Cornell Autonomous Bicycle
Spring 2017 Final Semester Report

Professor Andy Ruina's Biorobotics and Locomotion Lab
Cornell University

May 26, 2017

Team Members				
Name	Year	Major	Team Position	Credits Taken
Arundathi Sharma	Senior	Mechanical	Dynamics/controls STL	3
Aviv Blumfield	Junior	Mechanical		3
Conrad McCarthy	Sophmore	Mechanical		3
David Miron	Sophmore	ECE		3
Dylan Meehan	Freshman	Mechanical		3
Foteini Kriezi	Sophomore	CS		3
Frances Bryson	Senior	Mechanical		3
Jared Frank	Junior	CS		3
Kenneth Fang	Freshman	Undecided	Web STL	3
Kyle Fenske	Sophomore	OR	Business STL	3
Michelle O'Bryan	Sophomore	CS		3
Olav Imsdahl	Senior	Mechanical	Steer-By-Wire STL	3
Pehuen Moure	Junior	CS	CS STL	3
Rohit Bandaru	Sophomore	ECE		3
William Murphy	Junior	Mechanical	Team Lead	3

Contents

1 Introduction

The following paragraphs are a summary of the main projects the team worked on and the main team accomplishments throughout the semester. They also include details about the current state of both prototypes, and the team goals going forward.

This semester, the CU Autonomous Bicycle Team set out with the overall goal of traversing a simple path on the Cornell arts quad with our first prototype. We also had the goal of developing our first prototype fit for a human to ride. We unfortunately did not meet our goal completely, instead the team is at the stage of testing navigation controllers on the first prototype, fine tuning the balance controller, and overall is well equipped to take the progress this semester and make more measurable of improvement in future semesters. The goal of next semester and the summer will be to finish our initial goal of traversing the arts quad, then move into track-standing and better evolution of the balance controller now that a standardized testing procedure is in place. The team also doubled in size this year, and a lot of growing pains and progress were made learning how to handle a team of 15 members. This included forming sub-teams, creating a system of peer-evaluations to hold each other accountable, having formalized sub-team meetings and general body meetings all to make each individual more productive. We are confident we will build on this team foundation in future semesters.

For the first prototype in order to work towards traversing the arts quad, there were multiple components unfinished on both the Hardware and Software systems of the prototype. Hardware Projects this semester included:

- Implementing Physical GPS chip and Verifying Data Received
  - Progress: Currently receiving data, but not consistently at the rate that we believe is necessary (minimum 5 Hz). Further data filtering is needed as well as ensuring data is being received consistently
- Debugging and Ensuring consistent Landing gear performance
  - Progress: Landing gear now consistently functions and respond to RC inputs
- Testing and tuning/improving balance controller
  - Progress: New testing procedure was developed with standardized formula for identifying goals of tests, a procedure for working with the prototype, and recording data.
- Develop and tune rear-motor controller

Software Projects this semester included:

- Restructuring the Arduino code which interacts with each hardware component of the system
- Restructuring and modernizing the current data transfer and processing system

  - Progress: Implemented a system called Robot Operating System, which many robotic systems use. This system architecture is designed to facilitate data transfer between "nodes" which can be segmented components of the system. After developing the navigation algorithm using python code, the need for a streamlined communication protocol between the Raspberry Pi (which locally hosts the navigation code) and the Arduino (which hosts the code which interacts with the prototype) became apparent. ROS also allows for streamlined data recording.

- Developing a navigation algorithm in python to input the current state of the bike on a map and output a steering command to the front motor with the goal to converge onto a desired path
  - Progress: testing for the navigation algorithm was performed throughout the semester using a 2D simulator, then a dynamic model which yielded more problems. It was then refined, however the newest algorithm has only been tested a couple times with the actual prototype and further troubleshooting will occur this summer 2017.
- Implementing a database to store and easily retrieve testing data and notes

  Progress: the basic structure of the database is written, it is being worked on this summer
- Implementing a web system which inputs desired Waypoints for navigation
- Developing a public team website

Steer-By-Wire

The team also began developing a new prototype this semester, and formed a new sub-team called Steer-By-Wire. The long term goal of the Steer-By-Wire Sub-Team is to develop a self-stabilizing bicycle which a user can ride and steer remotely. For more detailed goals and progress please see Mission in section 2.1.

2 Steer-By-Wire

2.1 SBW Subteam Final Report

Olav Imsdhal, David Miron, Conrad McCarthy,
Dylan Meehan, Frances Bryson May 2017

3 Mission

The steer-by-wire (SBW) subteam is building a new self-stabilizing bicycle that will be rideable by a human, and implements a steer-by-wire method of steering, in which the user remotely, rather than directly, controls the turning of the bicycle's front wheel. The front wheel itself will turn according to the lean of the bicycle, and the steering input from the user.

The work of the SBW team this semester has consisted of obtaining new parts for this bicycle, designing and building the mechanical system, and undertaking preliminary testing of the front motor controller, including investigating PD controller gains.

Future work of the subteam will include fine-tuning of the PD controller gains and investigation of the balance controller gains, implementing a four-quadrant motor controller, and building and programming the steer-by-wire mechanism.

4 Mechanical Design

Conrad McCarthy

4.1 Power Transmission

We considered initially four different power transmission systems for controlling the front wheel via our front motor: direct gear to gear contact, planetary gear system, belt drive, and chain drive. We had two primary design factors which influenced our decision. First, we wanted to keep open the end of the shaft extending through the front fork. This would allow us to couple the shaft with a handlebar to test differences between the normally fixed connection of handlebar and wheel and our steer-by-wire system. Second, we needed to reduce the motor's angular speed as to offer a finer degree of angular position control and to produce a higher torque to counteract friction between the wheel and the ground.

To satisfy our first constraint, the motor needed to be off-axis with respect to the shaft of the front fork. We therefore first thought it necessary to rule out the planetary gear system, since it would have to connect the motor and the shaft through the fork while maintaining alignment through the shaft's vertical axis.

After deciding on an off-axis power transmission setup, we knew we needed to devise a mechanical system of machined parts to secure the motor to the bike frame. Though a direct gear to gear connection offers a high power transmission efficiency and low slippage conditions, the gears must be aligned perfectly. We decided it would be too difficult to ensure this while machining the attachment of the motor to the frame, and thus did not use choose this system.

We used our second design constraint to decide between a chain drive and belt drive. We approximated the ratio of angular velocities between our motor and wheel by comparing our motor specifications and wheel size to those of the other bike. Our motor has a maximum RPM of 3000 and maximum torque or 310 oz-in, while the other bike's motor has a maximum RPM of 1700 and maximum torque of 710 oz-in. Our front motor will have to provide a higher torque than the other bike since a rider on the bike will increase the normal force between the wheels and the ground, and thus increase friction opposing the turn of the wheel. Additionally, our bike has a larger wheel, which has a larger moment of inertia (directly proportional to the mass of the wheel and squared radius) and longer moment arm, again increasing the torque required to turn the wheel. We therefore needed a higher gear ratio than the other bike to step down our motor's angular velocity but increase the torque about the wheel.

The other bike had a gear ratio of approximately 5:1. Initially we wanted to double this ratio, since our motor's RPM is approximately twice that of the other and can offer approximately half of the torque. To increase the ratio of a belt or chain system, one must increase the ratio of the number of teeth on the pulley or sprocket. Since the number of teeth per circumferential distance must be constant between the two pulleys or sprockets, increasing the ratio of the number of teeth effective increases the ratio of sprocket or pulley radii. This poses an additional physical constraint in our system. Since the physical size of the smaller sprocket or pulley is constrained by the size of the motor's spinning axle, the increase in the ratio means an increase in the radius of the larger sprocket or pulley. To ensure proper contact between the chain and the sprocket or the belt and the pulley, the distance between the sprockets or pulleys must increase. This means that as we increase our gear ratio in these two systems, we must increase the distance of the motor from the front shaft. By doing so, the moment arm that the weight of the motor creates increases, which adds to the challenge of attaching the motor to the bike frame. If the distance becomes too large, the attachment can become excessively obstructive to the rider as well.The advantage of a chain drive system compared to a belt drive is that it has a lower minimum requirement for the number of teeth the chain needs to engage in order to avoid slippage and provide proper torque. Therefore, with a chain drive, we could keep the motor closer to the bike's frame.

We chose our ratio based off the maximum gear ratio we could obtain from ordering parts on McMaster Carr. Our gear ratio is 7:1. Going forward, we may increase this gear ratio because we need to overcome the torque of the ground on the wheel. While testing our balance controller, we noticed that our motor could not turn the wheel when offset from its desired position while a person sat on the bike. A higher gear ratio also makes sense for our motor since it has a high maximum RPM but a relatively low maximum torque. Additionally, the motor frequently surpassed a current of 30 amperes, before obtaining its top speed, which triggered an over current protection feature in the motor controller and subsequently stopped powering the motor (See HERE for motor controller spec sheet.). Next semester we will look into ordering a planetary gear system to increase our gear ratio. The system would sit on top of the off-axis motor and still require a chain drive or belt drive to transmit power to the shaft.

4.2 CAD Software

Fusion 360 is a free, cloud-based computer-aided design (CAD) software application. We chose Fusion 360 to model the bicycle primarily because its promotes collaboration within a group setting. Since it is cloud-based, one can upload and access designs easily across different device platforms. In a group, it is easy to create a shareable folder in which all members can contribute to the design. Additionally, Fusion 360 incorporates a timeline feature, which makes it easy for group members, and individual users, to track changes made to the design over time.

Fusion 360 is perfect for conceptual modeling but has limitations in dynamic simulations. One must define “Contact Sets” to simulate interactions between components upon contact. For even simple interactions, the software requires high-processing which can be overbearing and limit the graphical functionality of the program. We could not simulate the motion of the chain for this reason, though we could still correctly implement a rotational ratio between the sprockets.

4.3 Attachment of Motor, Encoder, and Handlebar

To implement our chain drive system, we designed a method to attach the motor to the bike frame. In the development of our design we had to account for the encoder’s attachment to the shaft through the fork and the future attachment of a handlebar and a second motor to complete our steer-by-wire system.

First, we removed the bike’s handlebar and replaced it with a shaft that extends through the front fork and matches the diameter of our encoder and large sprocket. The shaft is secured via a through bolt at the cusp of the fork (Figure 1).

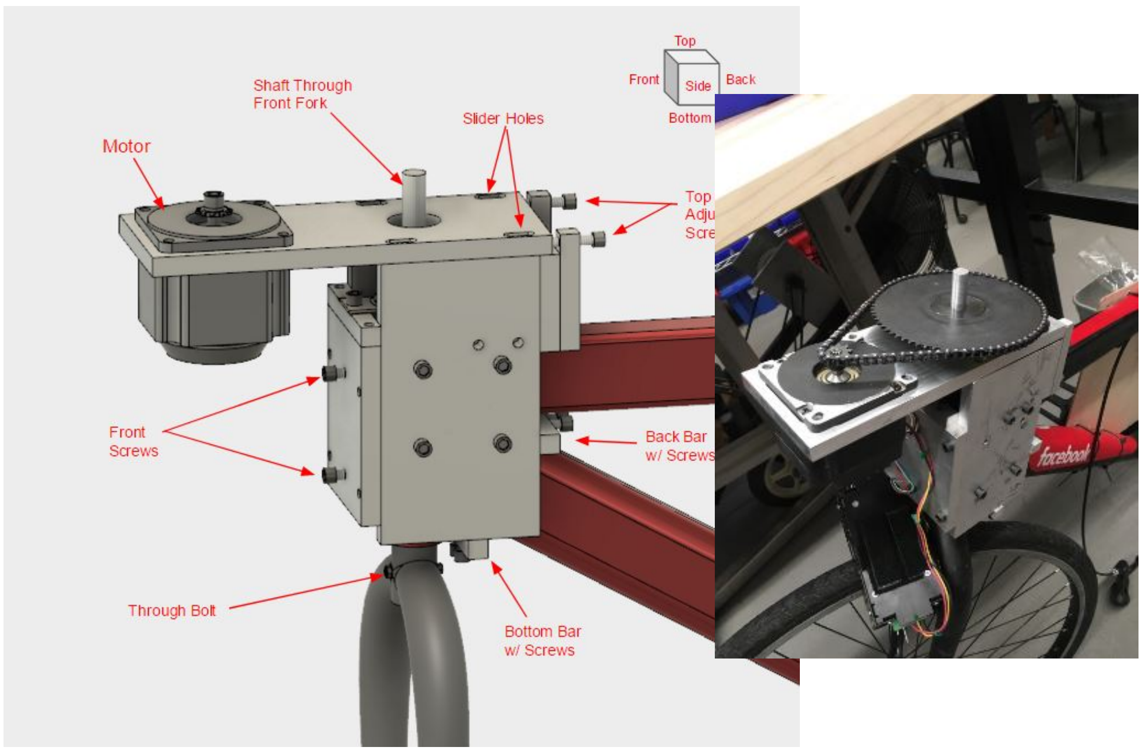


Figure 4.1: Front view of mechanical assembly, depicted in Fusion 360 and real life

We thought it best to have the motor in front of the bike to minimize its obstruction to the rider. The motor had to be perpendicular to the fork’s shaft so that the two sprockets were coplanar. Because of the bike’s irregular shape and asymmetric geometry, we designed a box to fit around the frame with methods of adjusting its alignment with respect to the fork's shaft as shown in Figure 1.

The motor sits in a hole cut out to its geometry on the top face of the box. We cut slider holes for the screws securing the top plate to the side plates so that we could adjust the motor's distance from the shaft to ensure tension in the chain. On the box's back face, we added two bars to hold screws ("Top plate adjustment screws") which press up against the top face to maintain its distance and counteract the force of the chain on the motor.

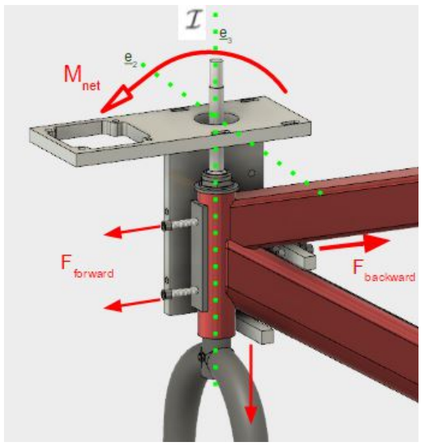


Figure 4.2

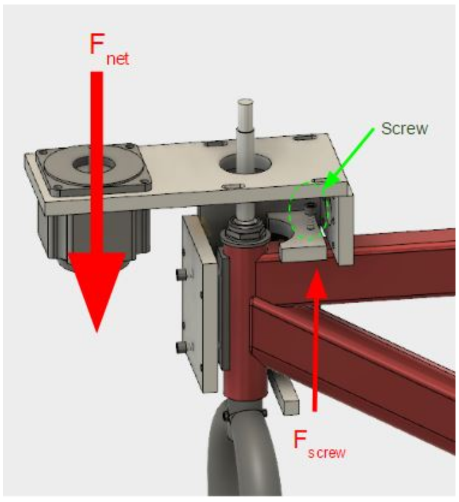


Figure 4.3

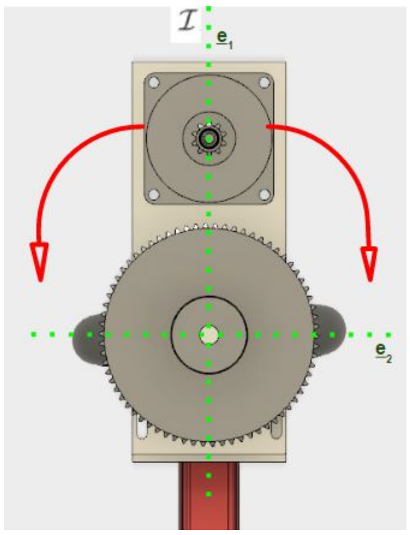


Figure 4.4

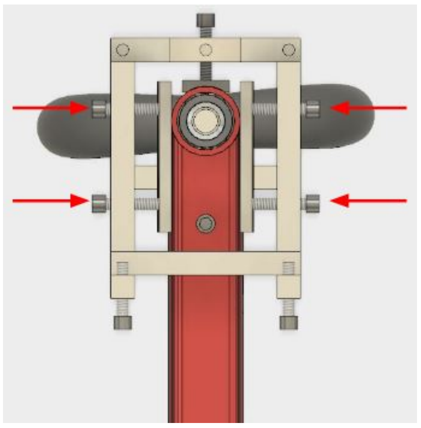


Figure 4.5

We adjust the box's orientation from all six sides (See Figure 1 for orientation reference cube). The motor causes a moment (Mnet) about an axis perpendicular to the bike's frame ("e2" axis in Figure 2). Screws in the front face of the box press up against the bike frame via a plate which matches the frame's curvature. These screws counter the majority of the moment (Mnet) but cause a forward force on the box (Fforward). We added a bar with two screws which press directly into the bike's frame on the back (Fbackward) and bottom faces of the box to negate the forward force and help counter the motor's moment (Figures 1 and 2). To aid in supporting the total weight of the motor and box itself, we add a screw on the inside of the box which presses down on the bike frame to provide an upward force ("Fscrew", Figure 3). Four screws on the left and right faces of the box press against flat plates and the bike frame to support moments about an axis through the shaft caused by the chain ("e3" axis, Figures 4 and 5). We tighten the screws at each of these various locations to ensure that the two sprockets are coplanar and the chain drive operates smoothly.

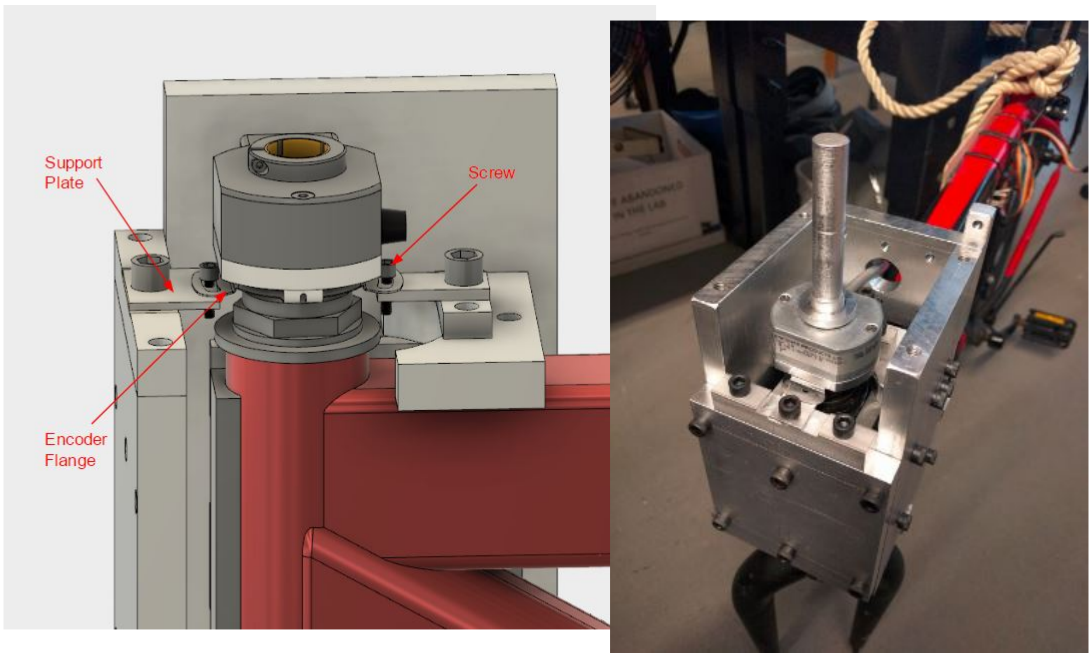


Figure 4.6: Encoder attachment inside box

The encoder sits inside the box itself to maximize the area on the shaft above the box. Two plates support the encoder and prevent its rotation with screws through its flanges as shown in Figure 6.

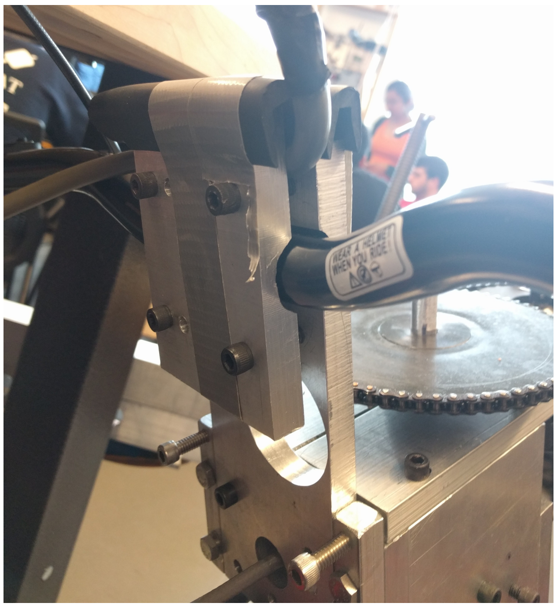


Figure 4.7: Rigid handlebar attachment

For testing purposes, we added a rigidly mounted handlebar which cannot rotate with respect to the frame. We attached a plate to the back face of the box with a groove for the handlebar which we secured with another grooved plate. Figure 7 shows how the two plates act as a clasp.

4.4 IMU

Dylan Meehan: dem292 and David Miron: dm585

The Steer-By-Wire Bicycle uses the same IMU (Inertial Measurement Unit) as the Autonomous Bicycle, specifically, a Yost Labs 3-Space Embedded IMU on an Evaluation Board. The IMU implementation is the same as on the Autonomous Bicycle, bar one exception. The roll angle of the SBW bike is offset by 1.57 radians (90°) to account for the IMU being mounted vertically (See **??**). This adjustment is done by subtracting 1.57 from imu_data.angle after the updateIMUData() is called[1]

The IMU is mounted vertically to not exceed the width of the frame so to not interfere with the legs of someone riding the bike. The IMU is mounted with its short axis up (as opposed to long axis) to minimize differences between the SBW bike and the Autonomous Bike. Mounting the IMU with the short axis vertical is equivalent to rotating it 90° CCW about its z-axis (from a flat starting position). Thus we can use the same output (roll) as the IMU as the autonomous bike, but with an offset. Were we to mount the IMU with its long axis vertical, we would need to measure a different IMU output (ie pitch) in order to get the lean of the bicycle. This would be more difficult than simply offsetting the roll angle. Thus, we mounted the IMU like so. Note that the location of the IMU on the bike frame does

[1]Specifically the IMU offset is accomplised with:

```
roll_t imu_data = updateIMUData();
imu_data.angle = imu_data.angle - 1.57; imu_data.rate = imu_data.rate;
```

not affect the IMU data. The IMU measures roll angle (and roll rate) and since the bike frame is a rigid object, roll angle is the same for each height along the bike frame. Therefore, the height we mount the IMU at does not effect the data.

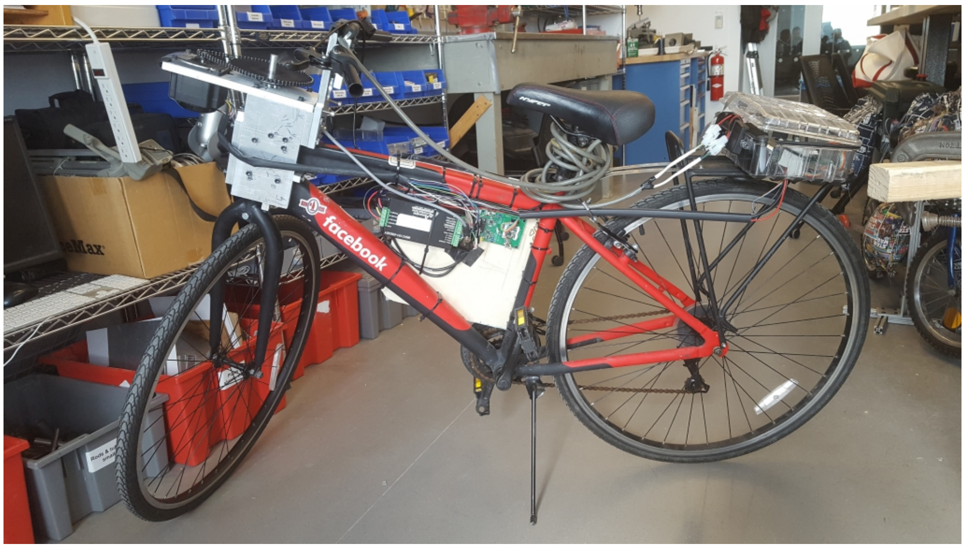


Figure 4.8: Components mounted in front triangle of bike frame.
Note the components are not wider than the frame itself, so they do not interfere with the legs on someone riding the bicycle.

