Biorobotics and Locomotion Laboratory

Development of a Testable Autonomous Bicycle

Authors: Jason Hwang, Olav Imsdahl, Weier Mi,
Arundathi Sharma, Stephanie Xu, Kate Zhou
Supervisor: Professor Andy Ruina

Cornell University
Ithaca, NY

Final Semester Report

Spring 2016

Abstract

The Autonomous Bicycle Team is developing a robotic bicycle that should balance better than any other. Others have tried using a variety of balance strategies, including gyroscopes and reaction wheels; our bike will use only steering for balance, much like a human does. The mathematical model we use to develop our controller uses a point-mass model of the bicycle and rider. We also use simplified geometry with a vertical fork and no offset. The equations describing such a bike are more manageable than the non-linear equations of a full bicycle. The eventual goal is to use massive computer optimization to find an ideal steering strategy. In the meantime, we will see how well we can balance the bicycle using a steering angle rate (controlled by a motor) that is a linear function of the instantaneous steer angle, the bike lean angle, and the bike falling rate. Ultimately, we intend to demonstrate the bike riding around Cornell campus on its own. To that end, our achievements this semester were the following: implement the circuit layouts we designed in previous semesters using the printed circuit board (a design which is shared with the Autonomous Sailboat Team/CAST); write code for a new microcontroller, the Arduino Due (shared with CAST); implement a new optical encoder and inertial measurement unit (sensors that inform the steer angle rate); improve software controllers using new simulations for balance and navigation control. We hope to test a working bicycle this summer.

Contents

1

Electronic Subsystems (Jason Hwang)

1.1 Arduino Due

1.1.1 Overview

The Arduino Due is a microcontroller used to process and control all components of the bike. A microcontroller is essentially a small computer with processing power, memory, and input/output capabilities. The Due lies at the heart of the bike and enables the bike to execute instructions programmed by the user. The Due was chosen as the microcontroller for the bike due to its many input/output pins and its high clock speed, running at 84 MHz. This allows the Due to communicate with multiple devices and to calculate many instructions every second.

1.1.2 Hardware

The Arduino Due is powered by a voltage source between 7-12 V. Inputting any signal larger than 3.3 V will harm the microcontroller since the Due reads signals up to 3.3 V. The Due is also capable of outputting 3.3 V and 5 V. The Due utilizes the Atmel SAM3X8E ARM Cortex-M3 32-bit processor and contains 54 digital input/output pins. Having many pins (more than other Arduino microcontroller options) allows the Due to connect with more devices and be more flexible. Digital pins are pins that only take low (0) or high (1) values. All digital pins have interrupt capabilities, which allows the program to pause its current task when a user-specified event occurs to perform another function. Once the function is completed, the processor returns back to where it left off.

Additionally, twelve of the digital pins support Pulse Width Modulation (PWM). PWM is a method for producing an analog value ranging between 0 (off) and 3.3 V (fully on) by using a digital signal. This is done by changing the duty cycle of the digital output i.e., controlling the fraction of time in each cycle that the signal is high or low. A duty cycle is the ratio between the duration a signal is high to the overall duration of the period. The average of the outputted PWM voltage is equivalent to a constant voltage between 0 V and 3.3 V. This is relevant for our purposes, because we want to control the speed of a DC motor, and this can be done by supplying different voltages, or by applying PWM signals. It is also possible to control a servo motor using PWM signals, but that is not relevant to our project. Besides digital pins, the Due also contains 12 analog input pins and 2 analog output pins.

1.1.3 Main Code

The Arduino Due's main code is the central program for the entire bike; this code should be the same in structure and content as that implemented by the Autonomous Sailboat Team (also in Prof. Ruina's lab). Every component's software is incorporated into the main code and is where all instructions are laid out. The main code is structured as follows: define libraries used, define pins and variables, setup pins and interrupts, and the main loop. After the main loop are interrupt subroutines for the RC which are run only when interrupts occur. While setup is run only once, the main loop never ends once it begins. The only situations in which the main code exits the main loop are in case of emergencies; this is described in further detail later in this chapter (refer to Watchdog section).

Within the main loop, the code is structured as such: reset watchdog timer, calculate RC inputs, input sensor data, control algorithm to determine outputs from inputs, and output PWM values to motors. It is important to reset the watchdog every loop so that the timer will not reach zero and shut off the battery (more about this in the Watchdog section). The main loop then calculates the appropriate values corresponding to each RC input (i.e., converting from the duration of an RC channel to an angle, more about this in the RC section).

The inputted sensor data includes the Euler angles (angle position in rad) and gyro values (rate at which Euler angles are changing in rad/s) from the IMU, as well as the front motor angle and angular rate from the encoder. Using the roll angle, roll rate, front motor angle, and desired turn angle, the control algorithm then calculates a desired steer rate. The desired steer rate is converted into a PWM value and outputted to the front motor while another PWM value is outputted to the rear motor to set the velocity.

1.2 Inertial Measurement Unit

1.2.1 Overview

The Inertial Measurement Unit (IMU) is a sensor used to provide information regarding the bike's current angle (Euler angle) and angular rates (gyro rate). To balance the bike, it is necessary to know how much the bike's frame is off center (center is when the bike stands vertically, equivalently a lean angle of 0 rad) and how quickly it is moving off center (rad/s). The IMU used for the bike is the Yost Labs 3-Space Embedded IMU. Acceleration along the xyz axis is found with the IMU's accelerometer and rotation about the axis is found with the gyroscope. The gyroscope is able to measure yaw, pitch, and roll and follows airplane conventions.

The Euler angles and gyro rates are accurate to $\pm 1^\circ$ in dynamic conditions, which is adequate for the control algorithm to produce an accurate steer rate. The control algorithm to keep the bike balanced uses the lean angle (roll) and lean rate. The yaw angle is useful when a GPS is equipped, so the bike knows which direction it's turning in. The pitch is useful when the bike is climbing uphill or downhill, but is not relevant on flat surfaces.

1.2.2 Software

1.2.2.1 SPI

The IMU receives and sends data through Serial Peripheral Interface (SPI) communication. SPI is a synchronous serial communication protocol, meaning it transmits data based on a specified clock frequency (6 MHz in this case). For every bit the Due transmits through SPI, it also receives a bit back from the IMU. The data transmission rate is kept by the clock.

SPI is composed of four signal lines: SCLK, MISO, MOSI, and SS. SCLK stands for serial clock, MISO for 'Master In Slave Out', MOSI for 'Master Out Slave In', and SS for 'Slave Select'. The master is the Arduino Due and the IMU is the slave (although this can be changed based on the programmer). Essentially, MISO is the input line on the Due which receives IMU data and MOSI is the output line on the Due which sends commands to the IMU.

1.2.2.2 Commands & Statuses

Commands sent to IMU
0x01 - Tared Euler Angle (Tared - Uses the tared orientation as the reference frame/zero orientation)
0x26 - Corrected Gyro Rates (Corrected - Biased and scaled to real world units)
0xF6 - Signals the start of a packet
0xFF - Gets status and data from IMU
∗Hexadecimal values (sent to IMU) identify the desired command
∗Note that sending any value besides 0xF6 or 0xFF clears the internal buffer when the IMU isn't processing a command

Statuses received from IMU after sending 0xFF
0x0 - (Idle) IMU waiting for a command, any command besides 0xF6 has no effect
0x1 - (Ready) IMU has processed command and has data to send back
0x2 - (Busy) Currently processing a command
0x4 - (Accumulating) Currently accumulating data

1.2.2.3 Union

A union is a data type that is able to store different types of data in the same memory location. Unions are used for the IMU since there are two data types that are used, bytes and floating values. Using unions make it easier to store the IMU data (which is received in bytes) into an address and accessing floating value equivalents from the same address. The floating values are the actual angles and angular rates of the bike. For the IMU, two union arrays are created, one Euler union array and one gyro union array. Each array contains three unions, denoting roll, pitch, and yaw, for a total of six unions.

1.2.2.4 Endian

Endian is the order in which bytes are stored. Big endian is ordered so that the Most Significant Byte (MSB) is stored in the lowest memory address and other bytes follow in decreasing order. In little endian the Least Significant Byte (LSB) is stored in the lowest memory address and the following bytes are stored in increasing order. Since the IMU is in big endian and the Due is in little endian, it is necessary to swap the order of the bytes. Calling the function 'endianSwap', located in IMU.cpp on the Due, will swap the endianness of the bytes.

1.3 Radio Control

1.3.1 Overview

Using radio control allows the bike to receive external instructions and commands. Information such as how fast the bike should go, which direction the bike should steer into, whether the bike should be on or off, whether the landing gear should be deployed, and the mode of operation can all be transmitted through RC. The RC consists of two components, a receiver and a transmitter. The transmitter is a remote control that connects wirelessly to the receiver. The remote control consists of throttles, switches, and knobs. The receiver used is the Tactic 624 Receiver, and the remote control used is the TTX600.

1.3.2 Background

The RC works by having the transmitter send over a pulse to the receiver. The duration of when the pulse is high corresponds to the value the remote control was set to. For example, if a throttle was all the way down, it would correspond to a pulse duration of 1000 ms. When the throttle is pushed all the way up, it corresponds to a pulse duration of 2000 ms. For all values in between, a linear relation exists, so a throttle at the halfway point would produce a pulse of 1500 ms. The Arduino Due, which is connected to the receiver, calculates the duration of the pulse from the receiver. It would then translate that time duration to a value through a specified formula.

1.3.3 Polling

One method for determining the duration of the pulse is for the Due to constantly check the logic level of the signal from the receiver. This is called polling. The Due would check as fast as it can what the logic level was and wait for it to change. Once the pulse goes from low to high, the Due would start a timer. It would then check for the signal to go low again. Once it has gone low, the Due would stop the timer.

This method of obtaining the duration works, however, is extremely inefficient. The Due has a high speed processor but wastes all of its clock cycles waiting for the signal to change when it could have done thousands of other tasks in the meantime. Another method which uses interrupts allows the Due to perform other tasks but also keep track of when the pulse changes.

1.3.4 Interrupts

An interrupt is a method that pauses what the processor is currently doing to tend to another task immediately. Interrupts are called when an event occurs and can be triggered both through software or through hardware. A software interrupt is triggered when an internal event occurs, such as dividing by zero. The processor then specifies what should be done in such a scenario. In the case of the RC, the Arduino Due uses hardware interrupts which is triggered when the logic (equivalent to voltage) of an input pin changes.

For example, a pin can be set to be triggered with a logic HIGH signal. If the signal were LOW, the program would run normally. However, once the signal becomes HIGH, the interrupt is triggered. Once the hardware senses the change, it calls an Interrupt Service Routine (ISR), which is a pre-defined subroutine located elsewhere in the code.

