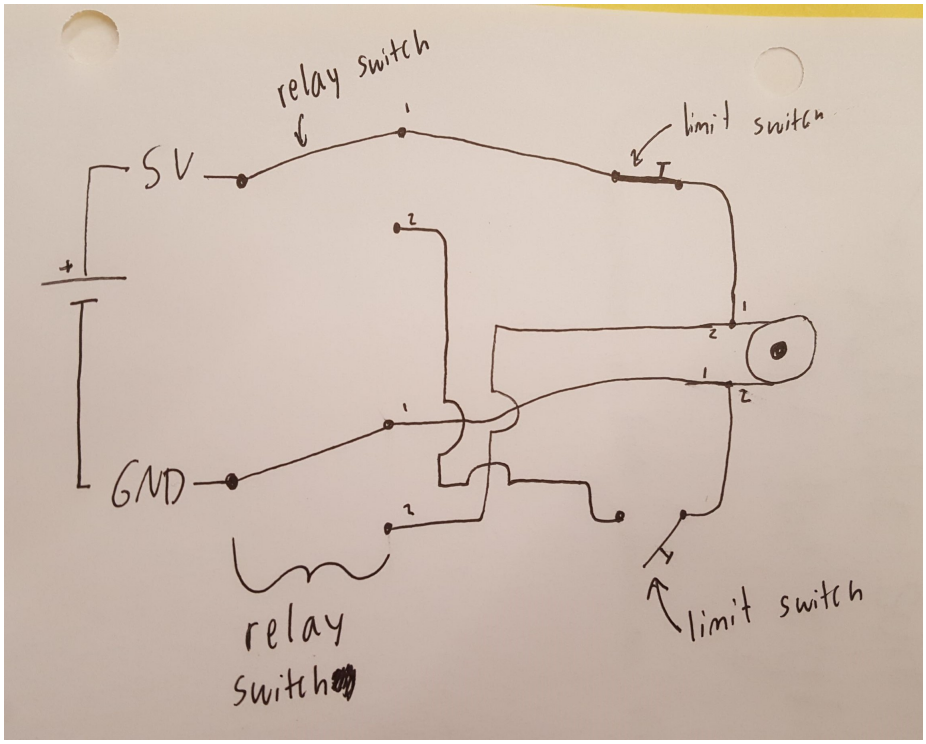

Figure 3.10: The Arduino controls two relay switches which can reverse the direction of power running through the circuit for the motor for the landing gear. This power is always running however, limit switches on each end of the motion physically break the circuit which serve to stop the landing gear when it gets to the end of its range of motion. We tested the circuit by hooking it up to a power supply and a spare relay (ie. one not controlled by the Arduino) which showed that the circuit worked.

Next we connected the circuit to the relay controlled by the Arduino. See figure 3.11 below.

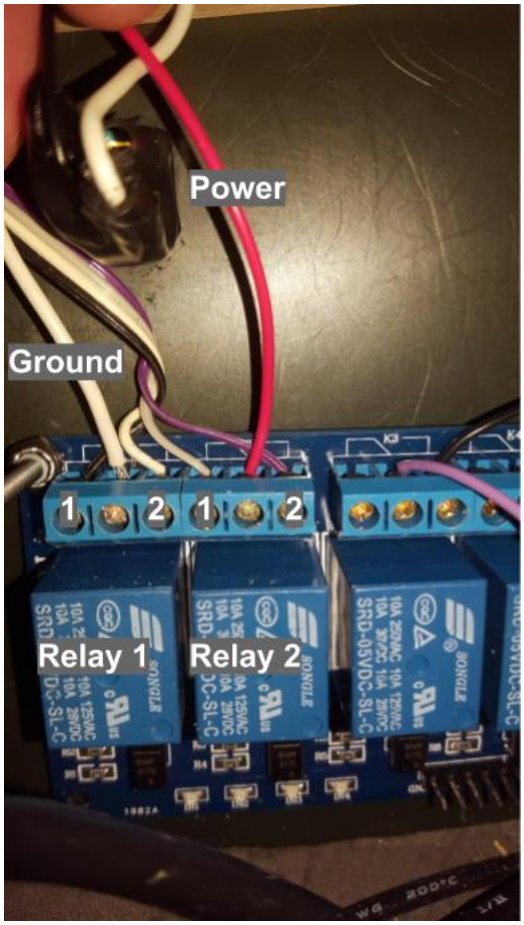

Figure 3.11: The power goes to the middle terminal of the relay 2 and the ground goes to the middle terminal of the relay 1. This configuration ensures that the proper limit switch is tripped while the landing gear moves in a certain direction (reversing the power and ground would change the direction that the motor would try to move the landing gear.) On each relay, the ports to the side of the middle port are the output terminals. The outputs get connected to the motor wiring.

Relays 1 and 2 correspond to pins 48 and 47 on the Arduino, respectively. The relay switch attaches to the PCB with a ribbon cable at the pins marked RELAY. The blue wire of the ribbon cable, which is the ground, goes to pin

1 on the PCB and the pin marked GND on the relay. After connecting each the wires, we checked the resistance in the wires which showed they were wired properly. Note: there are 4 wires to the motor which creates 2 circuits that the relay can switch between. When checking the resistance in the wire, 3 wires showed to be connected. This is because without either limit switch being pressed (ie when the landing gear was in the middle of its arc) 3 of the wires connected to their respective power/ground (both ground wires and one power wire connect to their relay switch). However, this still resulted in only one of the two circuits being closed which is expected.

To power the landing gear we use a separate 9V battery. This battery is placed in the box and secured with zip ties (See figure 3.2). The battery may die and need to be replaced periodically. Originally, we tried to use the 5V screw terminal on the PCB, however this source did not provide sufficient voltage to power the landing gear. While the source was rated for 5V it only supplied 0.3 volts when connected to the landing gear. We do not know why the 5V terminal does not supply 5V when connected to the landing gear, but regardless we could not use that 5V terminal.

We created two functions, *landingGearDown* and *landingGearUp*, which changes the position of the landing gear to down and up respectively. Due to the specific wiring of the circuit, HIGH outputs on the Arduino correspond the the UP position of the landing gear. When the landing gear is running in the main code, its default position (no signal from the Arduino) is down. Once the IMU is set to one state, it will stay in that state until it is commanded to go to a different position. Additionally, when the power to the Arduino is disconnected, the default position changes to UP. This has the effect of making the landing gear go up when code is uploaded to the Arduino. For this reason we installed a switch on the side of the box to control power to the landing gear so that the landing gear can be turned off when uploading code. Additionally, disconnecting the ribbon cable or not

initializing Arduino ports 47 and 48 to OUTPUT cause the default position to be UP.

One idea we considered was to make the landing gear go down in the battery to the bike is killed. By default the landing gear goes up when power is killed. We considered that having the landing gear go down if the power is killed would stop the bike from falling. However we now believe that the landing gear would not be fast enough to deploy while the bike is falling. Also, the landing gear is not designed to take the weight of a falling bike: if the landing gear is in the middle of deploying (for example being 45° from its starting position) while the bike falls, the weight of the bike would simply push the landing gear back to the UP position. Furthermore, we believe the landing gear is not needed to stop the bike from falling because the bike is designed to fall on its side and as such has airbags to act as cushions. If the bike takes a fall, the landing gear legs may bend. This may cause them to touch the ground before they reach the desired position or it may prevent both wheels from touching the ground at the same time. If the landing gear is bent it simply needs be bent back into its default position. (See figure 3.9)

3.5 Skill developing tasks for Dylan

3.5.1 Assemble/Disassemble Bike

In order to gain more experience about bike maintenance, I worked with Olav to assemble a bike from its various components. We connected the brake lines, attached gears and pedals, and attached the wheels. I also took apart the bearing in a wheel to see how that worked. I learned that much force (ie. more than I originally anticipated) was needed to loosen the nuts clamping the bearings. This exercise allowed me to feel how tight bearings

should be: they should be tight, but they should be loose enough to spin freely - ie. if you turn the wheel and feel bearing grinding, they are too tight. From this exercise, I learned the tools needed to service a bike, which was important in choosing what tools to include in my bike tool.

3.5.2 CAD Bike Tool

In order to practice CAD skills and learn what tools are needed to service a bike, I designed a bike tool in SolidWorks. Taking apart the bike and wheel helped me learn what tools, such as a 14mm crescent wrench, were important to include in the bike tool. The tool includes hex screwdrivers ranging from 2mm to 8 mm, a 14mm and a 15mm crescent wrenches, a phillips screwdriver, and a flat head screwdriver. A key to drawing each tool was to start with the side of the part that attached to the bolt and then design the tool off of that. This allowed me to ensure that each tool had a uniform way of connecting to the frame.

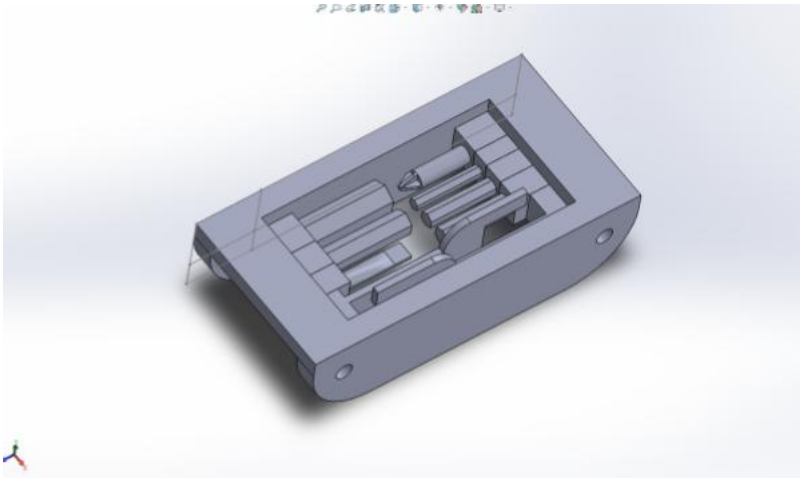

Figure 3.12: Bike Tool

4

Rear Motor and Electronics (Kenneth Fang, Michelle O'Bryan & Hayley Sopko)

4.1 Introduction

There are many applications of our autonomous bicycle that will depend on the speed of the bike. For example, one of the Autonomous Bicycle Team's first goals is to execute a track stand. Details of what a track stand are can be found in Chapter 1. A track stand requires the ability to accurately control

the speed, acceleration, and direction of the rear wheel at very low speeds. The purpose of our team is to accurately control the rear motor. To achieve this, first we need to find a way to measure the speed. If we can measure the speed, then we can characterize and control the motor in the future. In Fall 2016, the rear motor team focused on 2 goals: accurately measuring the speed and controlling the direction. After many failed attempts and some mistakes, the team can measure the speed with an accuracy of less than 1% error, and can change the direction the motor spins.

4.2 Background

4.2.1 Skill Developing Task: H-Bridge

We were assigned a side project of building an H-Bridge from components in the lab. This side project is designed to teach us about several common circuit components (such as transistors and diodes) as well as some important concepts used throughout the bike, such as Pulse Width Modulation.

An H-Bridge [3] is a commonly used circuit element composed of four switches and some load in the center. For our case, this load was a brushed DC motor that operated at 5-10V.

By controlling which switches are on and which ones are off, we can control the direction current flows through the load. For our case, this lets us switch the direction the motor spins. Figure 4.1 displays the general layout of an H-bridge, which is explained in the next paragraph:

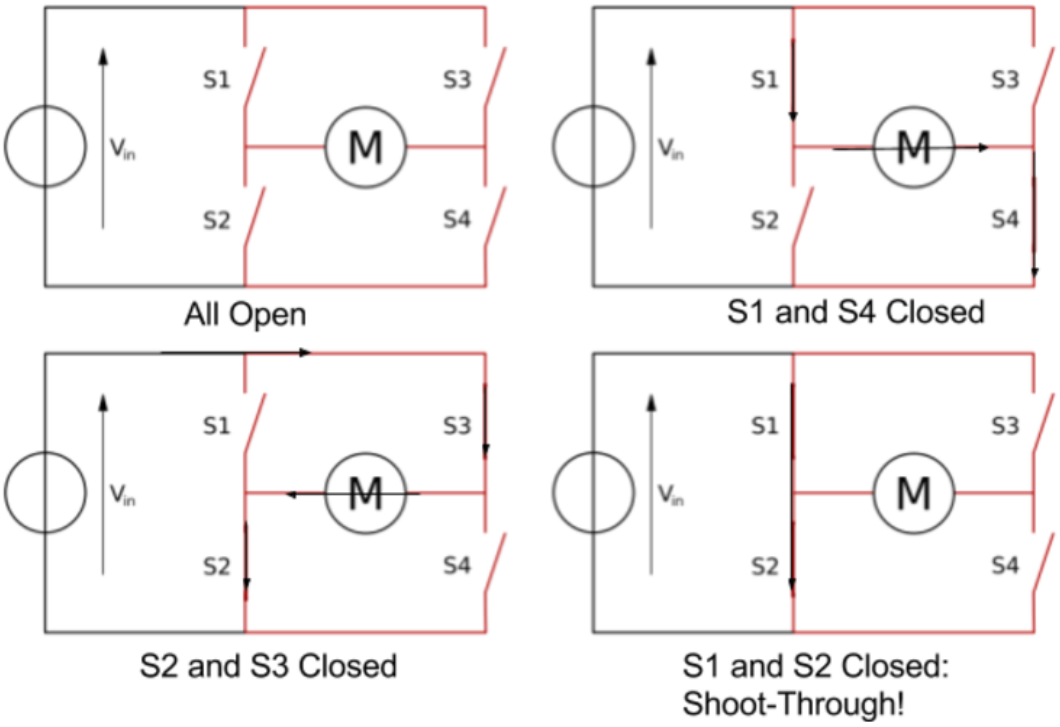

Figure 4.1: Four example states of an H-Bridge [6]

The top left of Figure 4.1 displays the H-Bridge with all switches off. This results in no power to the motor and an open circuit. The top right and bottom left show the two states in which the H-bridge can power the motor with arrows pointing in the direction of current flow. The direction of current flow determines the direction that the motor spins. Lastly, the bottom right diagram shows a very dangerous condition, called a shoot-through. In cases like this where current does not go through the motor, but is still able to flow, a short circuit is created that can quickly fry a battery supply. These four states show the H-bridge's most basic functionality.