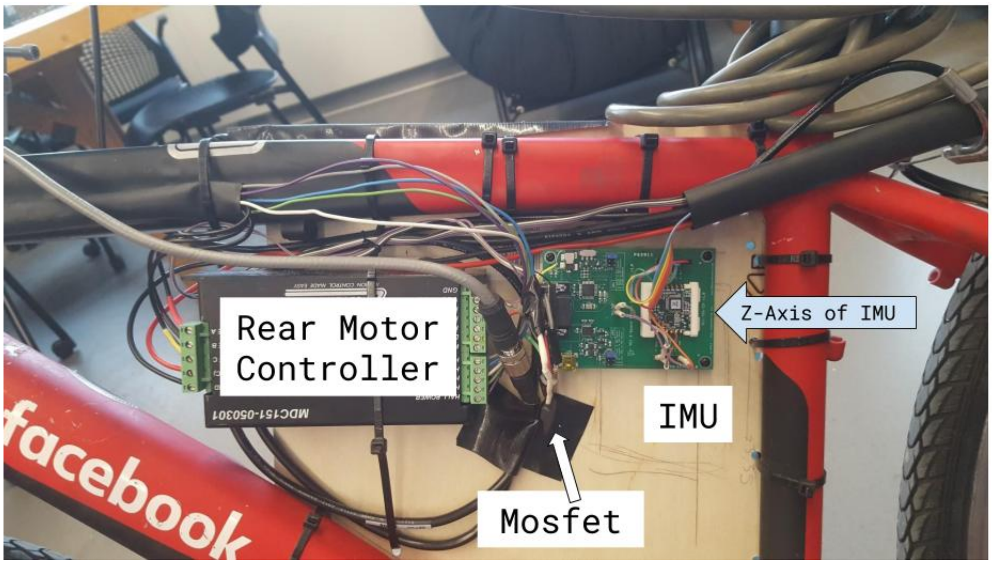


Figure 4.9: IMU and Front Motor Controller Mounted

5 Printed Circuit Board

David Miron: dm585

The flow of signals through the self-balancing bike prototype that Cornell Autonomous Bike made is controlled by an Arduino connected to a printed circuit board. A printed circuit board, PCB for short, is a copper sheet on top of a non-conductive material, and is used to connect electrical components. In our case the PCB we use is one that is designed to be compatible with the Cornell Bike and Sail teams.

The first thing to understand about the PCB is the placement of components on the board. On the Autonomous Bicycle Google Drive in the miscellaneous folder there is another folder titled "PCB Eagle Files". Click HERE for Eagle files. Eagle is a piece of software developed by Autodesk. It is used to design printed circuit boards. The files on the drive can be opened with a free download of Eagle, and they show the design of the printed circuit board used on the self-balancing bike. The Eagle files can be used to find the pin connections for the integrated circuits, circuit elements and, most importantly, the Arduino.

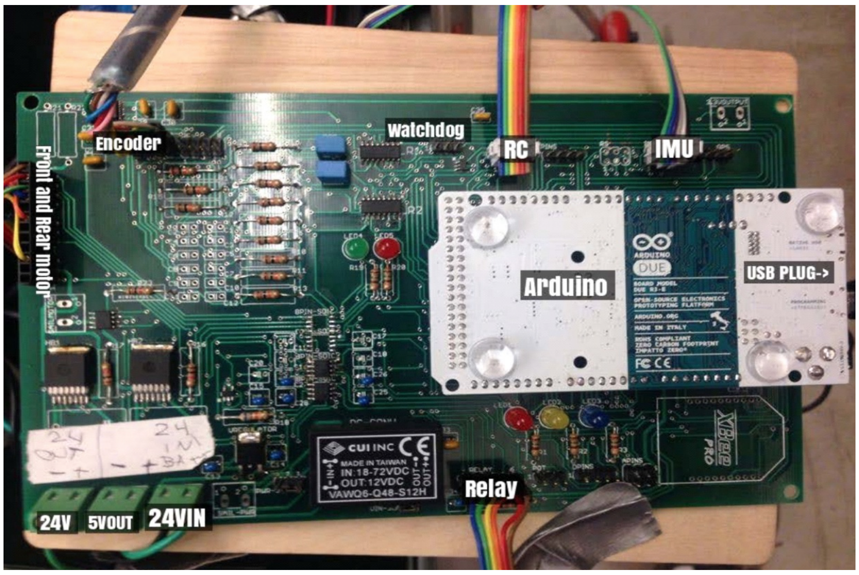


Figure 5.1: This image shows the PCB used on the Autonomous Bicycle prototype. This is the PCB used as a reference for the Steer by Wire PCB.

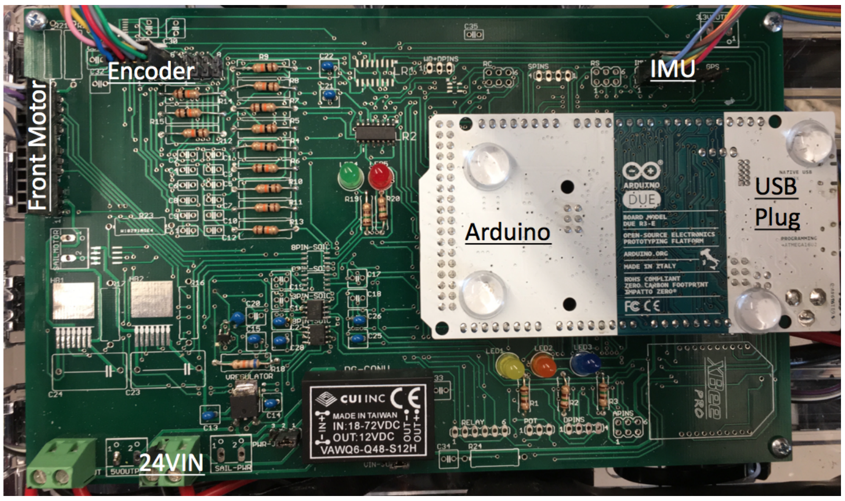


Figure 5.2: This image shows the PCB created for the Steer by Wire Bicycle Prototype. (Note the blank board was purchased from PCB works, and then circuit components were soldered onto the board.)

The images show that the Steer by Wire PCB uses less circuit components than the other prototype. The next step in understanding the PCB is analyzing the flow of signals throughout the board. There are two main areas to focus on, the signals that power the front wheel motor controller, and the signals that control the encoders and IMU.

5.1 The Front Wheel Motor Controller

The PCB can be powered with one of the battery packs found in lab that can hold of a charge of up to around 29V +/- .5V. The battery loses charge with time, and is useful until it drops to around 26V. The battery is connected to clips that are wired to "24VIN" on the image of the PCB above.

The batteries 26V – 29V flows to a 5V regulator and a 12V DC-DC converter. The 12V DC-DC converter takes the batteries Voltage as an input and outputs 12V DC. The 12V flows to the Arduino to power it when it is not connected to a computer via USB. The 5V regulator takes the battery's voltage as an input and outputs 5 volts. The 5V output is used to power different integrated circuits on the board, that require a Vcc of 5V.

One of the integrated circuits is a digital isolator. The digital isolator used takes two 3.3V PWM signals as inputs, and outputs the PWM signals scaled to a 0-5V range. Another integrated circuit element powered with 5V is a low-pass analog filter. A low-pass filter outputs input signals that have a frequency below its specified cutoff, and attenuates (decreases the intensity) signals with frequencies above its cutoff. Lastly, 5V powers an Op-Amp, which is used to amplify signals.

Only one of the inputs and outputs of the digital isolator is used. This signal comes from the Arduino. This signal is a PWM that controls the speed of the front motor. The signal outputted from the digital isolator passes through the low-pass filter. The output of the low-pass filter passes through the op-amp, and finally the op-amp's output is sent to the motor controller.

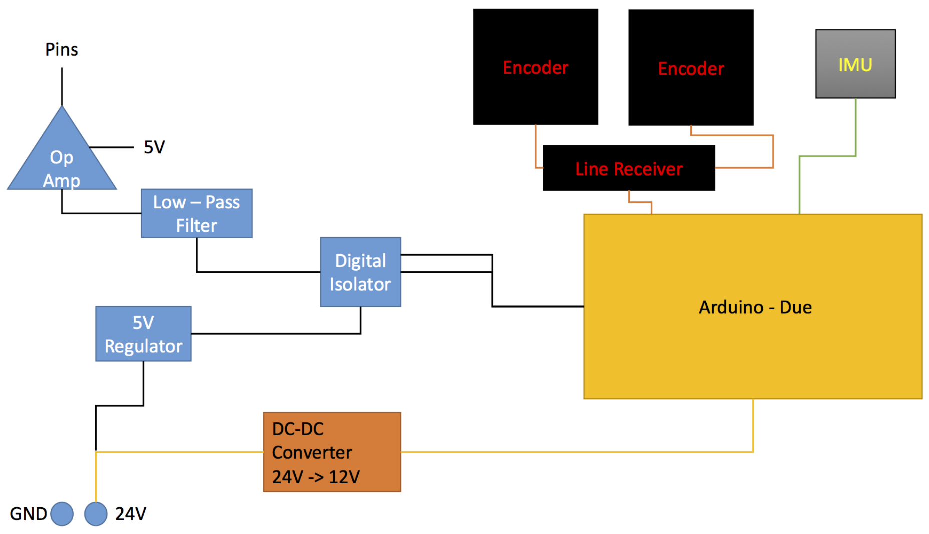


Figure 5.3: This image shows a simplified diagram of signal flow between the Arduino, front wheel control, encoder and IMU. Note that two encoders are included but only one is being used on the bike at this time.

5.2 Encoder and IMU

The Steer-by-Wire bike utilizes one encoder (Encoder Products Company Model 260) on the shaft of the front wheel. The encoders are powered with 5V. The 5V comes directly from the Arduino. The encoder outputs on six pins. To understand the outputs, and how we make sense of them, please read Chapter Two of the Spring 2016 Final Report written by Weier Mi and Stephanie Xu. The output signals from the Encoder are inputted to the Arduino to record speed and position of the motor.

Click HERE for Spring 2016 Final Report.

There is one IMU, inertial measurement unit, on the bike to record the bike's current angle. It outputs on six pins to the Arduino, providing information on the angular rates and current angle of the bike. For more information on what information the IMU records, please read Chapter 1 Section 2 of the Spring 2016 Final Report, written by Jason Hwang. The IMU is also powered by a 5V output of the Arduino.

5.3 Current State of PCB

The PCB is currently mounted in a box that rests on the back of the Steer by Wire Bike. The PCB is equipped to fully control our motor controller and read data form the IMU and Encoder. In the future we may add functionality for another encoder and motor controller. These components would be used for the steering assembly. We will need both a motor and encoder to create a resistance to a rider turning the handlebars, and to make the front wheel responsive to handlebar rotation. The encoder and motor pair are important to simulate a real life riding experience. When riding a bicycle, there is resistance to turning the handlebars because they connect directly to the wheel that is experiencing the force of friction from the ground. Our handlebars do not directly connect to the front wheel, so we need a motor to create resistance in the form of spinning the motor against the direction the rider tries to turn the handlebars. The encoder will tell us when and how fast the rider is turning the handlebars, so the motor can respond with an appropriate speed and direction.

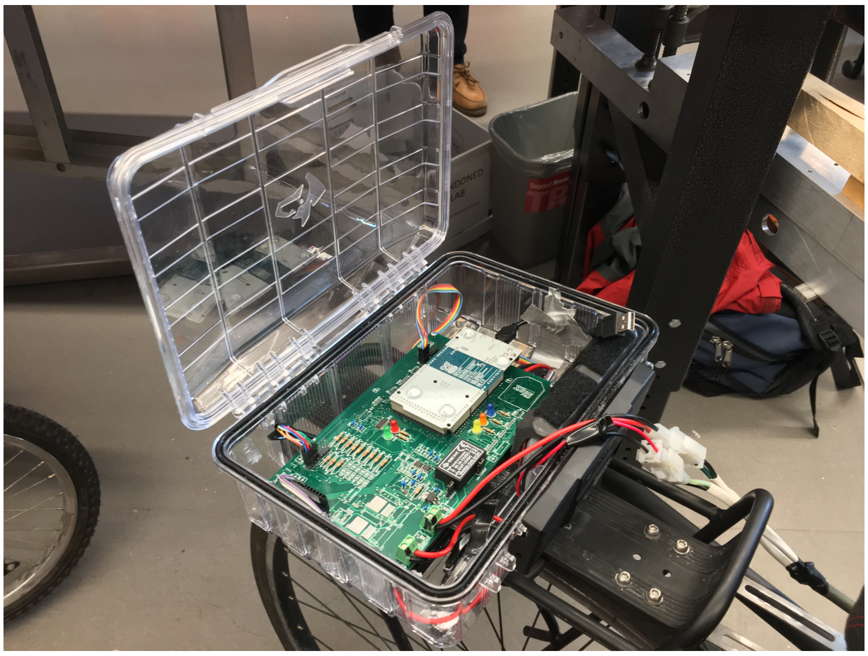


Figure 5.4: This image shows the PCB created for the Steer by Wire Bicycle Prototype mounted to the back of the bike. The necessary circuit elements to control the electromechanical function of the bike are all implemented.

6 Front Steering Motor Implementation

To balance the Steer by Wire bike we need to control the rotation of the front wheel. To control the front wheel we have implemented a Brushless DC motor and motor controller.

6.1 Brushless DC Motors

David Miron: dm585

The motor we are using is a 24V Brushless DC Motor rated to 3000 RPM from Anaheim Automation. Model number is BLYSG342S-24V-3000-R75. See HERE for specification sheets.

This motor is a brushless DC motor, meaning that these motors do not have any carbon brushes. This is advantageous; carbon brushes usually wear with time, making brushed motors less reliable for long term operation. Brushless motors spin when a voltage is applied to its coils - the application of a voltage creates a magnetic force between a permanent magnet, and electromagnet causes a rotor to spin around a stator. In a brushless motor, the rotor is the permanent magnet that surrounds the stationary stator, which has coils around it that are used as electromagnets. To keep the rotor moving there are multiple "poles" within a motor. A pole is simply two coils that are aligned across from each other and connected. When voltage is applied to a pole, one side acts as a positive magnetic field, while the other acts as a negative magnetic field, hence the name "pole".

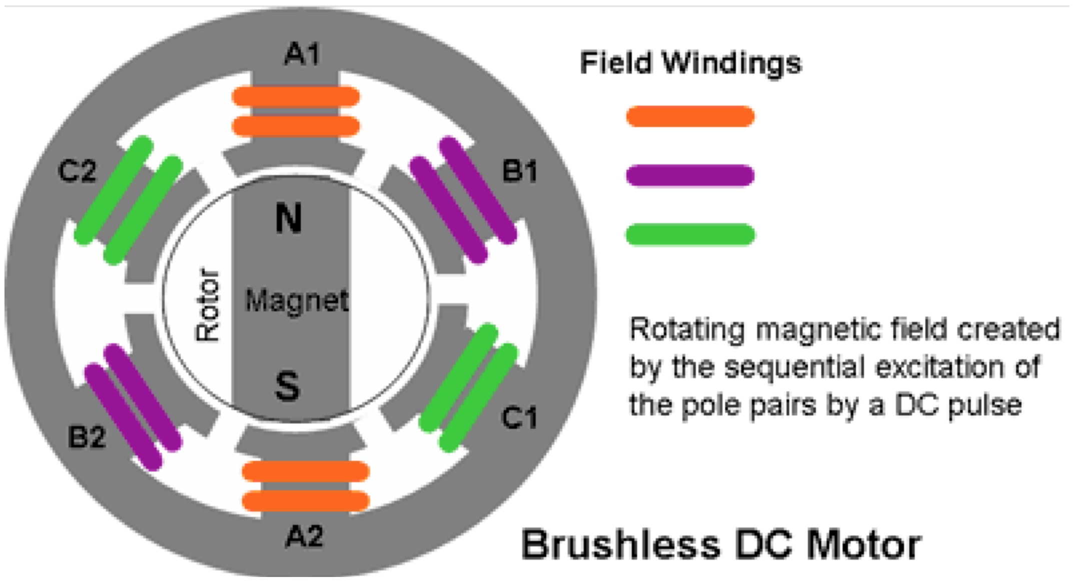

A1
Field Windings
C2
B1
N
Rotor
Magnet
S
Rotating magnetic field created by the sequential excitation of the pole pairs by a DC pulse
B2
C1
A2
Brushless DC Motor

Figure 6.1: This image shows the inside of a brushless motor. Notable features are the rotor in the middle (permanent magnet) surrounded by the stator that has the coils around it

As the rotor spins, a motor controller tells the motor which coils need voltage so the rotor is not only attracted to one pole, but also repelled by another, which increases power efficiency. Our motor uses eight poles, which become charged as the rotor moves to have a constant magnetic force to spin the rotor. See HERE for more information.

Internally, a 3-phase motor can be configured to a "Wye" or "Delta" configuration. The primary advantage to the "Wye" configuration, also known as the Star configuration, is that the phase-to-neutral voltage is equal in all three legs. The arrangement is a parallel circuit in a shape of the letter Y, where all windings are connected at a central point, and power is applied to the remaining windings. We use a "Wye" configuration for our motor, as it is the default.

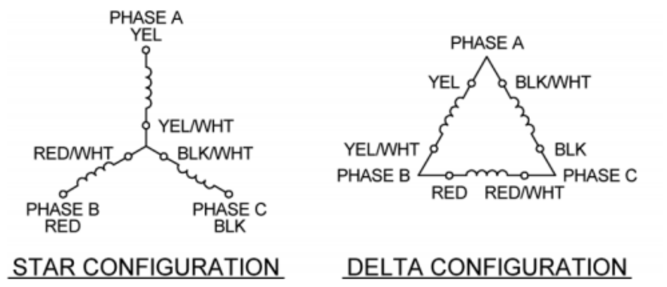


Figure 6.2: This image shows the configurations. Note Star and Wye are the same.

This brushless DC motor is controlled by a motor controller, which utilizes The Hall Effect, and three hall sensors within the motor. Hall sensors measure electric field within the Brushless DC Motor, and feeds data to the controller. The data is in the form of voltage, when a sensor is triggered a pulse is sent to the controller for that sensor (there are three sensors A, B and C) This data is interpreted by the controller to determine speed and positioning of the motor. See HERE for more information.

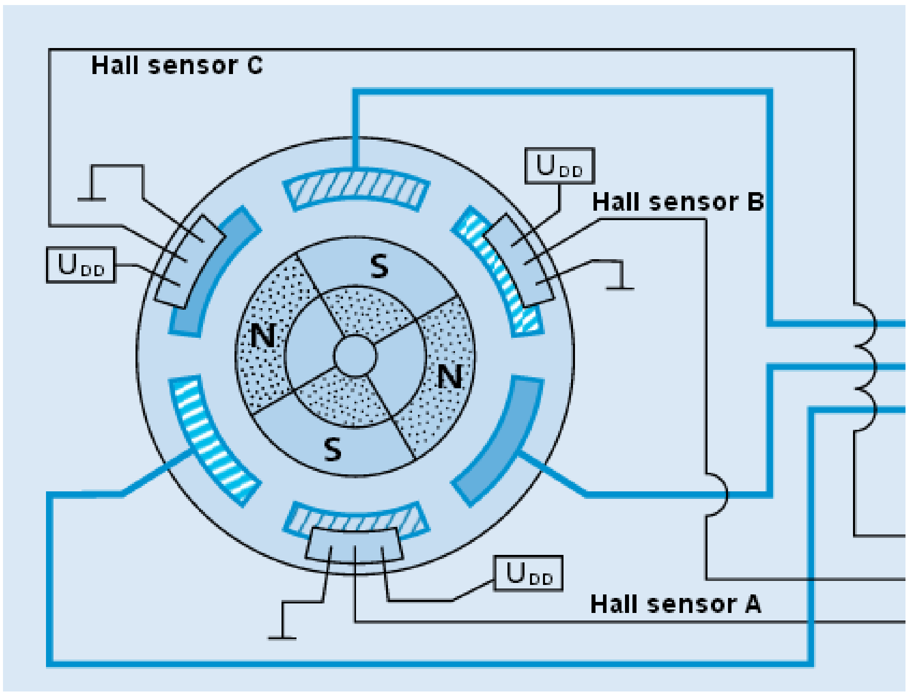


Figure 6.3: This image shows the location of hall sensors within a motor

6.2 Motor Controller

David Miron: dm585

The motor controller we are using is also from Anaheim Automation. Model number is MDC151-050301. Notable specs: 50V max, 30A max, velocity controlled, 2-Quadrant Operation (See **??** for discussion of 2-Quadrant Operation). See HERE for specification sheets.

The main purpose of a Controller/Driver is to control the speed and direction of the motor. This controller uses voltage to control the speed of the motor. The speed of a DC motor is directly proportional to the voltage input on its speed control pin.

When a supply voltage is constant the motor controller does something special. It varies the average voltage sent to the motor to control the speed. It could do this by adjusting the voltage sent to the motor, but this is inefficient to do. A much more efficient way to do this is, to switch the motors supply on and off very quickly. (efficiency in this case is saving power) When the switching is fast enough the motor will only recognize the average effect. It will not notice that it is actually being switched on and off. The average speed of the motor increases, as the amount of time that the voltage is on increases compared with the amount of time that it is off. This is how our motor controller translates the input signal to driving the motor at a specified speed.

6.3 Motor Controller Inputs, Outputs and Settings

David Miron: dm585

The first step to testing was reading through the documentation for the controller and motor. This is extremely important, and I suggest that even with the summary I provide, anyone trying to use these components should familiarize themselves with the instructions and specification sheets. I have provided links earlier in this section.

The motor controller has two groups of switches, and three potentiometers, which work as variable inputs, on its side. The potentiometers are used to control a variable resistance for inputs to the controller.

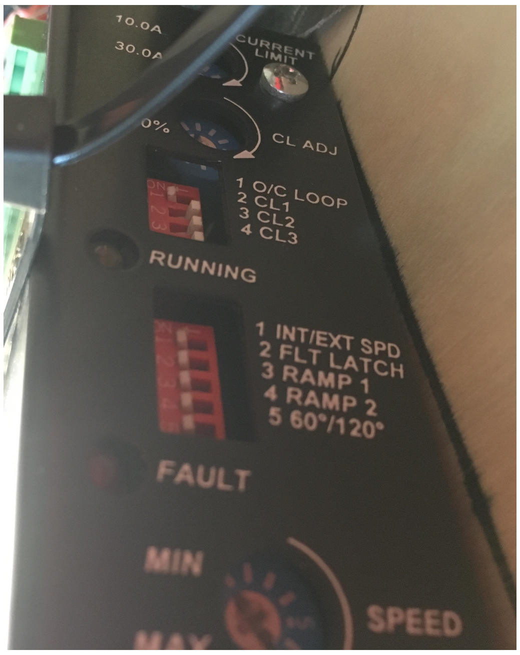


Figure 6.4: This image shows the side of the motor controller

The two LED's labeled running and fault tell us when the controller thinks the motor is spinning as expected and when the motor acts unexpectedly, respectively. When the fault light turns on, the motor usually stops spinning to prevent any damage to components.

6.3.1 The first group of switches is:

(a switch is "on" if it is flipped to the left)

•1 O/C Loop - When this switch is on we are operating with an Open Loop

•2 CL1, 3 CL2, 4 CL3

CL1–4 are switched on or off depending on the motor being used, and whether open or closed loop operation is needed. For our motor we want ON, OFF, OFF, OFF. This enables the motor to run in an open loop (vs closed loop operation. Open loop means the motor is NOT constantly adjusting speed. In closed loop operation there is feedback, telling the controller the actual speed of the motor. The controller compares this to the expected speed and adjusts constantly to maintain a set speed.) CL1-3 are needed to optimize closed loop operation of a motor, but we do not need to do this in open loop operation.