Interrupt Service Routine

An Interrupt Service Routine is a set of instructions that are performed when an interrupt occurs. Once the ISR has been completed, the processor returns back to where the main program was left off, prior to entering the ISR. It is good practice to make the ISR short so it doesn't affect the efficiency of the main program, since the main program becomes paused. Note that each digital pin on the Due has interrupt capabilities and can be assigned to trigger an interrupt on a rising/falling edge or when there's a signal change.

For the RC, interrupts are attached to the digital pins of the six channels. Each channel has its own ISR that is called when there is a logic change on the channel's input signal. Within the ISR, the first task is to disable the Due's interrupts so that the ISR is not interrupted in the chance that another interrupt occurs. Afterwards, the program checks to see if the logic is high or low. If the signal is high, it means that a pulse has just arrived, and the Due sets the start time of that channel to the current time by using micros(). If the signal is low, it means that the pulse has just ended, and the processor sets the channel's end time as the current time.

After the processor sets the end time, it then finds the total duration of the pulse by subtracting the end time with the start time. Since the start and end time variables are global variables, they can be accessed from anywhere. Once the ISR is done, interrupts are enabled again, and the processor returns to where it left off and resumes running the main loop.

1.3.4.1 Channel 1

Steer Angle - Channel 1 is controlled by the right throttle's horizontal movement.

Pulse Duration
Left: 1198
Center: 1502
Right: 1803
∗The values correspond to the pulse duration of the signal from the receiver, in ms. The values are hard-set by the RC hardware. For example, left corresponds to the leftmost position of the throttle and produces a signal with a duration of 1198 ms.

1.3.4.2 Channel 2

Channel 2 is controlled by the right throttle's vertical movement and is currently not used as the right throttle is broken.

Pulse Duration
Down: 1070
Center: 1394
Up: 1713

1.3.4.3 Channel 3

Velocity - Channel 3 is controlled by the left throttle's vertical movement. Since the throttle can stay at the level it is set, it is used to control the velocity of the bike.

Pulse Duration
Down: 1041 (Anything below is a velocity of zero)
Up: 1992 (Max velocity)

Equation
Velocity $= \frac{35}{317}$***duration -** $\frac{15830}{317}$

1.3.4.4 Channel 4

Steer Angle - Channel 4 is controlled by the left throttle's horizontal movement. Since the throttle can stay at the center with no user intervention, it is the equivalent to the bike moving in a straight line . Moving the throttle right causes the bike to steer right, left causes it to steer left.

Pulse Duration
Left: 1818 ($-\pi/4$)
Center: 1501 (0, Straight Line)
Right: 1178 ($+\pi/4$)

Equation
Steer Angle $= \frac{-9}{64}$***duration** $+ \frac{6741}{32}$
$*$The steer angle is in degrees and is converted to radians prior to being used by the control algorithm

1.3.4.5 Channel 5

Kill Switch - Channel 5 is controlled by the upper switch on the left side of the remote control. The switch has to be away from the user for the bike to be on. Once the switch is flipped so that it is closer to the user, the bike's motors are killed by entering a while(1) loop and setting off the watchdog (refer to Watchdog section).

Pulse Duration
Towards User: 989 (Off)
Away from User: 2009 (On)
∗The bike is on if the pulse is ≥2000

1.3.4.6 Channel 6

Landing Gear - Channel 6 is controlled by the knob on the upper left of the remote control. When rotated to the left, the landing gear is deployed. When rotated to the right, the landing gear is not deployed. The boundary is at the middle of the knob. Ideally, a switch would have been used for the landing gear however no more switches were available. Therefore, even though the knob allows for thresholding, it was used for the landing gear switch.

Pulse Duration
Turned left: 1505-2010 (Deployed)
Turned right: 989-1504 (Not deployed)

1.4 Watchdog

1.4.1 Overview

A watchdog is a timer circuit used to set off safety procedures when the system fails. A watchdog has a built in timer that is constantly decrementing. Once the timer reaches zero, the output of the watchdog changes logic (by becoming high or low), indicating that the watchdog was not reset properly and has timed out. Watchdog's are useful in making sure all systems work properly. When one system fails, the watchdog will be unable to reset and it's output will change. Once the output logic changes, an event will occur in response to the system failure (such as killing the battery).

Therefore, it is necessary to reset the watchdog's timer within the timeout period and before the timer reaches zero. When the processor fails to reset the watchdog, it means that something within the system has failed.

1.4.2 Implementation

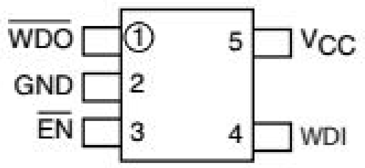
The watchdog used for the bike is the STWD100 (SOT23-5, WY) and has a timeout period of 2.4 milliseconds. The output of the watchdog is attached to a relay switch that will turn off the bike's battery once the logic changes. This way, the bike will shut off when one of the systems fails or a pre-defined condition occurs, such as if the bike is leaning too much. When such a conditions occurs, the program enters an infinite loop by calling while(1). Entering an infinite loop essentially causes the processor to endlessly spin in place, without progressing any further in the program. This is because an empty while loop was used where the condition is 1 (ie. always true).

As a result, since the processor is stuck in the while loop with no ability of escaping, the watchdog timeout period of 2.4 milliseconds will eventually elapse. As the program is unable to reset the watchdog since it was stuck in the loop, the watchdog times out and changes output logic. This causes the relay switch to set off and turn off the battery.

A watchdog circuit is implemented so that if the Arduino Due itself fails, the bike will proceed to shut off. This is done by connecting one of the Arduino's digital output pins to the watchdog's input. If the Arduino fails, it will not be able to properly output to the watchdog and successfully reset it. The instruction to reset the watchdog is placed at the beginning of the main loop.

One loop takes around 2 milliseconds to complete and the timeout period of the watchdog is 2.4 milliseconds. Since it takes 2 milliseconds to reach back to the instruction to reset the watchdog, and the timeout period is 2.4 milliseconds, there is still .4 milliseconds of slack leftover. Therefore, if all systems are working properly, the watchdog will be reset successfully in each loop, before the timeout occurs.

1.4.3 Pinout



Pinout.JPG

Figure 1.1: Watchdog Pinout

1.4.3.1 EN

Having the Enable (EN) pin set to a constant logic low will cause the watchdog timer to always be on and decrementing the timer. This is the recommended input for EN. It is also possible to reset the watchdog by toggling the EN pin by setting it to high for 1 microsecond, then back to low again.

1.4.3.2 WDI

Another way to reset the watchdog's timer is by using the Watchdog Input pin (WDI). It is possible to reset the watchdog by toggling the WDI pin to high for 1 microsecond then back to low again, within the timeout period.

1.4.3.3 WDO

The Watchdog Output (WDO) is connected to the watchdog output pin on the PCB. If the watchdog successfully resets within the timeout period, the output on the PCB pin would be high. If the watchdog is not reset within the timeout period, the output will be low.

It is necessary to use a 10 kΩ pull up resistor connected to a 5V power supply to read the WDO, otherwise the WDO would always read zero due to the active low open drain. Use the following schematic to connect the watchdog output.

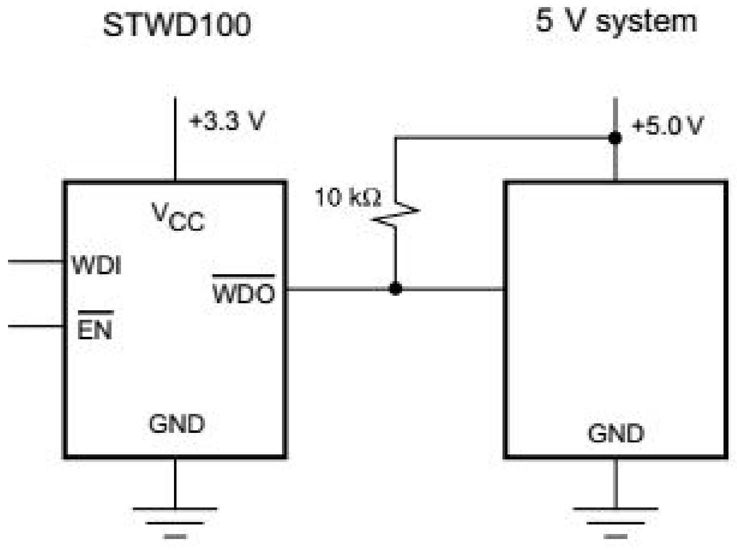


Figure 1.2: Watchdog Output Schematic

2

Rotary Encoder & Front Motor (Weier Mi & Stephanie Xu)

2.1 Overview

Being able to control the bike's wheel position and movement is an essential part of a working autonomous bicycle. Rotary encoders are electromechanical devices that convert the position and motion of a shaft or axle into analog or digital signals. In this case, the encoder is fixed around the bicycle fork, which is directly attached to the front wheel, so even small changes in position of the wheel is detected by the encoder. The Arduino

Due micro-controller sends a pulse-width modulated (PWM) voltage signal to the front motor. The duty cycle of this signal translates into an average constant DC voltage applied, which is (ideally) proportional to an angular velocity in the motor. The encoder reads the position and rate of rotation of the bicycle fork. The readings are sent back to the micro-controller and used to make calculations to determine the rate at which the motor should turn in order for the bicycle to stabilize. The process is then repeated using new data from the encoder.
This chapter goes deeper into how the encoder works, its properties and outputs, the calculations to find angle and angular velocity, encoder hardware and software, testing procedures, error and debugging, as well as possible future developments.

2.2 Mechanisms and Importance

There are two types of rotary encoders: incremental and absolute. Absolute encoders maintain their position data even when the device is powered off. Incremental encoders, on the other hand, read position based on movement from where it starts when it is first powered on.
Currently we have two encoders installed on the bike: HEDR-55L2-BY09 and the Encoder Products Company Model 260. Both are incremental rotary encoders, chosen carefully as the incremental encoder allows a variable initial position of zero where the absolute encoder does not.
The HEDR encoder (motor encoder) installed on the top of the motor is used to measure the motor's rotation angle and angular speed. Meanwhile, the 260 encoder (which is directly coupled to the wheel) is supposed to measure the front wheel's rotation angle and angular speed.

The reason behind having two encoders is that there is a bit of slack between the front motor and its gearbox, which means that the gearbox's rotation will have a small time offset from the motor's rotation. Therefore, it would be necessary to measure the shaft rotations at both the motor and the gearbox to keep track of that difference.

Despite having two encoders, we have decided to focus on only the wheel encoder this semester in order to reach our goal in building a functional bike, and we will incorporate (and add software to read) the motor encoder in the future to further improve upon the design we already have.