Most conventional H-bridges are made using semiconductor switches, which allow high speed switching. This high speed switching is very useful when

combined with Pulse Width Modulation (PWM), described below. This combination enables an H-bridge to control a motor's speed in addition to its direction.

4.2.2 Pulse Width Modulation

Pulse width modulation is a method for simulating analog signals using digital output. Digital signals are defined on two discrete states, "on" and "off," or alternatively "high" and "low." Discrete means every value in a signal is clearly distinguishable from every other value. On the other hand, analog signals are continuous. This difference between discrete and analog signals can be seen in Figure 4.2, which displays an analog signal and a discrete approximation.

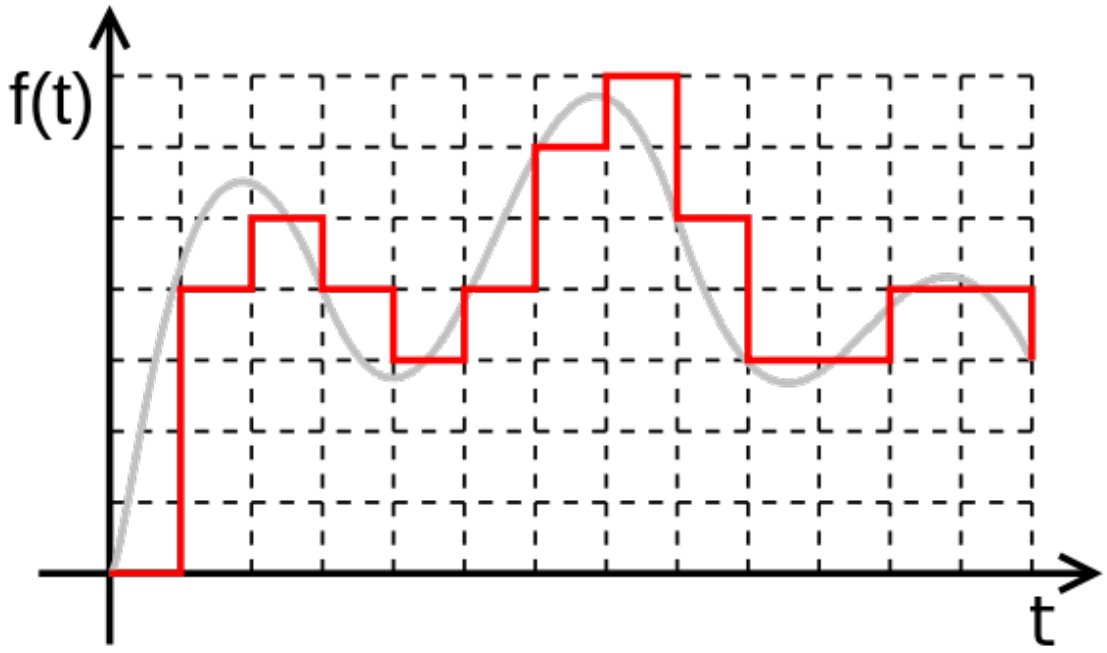

Figure 4.2: Analog vs. Discrete Signal [7]

When you want to use an analog technology, such as our DC motor, but control it with a digital signal, you cannot simply use a digital signal. Instead you must rapidly switch, or modulate, between the two states of your digital signal, so that the simulated signal coming from the digital device is the average of the time it is "on." The Arduino output is an example of this situation.

Arduino can output either 5v or 0v from its digital pins. If we wanted 2v, 3v, or any other voltage in between the maximum "on" state and the minimum "off" state, we use PWM to simulate it, as shown in Figure 4.3:

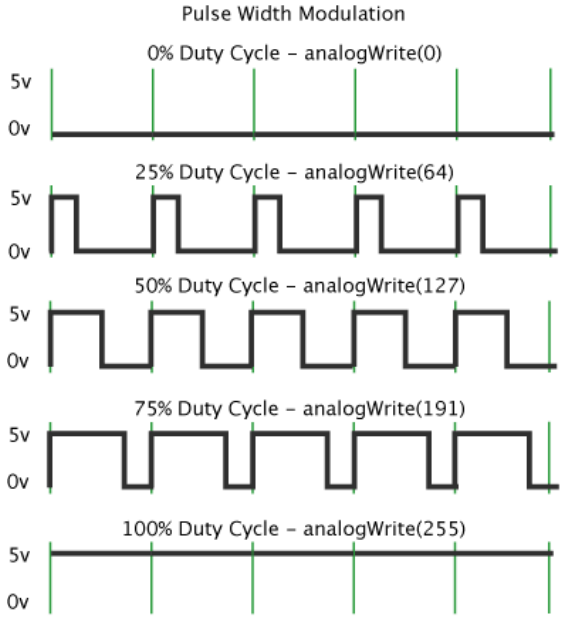

Figure 4.3: PWM graph from Arduino website which displays square waves for different duty cycles. [5]

Duty cycle is a term for the percentage of time that a signal is "high" in a PWM. The simulated analog output in each of the cases in Figure 4.3 is the average value of all of the "high" phases and "low" phases. In other words, the duty cycle times the maximum value (in this case 5v) will give us the

simulated analog signal. This relation is displayed in equation 4.1.

$$AnalogValue(V) = DutyCycle(\%) \times MaxValue(V) \tag{4.1}$$

The basic H-bridge will allow us to run the motor in two directions at a constant voltage (assuming an ideal circuit and battery). The two states would be running the motor at max power in one direction, and running the motor at max power in the reverse direction. Therefore, if our H-bridge can switch fast enough for us to effectively use PWM, we can theoretically run the motor at any speed up to its maximum, and in either of two directions.

In reality, motors can have internal resistance which can create a dead band where the motor will not start up until a certain threshold. We experience a dead band in the bike's rear motor however the exact cause has not yet been determined. Another important note is that an H-bridge using a manual switch would not have been able to use PWM, as it is not possible to manually flip the switch fast enough in order to get the desired speed. For this reason, many H-bridges come as integrated circuits and use transistors or other semiconductors as switching elements. Figure 4.4 below is an example of a proper H-bridge circuit diagram, and we will discuss the components contained.

4.2.3 H-Bridge Circuit Diagram

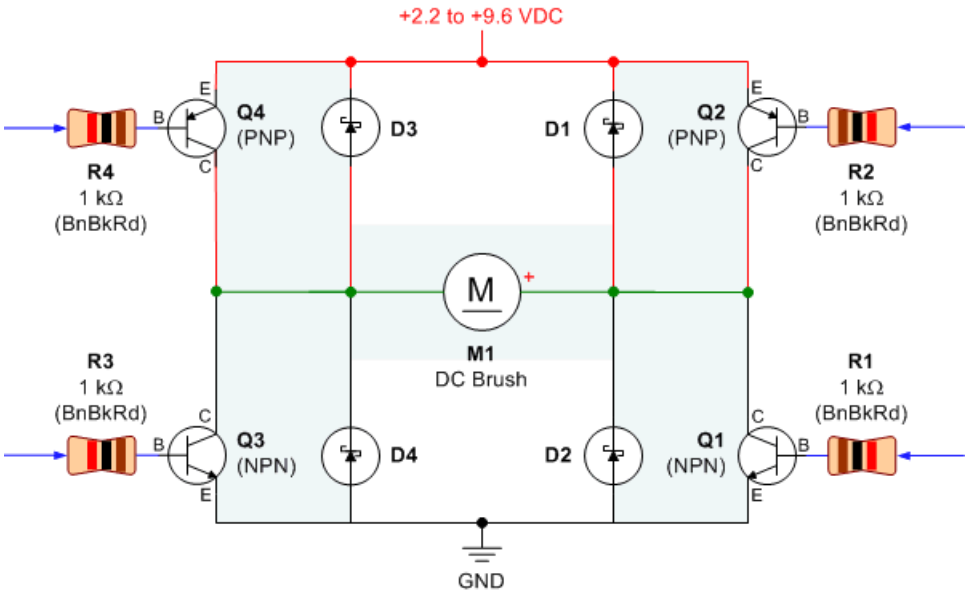

Figure 4.4: Circuit Digram for H-Bridge.[4]

In the diagram, current will run from top to bottom. All "Q#" components are transistors, which have an emitter, collector, and base, marked "E," "C," and "B," respectively. Transistors take a small voltage to their base that controls whether they are on or off. In the diagram, voltage from the battery goes through a resistor, labeled "R#," to attenuate the voltage to the "base." The resistors and transistors make up the switching elements in the H-Bridge.

The components labeled "D#" are safety-catch diodes, sometimes also called fly-back diodes. These function as safety features in the H-bridge. Ideally, when switching states in an H-bridge, the switch should happen instantaneously. However, in reality this is not possible. This means that each time you switch directions on the H-bridge, it is possible to have a temporary shoot-through condition. In order to prevent this, there must be a small

transition period in which all switches are off, in order to ensure that there is no accidental shoot-through. This is where the catch diodes come in. As DC motors are inductors, they will oppose changes in current, as per Lenz's law. So, during this temporary off period, there can be current generated that will flow backwards. In the diagram, backwards means from bottom to top. This can be an issue as without catch diodes, the induced current can overload the transistors. However, the diodes provide an alternate path for the induced current to flow, which prevents the previous issue of overloading. Also, as diodes act as one-way gates, this does not create any shoot-through conditions.

4.2.4 Isolated-Error Amplifier

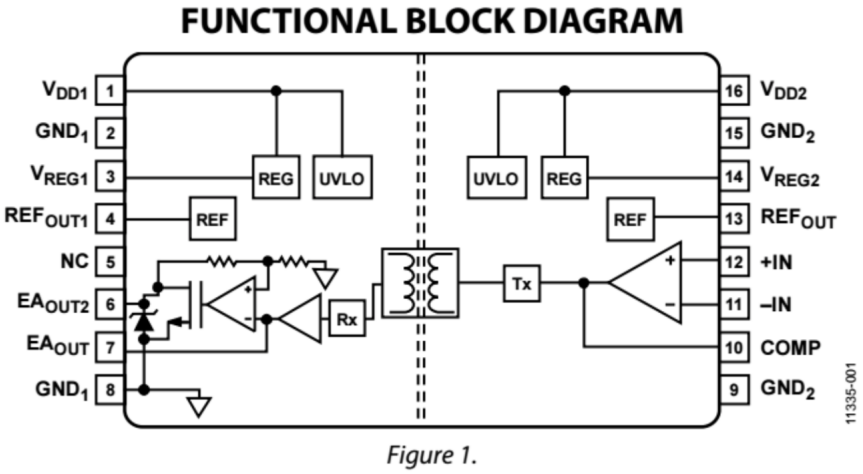

Figure 4.5: The pin diagram of the Isolated Error Amplifier

The isolated-error amplifier is a piece of hardware (ADuM3190) that is used to read the voltage on the main power supply using the Arduino. Since a

voltage above 3.3 V will fry the Arduino, we cannot directly connect the power supply (28V) to the Arduino. Instead, the power supply is connected to the isolated-error amplifier, which outputs a smaller voltage proportional to the power supply voltage, which is then safely read by the Arduino.

The output voltage, EAOUT, in Figure 4.5 is connected to Pin 63 (Analog Pin 9) on the Arduino. The reading from this pin is used in any calculations involving the voltage of the power supply.

Another method of reducing voltage is using a voltage divider. A voltage divider consists of two resistors arranged in a circuit, shown in Figure 4.6. We needed to use a voltage divider in our circuit to reduce the voltage before connecting to the input of the isolated-error amplifier. The range for the input voltages of the isolated-error amplifier is between 0.35 and 1.5 V. Therefore, the 28 V power supply cannot be directly connected to the input. The voltage divider circuit used on the protoboard is displayed in Figure 4.6 The voltage divider equation is as follows:

$$V_{IN} = V_{Battery} \times \frac{R_{21}}{R_{21} + R_{22}} \tag{4.2}$$

Using $V_{IN} = 1.5V$ and $V_{Battery} = 30V$, we calculated that the ratio of R_{21} to R_{22} should be at least 1:20. Therefore, we used $R_{21} = 470$ Ohm and $R_{22} = 10k$ Ohm.

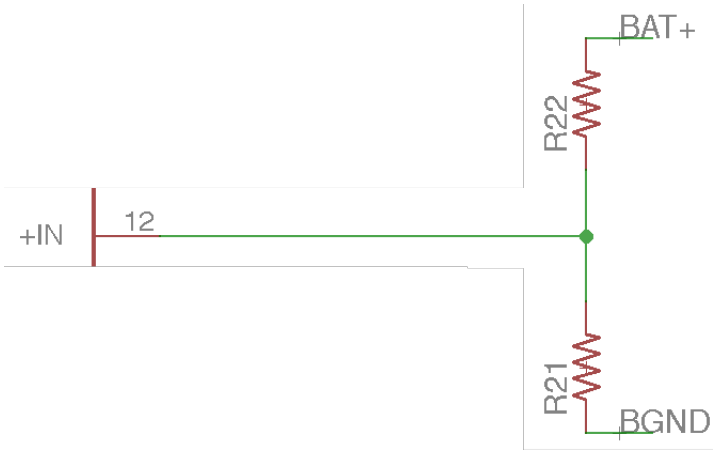

Figure 4.6: The voltage divider circuit used on our protoboard

One reason we need to use an isolated-error amplifier to connect to the Arduino, instead of a voltage divider, is because there should never be a direct wire connection between the 28 V circuit and the 3.3 V circuit on the protoboard. The isolated-error amplifier separates the 28 V power supply from the Arduino, and safely transmits the signal, without the risk of directly connecting the Arduino to 28 V.

4.2.5 Low Pass Filter

Sometimes signals can have noise that will distort data and give erroneous results. It is then useful to have components that can passively filter these signals. The low pass filter is one example of a passive filter.

The concept is that the filter can take in signals of various frequencies, and will attenuate, or weaken, any signals that are undesirable. In our case, we wanted to attenuate high frequency signals (as we were receiving more frequent readings than the hall sensor should have been outputting) and so we used a low pass filter. A low pass filter consists of a resistor and capacitor

arranged as in Figure 4.7. This simple addition of a capacitor only allows frequencies below a certain threshold (depending on the capacitor) to pass through, and is thus named "low pass."