6.3.2 The Second group of switches is:

•1 INT/EXT SPD

This switch determines whether we will be operating based on an internal or external speed. We switch it on to control speed externally with a voltage input.

•2 FLT LATCH

When on this switch provides an over current protection to the motor by implementing an over current latch. We turn it on to protect the components from high currents.

•3 RAMP 1, 4 RAMP 2

Ramp 1/2 are used to adjust the acceleration of the motor - not important to us at this time.

•5 60°120°

This switch is very important. When it is on the controller is expecting to rotate a motor with a 120°phase angle, which is the phase angle of our motor. We keep this switch ON.

6.3.3 The potentiometers are:

•Current Limit - sets the limit for current - we turn it all the way to 30A.

•CL ADJ - adjustment for closed loop operation - not used

•Speed - this potentiometer only controls speed when operating with internal control - not used

6.3.4 Electrical Inputs and Outputs:

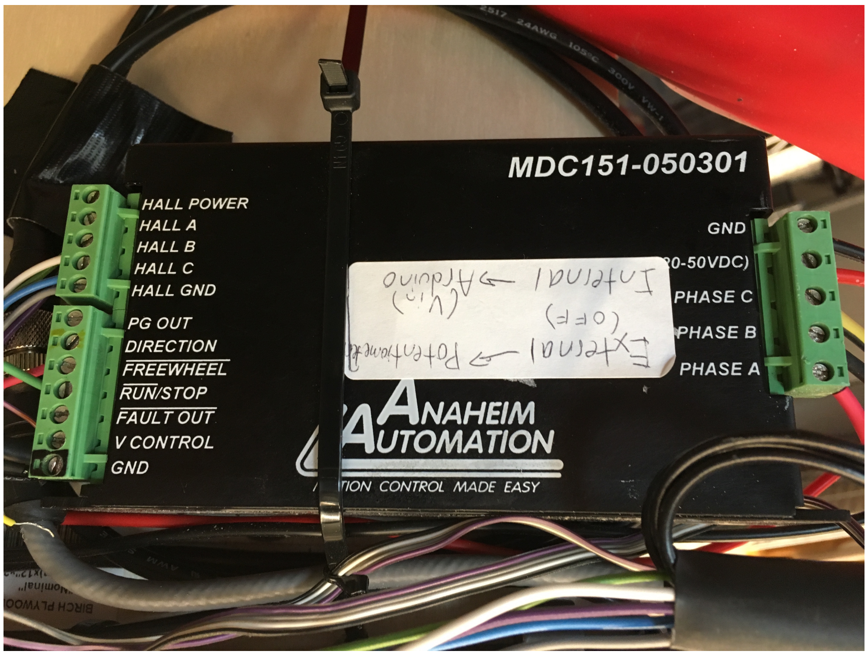


Figure 6.5: This image shows the front of the motor controller

•Hall Power (output) - power the hall sensors

•Hall A-C (inputs) - Hall A-C are the inputs from the motor's hall sensors, so position and speed can be calculated by the controller.

•Hall GND (output) - ground the hall sensors

•Direction (input) - control the direction the motor is spinning by connecting this pin to ground or an open circuit. To recreate a ground and open circuit, we used a transistor. This will be explained later

•Run'/Stop (input) - enabling this pin allows the motor to run, when it is not enabled the motor stays still. Like the direction pin, this pin functions by connecting it to an open circuit or ground. A transistor was used for this pin, and will be explained later.

•V Control (input) - this pin takes an input between 0-5V and controls the speed of the motor.

•GND (input) - ground for the inputs Direction through V Control

•Phase A, B, and C (outputs) - these pins are wired to the motor and control the activation of its three phases. The controller uses these outputs to spin the motor.

•Freewheel, PG Out, and Fault Out are pins that we did not use. See the specification sheet of the controller for more info on these.

6.4 Connecting the Motor Controller to the Motor

David Miron: dm585

The motor has many wires coming out of it. These wires connect to the motor controller's Hall A-C and Phase A-C. All of the wires had to be extended so they could reach the controller. We kept things color coded.

•Hall Power, A, B, C, and GND connect to the Purple, Green, White, Blue, and Grey wires leaving the motor.

•Phase A, B, and C connect to the black wires tipped with Red, Black, and Yellow shrink wrap leaving the motor

Note: The Phase wires are thicker than the Hall Wires.

6.5 Motor Controls

Dylan Meehan: dem292

6.5.1 Direction Control

The direction of the front motor is controlled via the Direction input on the motor controller. Direction has two setting: connected to ground (closed circuit) or disconnected from everything (open circuit). Simply, to change direction we needed to use a switch to open or close the circuit. This is achieved using a MOSFET transistor controlled by the Arduino (see next paragraph). Specifically, pin 11 controls direction with a HIGH Ardiuno signal corresponding a counterclockwise motor rotation.

In order to mimic a switch, we use a MOSFET transistor (**??**). The Gate of the MOSFET is connected to the Arduino signal, pin 11. The Drain is connected to the Direction input on the motor controller. The Source is connected to ground. A HIGH signal from the Ardiuino (to the Gate), electrically connects Drain and Source. A LOW signal from the Arduino disconnects Drain and Source. Both the Gate and Drain are run through 1kΩ resistors. Note: when testing the direction pin, the overall bike battery power must be on. Even if we do not want to spin the wheel, the battery power must still be on. An Op-Amp is used to keep the inputs to the motor controller steady. Therefore, for the motor controller to work properly (and for our tests of direction to be meaningful) the Op-Amp must be powered. To power the Op-Amp, the 5V regulator (which powers the high voltage side of the PCB) needs to have power. In order for this to happen, power (from the battery) to the PCB must be on. Therefore, the motor controller can only be tested with power from the battery on.

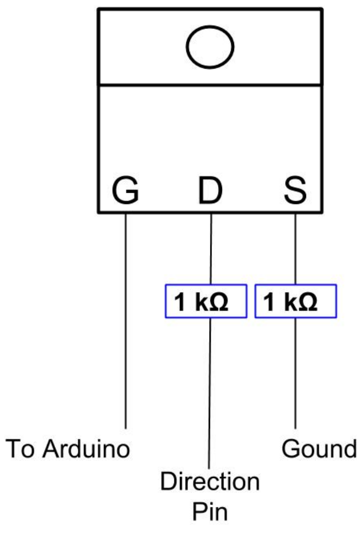


Figure 6.6: Diagram of MOSFET connections

Direction control of the front motor requires the direction pin be connected to either an open circuit or 0V (ground). The Arduino outputs 5V or ground. Thus, we could not simply connect the Arduino output the the motor controller. Doing so would most likely break the motor controller as the direction input is not designed to take any voltage.

Originally, we discussed using a relay instead of a transistor to mimic a switch. The benefits of a relay is that it is simpler to implement. However, a transistor uses electrical signals while a relay relies on moving a physical switch. Thus, a relay changes slower than a transistor. This delay may be problematic as the front motor needs to switch direction rapidly to converge on a position. Thus we choose to use a transistor to control direction.

We had also tried using a BJT (Bipolar Junction Transistor) rather than a MOSFET to control direction. The BJT proved problematic because when the Arduio signal switched from LOW to HIGH, Direction and Ground of the rear motor controller would remain connected (ie the BJT would not act as a switch as it would not open and close the circuit between Direction and Ground). Therefore, the wheel would not switch directions when the Arduino signal changed. This is because BJT transistors are current controlled while MOSFETs are voltage controlled. Since we want to use a change in voltage signal from the Arduino to produce an effect, we choose to use a MOSFET transistor. We should have researched the variation between different types of transistors before trying to implement one.

6.5.2 Run/Stop

The Run/Stop input on the motor controller functions in the exact same way as Direction input. Accordingly, run/stop is controlled with a MOSFET controlled by pin 46. That pin must be initialized to HIGH in order that the motor run.

6.6 Motor "Ticking"

Frances Bryson, feb45 After a few tests of the motor controller, we discovered that there were palpable "ticks" of the motor - that is, when we moved the front wheel from its equilibrium position, there were clear angles at which the motor would resist less the torque, and a noticeable tick could be felt. At first we believed this to be a mechanical flaw in the chain drive, but after further investigation we learned that this is a property of the motor itself, and not something that is easily changed. The motor we used utilizes the Hall Effect (discussed in more detail in the Front Steering Motor section). The ticks are effectively the angles in the motor at which there is a stator field winding, at which there is a greater force between the winding and the magnet. This creates the tangible ticks, as one moves the motor past each winding to angles at which there is a lower magnetic force..

This ticking effect may cause issues, and should be considered when calibrating the encoder - in our front motor testing without IMU data, the encoder began with an equilibrium position directly between two "ticks". This caused oscillations (additional to those from the PD controller gains) as the front motor attempted to return to a steadier position at a tick, but was forced back by a PWM. This problem was alleviated when we set the encoder "zero" position directly on a tick - combining both the PD controller and inclination of the motor to rest at steady state allowed for considerably less jittering of the front wheel for the same PD controller gains.

7 SBW Code

Frances Bryson, feb45

7.1 Overview of Code

The programming of the SBW bicycle is very similar to that of the RC bicycle, in that the structure of the code follows a reading of the IMU, determination of the desired steer velocity by the balance controller, an integration to a desired steer angle, and then a call to the proportional-derivative (PD) controller that converts the desired steer angle into a pulse-width modulation (PWM) and writes it to the front motor controller. These are currently the only elements in the SBW code, since we decided to build the code from the ground up, taking only applicable parts of the RC bicycle code (balance controller, IMU reading, PD controller, and applicable definitions and other setup functions)

7.2 PD Controller Gains

The PD controller is the same as used on the RC bike and was developed last semester. However, due to the dynamic (size, weight distribution) and mechanical (using a new motor controller) differences in the bicycles, the proportional (K_p) and differential (K_d) gains are difference, and needed tuning in the new bike, which was evident in the first test - the wheel violently vibrated around its 'zeroed' position.

We began a series of testing by just including the proportional controller and setting the differential gain to zero, and iteratively increasing the proportional controller by 100, starting at 100. In these tests, particularly those of lower proportional gains, the front wheel would oscillate with low frequency and decreasing amplitude around the zero position until finally coming to rest. The higher the controller, the higher this frequency would be - the proportional gain controls what might be thought of the "strength" of the motor controller. In a comparison of two proportional gains K_{p1} and K_{p2} such that $K_{p2} > K_{p1}$ by at least 100, at the same displacement angle of the wheel, K_{p2} will write a much higher PWM to the motor controller than K_{p1}, effectively making the motor feel stronger, and more resistant to a constant torque holding it in place.

After distinguishing the effect of the proportional gain, we began to include the differential gain. The differential gain effectively damps the oscillations of the wheel. We found that a very low differential gain, $K_d = -1.5$ (negative sign due to the implementation of the code, K_d itself is positive), is most effective, and tuning requires changes of .1 or less. Larger values of K_d result in overdamping - after the motor finishes its initialization the damping of the gain affects the wheel moving toward its zero position, resulting in the wheel moving more slowly than desired. Since the bicycle will experience a large damping effect from the ground and the weight of the rider, it is advantageous to keep the internal damping lower.

We noticed that there was a deadband in the motor controller - at angles $\pm 5°$ from zero the motor controller would not turn the wheel back to zero. This was due to the minimum PWM of the motor controller, and any PWM less than 10 was not recognized as different from 0. By adding 6 to every PWM that is written, not only those less than 10 (effectively linearly shifting the PWM), we were able to solve this problem by making the deadband smaller - now it is calculated PWMs that are less than 4 that are understood by the motor to be no different from 0. In the future, if the deadband is too noticeable, increasing the added value from 6 to a larger number would further decrease the deadband - however, it can not be so large that the PWM is never writing 0, since that would result in a constant oscillation of the front wheel.

Due to the increased damping we expect from a rider on the bike and the resulting increased torque due to the friction between the front wheel and the ground, we want the motor to be very strong, i.e. the proportional gain should be large. We began increasing K_p, and observed an interesting phenomenon - when we turned the front wheel there would be an angle at which the front motor shuts off. This caused the fault light on the motor controller to turn on. We deduced that the PD controller was writing a PWM that was too high for the motor controller, and implemented a maximum PWM cap. To determine the value of this maximum PWM, we iteratively changed the maximum PWM and turning the wheel until the motor turns off - if the motor can turn the wheel from angles we would expect to see when riding the bike (less than $\frac{\pi}{2}$), the maximum PWM is acceptable. From these tests, the videos of which are in the SBW folder in the Google drive, we determined that the maximum PWM was 130.

We began preliminary riding testing with a proportional gain of 400 and differential gain of 1.75, and the bicycle was able to turn its wheel to correct for a lean angle while moving. These gains still need to be tuned - when the motor first turns on the front wheel vibrates at a high frequency and low amplitude until it is steadied by someone physically holding the wheel until the vibrations ceased.

8 Conclusions

At the beginning of the semester we were maybe a bit ambitious with planning on having a fully working bicycle by the end of this semester including a moving and torque-controlled handle bar. During the semester we realized that having the bike 'just' balance on its own with a stiff handle bar would be a more achievable goal. By the end of the semester we were able to test the bicycle with a person and steer the bicycle by leaning rather than moving the handle bar without falling over.

9 Future Work

9.1 Hardware

1. Implement Tacometer on rear wheel. Bike velocity is input to the balance controller. A tacometer can measure rear motor speed which can be used to give accurate bike velocity to the balance controller.

2. Handlebar to complete steer-by-wire system. We will need to design a method of attaching a handlebar, rotary encoder, and motor above the current mechanical assembly. The handlebar will need to rotate about an independent shaft with the rotary encoder to read the user's control input. We want to include a motor and another power transmission system to allow the application of variable resistance to the user's control. This would help simulate the feeling of riding a normal bike, since a normal bike experiences a variable resistance to the turn of its handlebar due to the variable friction between the bike's wheel and the ground. Additionally, we want the handlebar's shaft to be conllinear with respect to the shaft through the bike's fork. If we can devise a way to attach the handlebar system to the bike frame while satisfying these requirements, we can implement a method of testing three different rider modes: one in which the handlebar's and fork's shafts are connected to simulate a normal bike, one in which the shafts are not connected and no resistance is applied to the handlebar's rotation, and one in which resistance is applied. This will allow us to test user experience and refine our future goals.

9.2 Motor Controller

Dylan Meehan: dem292

Moving forward, we need to implement a 4-Quadrant, current controlled motor controller. A 4-Quadrant controller will give us increased control over the motor (see discussion below). A current controlled motor controller will allow us to easily measure the torque on the front wheel (??). A 4-Quadrant, current controlled motor controller offers improvements over the 2-Quadrant, velocity controlled motor controller we currently use.

9.2.1 4-Quadrant vs 2-Quadrant Discussion

Motor controllers capable of working in two directions can be broken into two categories: 2-Quadrant control and 4-Quadrant operation. 2-Quadrant controllers are capable of supplying a torque only in the direction the motor is rotating (quadrants 1 and 3 of ??). This controller can move the motor in both directions, but torque can only be applied in the same direction as the motor velocity. 4-Quadrant controllers are capable of operating in all four quadrants of ??. Thus, 4-Quadrant controllers can not only spin the motor in two directions, but for each direction they can apply a torque in the direction of and opposed to velocity. [2]

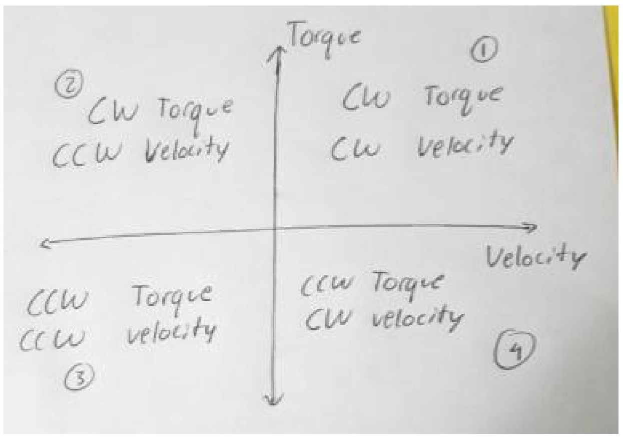


Figure 9.1: 4 Quadrant Motor Operation

4-Quadrant motor control is desirable because it allows the motor controller to more accurately and quickly decelerate the motor compared with a 2-Quadrant controller. With a 2-Quadrant controller, torque can only be applied in the same direction as velocity, as to accelerate the motor. 2-Quadrant controllers rely on motor friction and load torque in order to decelerate the motor. In contrast, a 4-Quadrant controller, by applying a torque opposite to velocity, is able to actively

[2] References for information about 4-Quadrant Control:

1. http://digital.ni.com/public.nsf/allkb/E768FE6E5663940A86256C46007DB743
2. http://www.lselectric.com/4-quadrant-motor-control-2-quadrant-operation/

decelerate the motor. This is done through the process of regeneration where the motor controller converts the mechanical energy of the spinning motor into electrical energy (similar to a generator), thus slowing the motor. This contrasts when the motor controller wishes to accelerate the motor by converting electrical energy of the battery into mechanical energy of the motor (similar to a traditional electric motor). Because we wish to accurately command the position of the motor, we want to quickly and smoothly change motor direction. Thus, the ability to actively decelerate the motor is beneficial.

9.2.2 Current Controlled Motor Controller

A current controlled motor controller would allow us to easily measure the torque on the front wheel. To mimic the feeling of a bicycle, we want to apply a torque to the handlebars proportional to the torque the ground applies to the front wheel. In order to measure how much torque the ground applies we want to measure the torque on the motor. Since the front fork of the bike is fixed to the steel shaft which is fixed to the large gear, we can treat the torque in the front wheel to be equal to torque on the large gear. This assumes small angular accelerations and minimal bearing friction between the steel shaft and the bike frame. These simplifying assumptions are valid because we do not need the measurement of torque on the front wheel to be very precise. The torque measurement is used only to simulate the friction from the ground in the handlebar. The applied torque in the handlebar is used only to give the rider a general feeling of resistance from the ground. Thus, the torque we apply to the handlebar does not need to perfectly match the torque the ground applies to the front wheel.

To calculate the torque on the front wheel given the torque on the motor, we must multiply by the gear ratio. The large gear is connected via the chain drive to the the motor. Thus, dividing the torque in the large gear by the gear ratio of the chain drive gives us the torque in the motor. So, with our gear ratio of 7, the torque on the large gear ($\approx$ torque on the front wheel) is 7 times the torque measured in the motor. Thus, from the torque on the motor we can calculate a reasonable value for the torque on the front wheel.

To measure torque, we will measure current in the motor. In Brushless DC motors, torque is proportional to current. Specifically, $\tau = K_T * I_A$ where K_t is the torque constant and I_A is the armature current. For our motor (BLYSG342S-24V-3000-R75), $K_T = 7.93\frac{oz-in}{A}$. So, by measuring the current in the motor, we can calculate the torque. In a current controlled motor controller, PWM is used to control current in the motor. Thus, since we command a certain PWM to the motor, we command a certain current in the motor, so we clearly know the current in the motor. On the other hand, in a velocity controlled motor controller, a given PWM corresponds to velocity, so current cannot easily be determined. Measuring the current in the wires to the motor is not easy. We cannot measure the current entering the motor controller because the current to the motor controller is greater than the current that enters the motor. We also cannot measure the current leaving the motor controller (going to the motor). Because brushless motor controllers have three phases, to measure the current to the motor we must measure and combine the currents in each phase. This would be hard. Thus a current controlled motor controller is the easiest way to know the current in the motor and thus the torque.

9.3 Software

Balance controller testing - the balance controller gains, as well as PD controller gains, may be different for different riders, and we will need to investigate the perceptible effects of changing each balance controller gain to obtain a better idea of how they might be changed for a new rider.

10 Hardware Updates

graphicx [margin=1in]geometry float array ragged2e mathtools caption amssymb gensymb graphicx caption longtable tabularx chronosys fancyhdr accents subfig fancyhdr rotating pdfpages listings [export]adjustbox cleveref todonotes [utf8]inputenc [margin=1in]geometry

Landing Gear Final Report Dylan Meehan, Aviv Blumfield, Olav Imsdahl May 2017

11 Landing Gear

11.1 Hardware Overview

Most of the landing gear hardware has been in place since last semester - (see Fall 2016 Report page 34 for more details). Here is an overview: The landing gear is powered by a 9V battery connected to a DC motor. The Arduino controls two relays which change the direction of the landing gear (UP or DOWN). Limit switches stop the landing gear at the end of its range of motion.

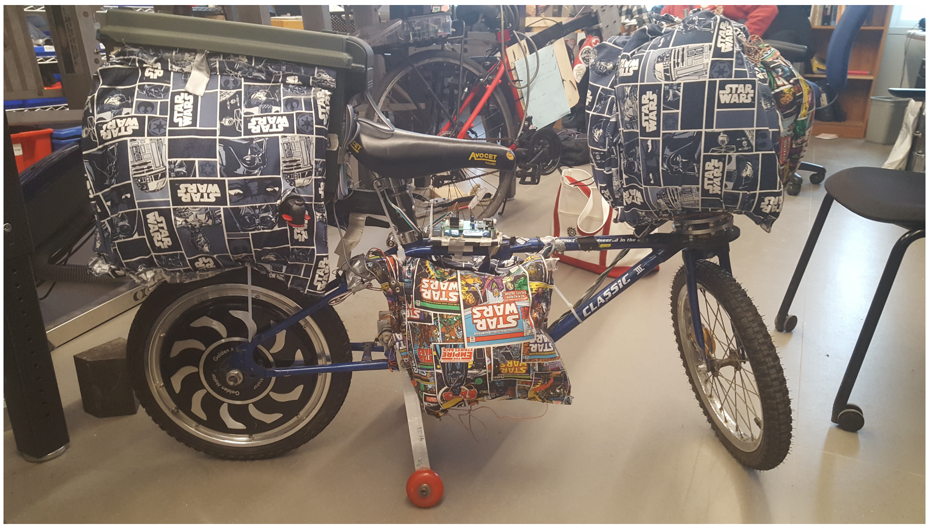


Figure 11.1: Landing Gear Properly Adjusted

The landing gear needs to be adjusted so that when it is DOWN the landing gear is in contact with the floor (??, ??).

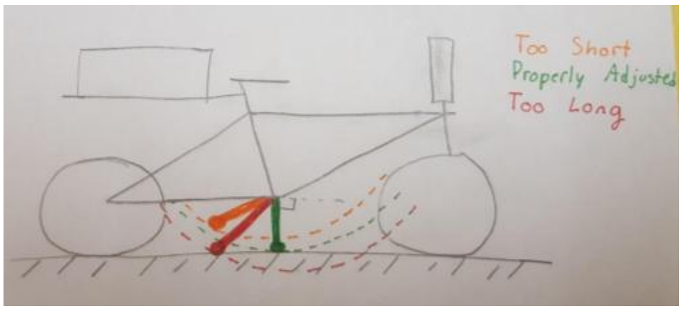


Figure 11.2: Good and Bad Adjustments of Landing Gear

Adjusting the angle the landing gear is done by bending the legs of the landing gear. The legs are aluminum and thus can easily bent by hand. They should be bent such that the landing gear when deployed is perpendicular to the ground (??). If the landing gear is too long, ie it hits the ground before vertical (red arc), it should be bent out. If the landing gear is too short, ie does not hit the ground (orange arc), it should be bent in. Both legs of the landing gear should be bent symmetrically. See ?? on bending.

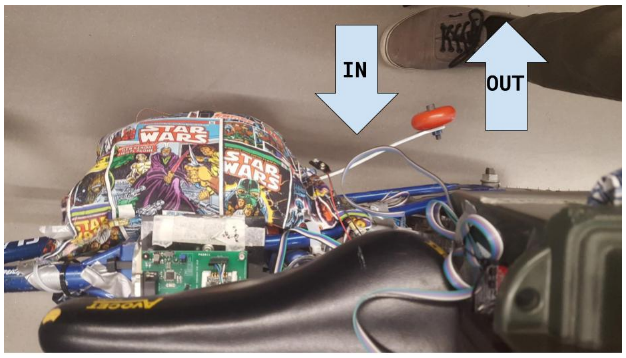


Figure 11.3: Landing Gear Bending

If the bike falls, the landing gear will get bent. This is because the landing gear hits the ground before the airbags (**??**). This is not a problem as the landing gear can easily be bent out after a fall.