2.3 Encoder Properties

Encoder properties are described in more detail in the Fall 2015 Final report, two encoder properties were especially important for us (Spring 2016): the encoder resolution and supply voltage. They will be briefly discussed below; refer to the Fall 2015 report for further information.
Encoder resolution, in units of Counts per Revolution (CPR), are the number of notches in the encoder disk that allows the encoder to track the position. An encoder with a greater CPR gives readings that are more accurate as there are more notches for the encoder to read and can therefore detect smaller angular changes. The encoder resolution for the wheel encoder (E.P.C. Model 260) is 4096 CPR, whereas the motor encoder (HEDR-55L2-BY09) has a resolution of 3600 CPR.
Supply voltage, in units of Volts (V), is the voltage that the encoder needs to operate. It has the same magnitude as its output signals for these encoders. The supply voltage for both the wheel encoder (E.P.C. Model 260) and the motor encoder (HEDR-55L2-BY09) is 5V.

2.4 Encoder Output

The encoder outputs three different signal pairs: A and A', B and B', as well as Z and Z'. In conjunction with one another, the A/A' outputs and B/B' outputs are used to determine the direction that the encoder is turning as well as the position it is currently in. Meanwhile, the Z/Z' outputs give us the index, which is a fixed point on the encoder that can be used as a consistent reference point.

2.4.1 Z and Z'

While the initial "zero" of the encoder is determined by the position of the wheel when the encoder is powered on, the index is a fixed point on the encoder disk. Currently, the encoder is positioned in such a way that the index is facing directly in front of the bike. The position readings reset back to 0 when the index signal is HIGH. We can use this fact to see how many counts the encoder may skip. The wheel must sweep past the index before the encoder registers any index signals. The number of times the wheel moves past the index is stored in register REG_TC0_CV1 on the microcontroller.

2.4.2 A, A', B, and B'

Most incremental encoders will use two channels to detect the position. In our case, we have channel A and channel B that count the number of notches that pass by, both outputting a square wave signal spaced 90 degrees out of phase. The two output channels of the quadrature encoder (quadrature referring to the square wave outputs being a quarter cycle out of phase) indicate the position and direction of movement. For example, if A leads B, the disk is rotating in a clockwise direction. On the other hand, if B leads A, the encoder rotates in a counter-clockwise direction. The number of notches the wheel has moved through is stored in register REG_TC0_CV0.

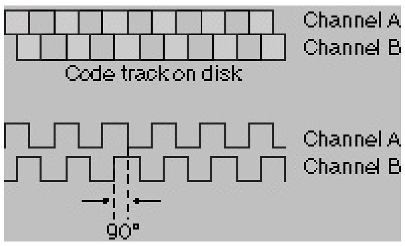


Figure 2.1: Quadrature Encoder Signals A and B

2.4.3 Counting Edges

Both of our encoders are able to send data on both the rising edge and falling edge of the clock cycle, doubling the counts per revolution to 2*4096 CPR = 8192 CPR for the wheel encoder and 2*3600 CPR = 7200 CPR for the motor encoder. However, since there are two channels (A and B), the counts per revolution is doubled again as the encoder sends back data on the rising and falling edges of both channels. This leaves us with 2*8192 CPR = **16384 CPR** for the wheel encoder and 2*7200 CPR = **14400 CPR** for the motor encoder.

2.5 Encoder Hardware

2.5.1 Line Receiver

2.5.1.1 Overview

A line receiver is a piece of hardware that takes a pair (or multiple pairs) of differential signals (like the encoder outputs) as inputs, and outputs a processed signal(s) by subtracting one from the other. The line receiver only outputs either low or high signals. In a pair of differential inputs, if one (plus-signed signal) leads the other (minus-signed) by a significant voltage (decided by line receiver itself), the output is 1. Otherwise it's a zero. It filters out potential noises in this way, assuming both signals in a pair of differential signals are affected by the same amount of noise, subtracting them would cancel out the noise, which is common for both signals. We could make the assumption because the two signals are transmitted by wires placed in the same cable.

Line receivers are used to utilize all of the encoder outputs. In addition to filtering signal noise, the receivers also isolate the Arduino from the encoder. The encoder generates 5V outputs while the Arduino Due could only take input signals with voltage level up to 3.3V. So, those 5V signals have to be transformed to a voltage level below 3.3V, and the connections should not be direct wires, or the Arduino board could be damaged permanently. This can be done by simply powering line receivers with 3.3V (which is within the specifications). Given the fact that the encoder has three pairs of outputs, it needs three channels on the line receivers, one for each of the outputs.

2.5.1.2 Models

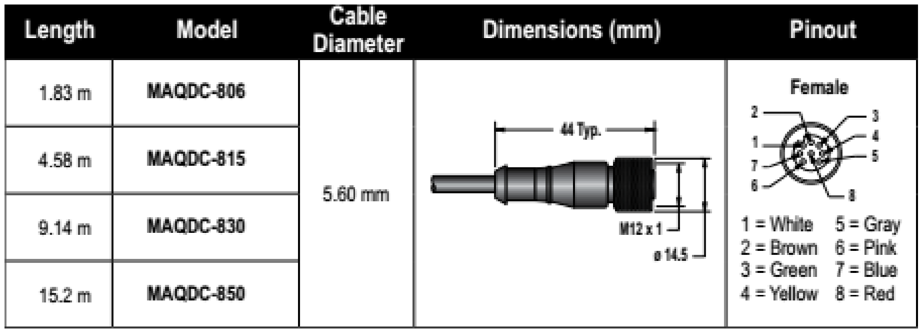
For the purpose of testing, we used the LTC1480 line receivers. The LTC1480 is a single-channel line receiver, so we need three of those to generate all outputs from one encoder. We connect pin 8 (VCC) to Arduino's 3.3V power supply output, and pin 6 (GND) to one of the Arduino ground pins. Pin 6 (A) and 7 (B) are for complementary differential outputs of the encoder, such as A+ and A-, respectively. Pin 1 (RO) is the output signal which has been processed by the line receiver. This should be connected to the Arduino to provide the encoder data input. The Arduino should always keep pin 2 (REnot, receiver enable not) and 3 (DE, driver enable) to be at 0V (LOW), or the line receiver could be operating in the driver mode and no valid output signal would be provided from pin 1. Pin 4 (DI) does not need to be connected to anything, as it is the driver input which we never utilize.

For actual use on the bicycle, however, we use a different line receiver model, ST26C32AB, which is soldered to the printed circuit board designed by Arjan Singh (Sailboat Team). Each chip can process four channels, so we only need one such line receiver to process the differential signals from one encoder. Thus, we have installed two line receivers to be installed on the PCB for both encoders.

The two line receivers have code names LR1 and LR2, and they are connected to designated Arduino pins as defined by PCB. R1 connects to pin 4, 5, and 10 (A, B, Z, respectively), and R2 connects to pin 2, 13, and A6. R1 is set to be used with the motor encoder, and R2 is set to be used with the wheel encoder.

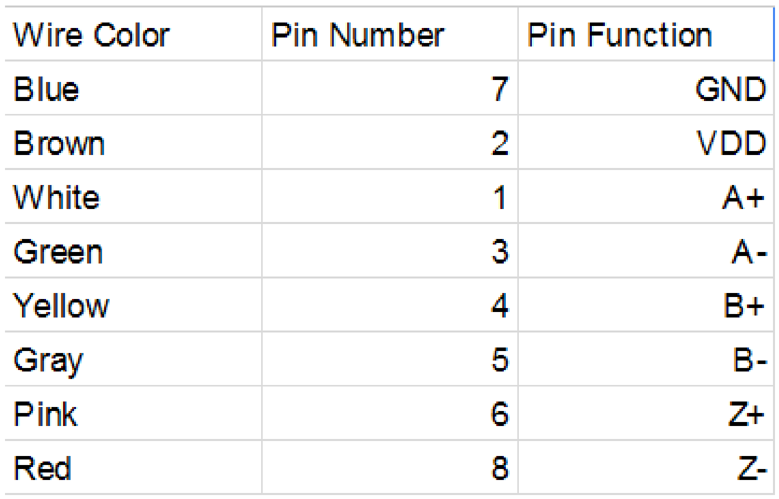
2.5.2 Encoder Cable

Previously, the team did not have the correct cable to connect to the wheel encoder and therefore could not use it. This semester, a new cord (8-pin M12/Euro-style cordset) was ordered with the correct number of pins. The cord characteristics and comparison between colors and functions of encoder wires are shown in Figure 2.2 and 2.3 below:



Length	Model	Cable Diameter	Dimensions (mm)	Pinout
1.83 m	MAQDC-806	5.60 mm	44 Typ.; M12 x 1; ø 14.5	Female; 1 = White, 2 = Brown, 3 = Green, 4 = Yellow, 5 = Gray, 6 = Pink, 7 = Blue, 8 = Red
4.58 m	MAQDC-815			
9.14 m	MAQDC-830			
15.2 m	MAQDC-850			

Figure 2.2: 8 pin M12/Euro style cordset



Wire Color	Pin Number	Pin Function
Blue	7	GND
Brown	2	VDD
White	1	A+
Green	3	A-
Yellow	4	B+
Gray	5	B-
Pink	6	Z+
Red	8	Z-

Figure 2.3: Wire colors and corresponding pin functions

2.6 Encoder Testing

2.6.1 Overview

The encoder code begins by setting up all the pins on the Arduino, namely for channel A (pin 2) and channel B (pin 13) which, according to the SAM3X (our microprocessor) data sheet, are pins that allow the code to use the Arduino Hardware Quadrature Decoder to count the number of rising and falling edges from the two signals from channels A and B. The setup stage of the code also makes sure that the correct clocks and timers are activated so the Arduino can receive encoder position data and save it into a register located on the Arduino (REG_TC0_CV0) and also receive index data and save it to a different Arduino register (REG_TC0_CV1). The looping stage of the code takes the value in the position register (REG_TC0_CV0) and uses it to find the angular speed in rad/s and angular position in radians, printing the two values out. It also uses updates to the index register (REG_TC0_CV1) in order to reset the position count to 0. The code is used for both testing the encoder and also for the main code used for testing the bicycle.