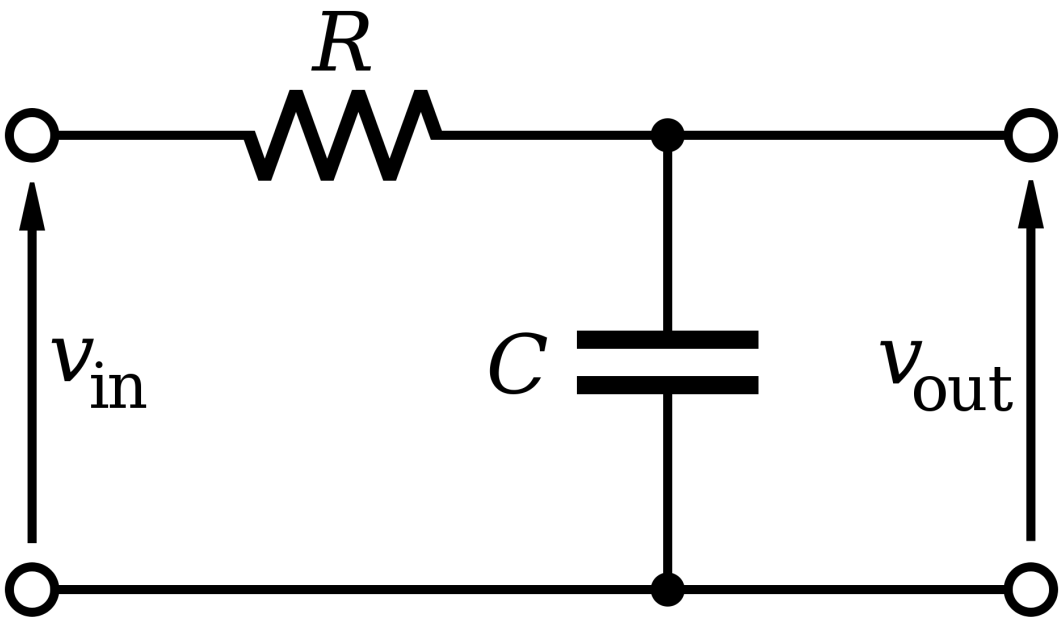

Figure 4.7: A low pass filter [9]

4.2.6 Rear Motor

The rear motor is controlled by a Golden Motor Magic Pie 2x brushless hub motor [1], which is a motor controller purchased from the company Golden Motor in 2014 (Figure 4.8). The power supply is connected to red and black wires, whose ports are labeled "Battery" in the figure. These wires are connected to the main 28 V battery. The speed is controlled using green and black wires, which are connected to the ports T2 and Z, respectively, in the figure. In order to control the speed, an analog signal must be sent through the green wire. The black wire is ground. The analog signal to the motor

controller is a value between 1.4 and 3.4 V. A higher voltage will produce a faster moving wheel. The yellow, green, blue, red and black ports labeled "Hall" on the right side of the figure are the outputs from the hall sensors. The yellow, green, and blue ports output signals each from one of the three Hall sensors on the motor. The port labeled R controls the direction and is connected to a purple wire. By shorting this wire to ground, the wheel spins in reverse. In order to switch from forward to reverse, the purple wire is attached to a relay switch, which is connected with pin 50 on the Arduino. Presently, the other ports are not used, but can be used with other features.

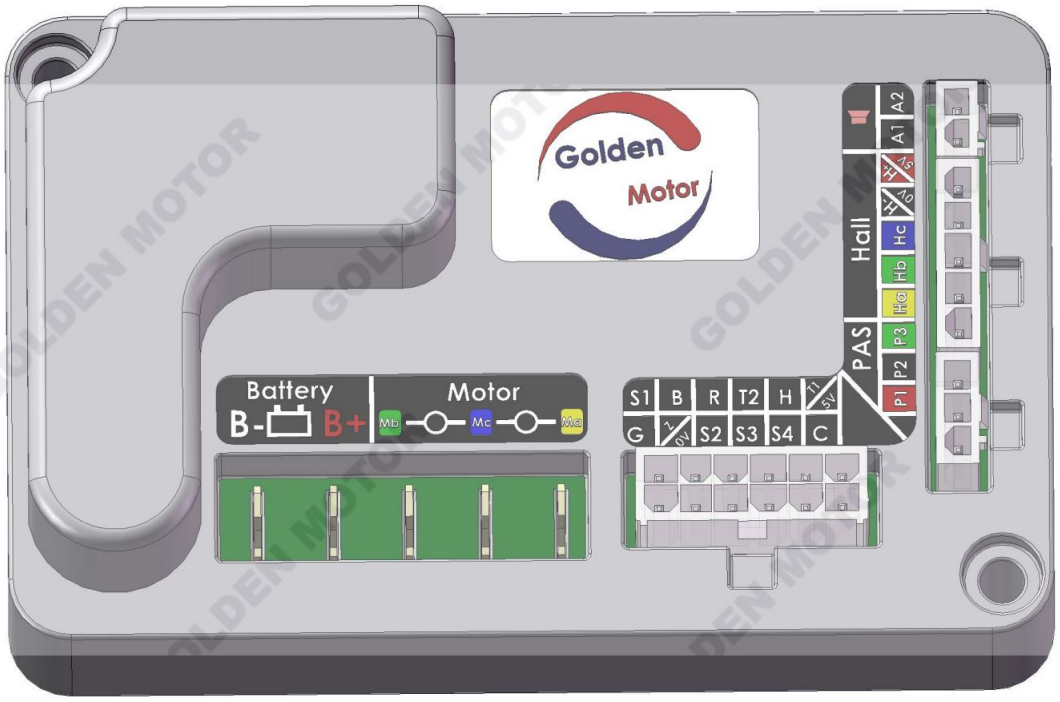

Figure 4.8: The Golden Motor Controller without connections

4.2.6.1 Hall Sensors

A Hall sensor is a magnetic sensor that measures the proximity of a magnet. There are two main type of hall sensors: digital and analog. Our rear motor uses digital sensors. There are also two types of digital hall sensors can be either latching or non-latching. Latching means that the voltage is high when the north end of a magnet is near the sensor and stays high until the south end is in proximity of the sensor. By contrast, non-latching means the voltage is high when the north end of a magnet is in proximity and low otherwise. The hall sensors used on the bike are non-latching, so they set the voltage to high when the north end of the magnet is within a small range.[2]

The rear wheel contains 28 coils, which function as magnets, spaced equally around the circumference of the wheel. This was calculated by attaching an LED to one hall sensor wire and to ground, then counting how many times it lit up in one revolution of the wheel. Figure 4.9 displays roughly how the hall sensors and magnets are arranged on the wheel.

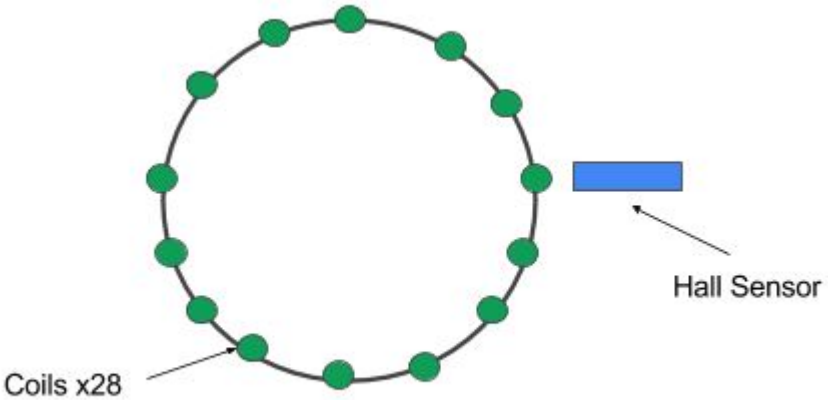

Figure 4.9: How the Hall sensors and coils are arranged on the motor

A typical signal from a hall sensor looks like a square wave. Presently only one of the three hall sensors is connected to the Arduino. This is the green hall sensor wire, which is connected to a white wire then connected to the protoboard. The Arduino pin associated with this signal is Pin 11. This is a digital pin, and therefore, the signal is digital when read by the Arduino. Figure 4.10 displays a digital signal from a hall sensor, before the addition of a low pass filter, read in the Arduino serial plotter.

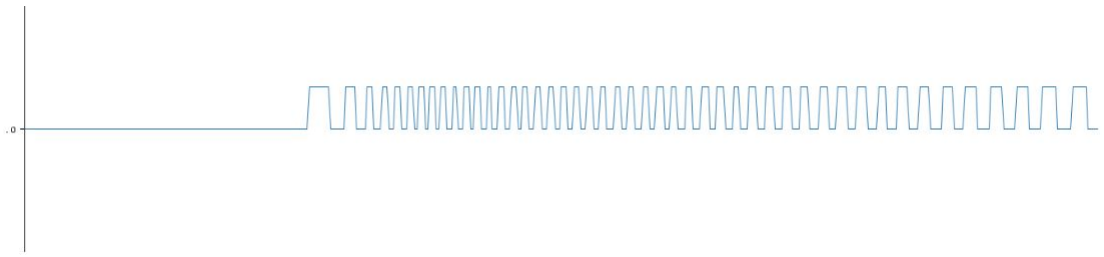

Figure 4.10: A typical signal read from the Hall sensor in the Serial Plotter. The x axis is time and the y axis is unitless.

The speed of the wheel can be determined by the period of the signal produced from the hall sensor. Since there are 28 coils around the wheel, the time it takes for the wheel to spin once is equal to the time it takes for the Hall signal to go "high" 28 times. Therefore, the angular speed of the wheel, in rev/s can be calculated by:

$$\omega = \frac{1}{28 \times T} \tag{4.3}$$

Where ω is the angular speed in revolutions per second, and T is the period of the pulse, as seen in Figure 4.11.

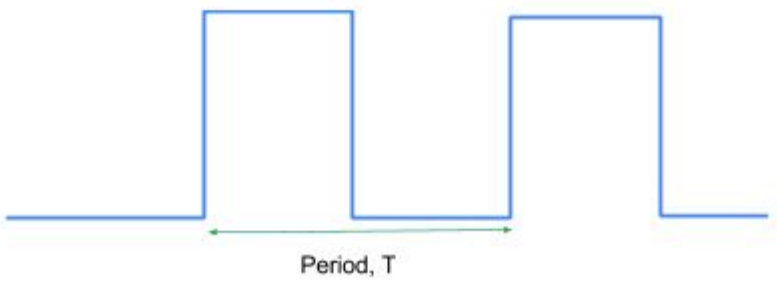

Figure 4.11: The period of a square wave, which is used to find angular speed of the wheel.

Using this fact, we wrote code on the Arduino that works as a speedometer and can calculate the angular speed of the rear wheel.

4.2.6.2 Turning on the Rear Motor

Before Fall 2016, the only way to get the rear motor running after uploading code was by physically going into the box and unplugging and replugging the wires that connect the throttle and the protoboard. This was a highly inefficient way to turn on the motor. We postulate that the reason the motor turns off when code is uploaded is due to a safety feature. When the Arduino sends a high PWM to the rear motor at rest, the motor immediately shuts off. This is useful in preventing spikes in speed but makes it difficult to turn on the motor at starting speeds within the motor's PWM range (around 70-180). To work around this safety feature we found that we can "ramp up" to the PWM we want. We do this by sending a very low PWM and incrementing it in a loop until we reach the desired PWM. This is useful for preventing the need to unplug the motor after uploading code to the Arduino.

4.3 Methods & Results

4.3.1 H-Bridge

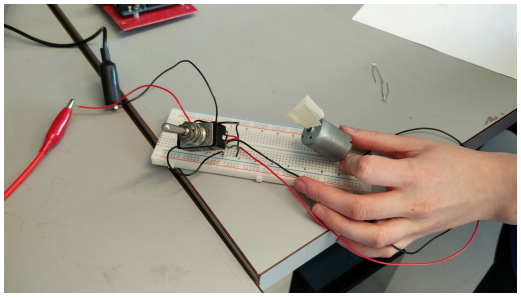

Figure 4.12: H-Bridge setup with DPDT switch

The H-Bridge can be implemented in a variety of ways. When we started making one, we first used a DPDT (Double Pole Double Throw) switch[8] we found in the lab. The DPDT switch is a physical switch that allowed us to easily switch the motor's direction. However, as stated above (section 4.2.2) the manual switch is not fast enough to use PWM. As a result, we moved on to remaking the circuit in Figure 4.4, and controlling it using Arduino.

In order to test the H-Bridge, we tested small circuits with a diode, transistor, resistor, and LED, as shown in Figure 4.13. The LED signalled whether or not current was running through the transistor. The resistor was connected to base, so as not to overload the transistor. When testing transistors, it initially appeared to function as intended, where the transistors are off (not allowing current to flow) when base is not powered, and on, allowing current to flow, when powered. When powered, the transistors should allow current to pass through the emitter to the collector. However, we noticed that the current seemed to be travelling through the base of these transistors (it did not matter whether or not the emitter was powered; the LED was

on whenever there was power to the base, despite not being powered by the emitter). This lead us to believe that the transistors we found in the lab were faulty, as either they were very poor transistors, or perhaps they had been burnt out previously.