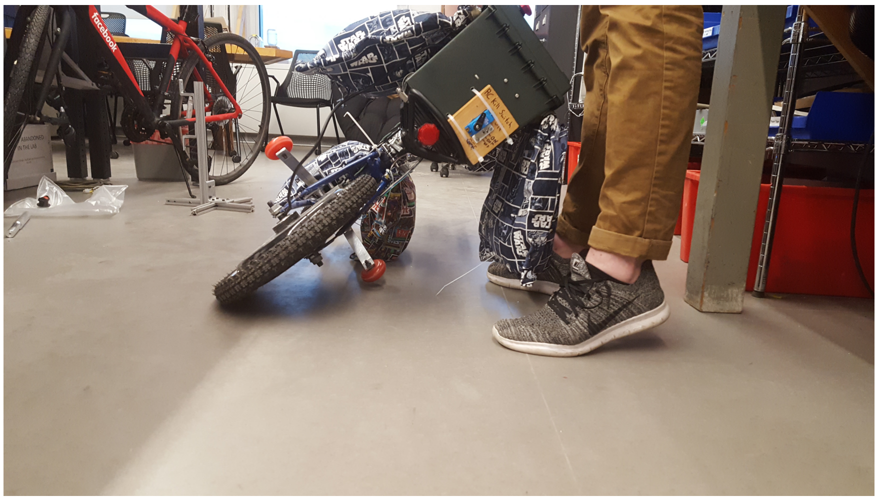


Figure 11.4: Landing Gear Hits Ground before Airbags

There is a peg attached to the shaft of the landing gear which hits a limit switch once the landing gear reaches the end of the swing arc, **??**. The limit switch breaks the DC circuit which stops the motor. The peg is wedged to the shaft of the landing gear and secured with a set screw on the underside of the peg. This set screw (and the one in **??**) can be problematic as they get loose over time. In the future, we should not use set screws to secure something to a round shaft.

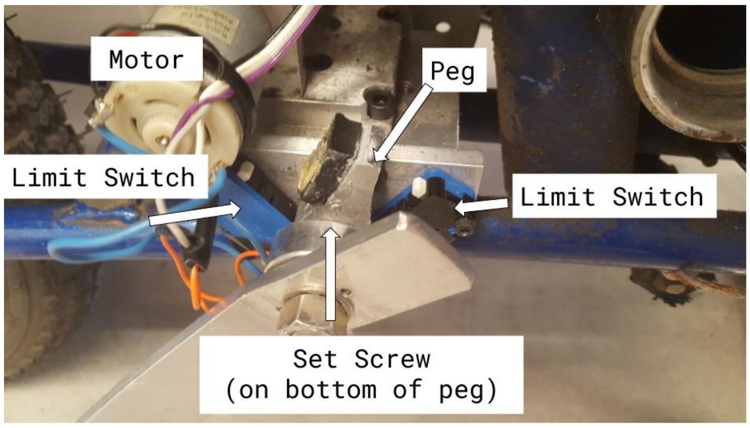


Figure 11.5: Peg, Limit Switches, and Set Screw

The landing gear itself is secured to the shaft via a screw (**??**). This screw should be tight. Test this by moving the landing gear manually: If the motor moves in couple with the landing gear than this screw is tight. Additionally, the screws securing the landing gear motor to the bike should also be tight.

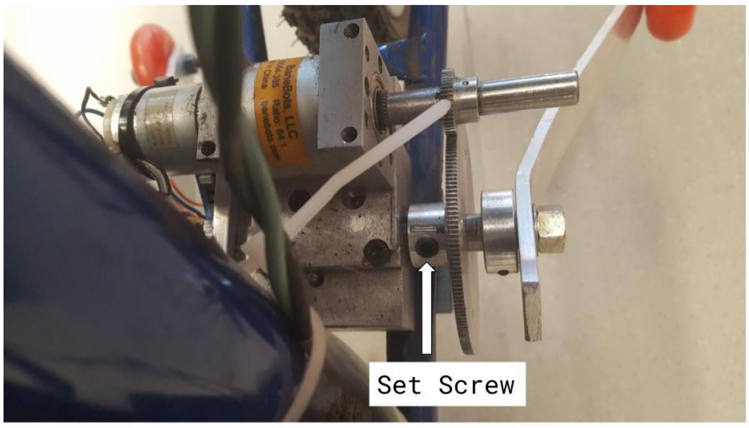


Figure 11.6: Set Screw to hold landing gear together

11.2 Software

The landing gear is controlled by RC channel 5. The switch for this channel is marked with masking tape on the RC controller. The RC control for channel 5 functions using interrupts.

Interrupts work by assigning a pin to the interrupt and setting a trigger mode. For our RC we set the trigger mode to change so that interrupt triggers both when the pin goes from high to low and from low to high. The RC signal transmitted from the controller is a PWM signal so the interrupt will trigger multiple times per cycle of the signal. When the interrupt triggers the attached function is run. The functions we use for RC signals calculate how long the pin was high in microseconds and assign that to a global variable called pulsetime. It is important that any variables used in the interrupt are defined as volatile variables so that the variables are loaded form RAM and not the storage register. The value of pulsetime corresponds to specific positions of the RC switches and analog inputs. Additionally, when an interrupt is triggered the code for the main loop pauses, then code for the interrupt runs to completion, then the main loop continues. Since this can happen multiple times per loop the amount of time spent inside the interrupt should be minimized. Therefore we use the interrupts to calculate pulsetime only and later on in the loop use pulsetime to deploy landing gear or preform other tasks.

RC channel 5 corresponds to pin 27. Note This pin should *not* be initialized as DigitalInput. The interrupt calls the functions LandingGearUp() and LandingGear-Down() which control the landing gear in the obvious ways. The landing gear should not be toggled in rapid secession (greater than once a second). Toggling the landing gear in rapid secession can occasionally cause problems with the other hardware of the bike. However, this is not a problem as the landing gear never needs to be called this fast.

11.3 Landing Gear & Rear Motor Bug

When testing, it was determined that calling the landing gear caused the rear motor to stop working. This was despite the landing gear and rear motor each working fine in isolation. [3]

11.3.1 Problem Diagnosis

The landing gear & rear motor problem only occurred when the 9V battery was connected to the landing gear (ie there was power to the landing gear circuit). Testing the landing gear and rear motor with the 9V battery unplugged caused no problems. This eliminated the theory that a software bug caused the issue as disconnecting the 9V battery has no effect of the code.

The Arduino circuit and landing gear circuit are designed to be isolated via two relays. The fact that a change in the landing gear circuit (connecting the 9V battery) caused a change in the Arduino circuit (the rear motor breaking) suggested that the landing gear circuit and Arduino circuits might have been accidentally connected. Carefully checking for a connection between the two circuits using the Ohm-meter showed the both circuits we not in fact connected.

Despite the landing gear and Arduino (rear motor) circuits being physically isolated, the problem was caused by magnetic interference between the two circuits. Specifically, the relays used to control the landing gear caused magnetic interference with the rear motor controller[4].

In more detail, relays use an electric current to generate a magnetic field to control a switch. This switch controls a second electric circuit. For each of our two

[3]The landing gear / rear motor problem was only discovered during a full rolling test of the bicycle and landing gear. Prior to this, it was believed that the landing gear was working perfectly. No problems were anticipated as the landing gear and rear motor are designed to be separate systems. This situation demonstrates the importance of testing every component integrated integrated with the rest of the bicycle, even after each component has been thoroughly tested individually.

[4]We would like to thank Jason Cortell for providing useful insights into this problem

relays, a signal from the Arduino (0V or 5V) is used to generate a magnetic field to control a switch which controls the direction of current in the landing gear circuit (See ??).

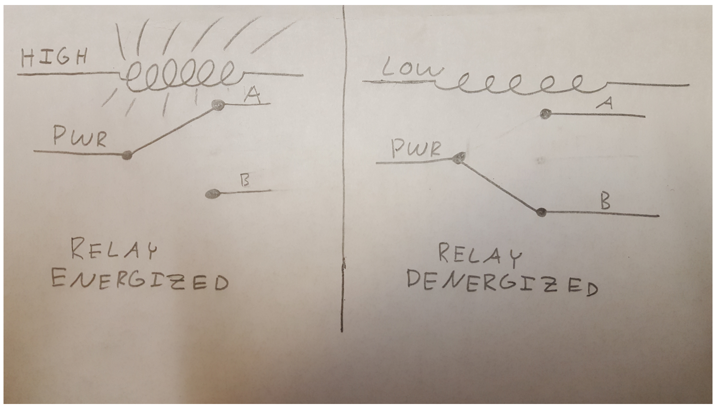


Figure 11.7: Diagram of single relay switch in energized and denergized positions. In the energized position, the relay generates a magnetic field. Note: the landing gear utilizes two relays which operate in unison.

The rear motor controller is a brushless DC motor controller which, in addition to 24V, takes low voltage signals from the Arduino. These low voltage signals include speed, direction, brake, horn and others. The autonomous bicycle only uses the speed control. The other inputs(eg brake) function as follows: Connecting the brake wire to ground causes the motor to brake (Braking is a built in function of the rear motor controller. We do not know exactly how the rear motor accomplishes this). Leaving the brake wire disconnected (an open circuit) causes the rear motor to not brake (function normally). Accordingly, there were about 5 wires (brake control, etc) leaving the rear motor controller which were not connected to anything (see ?? showing plug with loose wires). These unconnected wires could function similar to antennae and pick up magnetic signals near them (such as those from our relay). This was the crux of our problem: magnetic signals from the relays would interfere the rear motor controller causing it to shut off. The close proximity of the rear motor and relay module caused the magnetic field generated by the relays to interfere with the rear motor controller (??).

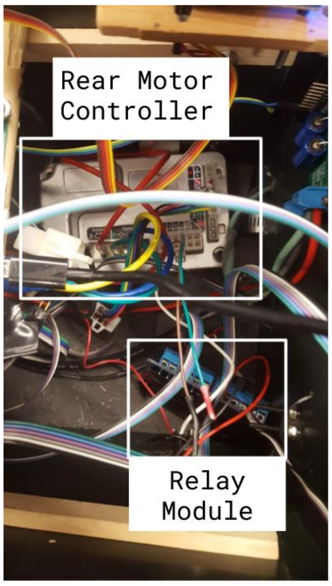


Figure 11.8: RMC and Relay in close proximity

We suspect that the rear motor shuts off as a built in safety feature. If it detects unusual signals (magnetic interference), it decides that something is wrong, and turns off for safety.

The follow tests supported the idea that magnetic interference from the relays caused the rear motor to shut off:

1. Assuming that the wires leaving the rear motor controller were acting similar to antennae, adjusting the direction they point should change the intensity of the signal they pick up (similar to how adjusting a radio antennae can adjust the signal of the radio). This was in fact the case. Taking one of the loose wires and pointing it towards the relays caused the rear motor to stop working. Then pointing the wire away from the relays caused the rear motor to work again. This test was done without changing the RC control of the rear motor or landing gear.

2. The rear motor only broke when calling LandingGearUp(). It did not break when calling LandingGearDown(). Calling LandingGearUp() sends a HIGH signal to pins 47 and 48, which creates a magnetic field in the relays. Calling LandingGearDown() sends a LOW signal to the relays which does not create a magnetic field (see **??** about relay operation). The rear motor also only broke when the 9V battery was connected. Thus, the rear motor only broke when *both* the relays were generating a magnetic field *and* power to the landing gear was on. This case generates the largest magnetic field. So, the rear motor only broke in the case with the largest magnetic interference.

3. The following procedure would *not* cause the rear wheel to break:
    (a) Have landing gear power on and landing gear in down position.
    (b) Use one of the limit switches to turn 9V power to the landing gear off.
    (c) Then use the RC controller to run the landingGearUp() function, this energizes the relays.

(d) After the relays have switched, close the limit switch to connect the landing gear circuit (9V power).

This practice has the effect of disconnecting the 9V battery while the relays switch. Similar to running the landing gear without the 9V battery connected, this caused no problems. However, performing step (b) using an additional relay instead of an external switch to disconnect the 9V power *did* cause the landing gear to break. This was strange as a physical switch and a relay should have the same effect, except that a relay generates a magnetic field. Thus, the magnetic field of the relay caused interference with the rear motor controller.

4. The following test did *not* help to determine that the relays caused the problem. However, this test shows an interesting symptom of the problem. Specifically, when the rear motor breaks, the Op-Amp (see figure below) can falsely appear to be broken. The op amp provides a steady signal to control rear motor speed (in the range of 0-5V). When the rear motor stops working, the op amp did not produce a proper output, despite receiving appropriate inputs. Initially, this suggested that the Op-Amp failed and caused the rear motor problem. However, this was not the case. Testing the Op-Amp in isolation demonstrated that it worked properly. The seemingly broken Op-Amp was not the cause of the problem, rather is was an symptom of the rear motor shutting off. We suspect the following happens: when the rear motor shuts off, the rear motor controller short circuits its inputs. One of these inputs is the signal from the Op-Amp. Thus if the rear motor controller shorts this circuit, it can appear that the Op-Amp is giving a bad output. In fact, the Op-Amp was fine and the rear motor caused the issue.

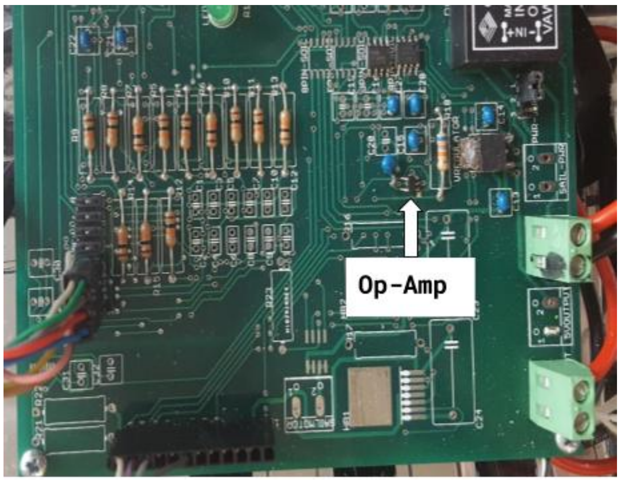


Figure 11.9: Op-Amp on PCB

11.3.2 Solution

The relay box was permanently moved outside of the main ammo box of the bike. This increased the distance between rear motor controller and the relay module. Thus, the magnetic interference between the relay module and the was reduced which eliminated the problem of the landing gear inadvertently turning off the rear motor. The landing gear now works properly and can be controlled with the RC controller.

Specifically, the relay module was placed on the bike frame, under the ammo box (see picture below). The relays were secured with duct tape (This will be adjusted Summer 2017 with a more permanent solution). The 9V battery was placed near the relay module and also secured with duct tape. The cable between the PCB and the relay module was extended to account for the larger distance between the two components.

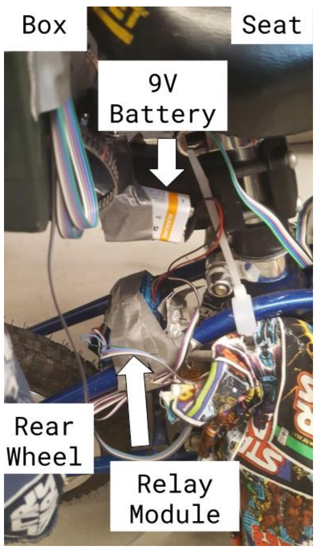


Figure 11.10: Relay in new location

11.3.3 Failed Solution

Change Rear Motor Connector: The connector to the rear motor control was changed to remove the unused wires (**??**). As the unused wires were believed to amplify the magnetic interference, this was expected to resolve the problem. However, after implementing the new connector, the problem persisted. The new conenctor did help to declutter the box.

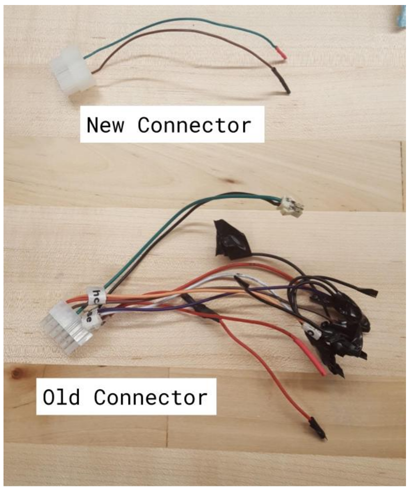


Figure 11.11: Old Plug vs. New Plug

amsmath mathrsfs graphicx [letterpaper, portrait, margin=1in]geometry cleveref

Sensor Bias and Fusion Research/Testing Frances Bryson

12 Sensor Fusion

12.1 Background

Last semester, we observed that when the bike remained still the IMU (inertial measurement unit) would sense a change in the lean angle, leading to a perceptible change in the front wheel angle (as it communicated with the encoder). This was traced back to an issue with the Yost Lab's software for finding the lean angle from the data the gyroscope and accelerometer sense. My first project was to write a new form of this software, named sensor fusion for the "fusing" of the data from the gyroscope and accelerometer on the IMU.
The gyroscope on the IMU measures a rate of change of the lean angle, and so this data must be integrated to find the lean angle. Over time, errors in integrating will build up to create a drift. On the other hand, the lean angle is found from the accelerometer by calculating the difference between the measured axes (x and y) and a known vector pointing in a known direction - for example, gravity. The basics of sensor fusion is to use the weighted sum of the lean angles from the gyroscope and accelerometer as the lean angle we use for the balance controller.

12.2 Equations

I am using the complementary filter for sensor fusion:

$$\theta = \beta\theta_g + (1-\beta)\theta_a$$

In this equation, θ_g refers to the lean angle found from the gyroscope, θ_a refers to the lean angle found from the accelerometer, and β is the weighting factor (Esfandyar et al.). Future iterations of the sensor fusion might implement a Kalman filter.

12.3 Analysis of IMU Data

After writing Arduino code to retrieve the raw data from the IMU (and then to pass it into the sensor fusion equation), I noticed that the values reported were not consistent with the orientation of the bike; for example, the data from the accelerometer rapidly oscillated values even when the bike remained still. In addition, the raw data retrieved by the Yost software, 3 Space Suite, was similarly noisy. This noise did not translate to the reported lean angle, which we recorded to analyze the drift.

In a 50 minute test of gathering data from the IMU using 3 Space Suite, during which time the bicycle remained motionless, we were able to collect the raw accelerometer data, as well as the orientation angles from the IMU's built-in sensor fusion.

?? shows the drift in angle that the IMU senses. This drift has an amplitude of 2 degrees, which meets the Yost Lab specifications (the sensor is rated for an accuracy of 2 degrees), so gaining greater accuracy will be difficult using only Yost's built-in sensor fusion.

The X-axis data in Figure 12.2 supports the results I found in my Arduino code - that the raw data is noisy and difficult to analyze directly. The same can be said for the Z-component of the gyroscope data (which corresponds to the raw data of the roll rate), in Figure 12.3.

Another method of correcting for the drift would be to use a bias estimator. The bias estimator is another controller that runs on a different time scale than the balance controller, and calculates the estimated difference between the current measurement and the expected measurement. From this, the expected measurement can be found, and the drift would be greatly reduced (Tereshkov).

However, after running simulations we learned that the balance controller is able to balance the bicycle in the presence of this sinusoidal drift pattern (Figure 12.4 shows the recovery of the bicycle from a 37 degree angle, while Figure 12.5 shows the long-term stabilization in the presence of bias in a simulation run over 20 minutes). Therefore, the bias estimator is not top priority for the bicycle to succeed in balancing as it roams the arts quad.

12.4 Conclusions

While the balance controller is robust enough to balance in the presence of the IMU drift, more investigation still needs to be done to determine if the drift will affect the navigation control. If the drift causes a problem for the navigation control, then it will be necessary to implement a bias estimator.

have we successfully been able to implement and test naviagtion to know that this is not a problem?

13 Sensor Bias

13.1 Background

Rather than implementing our own sensor fusion code (for background, refer to my PDR), another method of dealing with the IMU drift is a bias estimator. At any point in time, the bias estimator will determine the bias of the controller and subtract it out of the reported IMU angle. This would run at a different timescale than the main bicycle code so that the bicycle's physical lean are not confused as bias in the IMU. The estimation of the bias will be dependent on the current lean angle and rate (integrated to find a lean angle from the gyroscope), as well as the previous iteration's result for the bias estimation. The equation that I used was:

$$\phi_{corrected,t} = \phi_{IMU,t} - \Delta t(\phi_{IMU,t} - \phi_{expected,t})K_{gain}$$

13.2 Modeling on MATLAB

The first step was to determine the gain β. Since it was difficult to find time to do this on the bike, I wrote a Matlab code to run the bias estimation on IMU data we had previously saved from the bike over a twenty-minute test of the bike remaining upright and still. After I succeeded in making this simulation work, I set up a gain-finding code that looped through many different gains and scored each one based on how much it minimized the data whilst also keeping the mean of the data the same. From this code I found the optimal gain to be $\beta = .9675$, the result shown in ??.

13.3 Testing on IMU

After promising results from MATLAB, running on the data collected from a 20-minute test of the IMU, I moved on to writing Arduino code that implemented this estimator, and then repeated a long test of the data. To obtain usable data, I increased the number of decimal points output in the serial monitor, then copied the serial monitor into a txt file which could then be read by Matlab. However, after a ten-minute test, I saw that the data was not matching my expectations and the drift was still there, and the amplitude of drift remained after changing the gains. Furthermore, when I plotted both the raw data and data "corrected" by the sensor data in Matlab, they matched up very closely (overlaying each other), shown in ??

13.4 Conclusions

After live-testing on the IMU was found unsuccessful, and without being able to find solutions, we determined that since the drift does not greatly affect the performance of the balance controller (refer to **??**, **??**), and is not the issue with the circling of the bicycle during tests (since the circling occurs on a much smaller timescale than the drift), this project was not worth spending even more time on, and the bicycle would be able to run despite the bias without any attempts at correction.

14 Appendix

14.1 Sources

Esfandyar, Jay, Roberto De Nuccio, and Gang Xu. "Solutions for MEMS Sensor Fusion." Solutions for MEMS Sensor Fusion. Mouser Electronics, n.d. Web. 04 Mar. 2017.
http://www.mouser.com/applications/sensor_solutions_mems/.
Tereshkov, Vasiliy M. “An Intuitive Approach to Inertial Sensor Bias Estimation.”
Jul. 13, 2013.
https://arxiv.org/ftp/arxiv/papers/1212/1212.0892.pdf

14.2 Sensor Fusion Arduino Code

```
#include "IMU.h"
#include "SPI.h"
#include math.h


    const float pi = 3.1416;
float theta_0 = 0.0; //initial guess for theta for integral of gyro data
float beta = .5; //start with guessing the weight to be .5


    struct IMUdata {
float x;
float y;
float z;
};


    void setup() {
Serial.begin(115200);
initIMU();
}


    void loop() {
IMUdata raw_gyro = getIMU(0x41); //get raw gyro data
IMUdata raw_accel = getIMU(0x31); //get raw accelerometer data


    //print statements to view the data as it comes back from the IMU
Serial.println("Raw roll rate: ");
Serial.println(raw_gyro.z);
Serial.println("Raw pitch rate: ");
Serial.println(raw_gyro.x);
Serial.println("Raw yaw rate: ");
Serial.println(raw_gyro.y);
Serial.println("Raw accelerometer data: ");
Serial.println(raw_accel.x);
Serial.println(raw_accel.y);
Serial.println(raw_accel.z);

    float theta_accel = atan2(raw_accel.y,raw_accel.x); //Gives us roll from accelerometer
float theta_gyro = theta_0 + raw_gyro.z * .01 * 360 / (2 * pi);
Serial.println("Theta from gyros: ");
Serial.println(theta_gyro);
Serial.println("Theta from accel: ");
Serial.println(theta_accel);

    //sensor fusion equation
float theta_fusion = beta * theta_accel + (1 - beta)* theta_gyro;
Serial.println("Sensor Fusion Theta: ");
Serial.println(theta_fusion);
theta_0 = theta_gyro;
}
```

Authors: Foteini and Jared

15 Navigation

The navigation sub-team focused on the development of the navigation algorithm for the autonomous bicycle. In the initial stages of development, we used python plotting libraries[5] to simulate and test the results of the algorithm on predefined paths. Once the algorithm was developed, we worked on integrating the bicycle dynamics into the simulation of the bicycle's navigation. The integration of bicycle dynamics challenged some of the initial assumptions that had been made in the development of the navigation algorithm and led to the use of a proportional controller to navigate when close to the target path. By the end of the semester, we connected the whole system together to test the navigation algorithm with the actual bicycle.