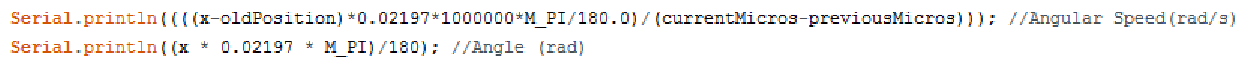
2.6.2 Calculations

2.6.2.1 Angular Velocity

As the angular velocity calculation is detailed in the Fall 2015 final report, the following is a brief explanation on calculating angular velocity. In order to calculate the angular velocity, the counts from the encoder needed to be converted into angle units (degrees or radians). For the HEDR-55L2-BY09 encoder with a CPR of 14,400, each count would be 0.025° (360°/14,400 = 0.025°). Likewise for the Model 260 encoder with a CPR of 16,394, each

count would be 0.02197° (360°/16,394 = 0.02197°).
To get the angular velocity, the old number of counts from the encoder is subtracted by the new count from the encoder. The result is multiplied by 0.02197° to get the the value in degrees, divided by the difference in time in microseconds, then multiplied by 10^6 to get the time back in terms of seconds. The result is the angular velocity in degrees/s.

The code is shown below:



```
Serial.println((((x-oldPosition)*0.02197*1000000*M_PI/180.0)/(currentMicros-previousMicros))); //Angular Speed(rad/s)
Serial.println((x * 0.02197 * M_PI)/180); //Angle (rad)
```

2.6.3 Testing

2.6.3.1 Hardware-Based Implementation vs Software-Based Implementation

One method to keep track of encoder counts would be to use software-based code, which relies on interrupts to get the counts from the encoder signals. Each time there is a rising or falling edge from either Channel A or Channel B, the code is interrupted by the signal and a count is taken in.

The code we're using instead, however, relies heavily on the Hardware Quadrature Decoder that is already built into the Arduino. The decoder takes the two signals from channels A and B and counts the number of rising and falling edges on the clock cycle. The data from the encoder is automatically sent to the register file on the Arduino (REG_TC0_CV0), and data can be pulled from the register file rather than making the code request it from the encoder each time it needs the value. The advantage would be that the hardware code would miss fewer position values since the encoder sends the data to the register file without the code calling for it to do so. Also, since the software requires the use of interrupts to go and get data from the encoder, it would be slower performance-wise compared to the hardware code.

This is further improved with an index signal that resets the position back to 0 every time the signal is at high voltage (when the encoder passes the index position). As the encoder was positioned to have the index notch facing forward on the bike, the index signal would help the Arduino keep track of any lost counts and reorient the position count by resetting back to zero, indicating that the wheel is facing forward. The encoder position count reset is automatic, and done by the Arduino hardware, so no extra code was written besides the initial configuration.

2.6.3.2 Procedure

The encoder was initially tested manually by starting the wheel in a certain position, generally in-line with the rest of the bicycle, and making full revolutions to see if the encoder reads approximately the number we were expecting (16384 CPR). The test was repeated both with the hardware code and the software code, and the readings were never off by more than 20-30 counts. Since the wheel needs to have no obstructions in order to turn completely, the only way to set where the wheel started and ended was to approximate it with a laser pointing at a fixed position. This, however, is not a very effective method to find the error since the laser cannot pinpoint an exact location as the laser light has spread. So, we also performed a test to specifically measure the error of the encoder readings.
To find the encoder error, a physical stop (a metal block held to the bike with a clamp) was attached to the bike, and we started the procedure by resting the wheel on the block, and then we moved the wheel quickly back and forth

to see if the encoder skips counts when the wheel moves too quickly. The wheel was then brought back to its starting position, and we took the readings using the Arduino. Each time, the reading was between -23 to 23 counts away from zero, which is approximately 0.505° (0.02197° * 23 = 0.505°) in either direction.

2.7 Motor Testing with Encoder

2.7.1 Overview

In order to control the front motor with the encoder, we need to find the relationship between the PWM signal that will be sent to the front motor controller and the angular speed that the motor responds with, which is measured by the wheel encoder. We provided the motor with a PWM signal and recorded the corresponding angular speed. After sweeping the PWM from 0 to 255 (minimum speed to maximum speed; the PWM duty cycle is defined in terms of bits out of 10^8 in Arduino), we should be able to generate a function that summarizes how fast the motor should spin given any valid PWM value. That way, when the controller code calculates a desired steer rate, the Arduino can output an appropriate PWM signal.

2.7.2 Approach 1

The first approach has the motor spinning for a set period of time with some PWM value. The average angular speed could then be obtained by dividing the angle it spins through by the amount of time. Then we loop through many PWM values and get a relationship between angular speed function

and PWM. We took measurements with the front wheel on the ground and off-ground. The resulting equation that we obtained is angular velocity = 0.1742*PWM*255-1.8482
However, this approach has setbacks. With the PWM injected increasing, it could result in a high angular speed that enables the motor to turn more than 90 degrees in a burst of time. The time interval we tested with was 200 ms. From our observation, with the same amount of power provided to the front motor, it should rotate in different speeds as its angle changes. For example, the front wheel should be easier to turn from 0 to 45 degrees than from 45 to 90 degrees. Considering that the angular speed also depend on the angle, we must make sure that during each test the front wheel rotates by the same amount of angle.

We also assumed that the motor's turning speed should be the same regardless to the turning direction and only tested one direction, while as pointed out by Professor Ruina and in practice we observed that the motor turns faster when it rotates clockwise than it does counter-clockwise. We currently don't have an explanation for this as we did not spend enough time researching on this phenomenon, but we concluded that possible reasons include friction, motor installment issues, and motor error. Since we have overlooked this problem, we decided to test both directions in the next approach, which is expected to yield more accurate results than those from the previous one.

2.7.3 Approach 2

For our second approach, we changed the measurement method: instead of setting up a certain amount of time for the front wheel to rotate with a certain PWM value and recording the changed angle, we make the front

wheel rotate to a certain angle with a certain PWM and measure the time spent. We set the target angle for the front wheel to be 45 degrees (counter-clockwise) and -45 degrees (clockwise). It provides more precise results than those from approach 1, because this approach bypasses the problem that the front wheel turns slower as it approaches 90 degrees. Since we only rotate it to 45 degrees now, the angular speed would not drop dramatically and therefore be more consistent.
We measured the wheel's angular speed in both cases when it was put on the ground and off the ground. We took three measurements for each case, and after averaging them we obtained this graph (Figure 2.4) and the following functions: y = -46.616x-193.43 (grey), y = -16.548 (blue), y = 17.275x (orange), and y = -43.902x-168.02 (yellow), where y is the PWM value and x is the angular speed.

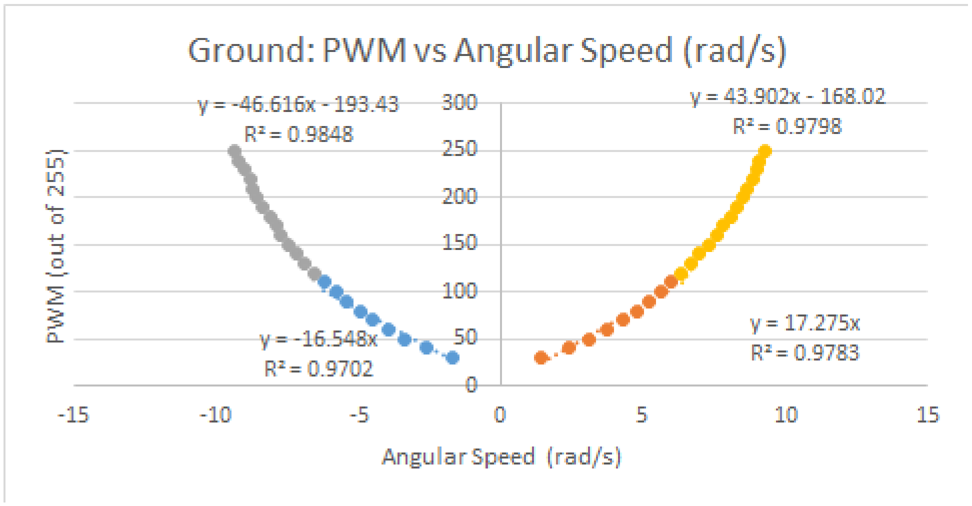


Figure 2.4: Angular Speed vs. PWM Value

The blue and grey graph is from spinning clockwise, and the orange and yellow graph is from spinning counter clockwise. Each side of the graph was split into two different sections to make four total sections in order to get a linear regression line that works better than if we had a single line for each side of the graph.

We also came up with a function that restores the wheel position to the starting point when the wheel reaches ±45 degrees. This made our testing progress a lot more efficient because we no longer had to manually correct the front wheel to the starting point after each test.

2.7.4 Approach 3 and Latest Equation

However, Olav and Arundathi pointed out that this approach includes a potential error: there is an overhead of time being counted created by the acceleration of the motor from zero speed to the speed indicated by given PWM. To exclude this overhead, a possible approach is to rotate the front wheel by multiple revolutions, such as 20 revolutions, and count the time taken. This approach minimizes the delay time caused by the overhead, because the overhead time could be a very small proportion of the time taken for 20 revolutions.
Nevertheless, we concluded that the approach that counts many revolutions is not realistic enough and cannot reflect the real application cases. When we run the entire bike, we probably would not want to rotate it to angles beyond ±45 degrees, as agreed by everyone in the team. The approach that counts many cycles would yield an ideal angular speed, but it only tells us how would the motor perform in ideal conditions (wheel off the ground). We have taken measurements using the more ideal method, and we obtained the plot (shown in Figure 2.5). We measured the time taken for 20 revolutions, and found the relationship between the angular speed and PWM to be: y = 5.37048x+5.24481, where y is the PWM and x is the angular speed. This

function is the average of clockwise and counter-clockwise rotations' functions.

Meanwhile, we took measurements using the original approach 2 again (ro-

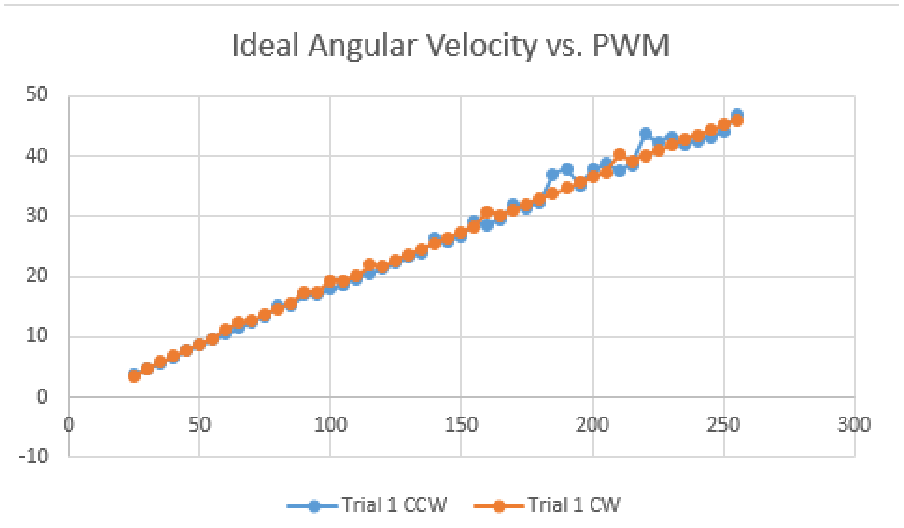


Figure 2.5: Ideal Angular Speed vs. PWM Value

tate to ±45 degrees each time), since Olav has managed to lessen the friction that caused the motor to rotate more slowly. We think this should make the motor turn faster, and we have obtained data supporting this assumption. The experiment was done with front wheel on the ground, so that we could simulate the realistic application better. The new function is y = 33.92591x-80.90888, where y is the PWM and x is the angular speed. We can observe that the same PWM now enables greater speed.