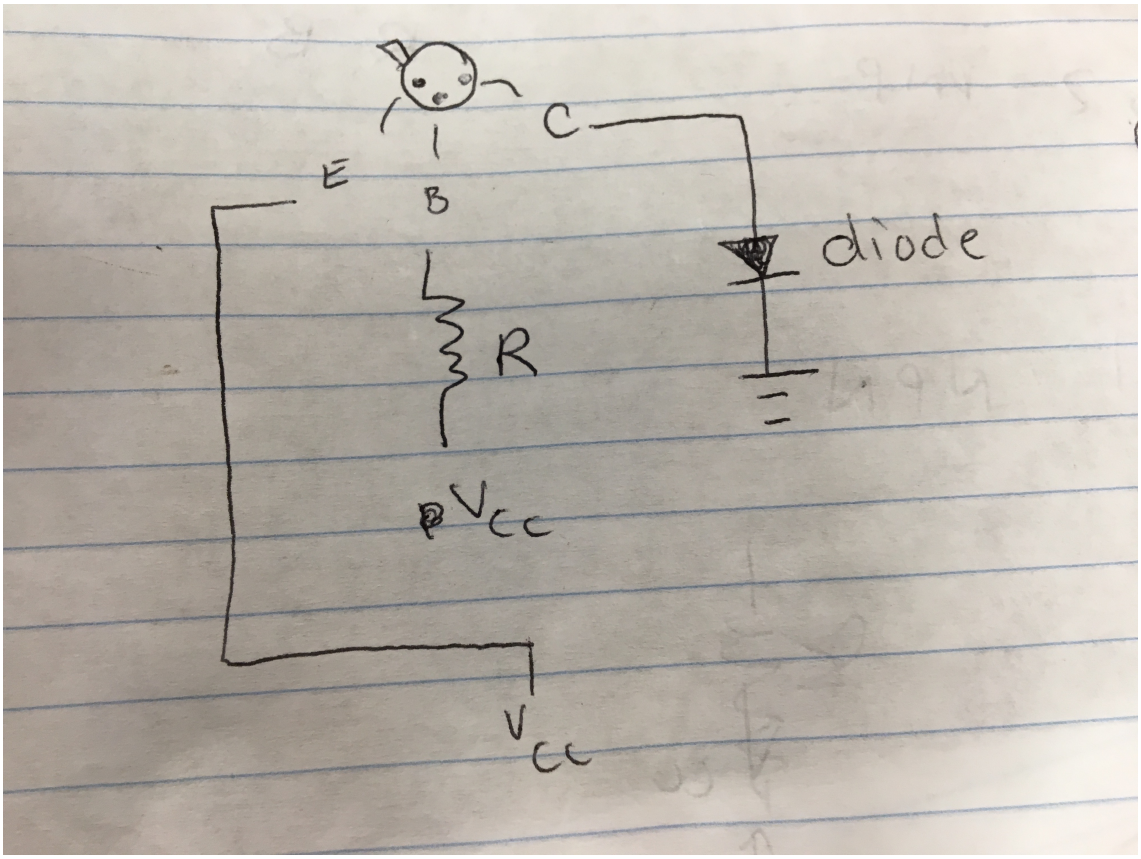

Figure 4.13: A circuit used to test transistors

After ordering new transistors (2N3906 & 2N3904), we rebuilt the circuit in Figure 4.4. This circuit is shown in Figures 4.14 and 4.15. The long yellow wires are connected to the base pins of the transistors. By switching which transistor bases were connected to ground and power, we could control which transistors were on or off. In the configuration Figure 4.14, the motor spun

counter clockwise. When we switched the yellow wires as in Figure 4.15, the motor spun the other way.

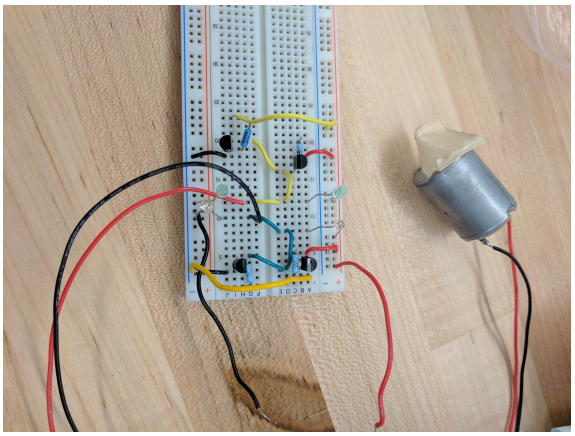

Figure 4.14: H-Bridge Set up one

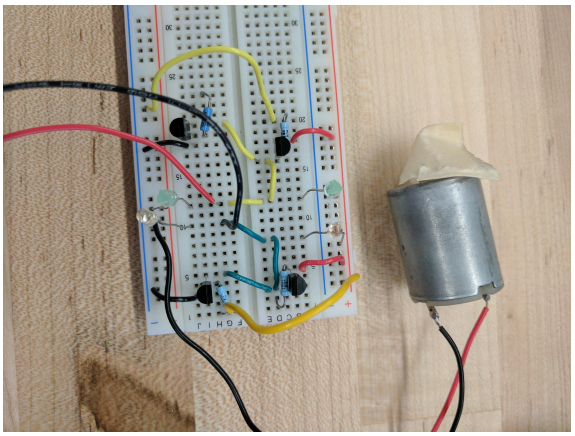

Figure 4.15: H-Bridge Set up two

4.3.2 Isolated-Error Amplifier

The Arduino pin 63 (analog pin 9) reads a value from 0 to 1023, since this is the standard range of an analog Arduino pin output. This reading is

associated with the output of the isolated-error amplifier. To find a relation between the voltage on the battery and the reading from the Arduino, we tested several batteries with different voltages and recorded the value on the pin. We also recorded the voltages on the output pin of the isolated-error amplifier, EAOUT. However, we decided to treat the system as a "black box" because we are only interested in the voltage on the battery, and the reading from the Arduino. A plot of the battery voltage versus the analog reading on the Arduino can be seen in Figure 4.16. From this plot, we found a linear relation between the Arduino pin reading and the battery voltage. Battery voltage can be found using this formula:

$$V_{Battery} = \frac{Pin_{63}}{14.201} \tag{4.4}$$

Where $V_{Battery}$ is the voltage on the 28V Battery, and Pin_{63} is the analog value read from pin 63 on the Arduino.

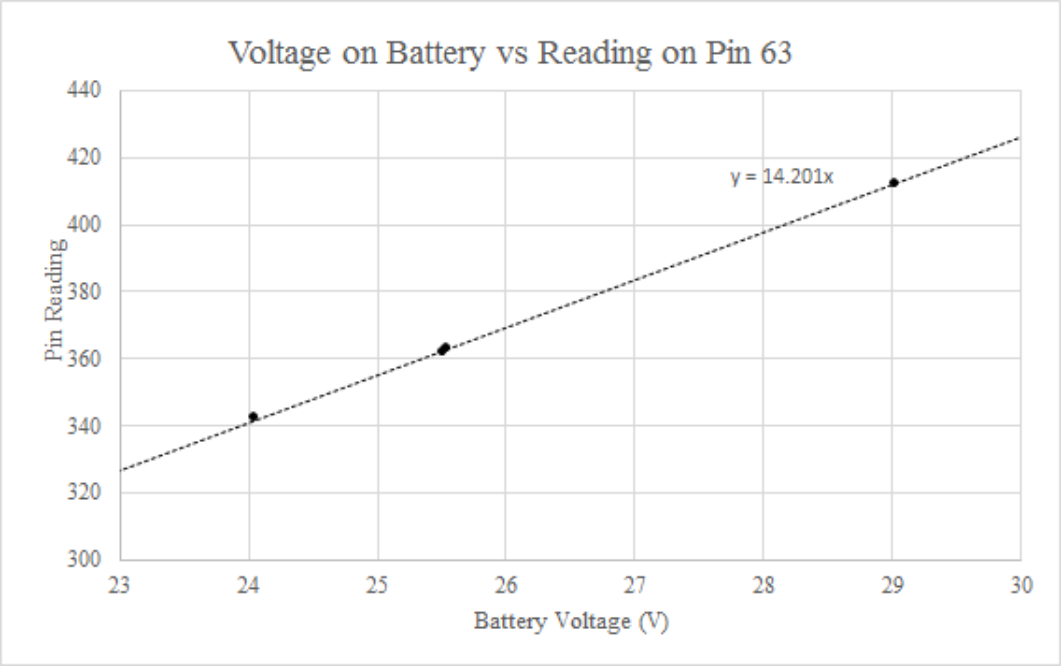

Figure 4.16: Different readings on Pin 63 corresponding to voltages on the battery ranging from 24V-29V

4.3.3 Rear Motor

One of the main goals of the team is to write code that uses Hall sensors to measure the speed of the wheel, and to relate that speed with the PWM signal from the Arduino. As of Fall 2016, the code accurately measures the speed, but we have not found the exact relation between PWM and speed. We ran into a few roadblocks, which deterred our ability to meet our goal. First, we found that the operational amplifier which produced the signal for the motor needed a capacitor. Before the addition of the capacitor, the wheel would always spin at very low jerky speeds, no matter how high the PWM. Another problem encountered was a noisy Hall sensor signal. The signal had too much noise, and our code was reading the frequency of the noise instead

of the wheel. This problem was remedied by adding a low pass filter to the signal. Some other methods that caused a delay in our progress are described briefly in this section.

4.3.3.1 Duty Cycle

A duty cycle is the percentage of time a signal from a square wave is high. We needed to know the duty cycle because our first code included PulseIn(), a built-in Arduino function. This function works by measuring how long a digital pulse is high. From this measured time, we estimated the period of the signal and the angular speed. We predicted that the coils on the wheel were spaced so that 50% of the time a coil was in front of the sensor, and 50% of the time no coil was in front of the sensor, which corresponds to a 50% duty cycle.

To find the duty cycle, we wrote code that constantly prints the state (1 or 0) to the serial monitor and took the average of the data. We ran the motor at different speeds to ensure that our results were consistent for all speeds. The results are summarized in Table 4.1.

Voltage	3.6	3	3	2.4	2	1.6
Duty Cycle	0.508	0.534	0.493	0.494	0.504	0.480

Table 4.1: The measured duty cycle for several voltages

The average duty cycle was found to be 50.21% and the standard deviation was 1.8%. Since the values are very close with a small standard deviation, and there is no downward trend, using a 50% duty cycle in any calculations is an accurate estimate.

We concluded that if the wheel is accelerating, this could have an effect

on the duty cycle. In this case our estimate of 50% for the duty cycle would not be correct. Therefore, the speed calculated using PulseIn() would also be incorrect. This duty cycle is still correct for constant speeds and helps us understand how the hall sensors are laid out. However, it is not useful for our purposes of measuring speed.

4.3.3.2 Measuring Speed Using the Arduino

The newest code for measuring speed is different from the code we worked on for a large amount of the Fall, and so most of the tests described below are invalid. However, we included them to show the troubleshooting involved so that others can learn from our mistakes, and learn how to solve problems.

The code we wrote utilizes an interrupt to calculate speed. The interrupt triggers every time that the reading from the hall sensor goes "high". The time, in microseconds, that the signal goes "high" is stored. By taking the difference of two times, the period is calculated. From this period, the speed is calculated.

We originally only wanted an average speed, and at first, the code would only measure the time after several passes of the square wave. The number of times it passes is called the "Sample Length." The total time is divided by the sample length to get the average period of one square wave. We tested this sample length by trying different values and evaluating the accuracy of the readings to discover the minimum length we could use. A few results from the test can be seen in Figure 4.17 below.

We used the oscilloscope to measure the speed, and compared those values to the values our code was generating. To do this, we displayed the pulse from the hall sensor by attaching an oscilloscope probe to the white wire connected to the Hall Sensor. Then, we calculated the speed using the "measure period"

function. To find the speed, we used:

$$\omega = \frac{1}{28 \times T} \tag{4.5}$$

Where ω is the angular speed in revolutions per second, and T is the period of the pulse in seconds, measured by the oscilloscope.

Test Number	Osc. Period (ms)	Osc. Speed (RP	Arduino Speec	Standard Deviatio	Effective Voltag	PWM	min T	Sample Length	Percent Erro
6	11.1	3.217503218	3.225785609	0.077117221	23.79	160	2000	300	-0.257%
7	11.3	3.160556258	3.342546419	0.126145343	23.29	160	2000	1000	-5.758%
8	11.1	3.217503218	2.189439252	0.290424326	24.1	160	8000	100	31.952%
9	35.6	1.003210273	2.321017274	0.351777248	14.5	95	2000	300	-131.359%
10	35.7	1.00040016	2.026706282	0.228416652	14.35	95	10000	300	-102.590%
11	35.8	0.997605746	1.083	0.2562901	14.42	95	10000	100	-8.560%
12	35.9	0.9948269	1.993420428	0.01843099	14.42	95	10000	200	-100.379%

Figure 4.17: Sample data while testing our original speed code. The 3 parameters were Min T, Sample Length, and PWM

Initially, the readings from the Hall sensors were very noisy. Figure 4.18 displays a reading from the Hall sensor that experiences this noise. To combat this, we added a section of code that filtered out very high frequencies by removing any period, T, measured in the code below a minimum length. We also tested different values of this minimum T, to find the value that yielded the most accurate results for all speeds. This test is documented in Figure 4.17. However, by Andy Ruina's suggestion, we added a capacitor to create a low pass filter in the circuit and filter out the higher frequencies. This yielded far better results, and so we removed the filter in the code.

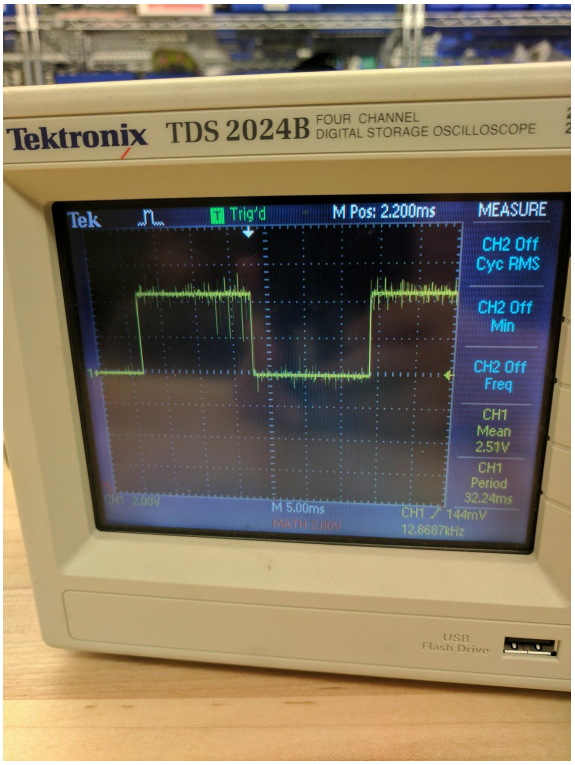

Figure 4.18: An example of the noise experienced before the low pass filter was added

The newest code for measuring velocity does not depend on sample length nor minimum period. Therefore, the previously discussed tests are mostly invalid. The newest code is very similar to the old code, but always operates with a sample length of 1 and has no minimum T value, due to the addition of the low pass filter. With this code, we measured speed at every PWM between 70 and 180 and found that there is a linear relationship between Arduino PWM and rear motor speed. The results are shown in Figure 4.19.