15.1 System Design

Our initial system was comprised of two main modules, the Nav module and the Simulator module. The two modules were connected with a one-way dependency. The Simulator module depended on the Nav module, as it used types and functions defined within the Nav module. There was a clear separation of responsibilities between the two modules, a characteristic which we made an effort to preserve throughout the evolution of the system. That is, we tried to have each module

[5]The python plotting library Matplotlib was used. Read more at https://matplotlib.org/

contribute to a single, well-defined task, thus, it could be easily replaced by a new module of similar functionality without disrupting the whole system.

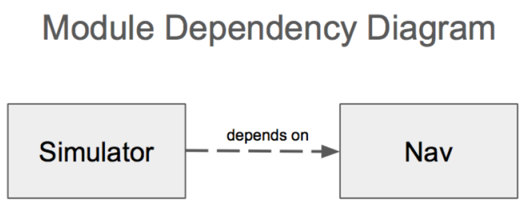


Figure 1: Visualization of initial system structure. Final structure visualized and described in section 1.5

The Nav module and Simulator module were both initially written in Python and the terminology that will be used is referencing the object-oriented programming paradigm[6]. The Nav module was designed to contain three classes: the Bike class, the Map_Model class and the Nav class. These same classes persist in the current version of the system with similar functionality and only incurred small alterations. The Bike class represents the bike and defines different attributes that describe the bike's state such as the x, y coordinates, its direction, its speed and its minimum turning radius[7]. The Map_Model class represents the map containing a bike object, a list of points representing the waypoints as well as a list of point pairs representing the paths. The Nav class contains the navigation algorithm for the bike and helper functions used in its implementation. By having a Map_Model object as an attribute, the Nav class uses functions from other classes as well as functions defined within this class to provide the navigation of the bike around a set of paths as will be described in "The Navigation Algorithm" section.

The Simulator module contained two classes: the Bike_Sim class and the Simulator class. The former class represented the bike in the simulation and defined attributes related to the state of the bike such as the coordinates, the speed and the direction. The latter class represented the simulation of the map model and navigation algorithm. This class contained attributes of types defined in the Nav module such as a Map_Model object and a Nav object. It was responsible for creating the simulation and updating the bike position according to the navigation algorithm.

15.2 The Navigation Algorithm

The algorithm for returning the bike to the path can be summarized with the following conditional statement:

if $\delta \geq d$*:*

[6]Read more about object-oriented programming in Python at https://python.swaroopch.com/oop.html

[7]We used the assumption that the bicycle would always choose to turn at its minimum turning radius

```
        turn parallel
else:
        turn perpendicular
```

Where δ represents the displacement toward the path that the bike would incur if it were to turn in a way as to become parallel to the path due to its turning radius. d represents the absolute distance between the bike and the path. *turn parallel* returns the direction the bike should turn in order to become parallel with the path. *turn perpendicular* returns the direction the bike should turn to become orthogonal to the path. This conditional statement tests whether the bike would undershoot the path if it were to attempt to turn onto it at any given moment. For example, when the bike heading is orthogonal to the path, $\delta = r$, where r is the turning radius of the bike. When the bike is parallel to the path, $\delta = 0$ because the bike is already parallel to the path and would not need to turn at all.

When $\delta < d$, turning parallel to the path would result in the bike undershooting the path by a distance of $d - \delta$. Therefore, in this situation, the bike should intuitively continue to directly approach the path (*turn perpendicular*). When $\delta = d$, the bike is the perfect distance from the path where turning parallel would place the bike directly on it. In this situation, the bike should turn parallel to the path because it will place the bike on the path. When $d > \delta$, the bike will cross the path regardless of the turning direction. In this scenario, the bike should still begin to turn parallel to the path because turning perpendicular would only place the bike further away from the path in the wrong direction. Once the bike has passed the path and is parallel to it, δ becomes 0 and $d > 0$. The algorithm correctly directs the bike to turn perpendicular to the path in order to approach. As the bike turns back toward the path, δ increases continuously and d decreases continuously. When $\delta = d$, the bike will turn parallel on to the path.

1.2.1 Computing δ

To calculate the displacement that the bike would incur during a turn, both the bike vector, $\overrightarrow{b}$, and the path vector, $\overrightarrow{p}$, can be placed on a circle of radius r. This is shown in the left subfigure of Figure 2 where the target distance δ is drawn in blue.

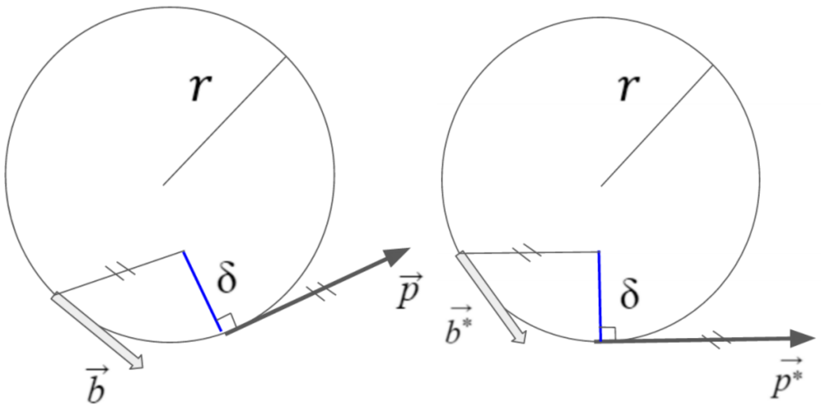


Figure 2: Computing δ. The left subfigure represents the generalized case. The right subfigure shows the generalized case, when rotated

Computing δ is greatly simplified by rotating both vectors such that p is perfectly horizontal. In this scenario, shown on the right side of Figure 2, δ may be easily calculated as $r - rsin(\theta)$ where θ is the angle associated with the point at which $\overrightarrow{b}$ is tangent to the circle. To convert the generalized case to the simplified version, the following rotation matrix is constructed:

$$R = \begin{bmatrix} p_0 & p_1 \\ -p_1 & p_0 \end{bmatrix} \tag{1}$$

where p_0 and p_1 are the endpoints defining $\overrightarrow{p}$. Using this matrix, $\overrightarrow{b^*} = R\overrightarrow{b}$ and $\overrightarrow{p^*} = R\overrightarrow{p}$ where $\overrightarrow{b^*}$ and $\overrightarrow{p^*}$ are the rotated vectors as seen in Figure 2. Although the vectors are rotated, the desired distance, δ, is unchanged and may now be calculated using the aforementioned equation: $r - rsin(\theta)$.

1.2.2 Computing d

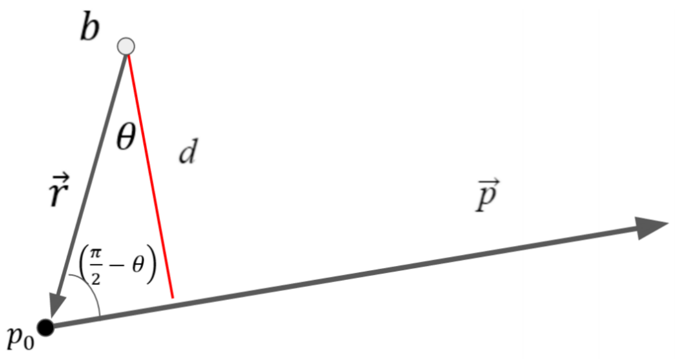


Figure 3: Computing the distance from the bike to the path

The distance from the bike to the path, d, is computed using simple projected distance. Let $\overrightarrow{r}$ be a vector from the bike, b, to the starting point of the path, p_0

as shown in Figure 3. Let θ be the angle between $\overrightarrow{r}$ and a vector orthogonal to $\overrightarrow{p}$, $\overrightarrow{p_\perp}$. The desired distance, d is then found as follows:

$$
\begin{aligned}
d &= \frac{||\overrightarrow{r}||sin(\pi/2 - \theta)}{||\overrightarrow{p}||} \\
&= \frac{||\overrightarrow{r}||cos(\theta)}{||\overrightarrow{p}||} \\
&= \frac{\overrightarrow{r} \cdot \overrightarrow{p_\perp}}{||\overrightarrow{p}||}
\end{aligned}
$$

1.2.3 Performing functions turn parallel and turn perpendicular

To compute the direction, clockwise (CW) or counter-clockwise (CCW), that the bike must turn in order to turn perpendicular to the path, the dot product is used. When the bike is above the path and $\overrightarrow{b} \cdot \overrightarrow{p} > 0$, the bike's direction is within π radians CCW of the path and the bike should turn CW. Otherwise, the bike should turn CCW. To normalize for the relative positioning of the bike and the path, the sign of the dot product is multiplied by the sign of the distance calculation, which results in the sign of the dot product giving a universal rule for finding the direction to turn.

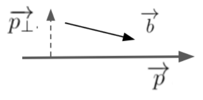
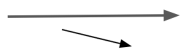
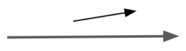
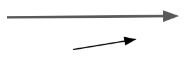
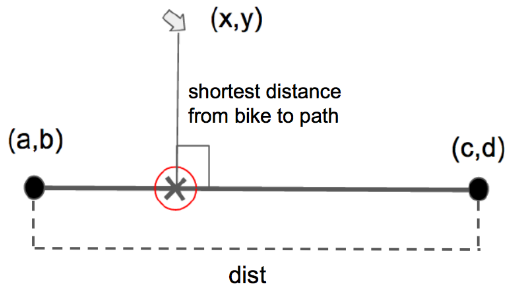
Possible turn_parallel cases ($\delta \geq d$)	Relation to path	Sign of $\overrightarrow{b} \cdot \overrightarrow{p_\perp}$	Nav Command
 $\overrightarrow{p_\perp}$ $\overrightarrow{b}$ $\overrightarrow{p}$	Facing Towards	Negative (-)	+1 (CCW)
	Facing Away	Negative (-)	+1 (CCW)
	Facing Away	Positive (+)	-1 (CW)
	Facing Towards	Positive (+)	-1 (CW)

Table 1: Calculation in nav algorithm that determines the direction to turn in the possible *turn_parallel* cases

Computing the direction that the bike must turn in order to turn parallel to the path is very similar to the calculations performed for the turn perpendicular operation. Instead of computing $\overrightarrow{b} \cdot \overrightarrow{p}$, the function computes $\overrightarrow{b} \cdot \overrightarrow{p_\perp}$. The above table illustrated how the dot product between the bike vector and the counter-clockwise perpendicular path vector is used to determine the direction to turn for the *turn_parallel* cases. In the navigation algorithm, an output of -1 is equivalent to turning CCW and an output of 1 is equivalent to turning CW.

1.2.4 Choosing closest path to bike

The route that the bicycle has to follow is broken down into a sequence of paths. To choose the closest path to the bike, when the bike is off path, we perform the following procedure. We iterate over the list of paths and find the closest point of each path to the bike. We do this by projecting (x,y) to the line and finding the proportion of the distance of the projection point from point (a,b) and point (c,d) as depicted in the figure below.



$$dist = \sqrt{(a-c)^2 + (b-d)^2}$$

$$d2 = (a-c)^2 + (b-d)^2$$

$$t = \frac{(x-a)(c-a)+(y-b)(d-b)}{d2}$$

If (t < 0): closest point is (a,b)
If (t > 1): closest point is (c,d)
else:
x_coord: (a + t * (c - a))
y_coord: (b + t * (d - b))
closest point is (x_coord, y_coord)

Figure 4: Calculation to find the closest point to the bike on each path

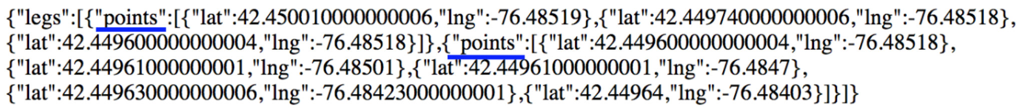
We then calculate the distance to each of those points and choose the path whose point has the smallest distance from the position of the bike.

15.3 Web Path Retrieval

The navigation algorithm requires to take the position and heading of the bicycle as inputs, as well as a list of waypoints that the bicycle has to traverse. In the final system, the position and heading of the bicycle come from the GPS sensor attached to the bicycle. The list of waypoints, which will describe the paths that the bicycle has to follow, will come from user input through the website. Specifically, the Google Maps application program interface (API)[8] is being used to select waypoints on a map. When waypoints have been selected by the user, the API creates a different list of waypoints, often separating segments between two user-selected waypoints into smaller sub-segments by adding more intermediate waypoints. Therefore, by providing a higher resolution of points, the new list of waypoints better describes the shapes of the paths. Using the terminology of the Google Maps API, the selected

[8]Read more about the Google Maps Web API at https://developers.google.com/maps/web/

waypoints are converted into a list of legs of the route. A separate leg is present for each waypoint or destination specified and each leg contains a series of steps. This list of legs is stored on the website server and can then be retrieved via an HTTP GET request.



```
{"legs":[{"points":[{"lat":42.450010000000006,"lng":-76.48519},{"lat":42.449740000000006,"lng":-76.48518},
{"lat":42.449600000000004,"lng":-76.48518}]},{"points":[{"lat":42.449600000000004,"lng":-76.48518},
{"lat":42.44961000000001,"lng":-76.48501},{"lat":42.44961000000001,"lng":-76.4847},
{"lat":42.449630000000006,"lng":-76.48423000000001},{"lat":42.44964,"lng":-76.48403}]}]}
```

Figure 5: Example of JSON representation of paths, represented as a list of two legs that each contain lists of points

The current version of the navigation system has a module named $requestHandler$ which contains a function named $parse\_json$, used to read the webpage containing the list of legs of the selected waypoints in json form. The function converts the json into a Python dictionary data structure and iterates over it to gather a list of all the waypoints. Each waypoint is specified by its latitude and its longitude when retrieved from the web. We created a function called $math\_convert$ inside the $requestHandler$ module to convert each of the points into x-y coordinates relative to a predefined origin. The predefined origin that we chose was the bottom left corner of the Cornell arts quad as depicted in the map of the arts quad in Figure 6 with latitude and longitude (42.44814, -76.48489).

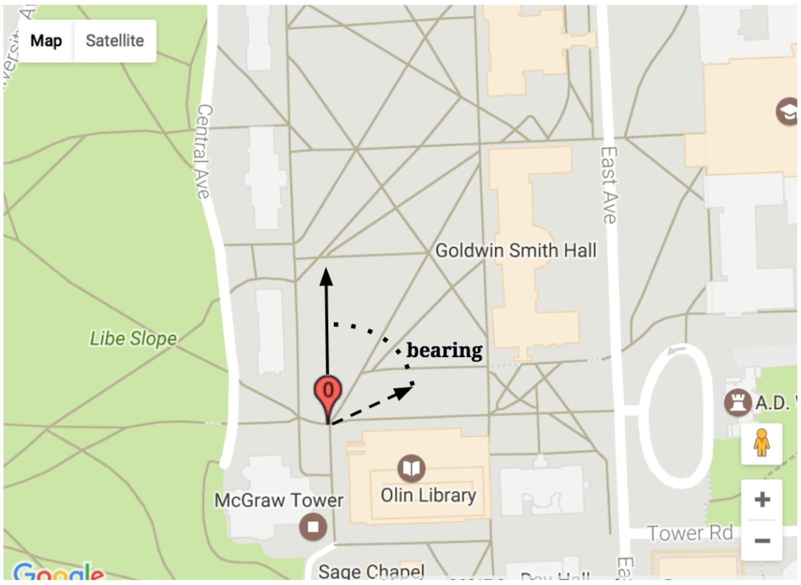


Figure 6: Pre-selected origin for x-y coordinate system of navigation algorithm and example of bearing measurement

We proceed to calculate the bearing, from the unit-vector directed towards the North, of the vectors directed from the hard-coded origin to each of the waypoints. Using the calculated bearing, we convert the waypoints from the latitude and longitude scale to an $x - y$ scale, which facilitates the visualization of the paths in simulation. In order to calculate the bearing of each $(lat, long)$ waypoint from the origin $(origin\_lat, origin\_long)$ we do the following:

$y = \sin(long - origin\_long)\cos(lat)$

$x = \cos(origin\_lat)\sin(lat) - \sin(origin\_lat)\cos(lat)\cos(long - origin\_long)$

$bearing = \arctan 2(y, x)$

The next step after finding the bearing of a point from the origin is to use the bearing and the distance between the point and the origin to calculate its x-y coordinates. The calculations are as follows:

$a = (\frac{1}{2}\sin(lat - origin\_lat))^2 + \cos(lat)\cos(origin\_lat)(\frac{1}{2}\sin(long - origin\_long))^2$

$\theta = 2 * \arctan 2(\sqrt{a}, \sqrt{1-a})$

$d = \theta * rad\_earth$

Thus, the final x-y coordinate value to represent the given waypoint is calculated as follows:

$(d * \sin(bearing),\ d * \cos(bearing))$

These calculations are made for each of the $(lat,\ long)$ waypoints in the augmented list of waypoints retrieved from the web and after each point is converted to an x-y coordinate, it is stored in a list. The final list represents the list of waypoints which is given as input to the navigation algorithm. The navigation algorithm uses the list of waypoints to create a list of paths by forming tuples of points for each of the neighboring waypoints in the given order.

15.4 Bike Dynamics Integration

Our original approach to the navigation algorithm determined the bicycle's direction to turn through relying on the bicycle's turning radius and the bicycle's calculated distance from the target path. We introduced a variable delta (δ) to express the displacement towards the path that the bicycle would incur if it were to turn parallel to the path due to its turning radius. Thus, in the case that the delta was greater or equal to the distance from the target path ($\delta \geq distance$), the algorithm would output to the bicycle to turn in a way as to become parallel to the target path. In the case that the delta was smaller than the distance from the target path ($\delta < distance$), the algorithm would output for the bicycle to turn in a way as to become perpendicular to the target path. The original algorithm was limited as it did not use knowledge of the past or future movement of the bicycle, to make a decision and output a command. In addition, the algorithm did not have knowledge of the bicycle's dynamics and thus operated with the assumption that the bicycle could turn instantaneously. The next step for the navigation team was to test the effectiveness of the navigation algorithm when used by the bicycle simulation in Matlab that took into account the dynamic model of the bicycle when updating the bicycle state after receiving the command from the navigation algorithm.

1.4.1 Approach to Integration

The equations describing the bicycle dynamics were initially written and simulated in Matlab[9]. Our approach to connect the navigation algorithm developed in Python to the bicycle dynamics simulation in Matlab involved calling the navigation algorithm from the Matlab loop that updated the bicycle. To use this model in conjunction with our navigation simulation, we called the python code that contained the navigation algorithm from the command line, passing it the bicycle's position, direction, speed and waypoints as string command line arguments. The arguments would then be parsed in python and converted to inputs of the navigation algorithm. This approach was observed to take a long time to run and terminate the while loop that updated the bicycle state in Matlab. The fact that we had to run python code from Matlab was the main contributor to the inefficiency of the loop that simulated the bicycle in Matlab. This created the need to convert the functions describing the dynamic model of the bicycle from Matlab to Python to facilitate the interaction with the navigation algorithm.

1.4.2 Observations and Problems

The fact that the navigation algorithm did not use knowledge of bicycle dynamics, to determine the direction that the bicycle should turn in, allowed us to test the effectiveness of our algorithm when used by the model of an actual bicycle. The connection of the navigation algorithm and the bicycle's dynamics rendered interesting results and created new challenges for the navigation algorithm. The main observation after the bicycle's dynamics integration was the oscillatory behavior of the bicycle's trajectory about a path. Specifically, we created the situation in which the target path was a straight line and the bicycle would start with a heading that differed from the direction of the path and at a location near the line. The observed outcome of the simulation can be seen in the following plot of the bicycle trajectory for the given straight line path.

[9]The development of this can be seen in Sp2016 report, Section 4 by Arundathi Sharma and Kate Zhou

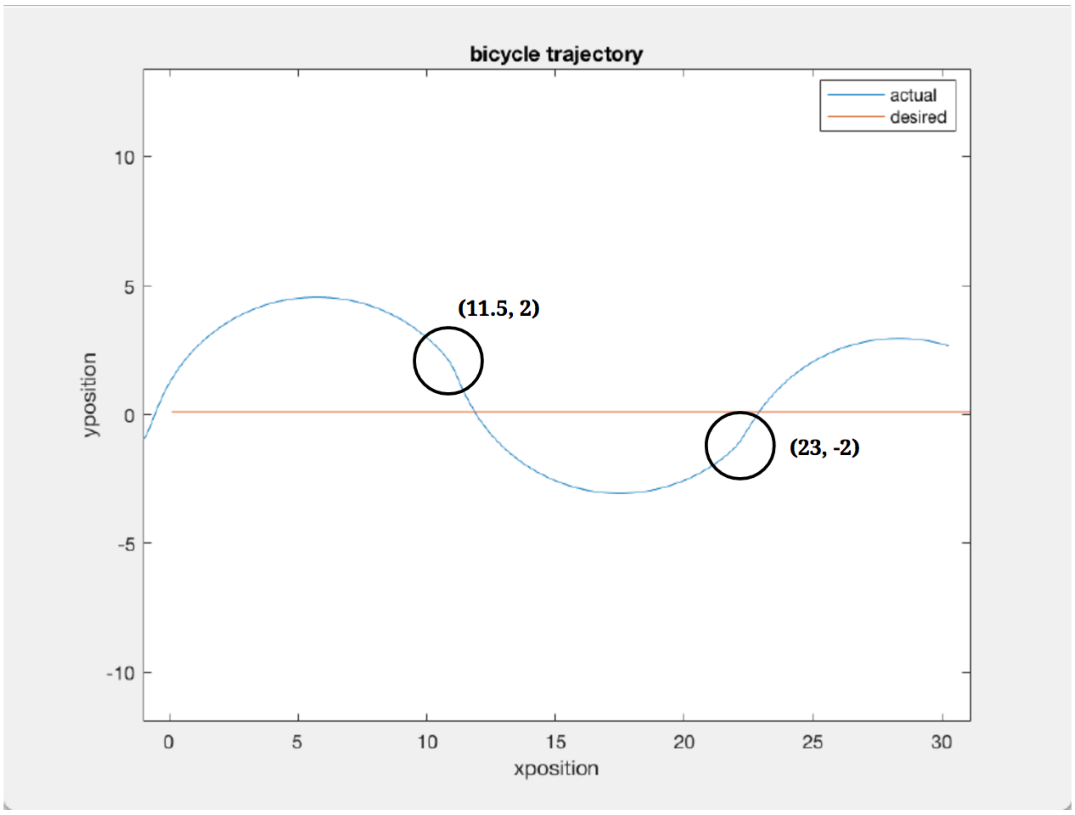


Figure 7: Bike trajectory about a straight-line path

As shown in the figure above, the bicycle overshoots at each one of its attempts to get on the straight line path. In addition, the actual trajectory of the bicycle depicted on the graph abruptly changes slope at a point on each new approach to the line. Specifically, this can be seen at approximately the points (11.5, 2) and (23, -2).

1.4.3 Hypothesis for Cause(s) of Problem

The initial hypothesis about the cause of the oscillatory behavior of the bicycle arose by closely observing the simulation of the navigation algorithm before the integration of bicycle dynamics. By slowing down the speed of the Python simulation of the algorithm, we observed a periodic oscillation of the bicycle about each line it was traversing. In higher speeds, this oscillatory behavior was not evident because the overshoot each time is very small. We determined that the reason behind this was the fact that the navigation algorithm operates in discrete timesteps, which poses a limitation in the ability of the bicycle to exactly converge on a path. That is, because of the constant speed of the bicycle, at each timestep it could only have a certain minimum value of displacement, regardless of the exact displacement which would be needed to converge exactly to the path. Thus, since the bicycle could only move a certain distance between the different timesteps, which remained constant due to the constant speed, it was highly unlikely for the bicycle to exactly converge onto the path and to not overshoot.