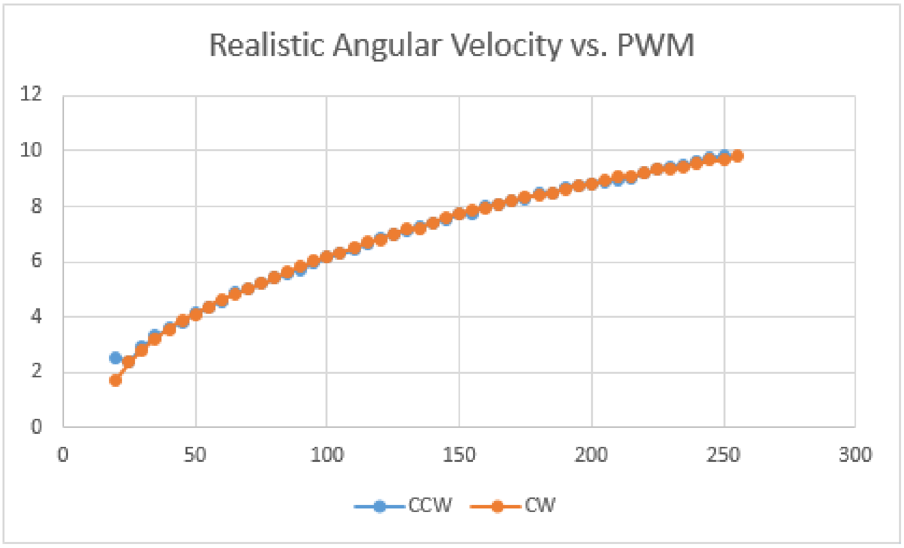
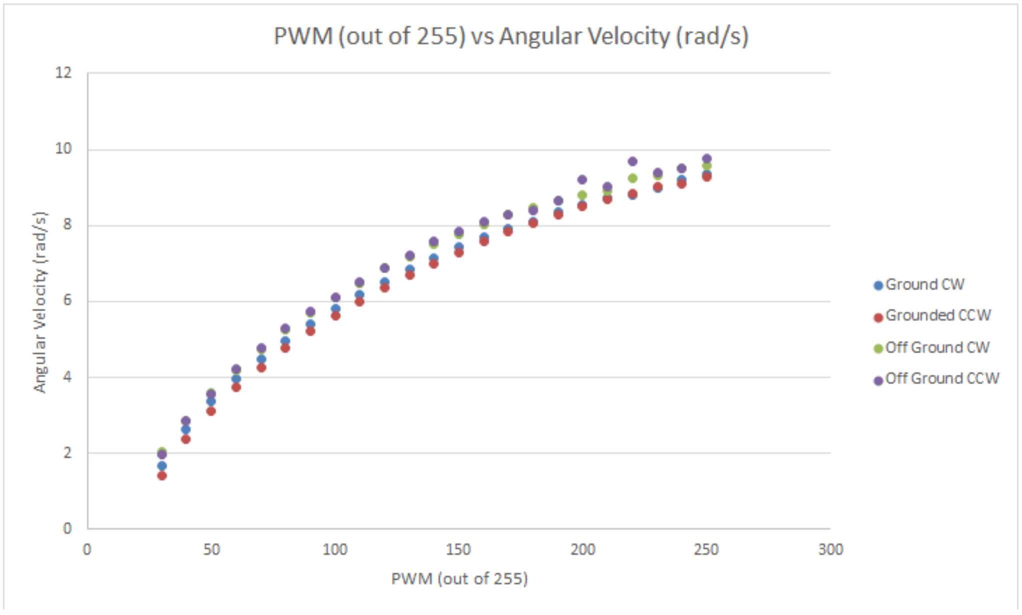


Figure 2.6: Realistic Angular Speed vs. PWM Value

2.7.5 Data Analysis

Using Approach 2, the following graph was created: The angular velocities measured from each PWM was plotted in the graph above. As expected, the off-ground data points were higher in angular velocity since the wheel isn't as affected by friction. The gap from 0 PWM to 30 PWM is due to the fact that the front motor does not turn when PWM is any value lower than 15 and we could not get any result with those PWM values. As the battery drains, this threshold would increase. We think generally it would be safe to choose the lowest PWM to be 25, so that the motor would always response and turn. The reason behind should be the friction between the motor and the axis, as well as the friction and voltage drop within the motor itself.



2.7.6 Other Factors

2.7.6.1 Voltage

During testing, as pointed out by Professor Ruina and Arundathi, the voltage level of power supply could also affect the relationship between the PWM and angular speed. We found that if the battery pack that was used as the power supply provided voltage below 26V, the front motor's rotation could be significantly slowed.

2.8 Solved Errors and Debugging

2.8.1 Pull-up Resistors

While we were testing the encoder with the LTC1480 line receiver which works with the circuit board made by Olav, we encountered a problem that the line receiver kept outputting a HIGH signal. After some debugging and research on the datasheet, we found that the reason was because the line receiver has a built-in mechanism that forces channel output to be HIGH when it detects floating inputs. To solve this issue, after consulting our mentor Jason, we installed a pull-up resistor to each of the line receiver channel input. These resistors connect the channel input pins and VDD, and they are thus called pull-up resistors. Pull-up resistors could help stabilizing floating inputs because they are always connected to VDD, any uncertain inputs could be interpreted as a fixed HIGH value.

2.8.2 DIR Pin

While we were attempting to run front motor tests described above, we found that we could not rotate the front wheel counter-clockwise. After some debugging, it turned out that the DIR signal, which controls the direction of the motor spin, had a problem. It was going from the Arduino Due to the Pololu motor controller. As the definition of how DIR works on the Pololu datasheet suggests, with the current installation of the front motor and connection between it and Pololu, if we write HIGH to the DIR, the motor should spin clockwise; if we write LOW, it should spin counter-clockwise. If we swap the two connections that the front motor takes as inputs, the direction of rotation would be reversed to the pattern described above. However,

we observed that the motor was spinning clockwise constantly, no matter what we wrote to the DIR pin. This means the DIR pin was always set to HIGH by an unknown source.
This observation stalled our progress for more than one week. After lots of debugging, we discovered the error was that we had been using wrong Arduino pins all the time, instead of our assumption that DIR being shorted to some other pin. We thought all pins on Arduino Due were digital pins which could be assigned as input pins and output pins, but it turned out that our assumption was wrong. The pin we used at first was pin 7, and we still are not certain of why it would not work till today. The pin was then switched to pin 6, and that somehow resolved the problem. To fully figure this out, we need to manage to recreate this error, but as time runs out we did not get to test it thoroughly.

2.8.3 Line Receiver Enable Pin

Similarly to the DIR pin's problem, the line receivers (LTC1480) that we used previously with the circuit board had a problem with one of the enable pins (REnot). At the beginning we assigned it to connect to pin 4 on the Arduino, and this pin functioned normally as it was always set to LOW. However, after the pin was switched to pin 22 in order to avoid conflict with an IMU pin, the REnot pin on the line receivers always received a 5V voltage output. To our surprise, the 5V voltage output pin on the Arduino was not even used anywhere.
We initially assumed that the DIR pin was shorted to another pin somewhere on the circuit board, and the 5V came from elsewhere. However, after lots of testing, such as checking if the resistance between DIR and any other pin is low enough to be considered "shorted" (a few tens of ohms), we concluded

that the circuit board did not cause the problem. Then we thought it could be an internal short in the Arduino Due, but the resistance between the 5V output pin and pin 22 was not low at all (more than 20Kohms), either. Eventually, we found that the source of the problem was rather simple: pin 22 was constantly outputting 5V, no matter what we attempted to write to that pin. Then we decided to switch the pin again, and the new pin, pin 6, was able to respond to what we write to it correctly. Consequently, the error was solved.

3 Hardware and Wiring (Olav Imsdahl)

3.1 Overview

This section will give insight into the wiring and hardware of the bike. It will give an overview on how things work and operate on this bicycle.