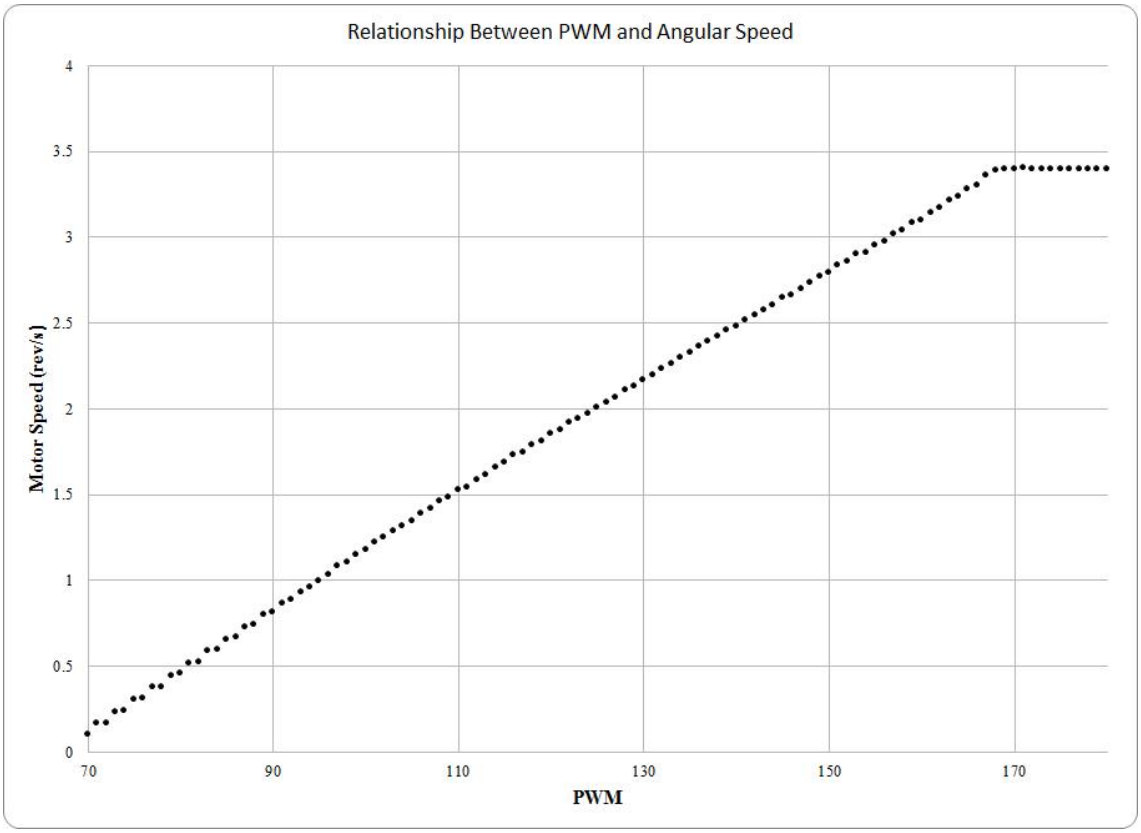

Figure 4.19: The relationship between PWM and speed is linear, as shown by this plot of PWM vs speed

4.3.3.3 Predicting Velocity

One goal that the rear motor team did not achieve in Fall 2016 was finding a relationship between speed and Arduino PWM. Knowing what Arduino PWM produces what speed is important for building a motor controller. If we know this relation, we can estimate what PWM from the Arduino will produce the desired speed. This will allow the reaction time of the wheel to be very fast. While working with the bike, we observed that when the 28 V battery was fully charged, the rear motor spun faster. Arduino PWMs

produced different rear motor velocities, depending on the battery voltage. This can be observed in Figure 4.20 Therefore, we postulated that there was also a relation between the voltage on the 28V battery and the speed of the wheel. However, as of this time, we do not know what the relationship is.

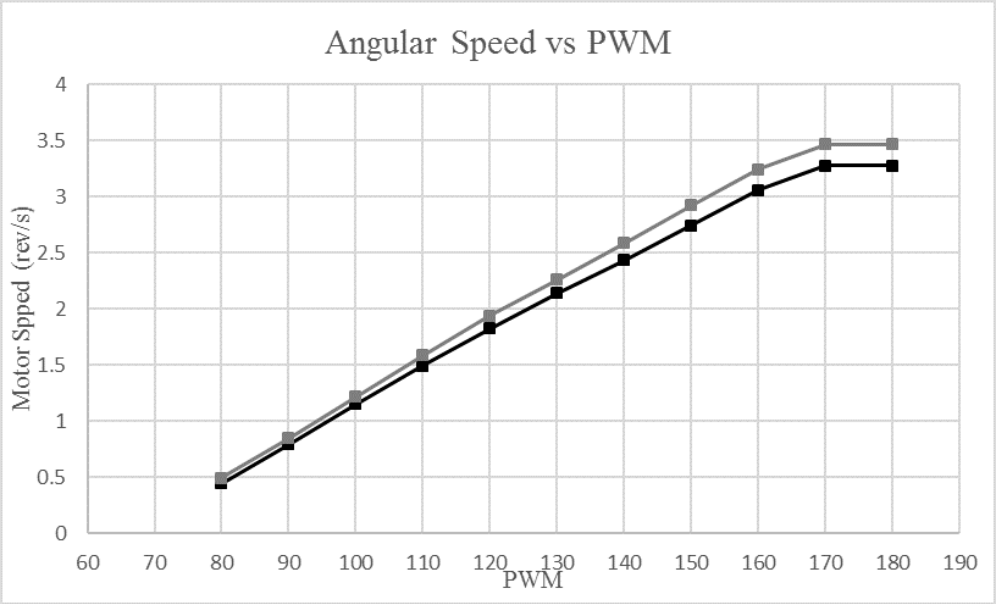

Figure 4.20: Two tests comparing PWM and speed. Note that the same PWM produces a different speed in each case.

We proposed that there may be a value that depends on the voltage of the 28 V battery, as well as the PWM, that can predict motor speed. We have called this value "Effective Voltage." However, it is not a useful value, and we recommend not using it in the future. We include it here so that others will not propose this relation in the future. Instead, we plan to determine the actual relationship between battery voltage, Arduino PWM, and output voltage directly to the motor.

We defined "Effective Voltage" by:

$$V_{Eff} = \frac{V_{Battery} \times PWM}{180} \tag{4.6}$$

Where V_{eff} is the effective voltage, $V_{battery}$ is the voltage measured by the error amplifier of the battery and PWM is the pwm value. This value is divided by 180, because the maximum value of the PWM is 180.

Figure 4.21 displays two tests done, where we measured effective voltage and motor speed. Both data sets have very similar slopes, but different intercepts. It is apparent that our defined "Effective Voltage" cannot predict the speed of the motor.

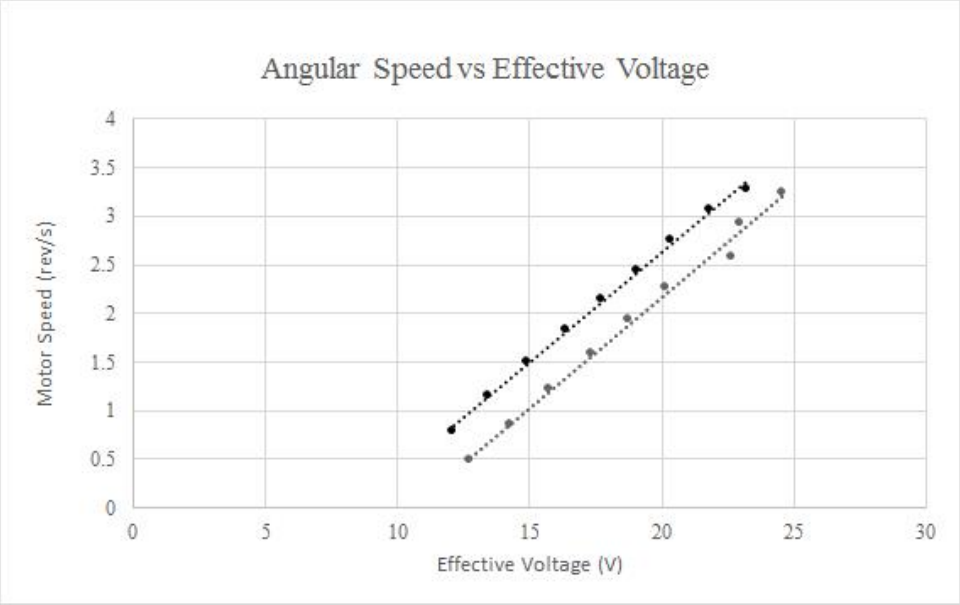

Figure 4.21: The same two tests as in Figure 4.20, measuring effective voltage and speed

4.3.3.4 Direction Control

The rear motor has the capability to switch direction. By shorting the reverse wire labeled R on figure 4.22 to ground we were got the motor to rotate in the opposite direction.

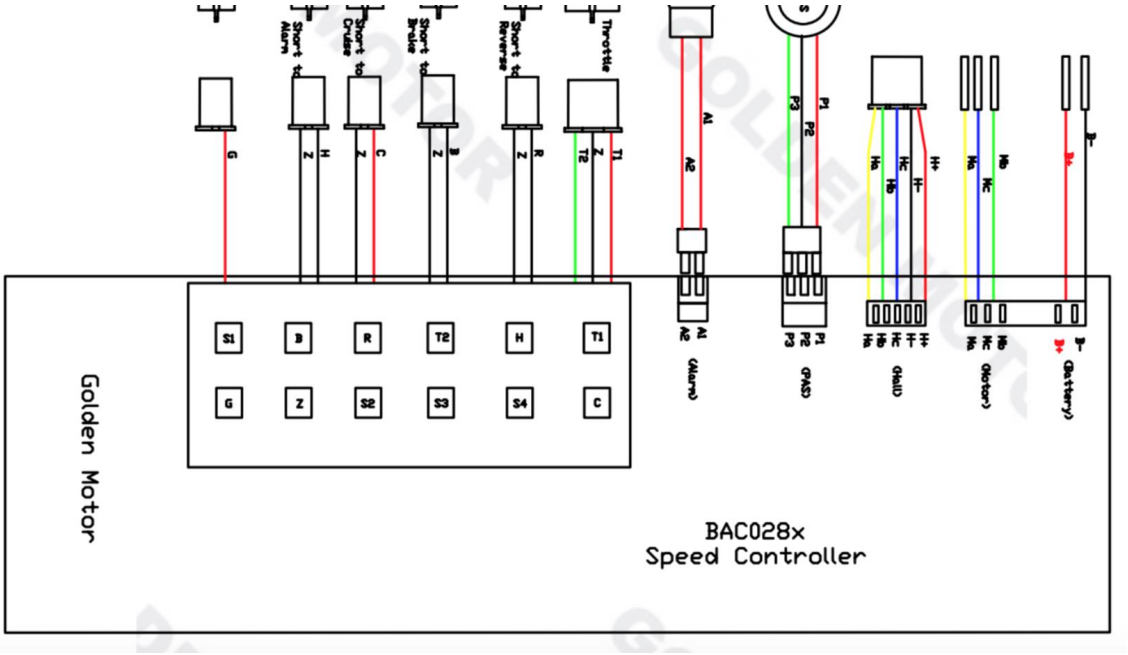

Figure 4.22: Golden Motor Magic Pie Wiring Diagram

This functionality will be very important for future projects that require both forward and reverse motion, particularly a trackstand. Further, we attached this wire to a relay. The pin to control this relay is pin 50 on the Arduino. When the pin is set to high the wheel will rotate forward when powered, when the pin is set to low the wheel will spin backwards when powered. This makes it possible to change motor direction with minimal Arduino code. For example we would write digitalWrite(reverse-pin,HIGH) or digitalWrite(reverse-pin,LOW) to set the pin to HIGH or LOW, respectively,

changing the direction of the motor.

4.4 Future Developments

One of our goals moving forward is to be able to characterize the rear motor of the bike, as well as the Magic Pie controller. At the moment, the Magic Pie system is effectively a black box. This is because we do not know the relationship between the power and signals we send to the motor controller box, and the actual voltage that gets sent to the motor. It will be beneficial to understand exactly what the Magic Pie controller is doing, as well as the characteristics of the rear motor itself. With this information, we will be able to perform a track stand and other tasks that require precise control of rear motor speed and quick transition between moving forwards and backwards.

One of the first goals of the team as a whole, is to achieve a track stand. A track stand is when the bike balances with minimal movement. To accomplish this, the rear wheel quickly alternates between forward and backwards movements at low speeds. Currently, our rear motor does not move at the low speeds needed to achieve a track stand. The lowest speed calculated in the simulation is about .01 rev/s, whereas the motor currently reaches a minimum speed of .1 rev/s. After characterizing the motor, we will be able to determine ways to move the motor slower. We suspect that the Magic Pie controller is preventing the wheel from spinning slower, since the minimum PWM we can send to the controller is 70. We also suspect that there is some internal resistance in the motor preventing low speeds.

4.5 Conclusion

The biggest achievement of this semester was accurately measuring the speed of the rear motor with an error of less than 1%. This achievement will be necessary and beneficial to the future endeavors of the entire bike team. The ability to change the speed and direction of the bike will also improve the overall functionality of the bike. These new features will allow the team to complete a track stand, and to test other features of the bike while the bike is moving. Next semester, we hope to determine the relationship between speed, Arduino PWM and the main voltage to the bike. This information will allow us to create a speed controller of our own, so that the speed of the bike can precisely be controlled.

Bibliography

[1] http://www.goldenmotor.com/

[2] http://www.electronics-tutorials.ws/electromagnetism/hall-effect.html

[3] http://www.modularcircuits.com/blog/articles/h-bridge-secrets/h-bridges-the-basics/

[4] http://www.robotroom.com/BipolarHBridge.html

[5] https://www.arduino.cc/en/Tutorial/PWM

[6] https://en.wikipedia.org/wiki/H_bridge

[7] https://en.wikipedia.org/wiki/Signal_(electrical_engineering)

[8] https://en.wikipedia.org/wiki/Switch#DPDT

[9] https://simple.wikipedia.org/wiki/Low-pass_filter

5

Business Team Functions (Kyle Fenske)

5.1 Introduction

The business subteam of the autonomous bicycle project team is responsible for handling and monitoring the team's purchasing and spending, public outreach and sponsorship events, as well as overall project team logistics.