We only attributed part of the reason behind the oscillatory behavior of the bicycle to the discrete nature of the navigation algorithm, however, we believed that there were more significant factors contributing to the oscillatory behavior. Thus, the question as to why the bicycle performed oscillations with a relatively large amplitude after the integration of bicycle dynamics remained.

We proceeded to shift our focus to generating different plots of data in Matlab to understand the nature of the relatively large oscillations occurring after the integration of bicycle dynamics. By observing the steer angle over time, we came to the conclusion that a major factor behind the oscillatory behavior of the bicycle was the counter-steering distance that the bicycle required before going into a turn. The navigation algorithm that we developed, operated under the assumption of instantaneous turning and did not account for limitations introduced by the bicycle dynamics. Therefore, the delay caused by the transient motion of the bicycle when turning is not accounted for in the implementation of the navigation algorithm, which led to the large overshoots of the bicycle when trying to converge on a straight-line path.

1.4.4 Possible Solutions

Following discussions with Matt Sheen and Professor Ruina as well as with the controls sub-team, we decided on two possible solutions to the oscillatory behavior of the bicycle. The first solution involved the use of an iterative approach for the prediction of the future overshoot of the bicycle through an internal simulation of the bicycle dynamics. The second solution involved the use of a proportional controller at close distances of the bicycle to the target path. We will present both solutions as well as their results and limitations.

The first of the possible solutions to the problem was the use of an iterative approach to calculate the future overshoot distance of the bicycle, if it were to follow the original nav algorithm, and use that information to adjust the delta (δ) that the navigation algorithm would actually use to decide on a steer command. More specifically, at any point that the navigation algorithm is called, an internal loop that simulates the behavior of the bicycle using bike dynamics would run and command the bike to turn parallel to the line at each iteration. In the internal simulation, when the bicycle reached the point where it would be, within an error bound, parallel to the target path, we would stop the iteration and calculate the vertical displacement of the bicycle from its initial position before the iteration and its final position when it is approximately parallel to the target path. This vertical displacement is depicted in Figure 8 as the red discontinuous line. We would then set the delta (δ) in the navigation algorithm to equal the value of the vertical displacement required for the bicycle to become parallel to the path at that point.

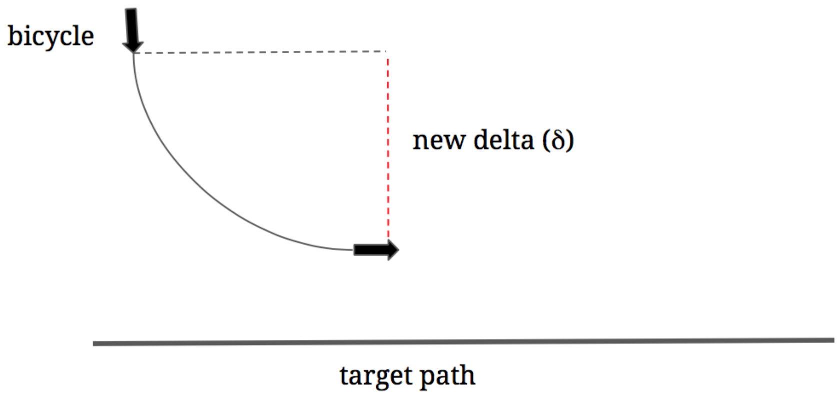


Figure 8: Calculation of delta using iterative approach through internal simulation

We found the calculated value of delta, through the iterative approach described, to always be greater than the original calculation of delta within the navigation algorithm which was mathematical and based on the turning radius of the bicycle. This justifies the fact that the bicycle would always overshoot when trying to get on a target path. This is due to the fact that the expected vertical displacement that the bicycle would incur to become parallel, as calculated by the navigation algorithm, would always be smaller than the actual vertical displacement that the bicycle incurs if the counter-steering distance is taken into account.

An alternative way in which we tried performing the iterative approach to correct the oscillatory behavior of the bicycle is the following. Instead of running an internal simulation that constantly commands the bicycle to turn parallel and then calculates the vertical distance that the bicycle would incur to turn parallel, we would allow the internal simulation run normally and let the bicycle overshoot until the point at which it would normally become parallel to the line. We would then find the perpendicular distance from that point to the target path and add the value of the distance to the navigation algorithm's calculation of the delta (δ).

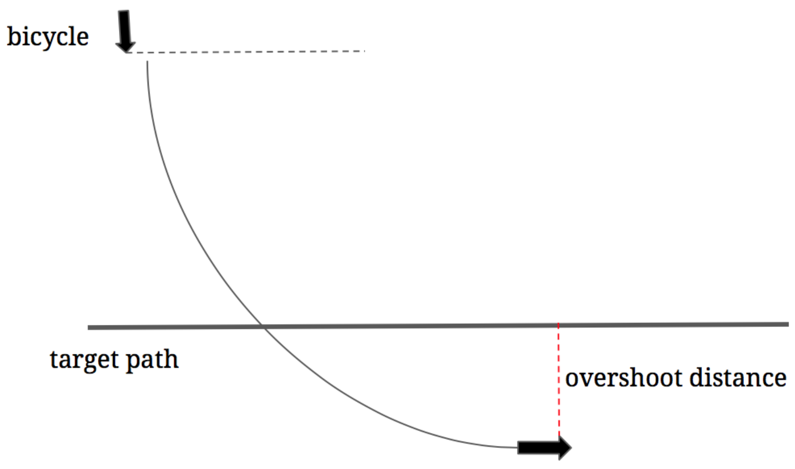


Figure 9: Calculation of the overshoot distance using iterative approach

In other words, we would calculate the future overshoot distance of the bicycle, depicted in Figure 9, and add it to the navigation algorithm's calculation of delta to prevent the overshoot. We implemented both of the methods of the iterative approach and verified that the calculation of the new delta was the exact same in both cases, rendering them both equally precise. The second method to implement the iterative approach required a longer loop of the internal simulator and thus further accentuated the runtime expense of the iterative approach.

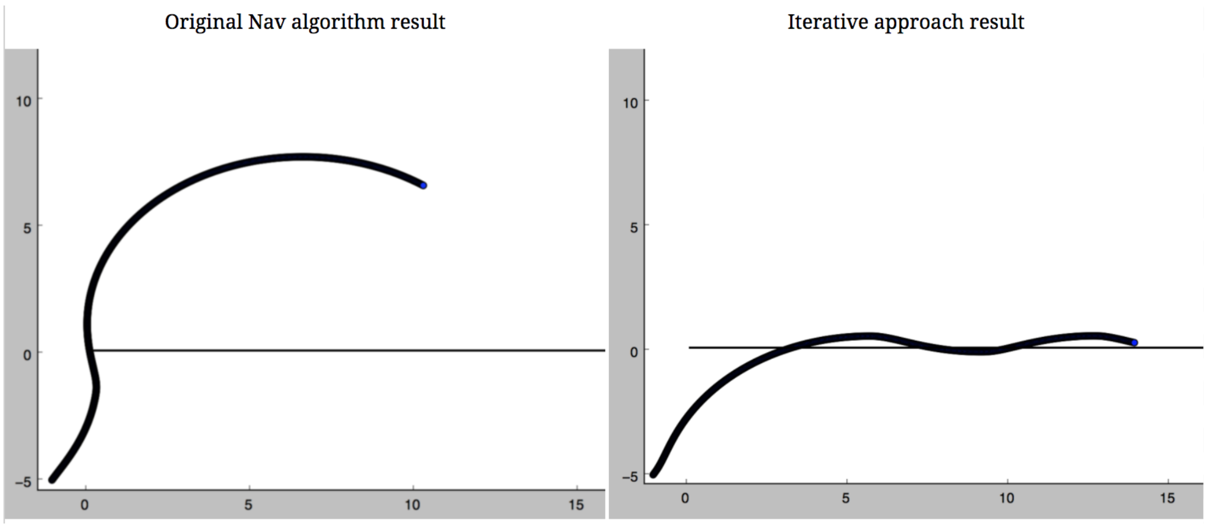


Figure 10: Result of the implementation an iterative approach through internal simulation. On the left we see original overshoot and on the right we see the result after iterative approach

The result of using the iterative approach was a great alleviation of the overshoot of the bicycle. However, it did not lead to the exact convergence of the bicycle on the path without oscillations. This is once again attributed to the discrete nature of the navigation algorithm. The point used to either calculate the vertical displacement or overshoot distance of the bicycle in the internal simulation was supposed to be the point at which the bicycle is exactly parallel to the target path. However, due to the discrete timesteps of the navigation algorithm, the point used was an approximation of the point at which the bicycle is exactly parallel to the target path. Furthermore, the point used was either slightly before or slightly after the point at which the bicycle is exactly parallel. This led to a small error in the correction of the delta, and thus prevented the bicycle from exactly converging on the target path.

A major limitation of the iterative approach that was briefly mentioned is its costly nature with respect to the runtime of the navigation algorithm. In particular, performing the internal simulation, to calculate or correct the delta used in the navigation algorithm, made the navigation algorithm significantly inefficient. Thus, this iterative approach could either be used scarcely within the algorithm or rather be used in advance to generate a look-up table of values of delta for different bike states.

The need of an alternative method to be used such as to make the bicycle converge onto a target path led to the use of a proportional controller. A proportional controller had already been developed in the Matlab simulation of the bicycle dynamics, thus, we worked on implementing it as a function within the NAV module.

We set the maximum steer angle to be a constant and equal to $\pi/6$ which functioned as saturation limit of the proportional controller. The controller accounted for the signed distance of the bicycle from the target path as well as the signed angle between the bicycle's heading and target path measured between $-\pi$ and π. The equation used to calculate the steer command is as follows:

$$steer = nav\_kp_1 * distance\_from\_path + nav\_kp_2 * angle\_from\_path$$

In the above equation, k_1 and k_2 are the proportional gains and the $distance\_from\_path$ and the $angle\_from\_path$ are both signed error values. If the steer value did not lie within the range formed between the negative and the positive value of the maximum steer, then the steer value becomes saturated and will equal the saturation limit for its respective side.

The proportional controller was observed to have the best performance at close distances to the target path and small angles between the heading of the bike and the target path.

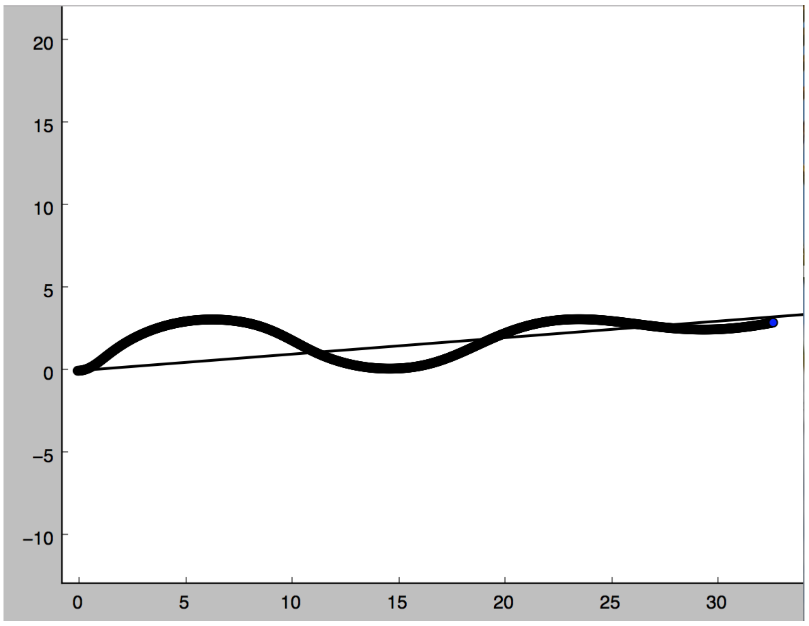


Figure 11: Result of the implementation of proportional controller

As seen in Figure 11, through the use of a proportional controller the oscillations of the bicycle about the target path are dampened and the bicycle gradually converges to the path. By testing different cases and observing the performance of the navigation algorithm vs the proportional controller at small angles and distances from the line, we concluded that the proportional controller is the better solution to the problem of the overshoot of the bicycle due to counter-steering. An additional benefit to the use of a proportional controller over the iterative approach is that the p-controller is efficient and does not result in a major increase in the runtime like the iterative approach.

15.5 System Redesign

The integration of bicycle dynamics in the testing of the navigation algorithm as well as the need to preserve modularity and cohesion led to the re-design of the system. We decided to move dynamic model of the bike and the simulation of the complete system from Matlab to Python. The reason behind this choice was also the inefficiency in the interaction between the two programming languages. In addition, in the Matlab code, which included the bike dynamics, we had a more extended representation of the bicycle state which included attributes such as the lean, the lean rate and the steer angle of the bicycle. In the new design of the entire system, we decided to include all attributes in the bike state class and thus have a single representation of the bicycle's state.

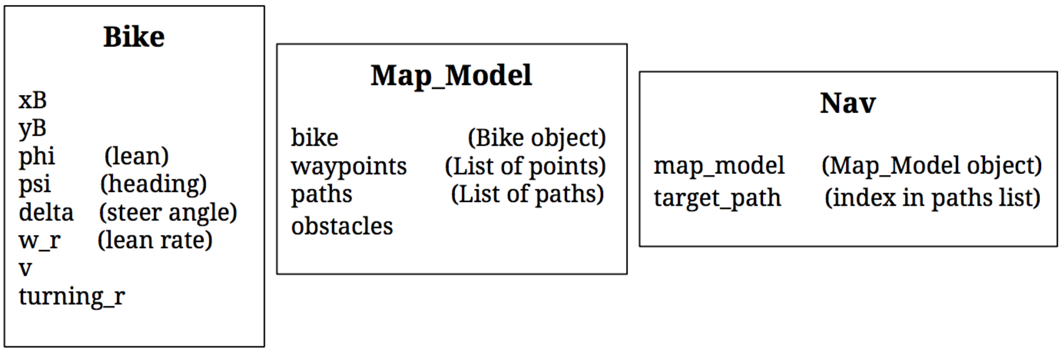


Figure 12: Structure of main classes within modules of the system. Includes bike state attributes after system re-design

The re-designed system has a main loop which has a variable to hold a nav object that contains the map_model object, which in turn contains the bicycle object that describes the bicycle state at any point. In the main loop, a call is made to the either the navigation controller or the proportional controller to get a steer command. Thus, as seen in the module dependency diagram in Figure 13, the main loop depends on the NAV module, which contains the two types of controllers, in order to get a steer command.

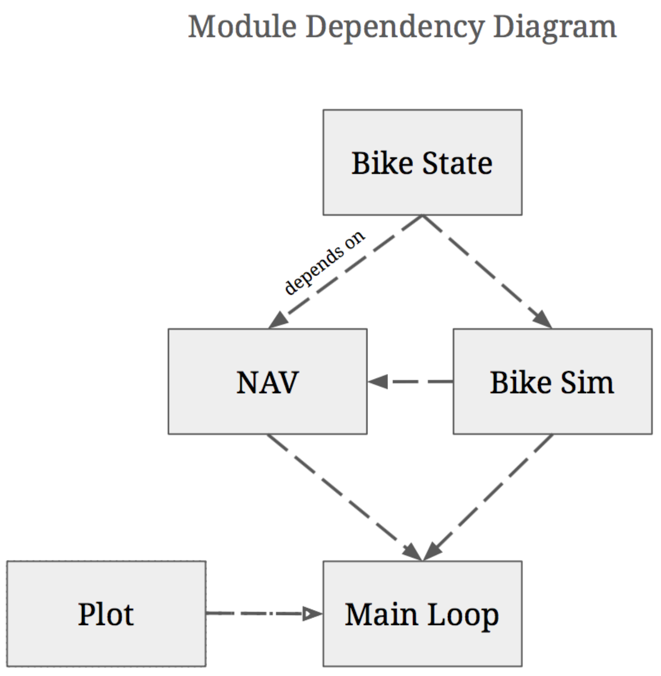


Figure 13: Module dependency diagram of new system design

After a steer command has been acquired, the main loop makes a call to the bike simulation (Bike Sim), which updates the bicycle state by taking into account the dynamics equations that describe the bicycle's motion. The main loop thus depends on both the NAV and the Bike Sim modules, both of which depend on the Bike State module. It is important to describe the existence of a one-way dependency between the NAV module and the Bike Sim module. Specifically, as seen in the diagram, the NAV module depends on the Bike Sim module. This dependency exists in the case that an iterative approach is used to predict the overshoot of the bicycle through running an internal simulation as described in the previous section.

In the new system, we also introduced a geometry module to contain all the functions used to perform mathematical calculations in the main modules. There also exists a constants module containing constant variables and gains used in the main modules. The system continued to have two main external library dependencies, one being NumPy and the other being Matplotlib.

The main benefit of the new system design is its modularity. The logical separation of functionality among the different modules allows for their replacement by different modules with equivalent functionality without disturbing the organization of the system. For example, when we wish to connect the system to the actual bicycle, the only change that has to be made is the replacement of the Main Loop module with a navigation node in ROS.

15.6 Future Plans

In the future semesters, we plan to have both the nav controller and the proportional controller integrated in a weighted linear equation. Professor Ruina stated that the proportional controller will behave worse the farther it is from the path and the greater the angle between the path vector and the bike vector due to non-linearities at large angles. Such an equation would be structured as follows:

$$steerD = w_1(p - controller) + w_2(nav\ controller)$$

where w_1 and w_2 are the weights.

16 Data Communication and Storage

17 Testing (Arundathi and Aviv)

17.1 Introduction

The motivation for a formalized test procedure comes from the first few tests we ran with data collection. Each test took several hours to run and for many either no data was collected because we forgot a step in the procedure, or no useful data was collected because we did not have a clear purpose for that specific test.

The goals of the testing procedure are as follows: simplify testing such that anyone on the team can run a test, decrease time per test and time in between tests, increase consistency between tests, and force each test to have a purpose. With these goals achieved each test will: produce relevant and useful data, does not require to person analyzing the data to be present at the test, can be run quickly and easily, and allow the team to pinpoint which changes in hardware or software cause which changes in the bike performance. In addition, by increasing the consistency between tests, we are setting up a system in which a member of the team can look back at a test from the past and analyze as if they had been at the test.

17.2 Individual Test Procedure

The need for a test procedure was immediate. We began with a general file structure and naming convention on Google Drive along with a document on how to run a test while the formalized procedure was written.

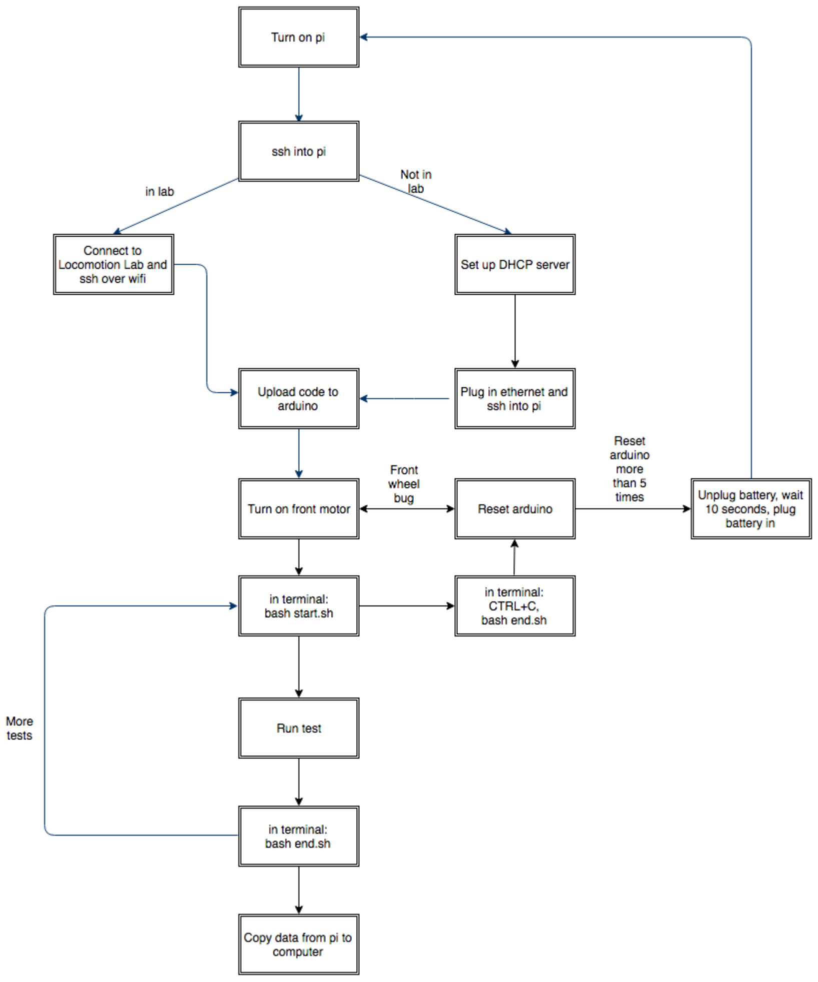
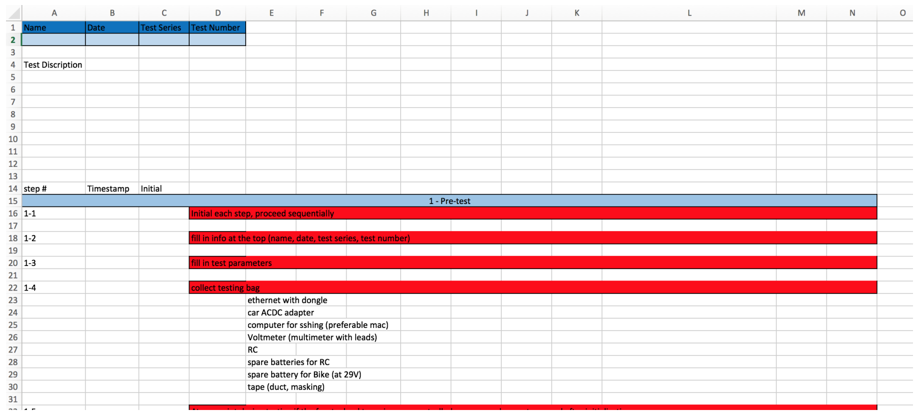

Turn on pi
ssh into pi
in lab
Not in lab
Connect to Locomotion Lab and ssh over wifi
Set up DHCP server
Upload code to arduino
Plug in ethernet and ssh into pi
Reset arduino more than 5 times
Front wheel bug
Turn on front motor
Reset arduino
Unplug battery, wait 10 seconds, plug battery in
in terminal: bash start.sh
in terminal: CTRL+C, bash end.sh
More tests
Run test
in terminal: bash end.sh
Copy data from pi to computer

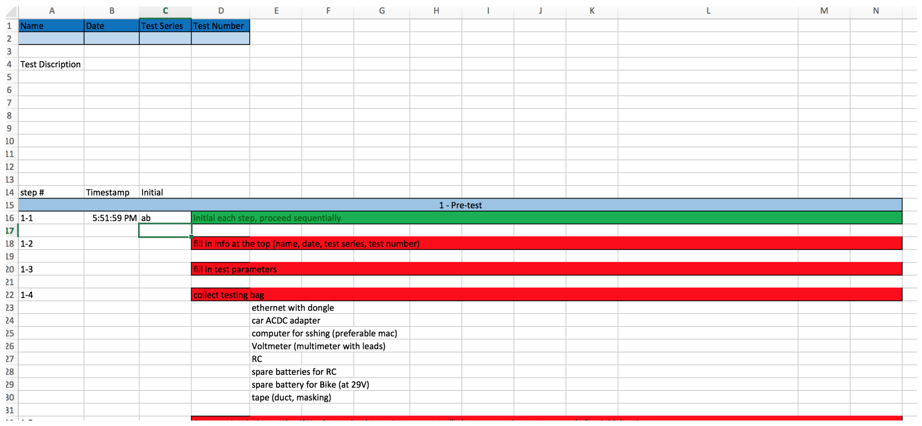
Figure 17.1: Overview of testing procedure

The formalized procedure is an excel spreadsheet with detailed instructions on how to run a test. Each of the steps in the flowchart above have instructions in this spreadsheet. The most updated procedure can be found in the team Google Drive under Spring 2017 → Testing → 0 - Procedure Archive. The procedure also has instruction on how to debug common issues such that someone can run a test and debug some issues without knowing anything about the bike. This spreadsheet is used by the test operator to keep track of which step in the procedure is next and when each step was performed. As more tests are run the spreadsheet will be revised based on requests from the test operators.