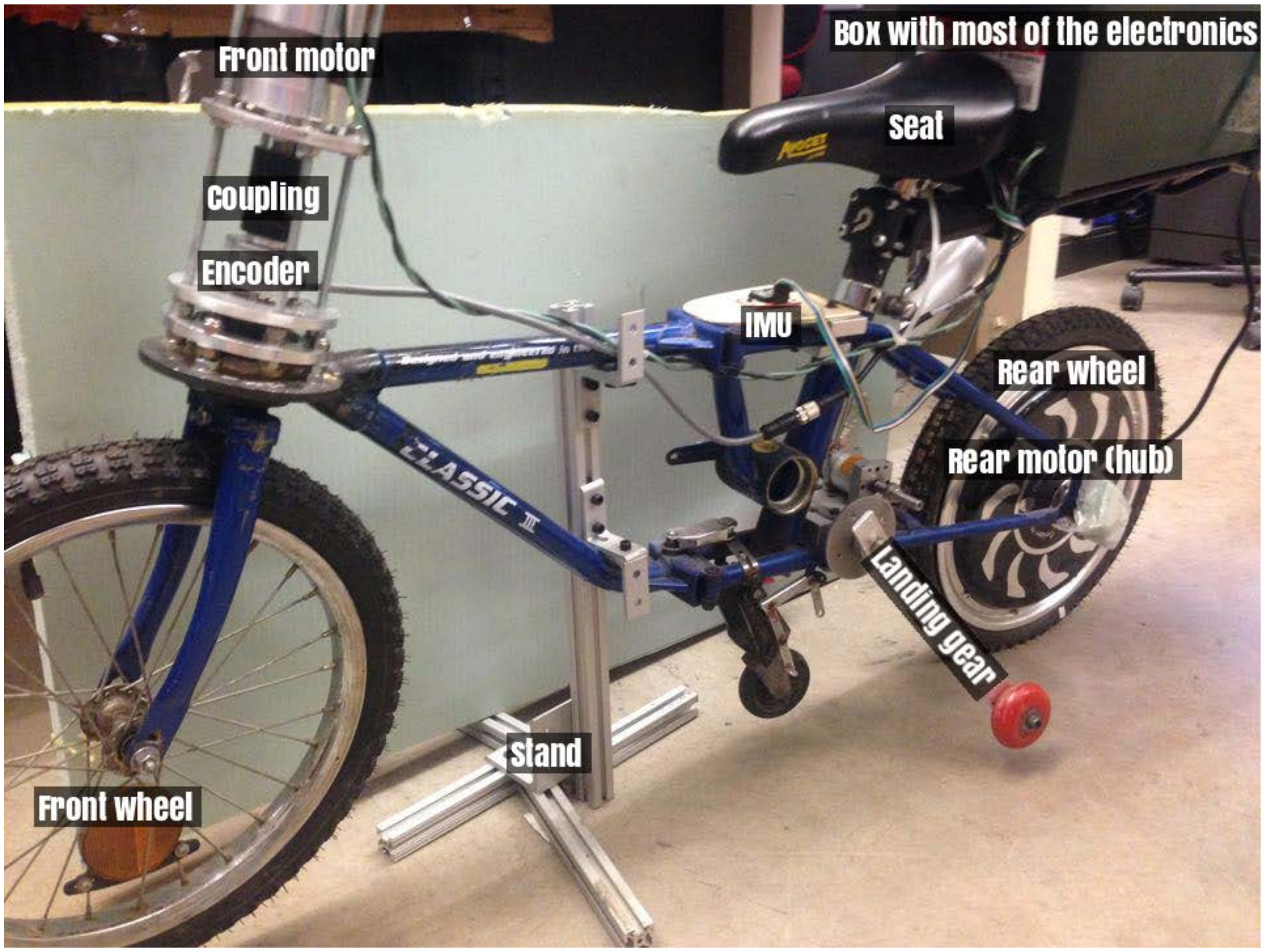


Figure 3.1: Complete bike diagram

3.2 PCB

The PCB, the main big circuit board, should be handled with care while being grounded, keeping loose wires from touching it. The electronic components of the bike all plug into the PCB with the designated plugs. The female plugs match to the corresponding male-endings on the PCB. The board indicates which plugs fit where and how to orient them. The power supply wires screw into the terminals at the edge.

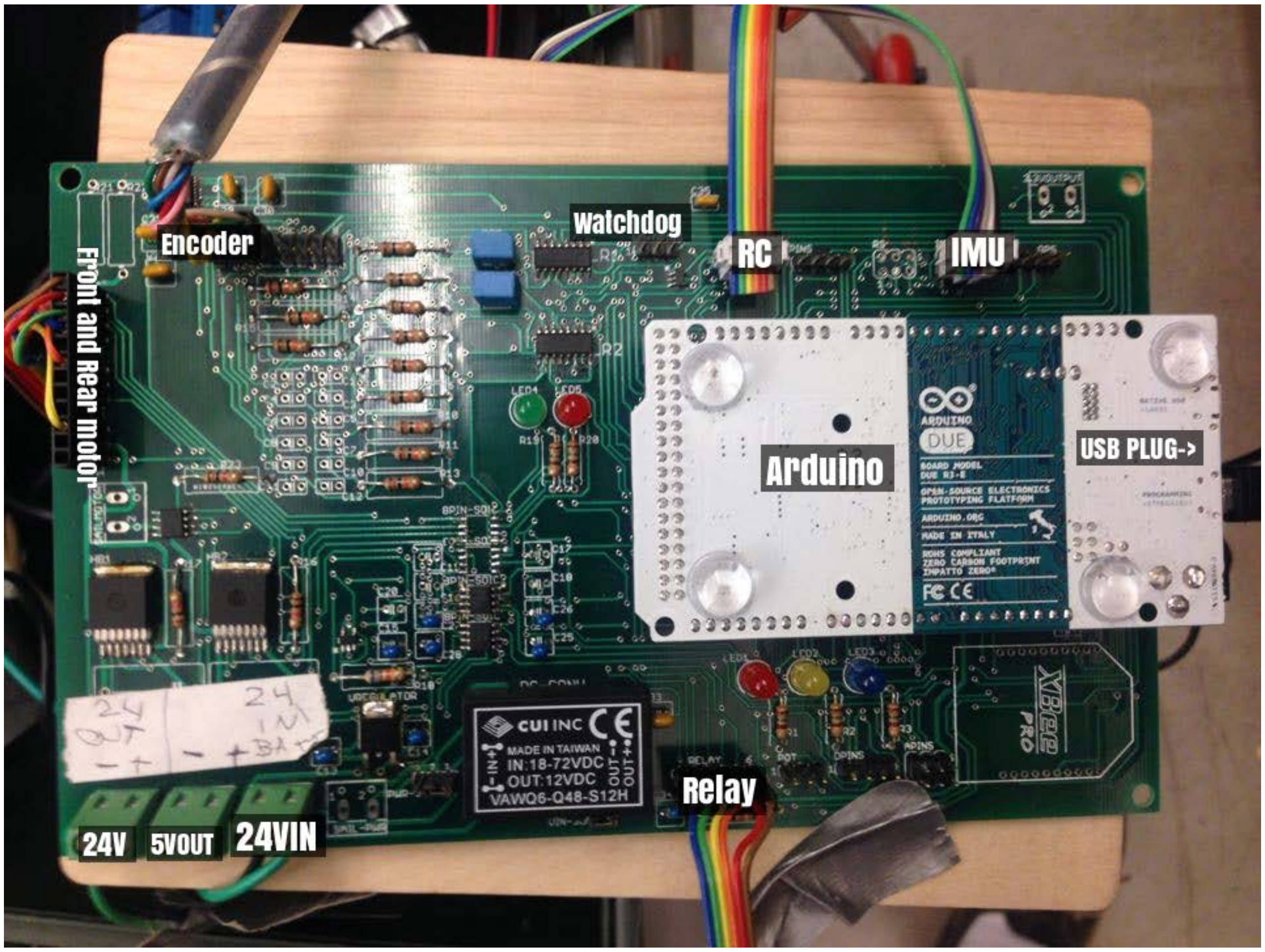


Figure 3.2: PCB with annotaded connections

3.3 Front motor setup and encoder

The front motor attachment can be tricky to access certain parts for adjusting. It is important that the front wheel axis is aligned with the motor axis, both concentric and parallel. The flexible coupling connecting them can account for some offset but the steering and encoder will get better results if they are positioned correctly. To align the motor with the front wheel the hex nuts (7/16, 1/4-20) can be loosened to adjust the motor. If the encoder index needs to be adjusted, a small set screw in a silver ring just above the encoder can be loosened to rotate the encoder without moving the wheel position. If more work needs to be done on the front motor, it can be taken off by loosening the nuts at the bottom ends of the long rods. In order for it to come off, the coupling screw need to be loosened as well as the set screw clamping the encoder to the shaft. Th front motor should only be clamped to the bottom plate and the top plate (holding the back of the motor) should not compress the motor.

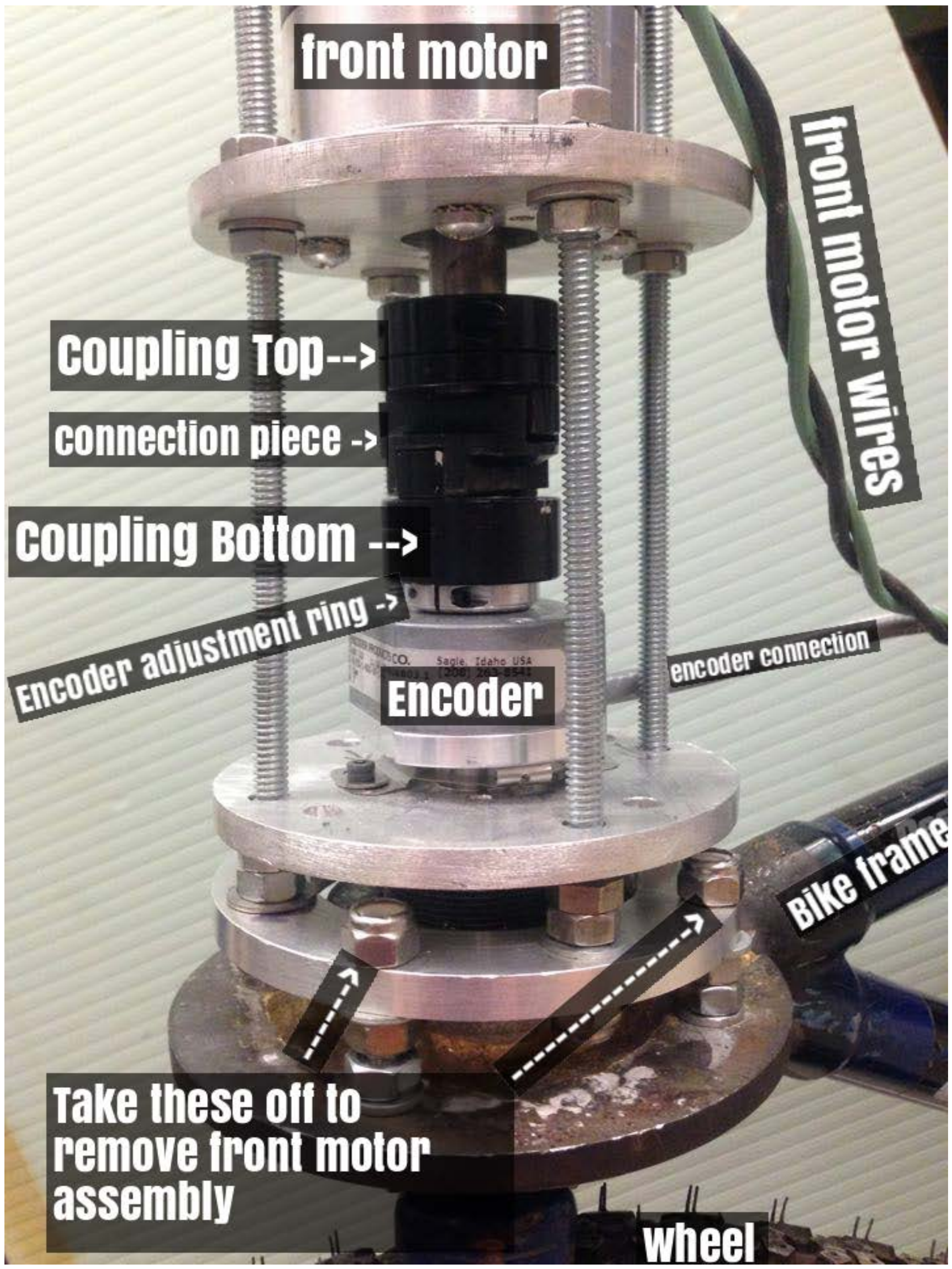


Figure 3.3: Front motor and Encoder setup

3.4 Landing-gear and Relay Module

The landing gear is setup with limit switches that will stop its motion. The motor is controlled by a relay module which switches the direction of the applied voltage to the motor. Once the landing gear reaches its end-position the limit switches turn off the supply voltage which stops the motor. Once it reaches an end-position it can only rotate in the opposite direction. The landing gear should be manually rotated past the limit switches.