5.2 Purchasing

As a small, undergraduate research group to a university recognized project team, the team had to acquire initial sponsorship and fund raising from project team donors and set up a purchasing account. The latter task took the majority of the semester due to certain holdups across different engineering departments. Even though this process held up progress initially, we were able to learn from the road block.

For instance, the first purchase we made was an embedded evaluation kit from Yost Labs. Yost Labs delayed the shipping for our new inertial measurement unit (IMU) due to production issues in their lab. Yet, I was unaware when it would ship until began to call the company and reach out in a more direct way opposed to exchanging emails. This small order emphasized the importance of communication and enforcing deadlines.

Our team's purchasing account is now officially set up. Members of the team can alert me of a part that the team needs and I can go ahead and purchase it without the help of a third party. One of the main parts of the purchasing account is the eShop access. This allows anyone on the team to assign a cart from certain stores to my account, streamlining the purchasing process.

5.3 Out-reach Events

Throughout the semester, autonomous bicycle has attended various local outreach events in order to demonstrate the progress that we have made thus far and to gain recognition on campus. The purpose of this is to garner interest from other students interested in joining our team. Additionally,

outreach events are important in the sense of becoming acquainted with people who would be interested in donating to our team.

5.3.1 Homecoming Project Team Showcase

Within the first couple weeks of the semester, we attended on campus showcase for alumni and families interested in engineering. During the showcase, the bicycle was not able to roll or showcase any major functions of the bicycle. However, we were still able to draw interest by talking about our plans for the future and our prototype. The main takeaway from the event was the need for team unity and branding, which other older teams demonstrated with well-developed posters and team apparel.

5.3.2 John Swanson Presentation

Autonomous bicycle presented our progress and goals to John Swanson, the main donor for Cornell Engineering project teams. This was a chance for us to practice presenting our project to someone who has technical expertise, but may not be specialized in our field. We refined our standard pitch to prospective donors and other interested people while developing this talk.

5.3.3 Northeast Robotics Colloqium

The Northeast Robotics Colloquium (NERC) was an event hosted by and at Cornell and attended by research institutions from all over the northeast like MIT, Carnegie Mellon, and Johns Hopkins. The event consisted of brief pitches from each poster-presenter followed by poster question and answer sessions.

For the event, we created a poster that highlights the interesting technical aspects from the project. We had to organize the status and direction of our project and present to a technical audience that does have some knowledge on the subject of our work.

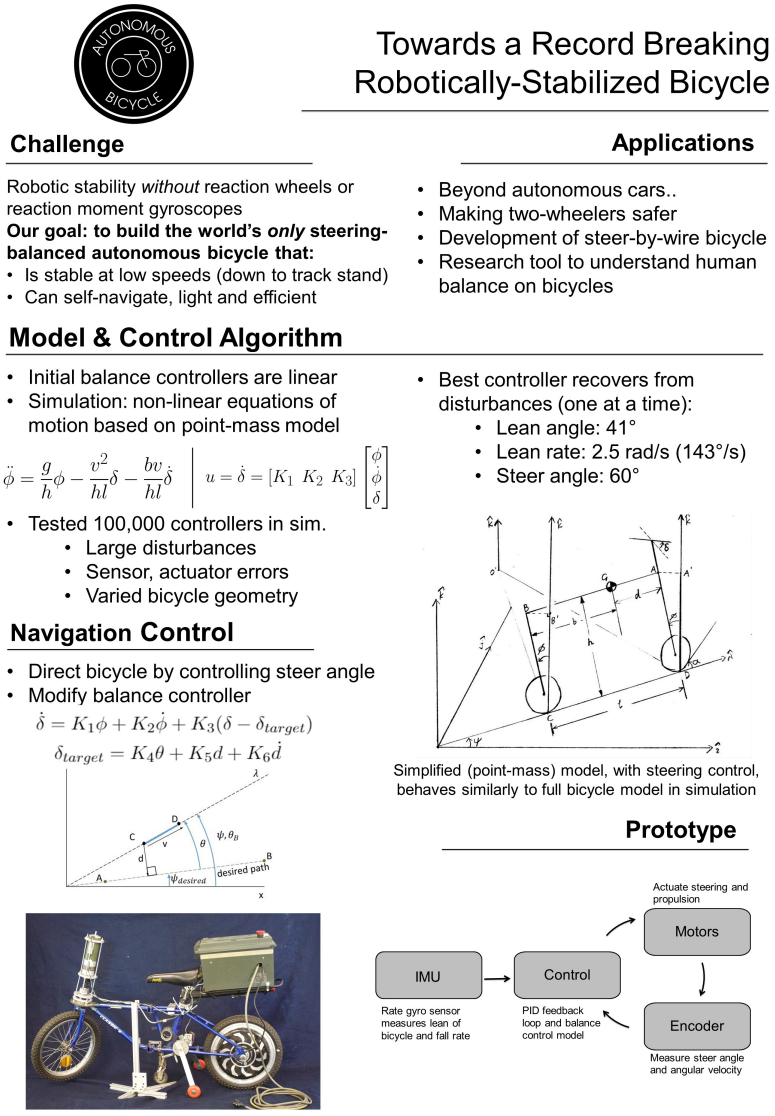

Figure 5.1: Northeast Robotics Colloquium Poster

5.3.4 Barnes and Noble Mini-Makers Faire

Our project team was invited to present our work at a local, family-oriented event at the Ithaca Barnes and Noble. This was the second annual mini-makers faire. We presented Scott Bollt's miniature bicycle to demo the servo's ability to balance the bicycle. The use of the miniature bicycle to demonstrate the concepts we want to apply on the large scale bicycle was very key in conveying a difficult concept to a younger audience. Essentially, the simplification of the bicycle in terms of sheer size and functionality made it easier to focus on the convey one aspect of our project. The manager and employees of Barnes and Noble asked us to come back next fall for their third annual fair, which will feature a more technical and higher level audience.

5.4 Sponsorship

In terms of sponsorship, we have reached out to several different companies about the prospect of sponsoring the autonomous bicycle project. Instead of repeatedly approaching companies with the prospect of sponsoring an autonomous bicycle project team that as of yet cannot operate autonomously, I decided that it would make most sense to focus on developing a sponsorship package for different levels of involvement and for different donation sizes. The package details different levels based on monetary value and the possibility of donating a bicycle. The tentative sponsorship package goes as follows: Bronze: 100 dollars - Company feature on website blog Silver: 200 dollars - Bronze Sponsorship package - Company logo on bicycle Gold: 500 dollars - Silver Sponsorship - Company logo on team apparel As the progress on the prototype develops, I plan to adjust the sponsorship package accordingly.

5.5 Technical Involvement

As the semester continued, it became evident that as business team lead, I would benefit from being more technically adept and knowledgeable in order to understand what our team does and effectively represent us. A good way to join in and learn about the mechanical design of our bicycle was to work on developing a CAD drawing of our prototype to use on posters. We are using Fusion 360 to design the bicycle into a three-dimensional, accurately scaled model of the bicycle. This allows me to collaborate more with other members of the theme while also gaining exposure to a new program. Dylan, Kenneth, Michelle and I worked together to split up components of the bicycle as a feature of Fusion 360 is to compile each part designed separately at the end.Thus, as long as we made sure our 3D drawings were drawn to size in respect to the actual prototype. Fusion 360's extensive video tutorials on how to draw, extend, and manipulate your drawings. Primarily, I worked on crafting the ammo box. I used a combination of freeform modeling and sculpting in addition to using solid modeling tools. Furthermore, the four of us decided to use Fusion 360 as opposed to another CAD software was the cloud connectivity of it.

6

Motor Control and Sensor Data Processing (Will Murphy, Haoyun Xu, & Pehuen Moure)

6.1 Overview

The goal by the end of this semester was to design a fully functioning, robust and quick motor controller for the front wheel. We wanted this controller to ensure the motor is moving at the desired velocity commanded by the balance controller. This was done using PD (proportional-derivative) control.

Additionally, we worked on processing the sensory data from the encoder and the IMU, in order to provide the balance controller with accurate enough inputs to allow the bike to balance. This involved updating the main software of the bike significantly in order to made it modular and more testable.

6.2 Main Code Updates

There was a large amount of time spent this semester on debugging and testing the code to make the entire system cooperate. The two main places the code was tested and debugged were the PD controller and the IMU code. The biggest improvement from a software structure standpoint was making the code more modular; being able to test the functions in a more piece-wise manner made debugging much easier. Figure 6.1 shows the structure of this code, with the different functions clearly delineated:

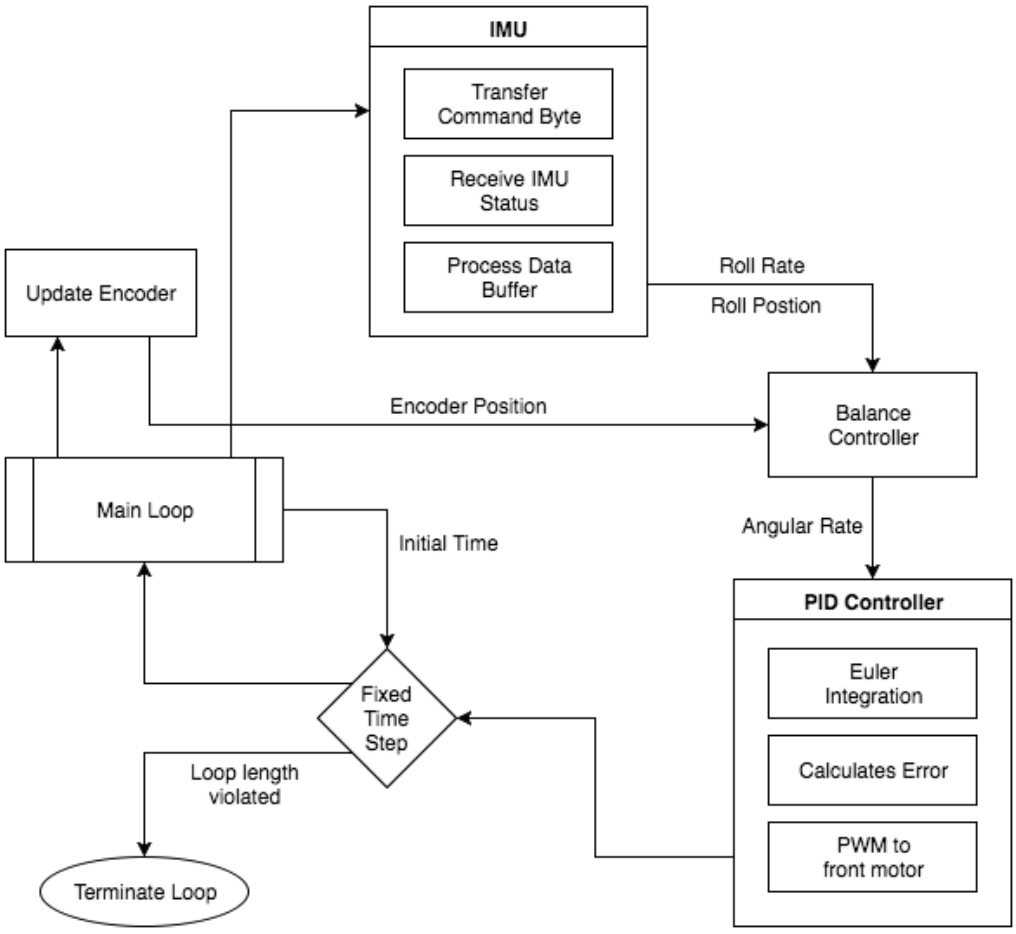

Figure 6.1: Main code structure

6.3 Optical Encoder

This section is written assuming that the reader has read Chapter 2 of the spring 2016 final report written by Weier Mi and Stephanie Xu. A change we made from spring 2016 to fall 2016 is that we no longer use the "motor encoder" which read the motor's top shaft. This encoder was initially installed in order to track the difference between the state of the motor shaft,

and what the fork of the bike was doing (after the mechanical coupling and the gearbox in the motor). We decided that the state of the wheel itself was more important, which is measured by the wheel encoder, and that the motor encoder was not giving us additional useful information. This decision was also because the controller for the front wheel changed (see section 6.5 on front wheel control). Because we now have a feedback loop controlling the motor speed, the difference between the two separate encoder readings is less valuable. If we were using a feed-forward loop, the characterization of the front motor and the front motor column would be more important, but because of the way a feedback loop functions, the continual adjustment only needs the current state of the wheel. The motor encoder has been removed from the bike, and we just use the wheel encoder.

6.3.1 Encoder Calibration

There are two types of rotary encoder: incremental and absolute. The encoder on our wheel is incremental. We have placed it on the bicycle such that the the Z (the index channel) channel goes high when the front wheel faces front. However, because it is an incremental rotary encoder, the 'A' and 'B' channels (the position channel) have a "zero" value which is not maintained when the encoder is powered off. The A and B channels for our purposes are the ones that tell us the angular position of the front wheel at any point, so they must be zeroed before the bike can run. The Z channel is mounted on the bike such that it is facing roughly forwards (it is subject to human error how perfectly aligned with the bike the Z channel front tick truly is, but up to this point it is as forwards as can be tuned by the naked eye). So we use the Z channel tick to set the zero of the position for the front wheel. This could be done in multiple ways. You could reset the angular value to zero each time the wheel passes by the front facing Z channel tick. You could

manually set the zero value once every time the encoder is powered up, or you could calibrate each time you reset your code, at the beginning of the program. We decided the last option was best. If you were to reset the value of zero each time the front wheel passed the front, if the encoder has drifted at all, and suddenly the value were to jump to zero when it was previously at a value that is far from from zero, it could cause a large spike in angular velocity. This is because we calculate the angular velocity of the front wheel from the position change over one time step. Let the front wheel angle (measured as shown in the dynamics section, chapter 1) = δ:

$$\dot{\delta} = \frac{\Delta\delta}{\Delta t} = \frac{\delta_t - \delta_{t-1}}{\Delta t} \tag{6.1}$$

So if the position jumped from some non-zero value to zero over one time step, we could unintentionally command a spike in motor velocity. The actual implementation of this velocity calculation is shown in the following code:

```
float current_vel = (((x-x_offset)-oldPosition)*0.02197*1000000*M_PI/180.0)
//Angular Speed(rad/s)
oldPosition = x-x_offset;

unsigned long current_t = micros();
previous_t = current_t;
```

Furthermore, calibrating the position channel to the front of the bike need only be done once because the encoder has been shown that it is consistently accurate to within $\pm 1^\circ$. Testing to validate this assertion was done this semester in addition to testing in previous years to verify the encoder was not skipping counts. This test was done by making a mechanical stop for the front wheel, recording the encoder value when it was flush with the stop, then quickly moving the wheel back and forth and recording the value when it was again flush with the stop. Our test confirmed previous encoder testing,

that it is accurate to within $\pm 1°$.