Name	Date	Test Series	Test Number	
Test Discription				
step #	Timestamp	Initial		
1 - Pre-test				
1-1			Initial each step, proceed sequentially	
1-2			fill in info at the top (name, date, test series, test number)	
1-3			fill in test parameters	
1-4			collect testing bag	
				ethernet with dongle
				car ACDC adapter
				computer for sshing (preferable mac)
				Voltmeter (multimeter with leads)
				RC
				spare batteries for RC
				spare battery for Bike (at 29V)
				tape (duct, masking)

Figure 17.2: Structure of testing procedure



Name	Date	Test Series	Test Number	
Test Discription				
step #	Timestamp	Initial		
1 - Pre-test				
1-1	5:51:59 PM	ab	Initial each step, proceed sequentially	
1-2			fill in info at the top (name, date, test series, test number)	
1-3			fill in test parameters	
1-4			collect testing bag	
				ethernet with dongle
				car ACDC adapter
				computer for sshing (preferable mac)
				Voltmeter (multimeter with leads)
				RC
				spare batteries for RC
				spare battery for Bike (at 29V)
				tape (duct, masking)

Figure 17.3: Conditional formatting and timestamps in the test to assist the operator keep track of the test

17.3 Main Challenges in Testing

When testing the bike we test the system as a whole. As a result we have found many unexpected bugs or errors with the system:

- Data communication bugs found from testing:
  - Improperly stopping the bag process at the end of the test (fixed). For more information about how data communication through ROS works and how data is stored look at the Data Communication section of this paper.
- Hardware/Electrical:
  - Rear motor controller cannot ramp up without first using RC to start the motor (still not fixed). The physical rear motor controller box has safety precautions to limit how quickly the the motor can ramp up. We think that our current ramp up method of increasing the PWM signal to the motor by 10 each loop (the loop runs at 100Hz) is too quick for the motor controller. When ramping up with the RC controller we are increasing the PWM signal at a much slower rate. Since we can ramp up with the

RC controller and then switch over to ramping up at a rate of 10 PWM per loop we believe that this issue can be fixed with a slower ramp up procedure during the bike's initialization procedure.

– Front motor spins uncontrollably after initialization process (still not fixed). There are a few possible reasons for this bug. When the issue occurs it is consistently after the front motor finishes calibrating so the issue could be related to how the front wheel is calibrated. Another possible reason is damage to the front motor. Whenever the bike falls during testing the front motor can potentially be damaged. In the past we have seen when a brush in the front motor is broken there is an increased resistance in the motor. With this motor we have noticed that the front wheel's resistance to turn has increased over time. The increase in resistance correlates to an increase of how often the front wheel spins uncontrollably after initialization so the two behaviors may be connected.

17.4 Future Work on Testing Procedure

The spreadsheet for individual tests is not enough to achieve the goals of a formalized test procedure. In addition we need a place where people can order specific tests to be run and a database.

17.4.1 Test Matrix

This is a collection of tests to be done that include the goal of that specific test. The goal for a test should answer the questions of why we are running this test, what data from this test are we looking at, what will the results of this test tell us about the performance of the balance controller/navigation algorithm. An example of a goal for a navigation test is: "Test if the GPS update rate is fast enough to provide the navigation P controller fast enough information. Start the bike on or near the path facing forward and see if the bike converges onto the path for a given GPS update rate. Isolate the navigation algorithm from the balance controller by running this test with the landing gear down. Relevant data will include comparing GPS update rate to navigation algorithm update rate and comparing the bike's desired and actual path." In addition to a goal, all of the parameters that the test should be run at such as controller gains, bike velocity, type of test (balance or navigation), GPS update rate, etc. will be provided in the test matrix.

17.4.2 Database

The database will be built off the SQL database to store the data files from the test (bag files and csv) as well as other data points. The operator will have this database open as they run a test and fill in the parameters of the test from the test matrix into the database. There will also be section for the operator to write notes about any anomalies/observations/notes regarding the test.

17.5 Hardware Debugging

When running tests the bike will sometimes fall. Described are the methods used to check the bike after a fall and to isolate hardware issues that might have occurred after a fall. After a fall a safety switch should be hit to turn off the motors and the bike should be taken to the bike stand. There, it is necessary to check that all hardware is still functional since it is sometimes the case that falling will cause electrical or mechanical issues. This semester, issues have included: rear electric box getting angled to the side (yaw), wires coming loose, landing gear becoming bent, and front motor brushes breaking. Once the bike is on the stand it is necessary to end the test file so that no data is lost from the test. End the test as normal, as described in the testing procedure spreadsheet. If you are unable to connect to the Raspberry Pi check that the red light on the Raspberry Pi is on. If it is not lit, the Pi is not receiving power, check your connection to the Pi. The bike should have been killed using one of the safety switches immediately after the fall. Once the bike is on the stand, these switches should be turned back on so that the bike is not turned off, but the front and rear motor switches located on the side of the Ammo Box should be turned off. If the bike is not able to be turned back on by just turning on the safety switches, check the LEDs on the board. If the red LED is on then the battery is dead or disconnected. If the green LED is not on there is no power to the board, which may also mean the battery is dead or disconnected. If necessary, check the battery voltage using a voltmeter, it should be greater than 23 volts, and check the connections to the safety switches, pictured below. If the plastic connections are not well secured the battery may not be connected to the PCB. The plastic connections may look as if they are correctly linked but the wire leads in the connection may have shifted, breaking the circuit.

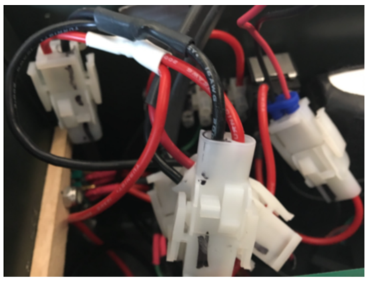


Figure 17.4: Safety Switch connections

Once there is power to the bike, check the front motor by turning it on. The front motor should rotate clockwise twice to calibrate and then point relatively straight. If the front motor starts spinning uncontrollably or acting erratic in other ways the arduino should be reset. Turn the front motor off and press the reset button located on the underside of the Arduino, pictured below.

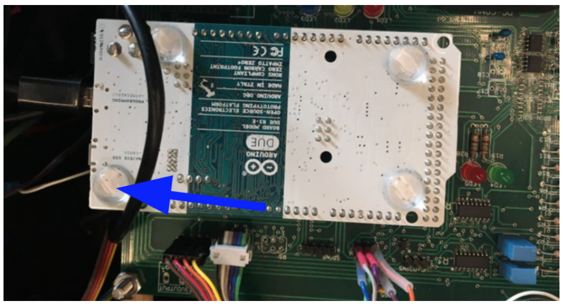


Figure 17.5: Arduino Reset Button

If the issue persists after resetting the Arduino and turning the front motor back on, the best solution we have found this far is to unplug and replug the battery. This method is not preferred since, in the current wiring configuration, unplugging the battery also shuts off the Pi. Also if the front or rear motor are not reacting at all, one should check the front and rear motor connections, pictured below.

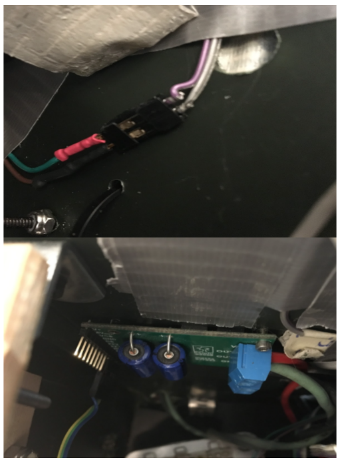


Figure 17.6: Rear Motor(top), Front Motor(bottom)

If safety switches are not activated, the battery and motor connections are good, the battery has enough volts, and the Arduino has been reset, the issue is most likely on the PCB and will need to be examined more closely. If the motors seem to be working, check that the Pi is receiving power by checking the red LED on the Pi. If it is not lit then check the power connection labelled PWR IN on the Raspberry Pi case. If the Raspberry Pi and both the motors appear to be working properly, one is ready to try running a new test. Note that this is a simplified version of all the hardware issues that could go wrong during a test, especially after a fall. Described in this section are solutions to hardware issues we did encounter during testing, but it does not cover all possible hardware issues. If these do not help in finding a hardware issue during a test, one should check all connections from hardware to the board.

17.6 Testing Goals and Data Analysis

With the above testing procedure in section 1.5 finalized, we were able to collect useful data from the bicycle prototype as it ran and compare the prototype behavior to that of the bicycle in simulation. Ensuring there is a close correspondence between simulation and prototype is important for future improvements and developments of controllers. With a reliable simulation, we will be able to test many, many controllers much more quickly, and find an optimal controller for our purpose more easily than if we had to test each one on our physical prototype.

The data we collect (or that we are particularly interested in for comparing simulation dynamics to prototype dynamics) is the bicycle state. That is: the lean angle, lean angular rate, steer angle, and forward velocity of the bicycle. These are the properties (state variables) we can compare between simulation and prototype to make sure there is close correspondence between the two.

Early tests were not used for comparison with simulation; in order for us to meaningfully compare the two, we had to make sure certain conditions in the simulation were met in the prototype. For instance, the simulation assumes the bicycle has constant forward velocity; so, we had to develop a controller and use tests run on the ground to develop a reliable forward velocity controller (see Michelle O'Bryan's section on this).

Regardless, we wanted to have an idea of how well the bicycle could handle disturbances and get a sense of the robustness of the controller. After the bicycle had reached stable straight-line motion, we hit it with disturbances of various sizes, and observed the recovery response.

It is **imperative** that there is no remote control interference in the bicycle behavior during the test period for balance robustness tests. If the bicycle is disturbed during a test it should not be steered with RC, because that would affect the recovery behavior. The data would therefore not tell us about the controller's ability to balance.

Unfortunately, we only learned this after our first round of tests, when we realized that there were non-constant RC-inputs, as shown in figure 7 below:

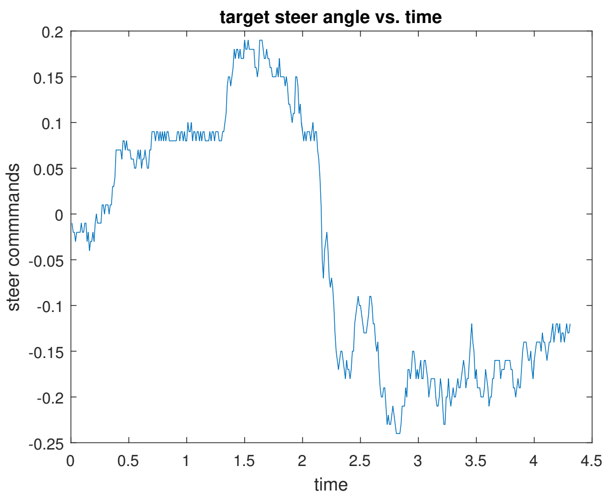


Figure 17.7: The figure revealed that the bicycle was being steered during our data collection, so data collected during this disturbance test will not tell us much about how well the bicycle balances

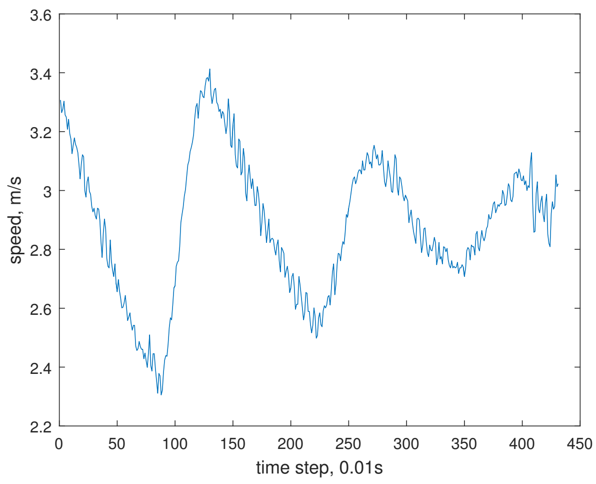


Figure 17.8: The velocity was also not constant during this test, and this would have affected our balance and how well the behavior would match up with simulation

Since the current dataset we have does not have good velocity control and also has RC interference, the following comparison plots between data and simulation were obtained using a modified bicycle dynamics simulation, in which the velocity and desired steer angle were updated using information directly from the data. Although the data cannot tell us much about the robustness of the controller, it can show us how well our simulation and actual bicycle behavior correspond. Sometimes the inputs were significant, like the ones shown in the above two figures; in other cases, there were minimal RC interference, or the velocity stayed mostly constant. Even in those cases, we used input from the data, for the sake of consistency.

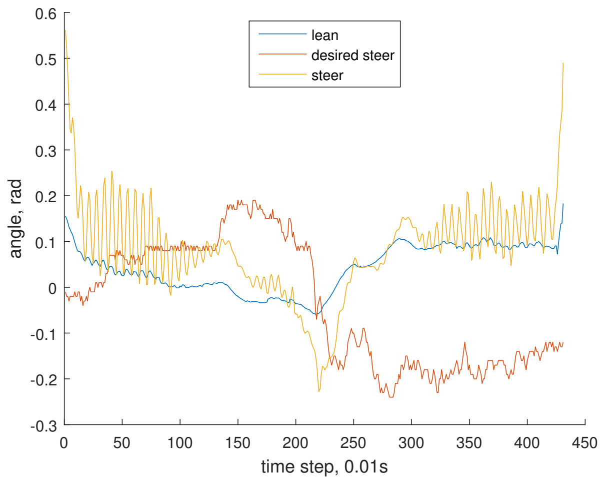


Figure 17.9: Example of a plot of data collected during our test: We want to look at the lean and the steer angles of the bicycle during the test and compare this plot to the one generated using the simulation

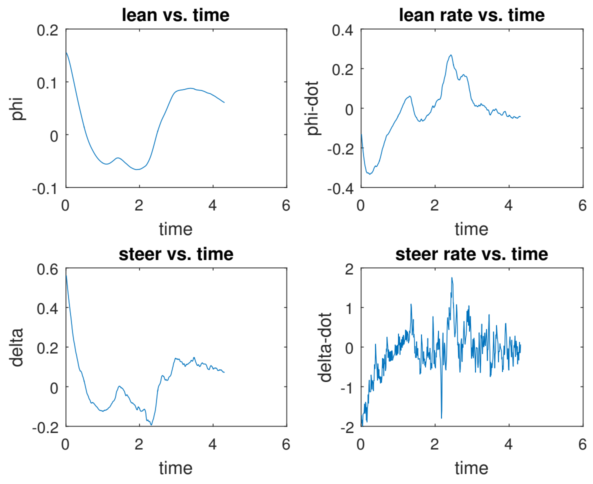


Figure 17.10: With the input velocity and desired steer angles, we ran the simulation and generated this plot of bicycle recovery behavior from a disturbance initial condition taken from the actual test data. This is a standard subplot figure generated from the simulation we have

It is more useful to see these plots on top of each other. We used the simulation plotter to show both the predicted and the actual behavior of the prototype on the same axes.

We made modifications to the existent simulation so that plotting this simply requires a user to plot the data using the plotting function (included in the appendix), choosing the range of data indices that are of interest, and then inputting that range in the simulation. However, choosing that range of data is non-trivial. We are particularly interested in knowing the behavior of the bicycle in response to a disturbance, during roughly constant-velocity forward motion. In the future,

tests should be run with timestamps at every relevant point (disturbance applied at such-and-such time) so that cross-checking data with videos and selecting what to plot in simulation will be easier. This should happen once the testing procedure laid out earlier in this section (section 1.5) is implemented.
The following four figures (1.11-1.14) are examples of data collected from disturbances of varying sizes.

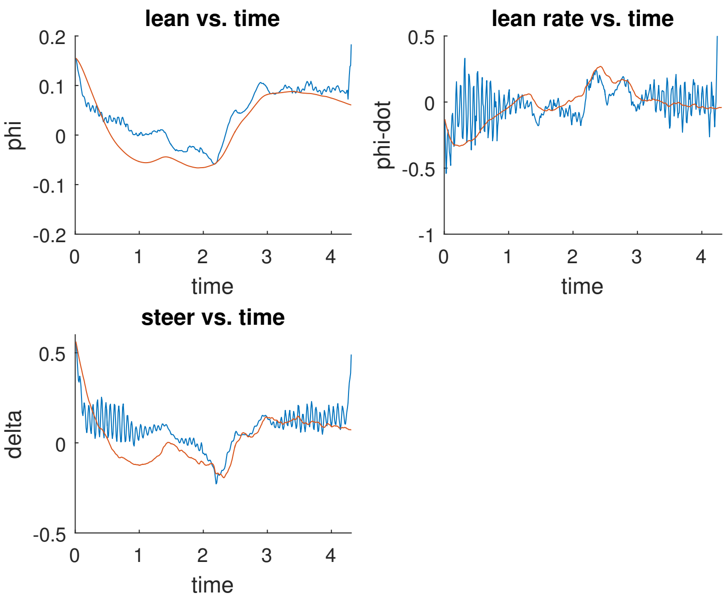


Figure 17.11: Blue plot is from data, orange plot is from simulation

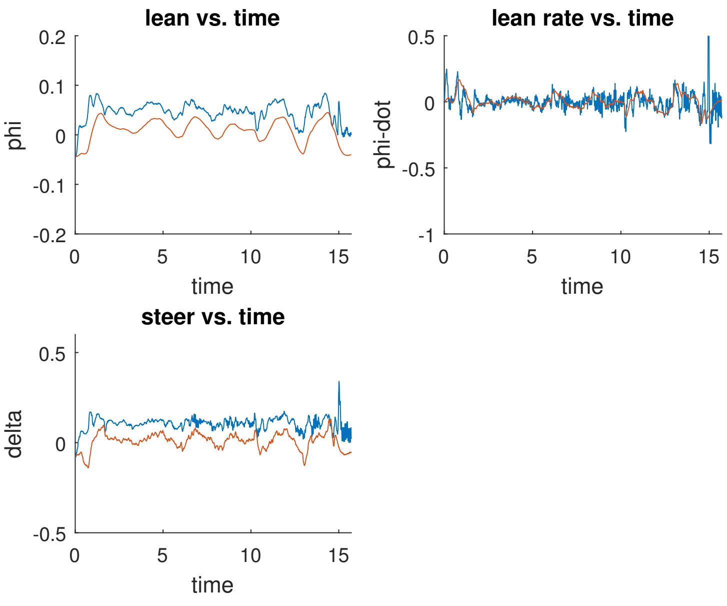


Figure 17.12: Blue plot is from data, orange plot is from simulation

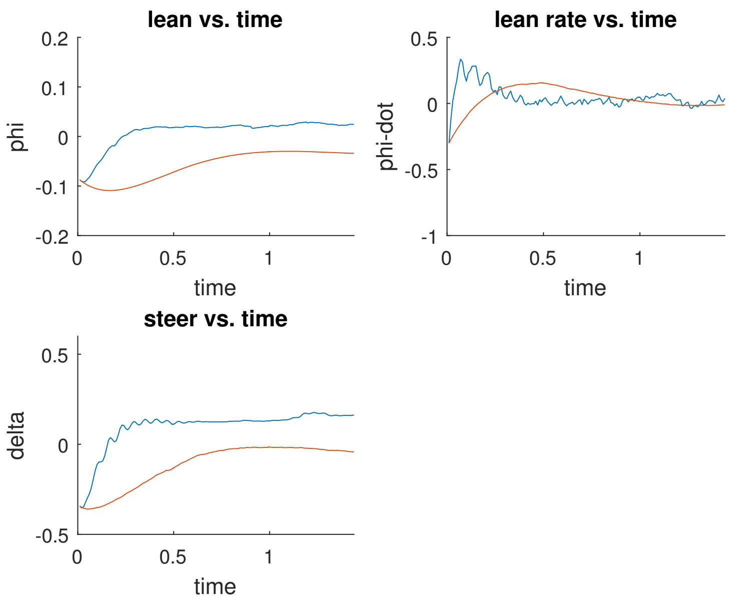


Figure 17.13: Blue plot is from data, orange plot is from simulation

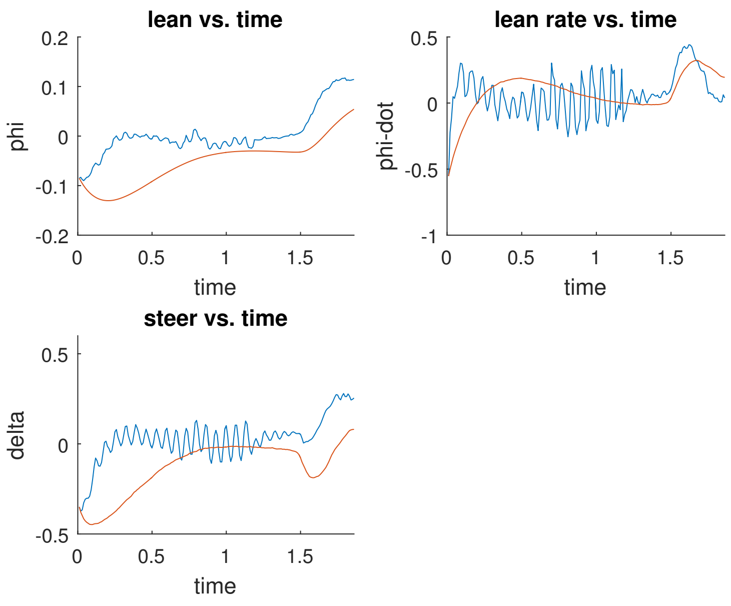


Figure 17.14: Blue plot is from data, orange plot is from simulation

In general, the shapes of the plots are similar and seem to correspond well, meaning the expected behavior of the bicycle, as far as how it corrects, is what the simulation predicts. Let's summarize some of the differences: first, a lot of the test data has oscillations that don't appear in the simulation. A lot of the noisy oscillations came from instabilities in the front motor during balance recovery or unstable behavior from the balance controller itself (adjusted but not tuned on the physical prototype) ***The front motor and balance controllers need tuning to reduce these instabilities***.

Figure 1.13 shows the largest discrepancy between simulated and actual behavior, while figure 1.11 shows the closest correspondence between the two. The discrepancy, given that the response behavior is roughly the same, has to do with an offset. We do not yet know the source of this offset, and it is inconsistent between tests. Figures 1.12-1.14 show similar shapes, as mentioned earlier, but show a substantial offset, especially in lean and steer (the lean-rate seems to be more consistently matching).

We do not know why the lean-rate behavior on the bicycle seems to match the simulation well, but the lean and steer do not. Since further tests on the balance controller were not done this semester, we do not have any data about whether this discrepancy would change when the controllers are tuned. It is possible that ***the reason they do not match up has to do with the tuning of the front motor or balance controllers***. Before we can make more conclusions on how well the simulation and prototype behavior match up, we would need to collect more test data to see if changing gains on controllers affects the correspondence between simulation and prototype behavior. These controllers must anyway be fine-tuned before the bicycle can be trusted to reliably and robustly balance on its own. And finally, they must be tuned before we can run many thousands of simulations on different controllers to find one that will make our physical bicycle balance better than any other steering-balanced bicycle before it. If there is not close correspondence between the data and real-life behavior of the bicycle, then these simulations will not yield useful results.

Note that in figure 1.15, the average RC input if we do not touch the controls is at roughly 0.08rad. On the bicycle the effect of this offset in steering on the lean angle is on the scale of 1-2 degrees, which is within the error bounds of our sensor and should not affect how well the bicycle can balance. Also, this steering input is accounted for in the simulation, so this does not explain the offset between data and simulation.