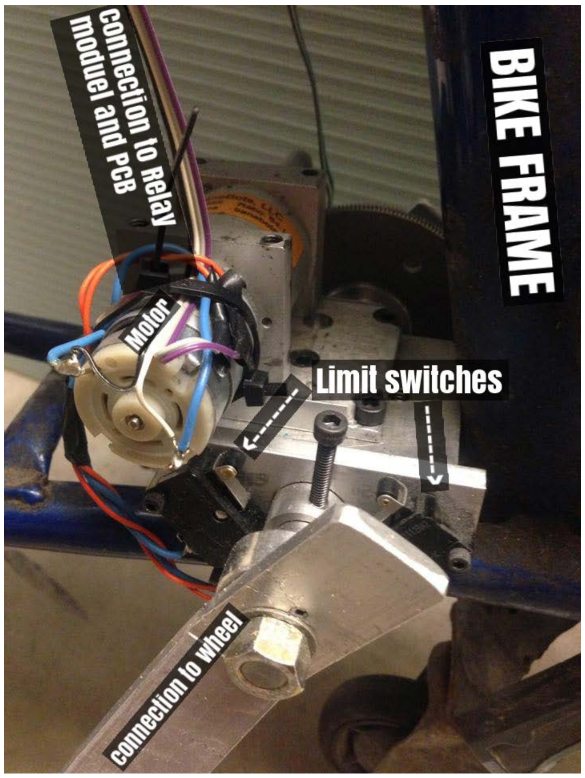


Figure 3.4: Landing gear mechanics

3.5 Power supply

The power supply consists of the battery, a switch in series and the divider (3 screw terminals). The switch allows us to turn off the supply for the whole bike with one switch. The divider has 3 terminals: one runs to the PCB, the second to the rear motor and the third to the front motor controller. The power supply will eventually also have a relay which is controlled by the watchdog so the power can be shut off automatically in an emergency.

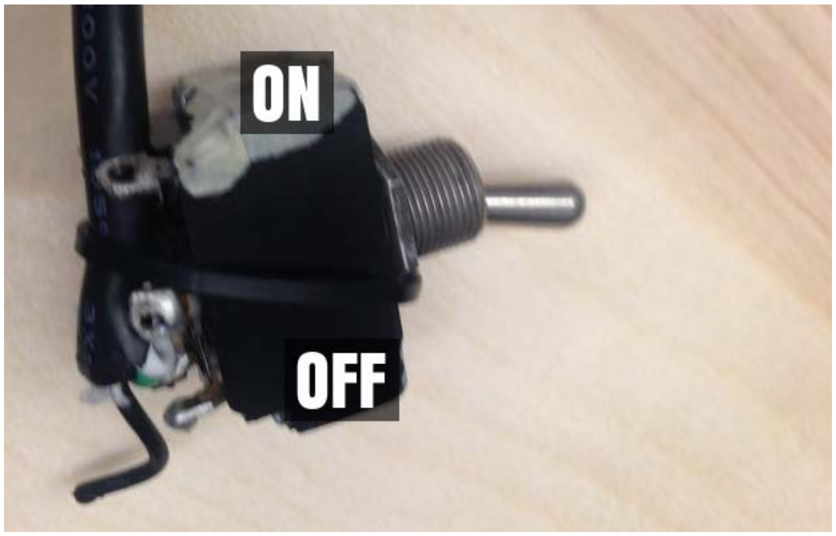


Figure 3.5: External switch for battery

3.6 IMU

The IMU mounts just under the seat and connects via ribbon-cable to the PCB. The direction of the IMU is important and needs to not wiggle during testing (should be on a stable platform).

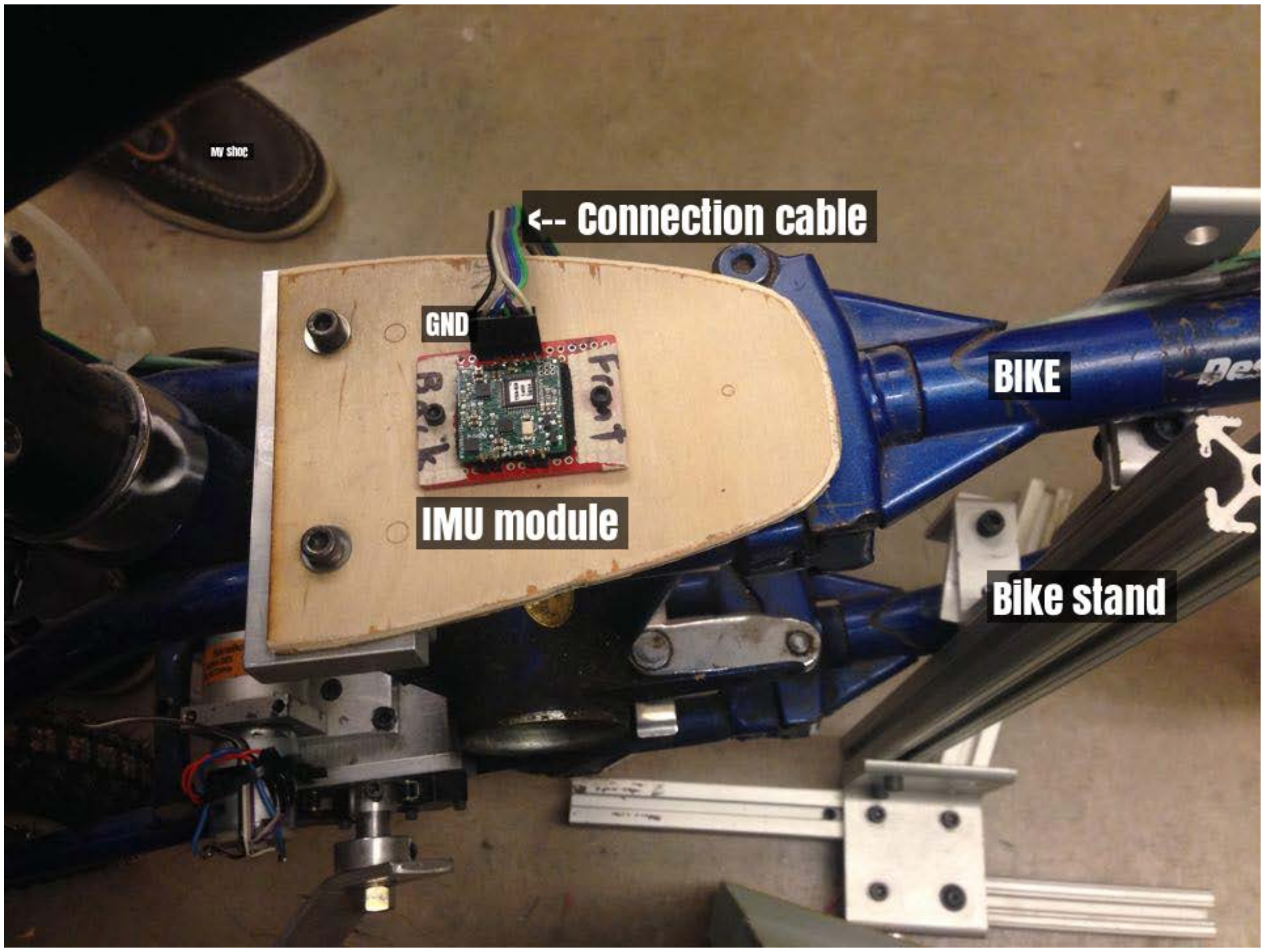


Figure 3.6: IMU on bike

3.7 Rear Motor

The rear motor is controlled by a separate motor controller which came with the original motor components. Since the motor is meant to be controlled by manual components on the handle-bar of the bike (throttle provided by the motor company) we need to mimic this with the arduino. The rear-motor control is done by sending a voltage to the motor-controller which corresponds to a motor speed. The motor speed can be accurately measured by using the hall-sensors built into the rear motor.

3.8 Motor Connections

The front motor has a motor controller which is separate from the PCB. On one end it connects to the PCB with a (yellow/green/blue) ribbon cable and on the other side it has 4 screw terminals. The inner two go to the front motor and the outer two run to the 24V supply voltage.

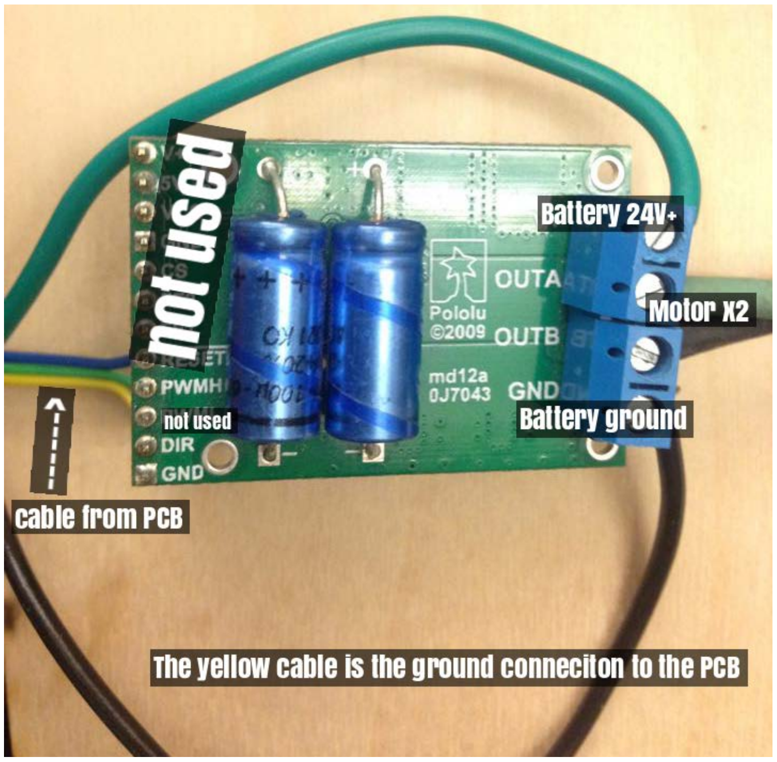


Figure 3.7: Front Motor Controller connection

The rear motor is connected by plugging the (brown/red/orange) ribbon cable into the (black/green/red) cable coming out of the rear motor controller. The brown wire is currently unused but can be used as a signal from the Hall sensors to have an accurate speed reading. The orange connection should line up with the black (ground) and the red with the green one (signal).