By using the quadrature hardware decoder, the state of the encoder can be read directly from registers on the microcontroller; To read the current value of the 'Z' channel of the encoder, we assign a variable "y" to store the initial value of the register REG_TC0_CV1 on the micro-controller. We then assign a variable "old index" to store that initial. Every time the front wheel passes the front encoder tick the value of the register REG_TC0_CV1 increments or decrements by one (depending on the direction the front wheel is turning). We write the front wheel to turn at an arbitrary slow speed (in this case, a PWM value of 20 is written to the front motor), and use a while loop to continually check if the value has changed. The instant the value of that register changes, it means the wheel is aligned at the front. This breaks the while loop and then immediately assigns an "offset" value which is the current value of the register REG_TC0_CV0. Each "x" position taken from then on is actually calculated as the current x_position minus x_offset, in order to take this initial calibration into account.

```
signed int y = REG_TC0_CV1;
oldIndex = y;
digitalWrite(DIR, HIGH);
while(y==oldIndex){
  analogWrite(PWM_front,20);
  y = REG_TC0_CV1;
}
x_offset = REG_TC0_CV0 ;
```

6.4 IMU implementation

6.4.1 Calibration

The IMU is initially configured and calibrated using the Yost Labs 3-Space Suite software. In order to use this software, you must move the two blue connectors on the IMU evaluation board itself to the third pin pictured on each side labelled with red outlining arrows. For this mode the blue connectors should be on the two rightmost pins of the three options, shown in figure 6.2 and the switch on the side of the evaluation mode must be set to usb connection mode. This is labelled with a solid red arrow in figure 6.2 .

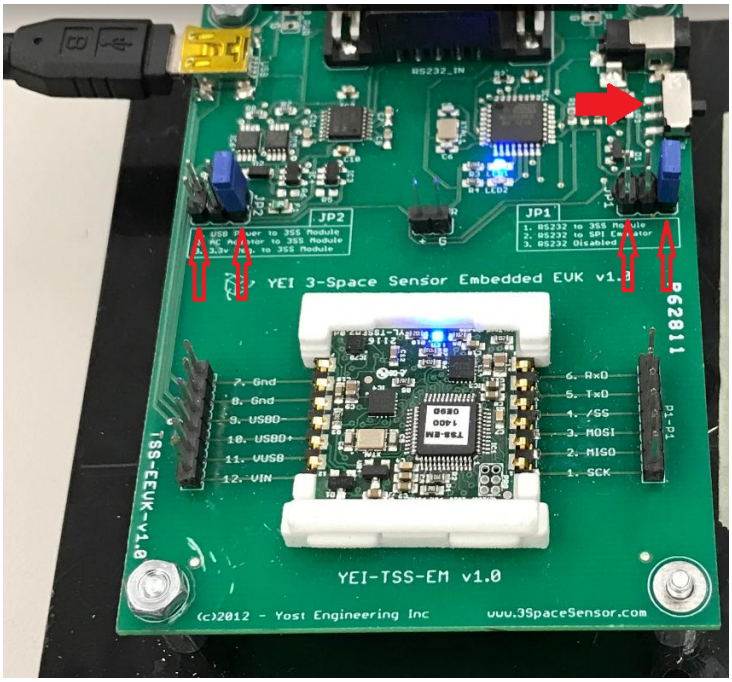

Figure 6.2: IMU calibration mode

You connect the IMU directly to your computer in order to calibrate it.

To calibrate the Angles, in the 3-Space Sensor Suite under the calibration tab, follow the given instructions. This will require orienting the IMU in 18 different angular positions, and each time you orient it in a new position you click next, and the IMU registers its position in space.
The IMU code that existed previously was adapted into a function. The argument of the function is the byte according to the IMU data sheet which, when sent to the IMU evaluation board, will return the desired value. The IMU and the evaluation board together have the ability to send back various different types of sensor data, which is why you must specify exactly which value you need. We have been using tared Euler angles for most of the semester. The IMU also has 3 main types of raw sensors that are used in different ways to calculate desired values. The three types of sensor are gyroscope, linear accelerometer, and magnetometer. The sensor fusion calculation is performed by the evaluation board. To modify settings of the IMU and evaluation board, you can use the check boxes on the bottom right of the 3-space sensor Suite software. This is how we disabled the magnetometer.

6.4.2 Drift

The values we were getting from the IMU were changing arbitrarily and significantly when we did our first full test run. When performing standing tests (holding the bike in place and leaning it side-to-side), the front wheel would behave as we expected, correctly finding zero-steer at the center. However when we rolled the bike down the hallway it would act as though a lean angle of 20 or 30 degrees was actually a zero lean angle. When rolled back down to different areas of the hallway, the "zero-location" would change.
When looking at the results for lean angle from the IMU (from the tared euler angle option) while rolling the bike down the hall, we noticed that the readings were changing wildly, even when the bike lean did not change much.

In order to narrow down the source of the problem, we connected the IMU to the 3-space sensor suite software, provided by Yost, where you can get direct readings for the different values requested from the IMU. With this program, we looked at Sensor -¿ advanced -¿ data plots and were able to see the values and plotted data coming directly from the evaluation board. Through the software, we saw that the readings for lean angle were still changing as we rolled down the hall. The sailboat team this semester had issues because their IMU was located close to other electrical sensors and wires, and this magnetic/electrical interference was causing their signal to have error. So we isolated the IMU from the bike to rule out electrical or magnetic interference from the bike components. Previously, because the front wheel causes vibration in the bike when it rotates, there were some cases where a feedback loop would occur, where the front wheel would rotate, causing the IMU to vibrate, which subsequently caused the front wheel to vibrate, which led to an overall oscillatory loop. So, we also removed the IMU from the bike in order to prevent this effect.

6.4.3 Modular IMU testing

The IMU was fixed to a table rigidly, then the table was rolled slowly down the hallway. We looked at lean angle, and observed the following:

- The location in the hallway had some relationship to the lean (and pitch and yaw) reading from the IMU.
- The relationship was significant enough to disrupt our result; we could not operate the bike with these large of errors
- The effect from taking the IMU in the elevator was symmetrical when going down the elevator and back up, suggesting that a given location

was associated with a given (incorrect) lean angle reading, assuming the physical IMU did not change orientation.

After presenting these results and discussing them with Prof Ruina, we then took a magnet and held it near the IMU and plotted the euler angles. The IMU was fixed on the bike, as the magnet was brought near it, all three euler angle results were affected. This means that fluctuations in the magnetic field around the IMU could have been causing the "arbitrarily" changing lean angle readings. To test this hypothesis, we disabled the magnetometer in the IMU by unchecking the box in the bottom right corner of the 3-Space-Sensor Suite and "committing the settings." When we ran the same tests, the large fluctuations due to rolling the IMU down the hallway no longer occurred.

6.4.4 Drift without the Magnetometer

In order for the system to be robust and versatile, we decided that the IMU should be able to handle magnetic interference, meaning we no longer wanted to include the magnetometer in the lean angle calculation. When we included only the gyroscopes and accelerometers in the calculation of euler angles, however, we saw a much smaller but still significant drift of the lean angle reading. This drift was roughly 1 degree/minute. Professor Ruina suggested that this could be an issue with how the evaluation board was using the three sensors to calculate orientation. He suggested we write our own sensor fusion code to combine the data from the accelerometers and the gyroscopes. This requires requesting the raw accelerometer and gyroscope data from the evaluation board. The idea behind rewriting this sensor fusion code is due to the capabilities of the different types of sensors in our 9-axis IMU, and the fact that the lean rate calculation done on the calibration board by Yost Lab's software relies heavily on the magnetometer, which we want to

disable. The 3 axis magnetometer is inaccurate over short time intervals but accurate in the long term, the 3 axis accelerometer is accurate over longer periods of time (not quite as long as the magnetometer) but is impaired when the bike is accelerating, and the gyro is accurate over short periods of time but drifts when integrating to calculate lean angle from lean rate. We decided we do not want to use the magnetometer for our lean rate calculation due to environmental magnetic interference, thus our sensor fusion will be a weighted average over time of the accelerometer and gyroscope readings. This should be implemented next semester (spring 2017).

6.5 Front Wheel Control

The front wheel control was re-done this semester with heavy input from Professor Ruina. After Will discussed the following idea for motor control with Professor Ruina at the beginning of the summer, the rest of the summer and fall semester was spend implementing and experimenting with this control design, shown in figure 6.3.

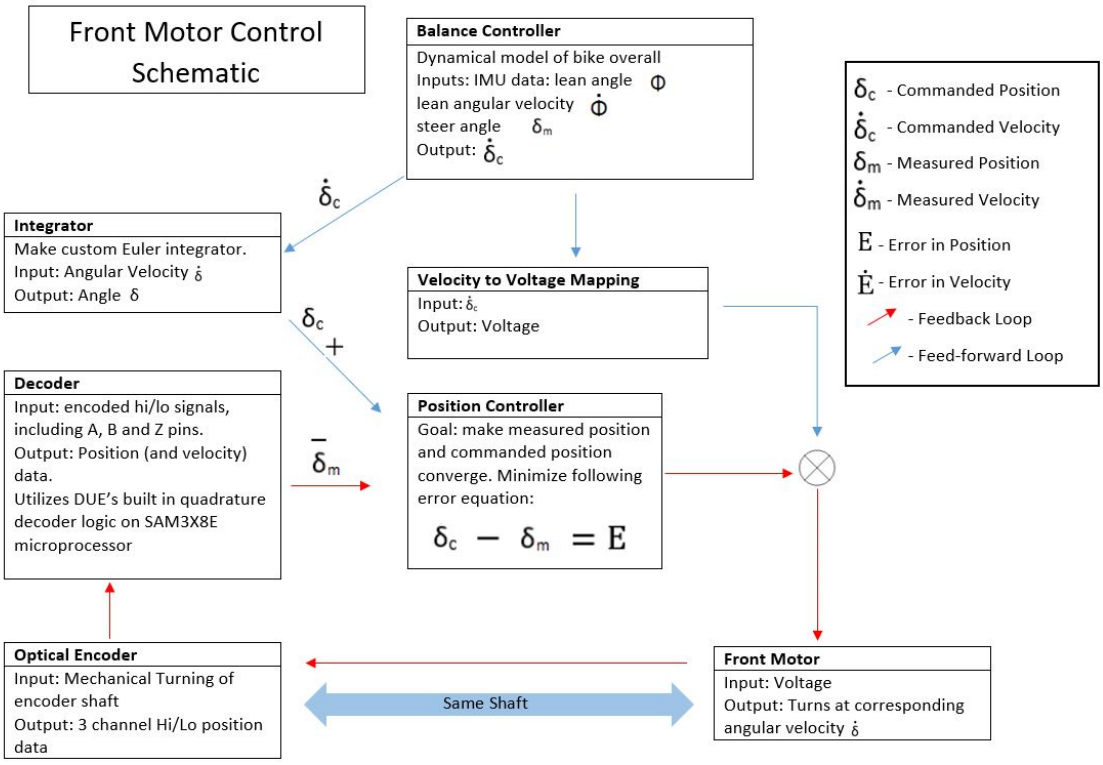

Figure 6.3: Front Motor Control Flowchart

6.5.1 Final decision: Abandon Feedforward loop

In previous semesters, there have been multiple attempts at characterizing the front motor such that a feed forward loop could be utilized for motor velocity control. The idea behind this would be as follows: Develop a function relating the independent variable (PWM commanded to the motor) to the dependent variable (front wheel angular velocity). If this characterization is accurate enough, then when the balance controller commands a certain angular velocity to the front wheel, the function would know exactly what pwm value to provide.

The difficulty faced in the past has been characterizing the motor in conditions representative of the bike rolling. There was an additional attempt this semester at characterizing the relationship between angular velocity and input voltage by taking the off-ground steady-state speeds based on inputted PWM. This relationship was mapped and we ran a linear regression to finalize the function (despite the non-linear findings in the past for this relationship; This comparison shows the difficulty of this characterization task and the potential inaccuracy). We then created a feedforward loop controller based off that characterization.

However, at this time we also had a breakthrough in the feedback loop controller. We found out that the feedback loop itself is strong enough to handle the task (will be explained further in the feedback loop section). The feed forward loop in preliminary testing also was not providing accurate results. If we were to continue to test the feedforward loop and tune it more rigorously, it possibly could be a viable option for a controller, or a helpful addition to the feedback controller as it was originally intended this semester. This idea is described by the summing junction in figure 6.3.

Furthermore, theoretically the feedforward loop is only effective to limited scenarios. For example, the above characterization is done with the front wheel in the air. The loop will work if the wheel is in the air. However, it is unlikely for the loop to still be effective when the wheel is on the ground because the friction is not considered in the feedforward model. It probably requires another characterization for when the wheel is on the ground. The point is the bike could encounter numerous scenarios but it is impractical to build a model for every scenario. In other words, the feedforward loop has minimal robustness. If the model is inaccurate for the current scenario, it will make the whole system worse. Even if it is only intended to augment the feedback controller, it still has the potential to make the controller worse if it is better equipped for a certain set of physical parameters, and the

environmental system parameters change.

The initial motivation for the feedforward loop was to help the feedback loop at the beginning to approach desired position faster. We thought that the feed forward loop would make a "ballpark" guess, and then this guess could be fine tuned by the feedback controller. Since bicycle performance and other testing proved that feedback loop is capable of outputting a satisfying response, we decided to abandon the feedforward loop.