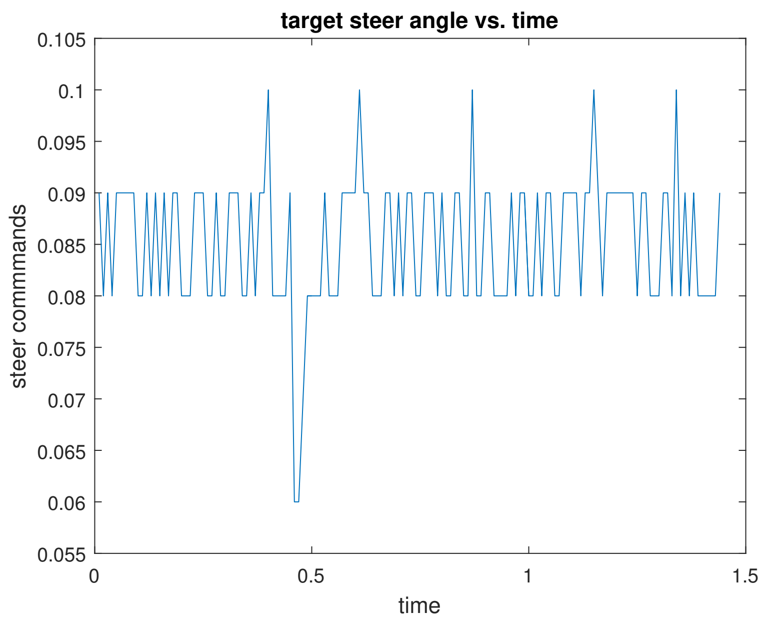


Figure 17.15: If we do not touch the RC controls, the desired steer angle input looks something like this.

A lot of testing work remains to be done–indeed, the majority of the testing work done this semester was spent on developing our testing procedure instead of actually running tests. Now that the testing procedure is finalized, the following goals should be top priority next semester:

- Run disturbance tests (like the ones shown here) without RC input. Change front motor controller to confirm that the motor is not lagging and is responding quickly and accurately to the steering commands.
- After the front motor controller is deemed satisfactory, adjust balance controller. We should understand when/what gains will make our bicycle behavior unstable and produce oscillations like the ones we see in our data now.
- If changing balance/front motor controllers does not seem to improve the discrepancy between actual and expected bicycle behavior, then we need further testing/investigation

18 Vision

Due to the state of the bike this semester, it was unlikely that a vision system could be incorporated into the bike. Therefore the primary role of vision this semester was to determine what goals the vision needs to accomplish and what hardware would allow for completion of those goals. The two main objectives that we identified were bicycle localization and obstacle detection/avoidance.

18.1 Obstacle Avoidance

For obstacle avoidance, the system must be able to generate some sense of depth from a single view. Therefore, there are only two feasible hardware options: 2D LiDAR and a Stereo Camera.

- 2D LiDARs measure depth by timing how long ejected light takes to return to the sensor. The longer the time of flight, the farther away the object. LiDARs are computationally cheaper and most hardware performs depth computation with an onboard embedded chip, but they are also limited to a single plane of detection. This means that if the sensor is placed horizontal to the ground at handle-height, it would be unable to detect curbs. This can be partially solved by angling the sensor downward so that its sensing is not restricted to a single horizontal plane, but this setup would still be extremely limiting.
- In contrast, a stereo camera uses image registration between two adjacent cameras to determine depth. These cameras identify pairs of pixels in the two cameras that correspond to the same real world point. The orthogonal distance to this real world point can be computed using the known distance between the two cameras and the relative positions of the two pixels corresponding to that point. Using a stereo camera, an entire point cloud can be generated to measure depth for each pixel. While this technique produces much more usable data than the LiDAR, it is also much more computationally expensive. Any system using a stereo camera would also require a computer capable of handling these computations. Another added benefit of using a stereo camera is that the same equipment can be used for object detection. This would help the vision system distinguish between different types of obstacles (i.e. people vs trees) so that the bike can respond accordingly.
- A third common system uses infrared structured light to measure depths. Structured light 3D sensors project light patterns and infer depth by measuring how the pattern is distorted. This technology is used in products such as the Microsoft Kinect and in Google ATAP's Tango. While these products work extremely well indoors, structured light approaches perform poorly for larger distances, making these options infeasible for the bicycle.

18.2 Localization

Although some localization is done with the GPS, the precision of our hardware is only approximately a meter. Visual odometry paired with the GPS data using Kalman filtering will allow for a much more precise localization. To perform visual odometry, we would use some implementation of the Simultaneous Localization and Mapping (SLAM) algorithm. SLAM works by taking combining views of the same scene to simultaneously create a global view of the setting and localizing the sensor within it.

This process can be done with one of three hardware options: a 2D LiDAR, a single camera (monocular SLAM), or a stereo camera. Because the SLAM has to match key points between viewpoints, the more data associated with the sensor, the better the results. Therefore, 2D LiDAR would most likely be infeasible to use for precise localization. Not only does 2D LiDAR only capture data from a single plane of the environment, it is largely limited by range so in an outdoor setting, the data would be extremely sparse. The second option, monocular SLAM using a single camera, provides significantly more data than the 2D LiDAR but also requires more processing power to compute. A single camera, although cheap, would also not be able to handle obstacle detection on its own and would therefore need to be paired with another piece of hardware. The final option for SLAM would be a stereo camera. Because a stereo camera has two views of the same scene and can compute a point cloud of its environment on its own, it has by far the largest amount of data at its disposal for image registration between views.

18.3 Hardware

The biggest limitations in choosing vision hardware are the size and, more importantly, the cost. Vision systems often have an immense capacity to become extremely expensive. For example, the 3D lidars used on many self-driving cars can cost as much as tens of thousands of dollars. For our own system, we aimed to keep the total price under $1000, preferably about $500. For LiDARs, the top two options we identified were the Scanse, a $250 device with a 40 meter maximum distance and a 500 Hz update rate, and the RPLidar, a $500 device with only a 6 meter maximum distance but a 2000Hz update rate. For stereo cameras, the only well-reviewed sensor within our price range was the Zed, a $500 device. In addition to the sensor, we needed to purchase a new computer to handle the computation because our Raspberry Pi would not be able to require the requirements of the system. While any system would need a new computer, the Zed camera also requires an NVidia GPU as well. After weighing the options, we settled on purchasing the Zed Camera with a Jetson TX1 developer kit for a total of $500 using an NVidia combo deal.

19 Business

The business sub-team 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.

19.1 Recruitment

I lead the outreach for recruitment this spring semester, and I would say it was very successful. One part of the team that I am most excited about is the passion behind the new members. During this process, I designed a series of quarter cards through the use of Microsoft Publisher. I distributed these quarter cards on the engineering quad always as a library on campus. In addition, I also posted messages across various Facebook pages including the Cornell Computer Science page as well as various class of pages. Another main thing that we tried to do during the recruitment period was essentially reach out to students that we knew from classes or other organizations on campus. I found this to be very helpful because in addition to the application and the interview we had another set of information on these potential new members. In total, we received 24 applications and interviewed a wide-variety of majors. I was in charge of scheduling and coordinating the interview times with each applicant. I found using Google forms was a very good tool throughout the recruitment period for both the application and the interviews.

This team has the opportunity to be very successful for reasons including skill-set and dedication. As the semester progresses, it becomes evident that a core aspect of how our project team succeeds is that the members of the team fit the "mold" of the team. In general, sub-teams and people of different majors are able to work cohesively and as a unit. Thus, as the team looks towards next semester, we are focusing on maintaining that mold in our new personnel for fall 2017 recruitment. In that sense, we are hosting a small recruitment event for students that are classmates and friends of current team members at the end of this spring semester. This will consist of a laboratory tour and demo of the bicycle, plus the chance to talk to many members of our team. All in all, hopefully this end of the year recruitment event gives us a head start for the upcoming semester.

19.2 Purchasing, Budgeting, Grants

To continue off last semester, the purchasing account is officially setup. Purchases can be made through either eShop or I Want Docs, which streamlines the ordering process. Through monthly finance meetings with Emily Ivory, I am able to track our different accounts, both gift and our main spending account. In combination with the spreadsheet that she provides us, I am looking to have a better forecasting system. This is especially important for our new sub-teams, steer-by-wire and visions teams. The steer-by-wire team provided a "living" spreadsheet of expected expenses. This was very helpful this semester as it let me know how much money we had as a team to spend for sub-teams. I have been much more alert this semester in being conscious of the budget and talking to team members about their plans for the coming semester.

I applied for the CEAA Student Project Team Award, which would 1000 dollar gift. This team award is awarded to the team that exhibited the most growth and highest-level of engineering skills as a team. Unfortunately, our team did not win the award. Instead, it was awarded to Data Science and Engineers Without Borders. Below is an excerpt from my application:

The Cornell Autonomous Bicycle Team is at the forefront of autonomous vehicles. We are in the process of making the world's best autonomous bicycle that can self-steer and self-navigate. We recently doubled our team size and each of our students are truly committed to this project. This semester we have expanded our electrical and mechanical sub-teams. We have formed four new sub-teams: Control, Navigation, Vision, and Web Integration/Data Processing. The sub-teams work together cohesively as follows: a user inputs a destination on an application which uses Google maps. The web team uses a server to transfer this data to the bicycle so that the navigation module can instruct the bicycle where to travel on the map. Subsequently, the vision team uses sensors to gain understanding of the environment and classifies traversable and non-traversable ground. All of this information is then used by the control algorithm to allow the bicycle to balance and follow the steering commands.

Regardless of the outcome, this was a valuable experience for me in terms of writing and talking about the team in a technical sense. This grant application will also be useful for other applications in the future. For instance, I am currently in the process of applying for the Engineering Innovation Award. This award is described as an unrestricted competition for the most innovative and best developed innovation concept. As a whole, I believe that we have a very strong chance of winning this competitive award.

19.3 Outreach Events

The only outreach event we have been a part of thus far this semester is the Project Team Blitz. The setup of this event was a reverse career fair in which employers were able to come up to respective project teams and discuss new activities that the teams are working on. The main takeaway from this event is that I want this team to have more of a presence among other project team organization. Autonomous Bicycle is one of the most unique and quickest growing teams. Our new members are very passionate about this team and our team deserves to be recognized for that.

Over the course of the semester I have compiled a list of potential events that our team should attend: BOOM, GiveDay, MakerFaire, Project Team Blitz, and other project team sponsored events. These possible events include being selected to present at Cornell Days, Parent Fellowship and possibly at Graduation. Currently, without a working prototype I do not believe we should present that these types of events; we are still a new project team and need to show more results in order to be successful.

19.4 Logistics

One of the main parts of remaining in good standings as an engineering project team is making sure we stay up to date logistically speaking. In terms of that, I attended the Leader's Roundtable meeting for Logistics with other team leads to discuss what steps we need to take to continue to be successful. I have updated the leaders access folder from the engineering project team blog. These updates include contact information, new team roster, hold-harmless forms, and a one slide description of the team. In terms of developing and marketing our team, I outlined a sponsorship package that details who we are as a team and why it is important to donate to our team. As the team continues to grow and sub-team projects evolve/come together, this will be a very important tool in terms of marketing ourselves to receive funding in the future. I worked on the application to renew our project team status. This is a yearly action that we will have to continue to do every spring. Similar to that of the CEAA application, this application was very useful in terms of selling our teams strengths and identifying our team's weaknesses. We were able to begin to start identifying our goals for the following two semesters, which was a difficult test. As a team without a competition, our goal is pretty set in stone: finish the prototype. We will have the opportunity to develop other prototypes along the way, but, in general, our goals are very subject to change as this team progresses.

19.5 Safety

This semester I became the safety officer for the team. This position includes the responsibilities of attending monthly safety meetings to discuss the new ELL lab space for project teams in Upson. I was also able to edit and rewrite our safety plan to meet the specific requirements (even though we do not fit the bill of most project teams. A majority of our report includes information on office safety as well as general organization due to the fact that our team does not work with high level machinery. Even though our team's safety report differs from that of others, it is still a very important living document to create.

20 Conclusion

21 Appendix

Figure 7.1: Olav testing the bicycle in the hallway of Upson 5th floor.

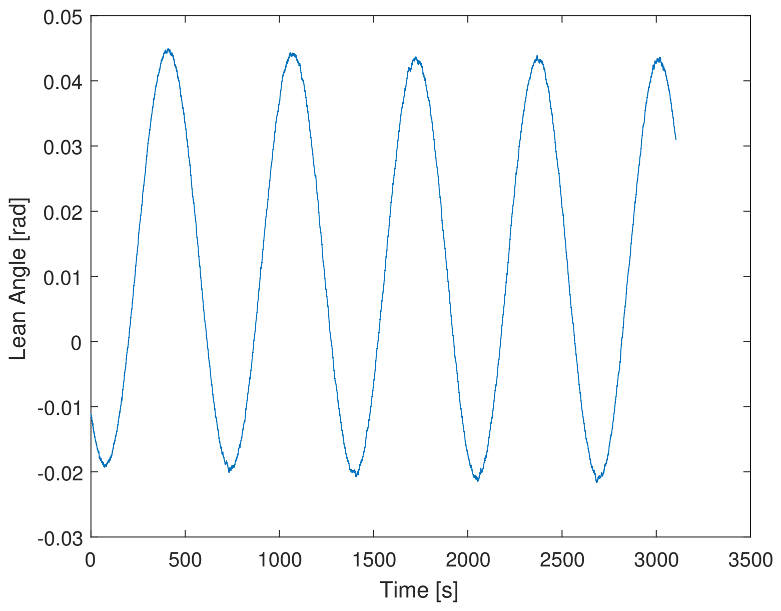


Figure 12.1: Roll angle results from a 50 minute test of a motionless bike

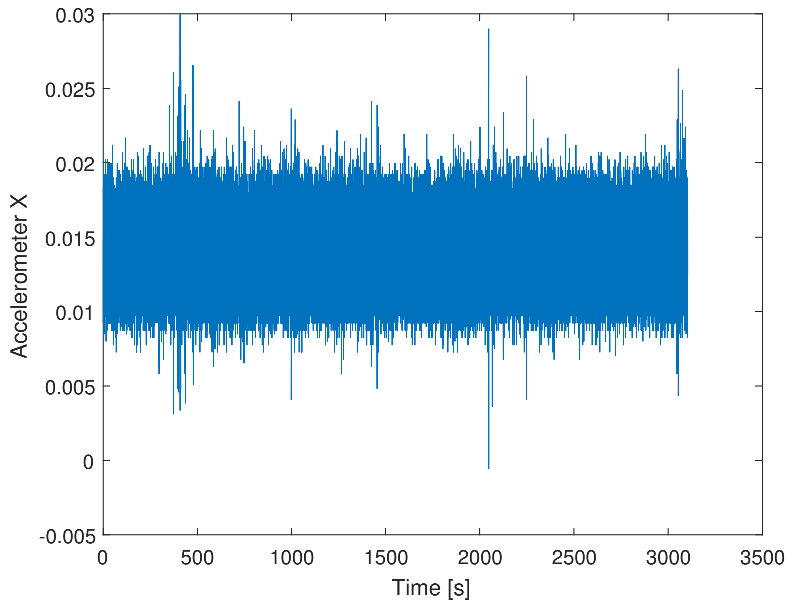


Figure 12.2: Accelerometer 'x' data from a 50 minute test of a motionless bike

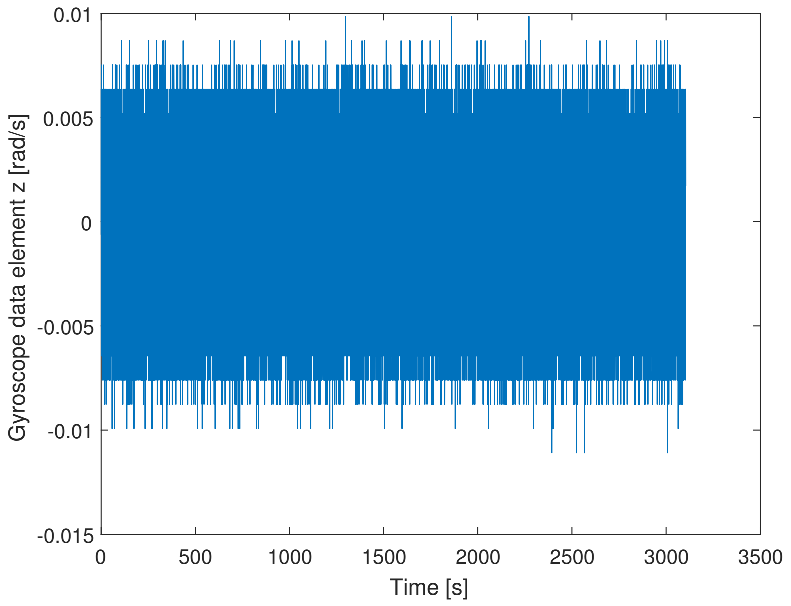


Figure 12.3: Gyroscope ‘z’ data from a 50 minute test of a motionless bike

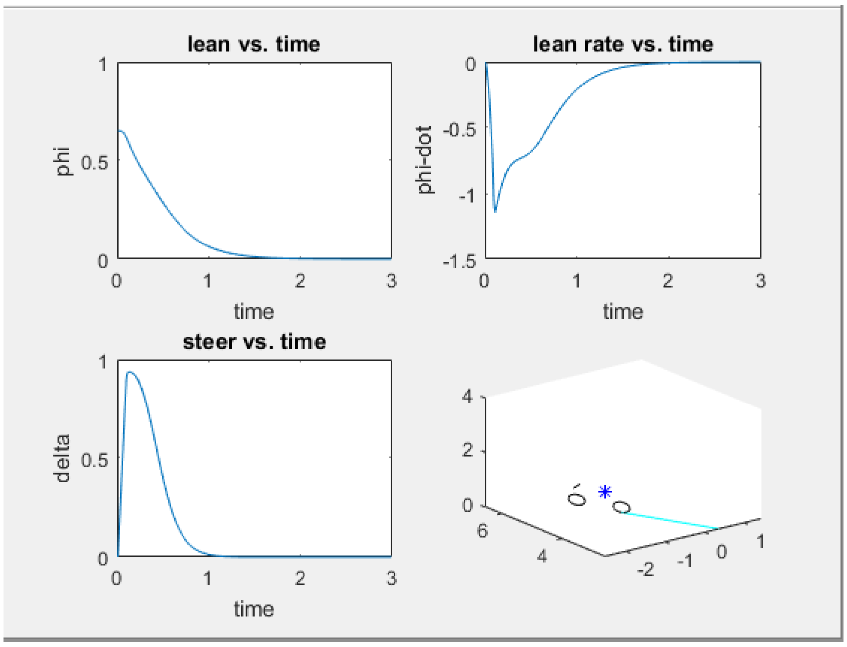


Figure 12.4: Bicycle simulation in presence of bias, short term recovery

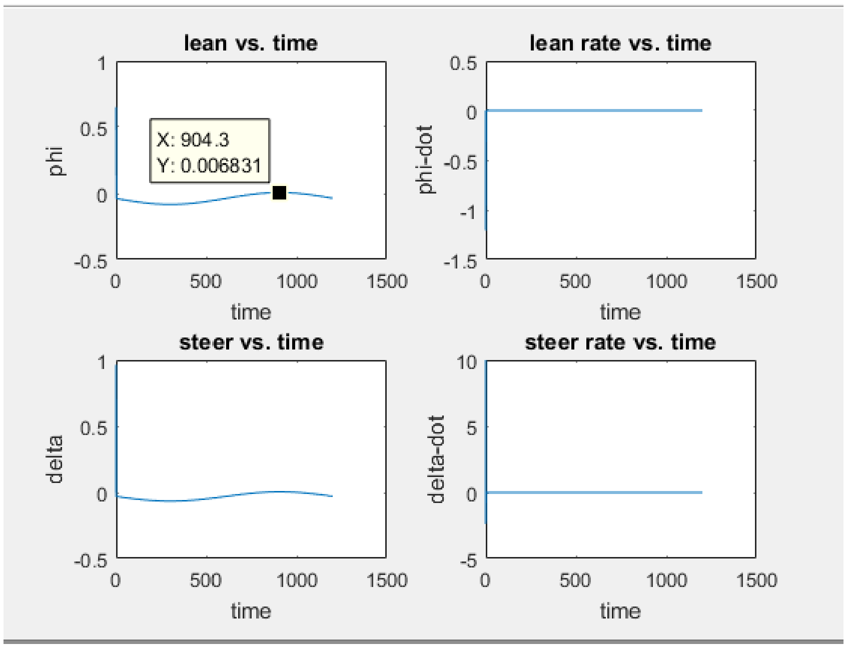


Figure 12.5: Bicycle simulation in presence of bias, 20 minute simulation

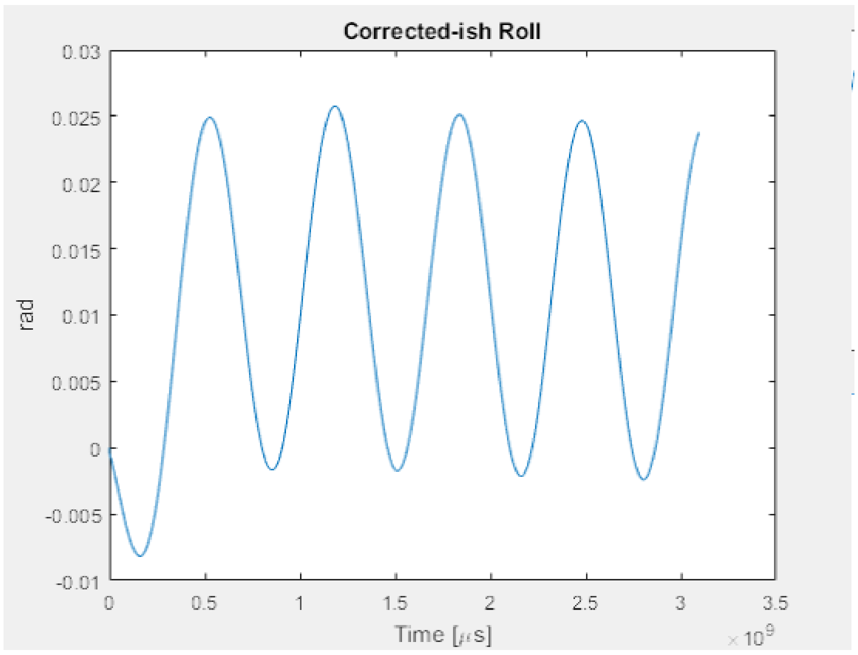


Figure 13.1: Matlab test of IMU data using sensor bias corrector

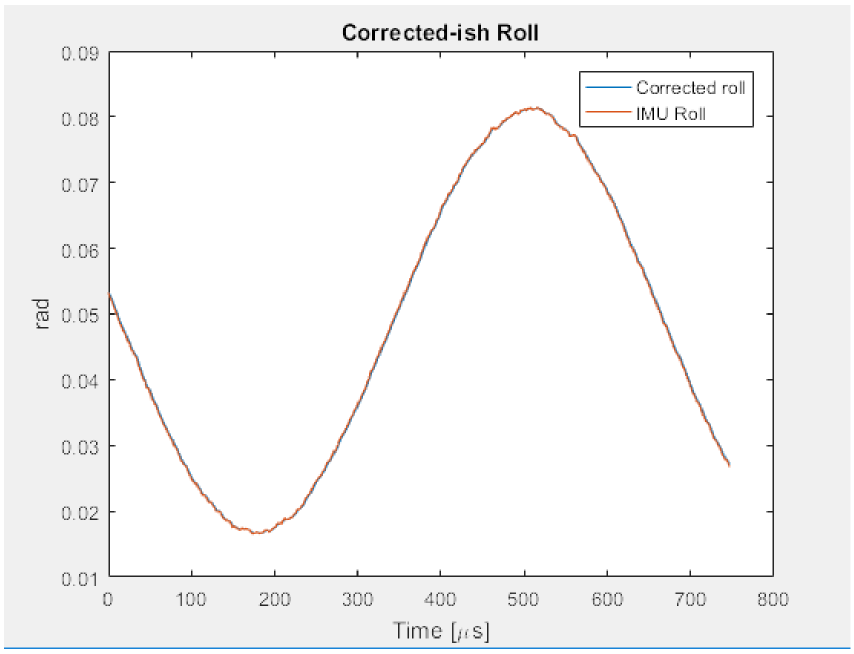


Figure 13.2: Roll data from sensor bias code and from IMU during Arduino test