6.6 Feedback Loop

The Feedback Loop consists of a PD (proportional-derivative) controller that enables the wheel to turn to the desired angle

6.6.1 Why PD controller

We chose to use PD controller for our feedback loop because Prof. Ruina proposed it and PD controllers are one of the most popular and well documented controllers. It is famous for being robust and easy to implement. We initially started with the intention to implement a full PID controller that would include all three terms, but after implementing the Proportional and Derivative terms, we observed that if we tune those two correctly, there is no need for an extra Integral term. Additionally, adding an integrative term may not be very useful because it requires multiple time steps in order to take effect, and our particular controller implementation requires a very fast reaction time, suggesting that the I term likely will not function quickly enough to improve the controller.

Integrative term is supposed to remove steady state error, and the alter-

native we used at the end to achieve that is giving a large proportional term ($K_p = 1000$), with a derivative term $K_d = 15$. For safety, we clip the signal at PWM = 100/255 to prevent the motor from going unstable. High proportional term shortens the deadband caused by friction of the front wheel system and scrubbing from the ground because the controller can output more voltage from relatively small error, hence decrease the steady state error. The side effect of having Kp value is large overshoot, but it can be compensated by the derivative term that will slow down the motor. Minimizing the dead band was the primary way of tuning the motor initially, because we knew the Kd value could dampen the effect of a high Kp value when the wheel was actually moving. Having a high Kp value is useful for very small adjustments that friction could otherwise overwhelm. For our purposes, small adjustments are crucial for balance and the majority of commands to the motor will be small adjustments, so a high Kp value is advantageous.

Besides the support from testing result, theoretically the disadvantage of an integrative term may also outweigh the advantage. An integrative term will inherently add a 90 degree phase lag at low frequency that will result in time delay. Since we want the response of motor to be as fast as possible, we should not implement the Integrative controller.

6.6.2 Testing PD controller alone

In order to test the performance of PD controller, we fed the controller with a sine wave as the desired position and observe how closely the wheel follows command.

The sine wave looks like this – sin(t), and t is drawn from the current time in Arduino. This sine wave has a amplitude of 1 rad, and we assume that the error between current position and desired position will not be bigger than

1. Since we don't want the voltage output to bigger than 80, we chose Kp = 70 as the initial value. Furthermore, according to several online sources, Kd is usually one order of magnitude smaller than Kp. Hence we chose Kd = 2.5 as the initial value. The result is shown below.

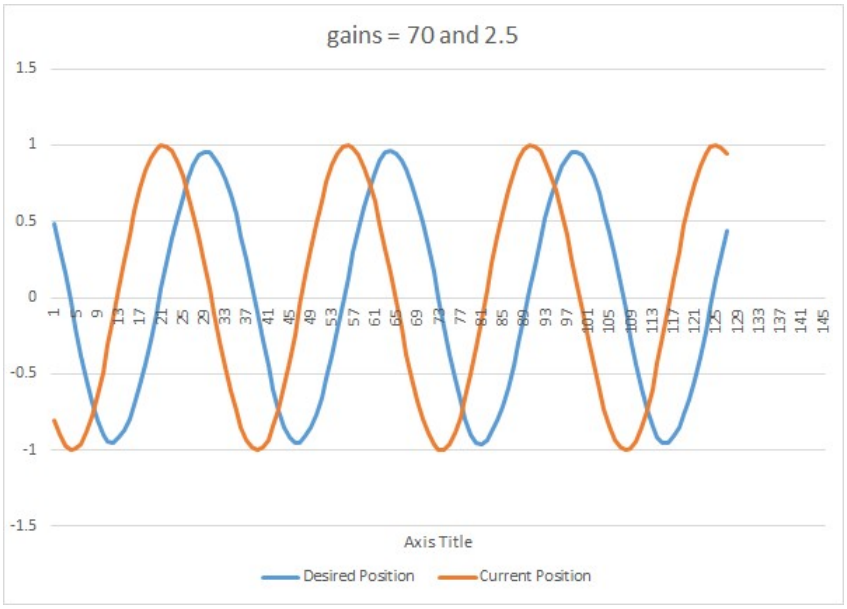

Figure 6.4: Sine wave input test for PD controller alone

As one can see from the plot, the feedback loop with the PD controller actually outputs a promising result as for the first trial. The wheel follows the command with around 10 time steps lag and some overshoot. These can be minimized through further tuning. This test validates our theory that the feedback loop alone can handle the task as position controller and the feedforward loop is not needed necessarily.

6.6.3 Tuning with balance controller and Euler Integrator

Based on the assumption that the balance controller works fine, we test the code with balance controller, euler integrator and PD controller altogether.

We fed the balance controller with zero lean angle and zero lean rate, and the result is unexpected. The controller seems much less powerful. We turn off the motor and turn the wheel to a random angle, then turn the motor back on expecting the wheel to go back to zero degree. The wheel does go back to zero but with a much slower speed comparing to the response without balance controller and Euler integrator. It looks like the gain is decreased significantly. We then increased the Kp by 300 to 800. The response became better but was still unsatisfying. Scott proposed to increase the Kp by an order of magnitude to 8000. Haoyun initially thought that is insane, but since the system is protected by the clipping mechanism, a trial won't hurt.

The result was surprisingly good. The wheel turned to 0 very quickly. Although there was drastic oscillations, at least this trial gave us an upper bound for kp. Additionally, as shown in the figure below, the steady state is zero which is great.

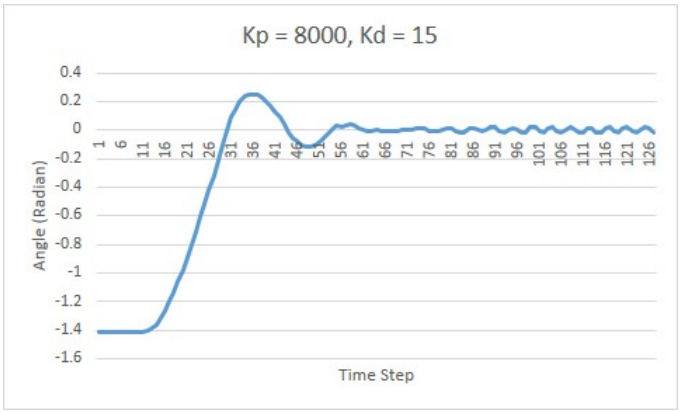

Figure 6.5: The response for Kp=8000 and Kd = 15. The system becomes marginally stable with oscillation between -0.02 to 0.02 radian.

Obviously, the next step is to decrease the proportional gain. We then tried Kp = 6000.

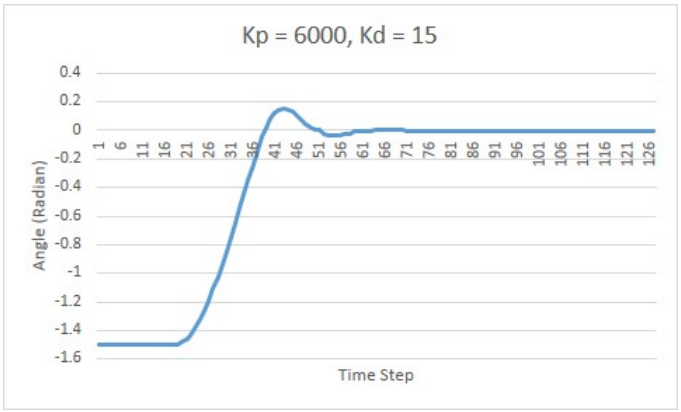

Figure 6.6: Kp = 6000 and Kd = 15. Maximum overshoot is significantly decreased.

As expected, the maximum overshoot drops from 0.25 rad to 0.15. However, the overshoot has to be decreased further. This time, we tried another method which is increasing the Kd value, or in another word, making the system more damped.

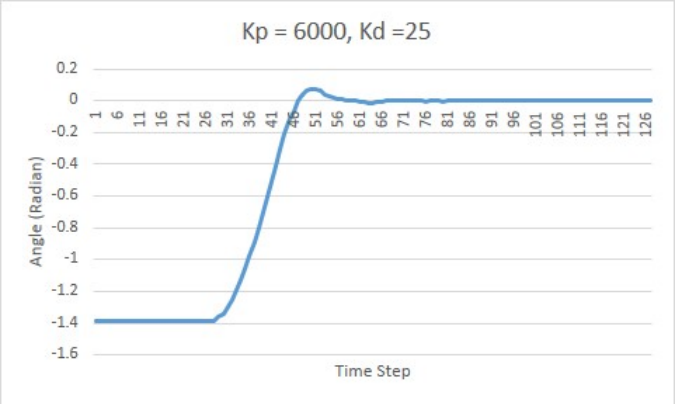

Figure 6.7: Kp = 6000 and Kd = 25.

The maximum overshoot drops to 0.07 rad which still needs to be improved. We then decreased the Kp to one third of previous value which is 2000, and changed Kd back to 15.

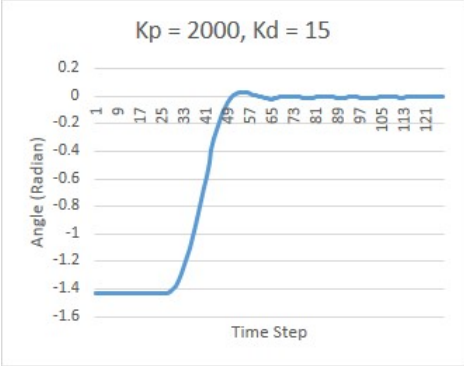

Figure 6.8: Kp = 2000 and Kd = 15.

The maximum overshoot drops to 0.03, but response becomes slower. If we define settling time as the time elapsed from the application of the step input to the time at which the wheel angle output has entered and remained in the range from 0.01 to -0.01 rad, then the settling time of previous trial is 35 time steps which is 0.35 seconds ((79-44)*10ms) and it rises to 0.40

seconds ((82-42)*10ms). Even though 0.05 seconds seems negligible, but we need the response to be as fast as possible. Therefore, we decided to increase the Kp to 3000. Moreover, the steady state error, that we worried about due to the huge reduction of Kp, actually does not increase at all. It remains zero.

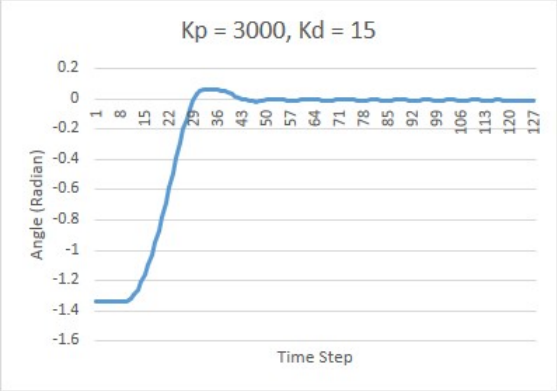

Figure 6.9: Kp = 3000 and Kd = 15.

According to the plot, the overshoot obviously gets bigger; and it is verified by the numbers. The maximum overshoot becomes 0.06 rad while the settling time declined to 0.27 seconds ((62-25)*10ms). In order to remove the overshoot, we decided to increase the Kd to 20.

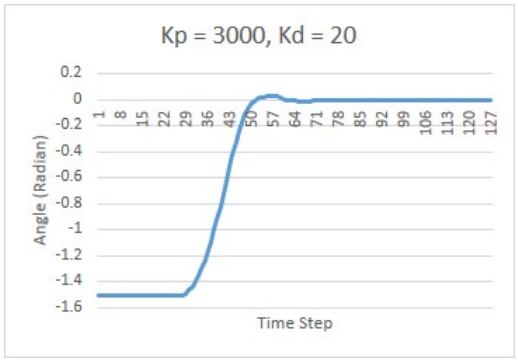

Figure 6.10: Kp = 3000 and Kd = 20.

We immediately realize this is the controller we want from observing the wheel response. The number and plot support our thoughts. The maximum overshoot is 0.03 rad and settling time is 0.31 seconds ((74-43)*10ms). Additionally, the steady state error still remains zero. Both Will and Haoyun believe that this is the optimal controller given these conditions, and no further tuning needs to be done.

6.6.4 Final Testing Results and Future Plans

For the current semester testing, we were able to roll the bike down the hallway using the IMU with only the accelerometer and gyroscope sensors enabled. This was still requesting tared euler angles from the IMU. The drift without the magnetometer enabled described in section 6.4.4 still existed, though it never was bad enough to where the bike wouldn't balance. However the bike will not drive straight down the hallway even if started straight. Because the controller is written just to balance, not to stay in a straight line, testing the bike in a narrow hallway was difficult because it would generally steer into the wall in order to balance.

Why might that be happening? First, it could be the drift or the inherent initial offset of the IMU. The optical encoder front tick could also be slightly off center (it is less likely that this is causing the steering bias). Additionally, it could just be a product of the way the system functions; the system is inherently unstable. If we use the analogy of the inverted pendulum, it will be impossible for the bike to be perfectly upright, that is, it will inherently fall to one side, that is the reason for having the balance controller. This certainty of "falling" to one side or the other before the system adjusts and balances will cause the bike to steer to one side, because currently there is no compensation or algorithm implemented to make the bike follow a particular

straight line. However, furthermore, the bike was observed to be biased to one side when doing completely free testing, that is, it was balancing but still slowly, consistently steering to one side in an arc shape.

We compensated for this consistent off center steering by implementing an RC control in order to steer the bike and avoid hitting the walls. By implementing the RC steering, we were able to make a proof of concept to show that the balance controller reacts quickly enough to balance the bicycle effectively.

During this testing, we realized a strange phenomenon about steering the bike. We initially had the front wheel controller steer the wheel left when you pushed the joystick left, as a standard intuitive controller would function. However we soon realized if we wanted the bike to go right, we would actually have to turn the front wheel left, because this would cause the bike to lean to the right, which would subsequently cause the balance controller to effectively balance the bike by turning the wheel right. So this initial turning of the wheel left had the ultimate effect of steering the bike to the right, because of the balance controller. This phenomenon will be looked into in more in depth next semester as a full steering controller is developed.

6.7 Conclusion

This semester was very heavily focused on debugging the current system established in previous semesters to finally arrive at a working prototype. The bicycle balanced itself using the balance controller when pushed at a high speed. The biggest change from previous semesters on this prototype was a different type of front motor controller to be able to accurately output the angular velocity of the front wheel, directed by the balance controller, in order to balance the bicycle. Furthermore, steps were made to prepare for future

semesters by the dynamics team to develop a trackstanding model, and the rear motor team to more accurately control rear motor speed. Additionally, younger members of the team got some valuable technical experience to be able to implement systems on new prototypes in coming semesters.