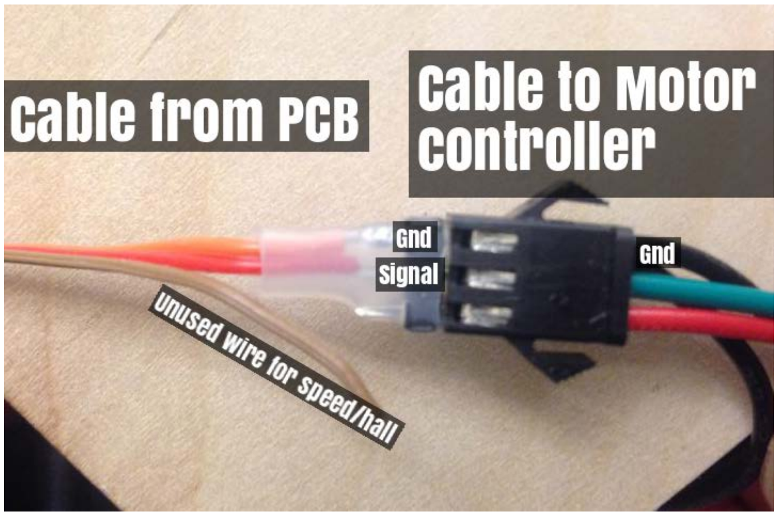


Figure 3.8: Rear Motor connection

3.9 Other notes

- Don't drop components.
- Don't short connections when using the voltmeter to measure things.
- Always keep the switch in one hand during testing in case something goes wrong.
- Make sure the bike is on the stand for testing.
- Be grounded when touching and handling electrical components.
- Think before you do something.
- Don't be stupid.
- Keep things tidy inside the box, no loose wires, no parts flopping around.
- Work systematically when debugging something and figure out what you are sure of and what you are not sure of.

4 Dynamics and Controls (Arundathi Sharma & Kate Zhou)

4.1 Overview

The dynamics and controls subteam of the autonomous bicycle project is responsible for deriving and using equations of motion (EOM) of the bicycle and developing control algorithms to balance and navigate the bicycle.

4.2 Equations of Motion

From research done in previous semesters, we found that the most appropriate bicycle model to begin with is the linear and non-linear point mass models. These equations provide a simple starting point for development of a controller. In addition, we have seen that it is possible to balance the bicycle with one control input using the three state model. The three state model keeps track of the changes in three variables, lean angle (ϕ), lean angular rate, ($\dot{\phi}$), and steering angle (δ).

The nonlinear point mass equations of motion are:

$$\ddot{\phi} = \frac{g}{h}\sin(\phi) - \tan(\delta)(\frac{v^2}{hl} + \frac{b\dot{v}}{hl} + \tan(\delta)(\frac{bv}{hl}\dot{\phi} - \frac{v^2}{l^2}\tan(\delta) - \frac{bv}{hl}\frac{\dot{\delta}}{(\cos(\delta))^2}) \quad (4.1)$$

In the above equation, the variables are defined as follows:
ϕ = lean angle (rad)
$\dot{\phi}$ = lean angular rate [rad/s]
δ = steering angle [rad]
$\dot{\delta}$ = steering angular rate [rad/s]
v= velocity [m/s]
b = distance from ground contact point of rear wheel to COM projected onto ground [m]
h = height of the bicycle COM [m]
l = distance between front wheel and rear wheel ground contact point [m]

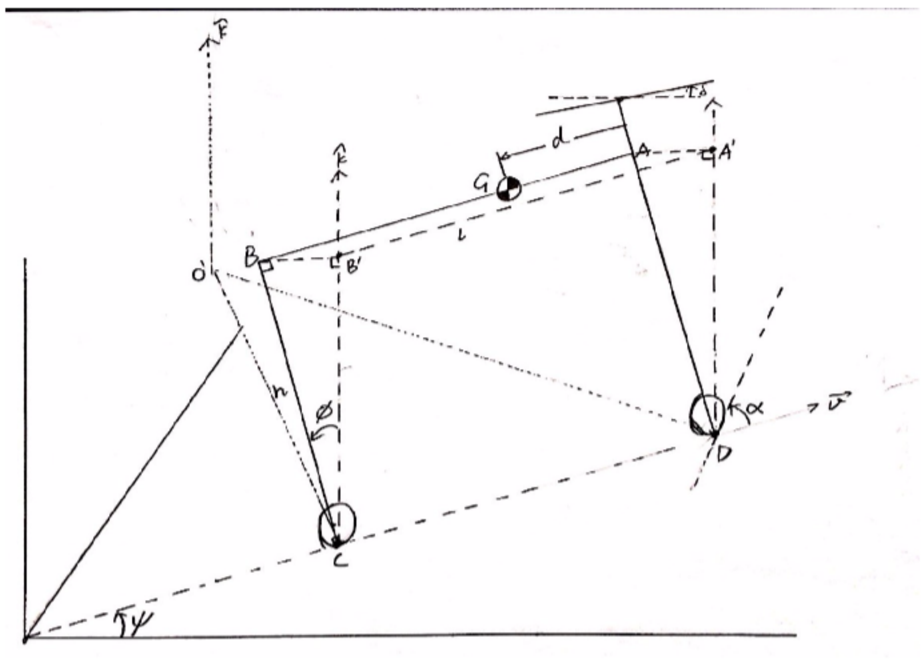


Figure 4.1: Point Mass Bicycle Model

The nonlinear point mass model (eq. 1) can be linearized for a constant speed, small perturbation bicycle model. The linearized equation of motion is:

$$\ddot{\phi} = \frac{g}{h}\phi - \frac{v^2}{hl}\delta - \frac{bv}{hl}\dot{\delta} \tag{4.2}$$

The linearized equation is written in state-space form to help with controller development. The state-space form of the linearized point mass bicycle model is:

$$\begin{bmatrix} \dot{\phi} \\ \ddot{\phi} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ g/h & 0 & -v^2/(hl) \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \phi \\ \dot{\phi} \\ \delta \end{bmatrix} + \begin{bmatrix} 0 \\ -bv/(hl) \\ 1 \end{bmatrix} \dot{\delta} \tag{4.3}$$

4.3 Bicycle Simulations

We wanted to compare our controllers with those developed by Shihao Wang (Spring 2015). So, we started from scratch and developed our own simulations and controllers. Our simulation makes use of some of the features from Shihao's simulation, including the animation from Diego.
We first checked to see how the bicycle behaved without a controller. Using the linear point-mass model, with no controller input. We found that the bicycle falls from an initial condition of:

$$\phi_0 = \pi/8$$
$$\dot{\phi}_0 = 0$$
$$\delta_0 = 0$$

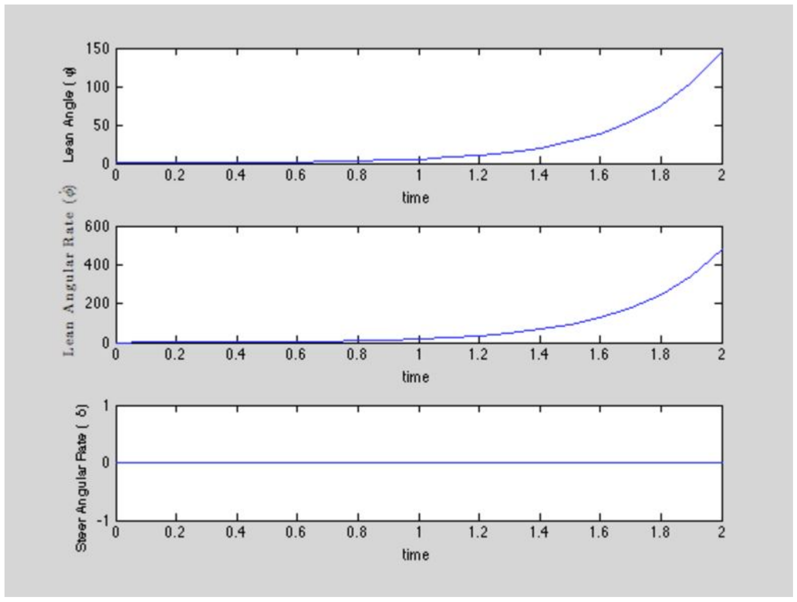


Figure 4.2: Uncontrolled Bicycle Simulation 1

Next, using the same initial conditions, we tried a random steering angular rate input of $\dot{\delta} = sin(t)$. The bicycle also fell this time.

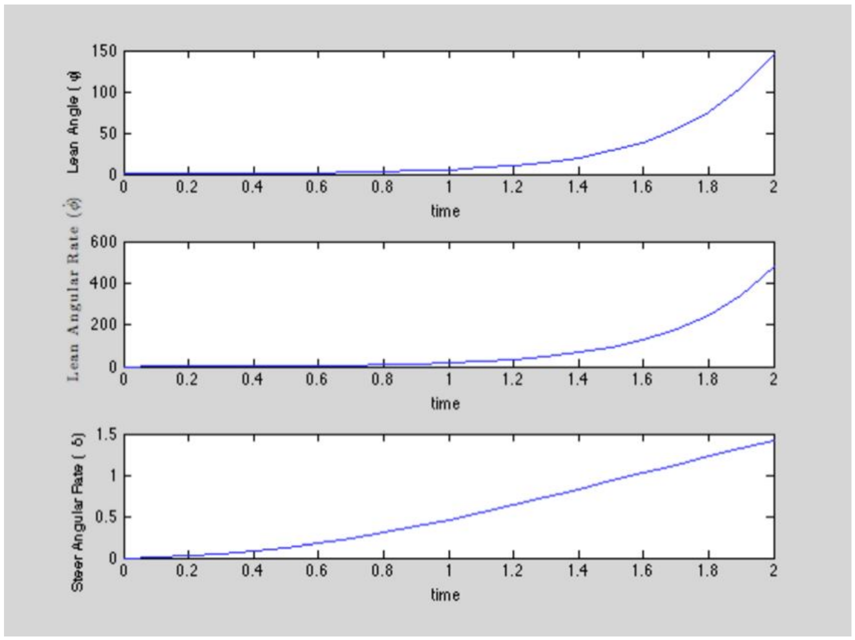


Figure 4.3: Uncontrolled Bicycle Simulation with Sinusoidal Steering Angular Rate Input

This result shows that the uncontrolled bicycle will fall regardless of $\dot{\delta}$ inputs.

We also compared the behaviour of the linear and nonlinear point mass bicycle model. We used different initial conditions to see how the two models compared with different size disturbances. One interesting observation is that the nonlinear simulation resulted in a bicycle that fell much slower.

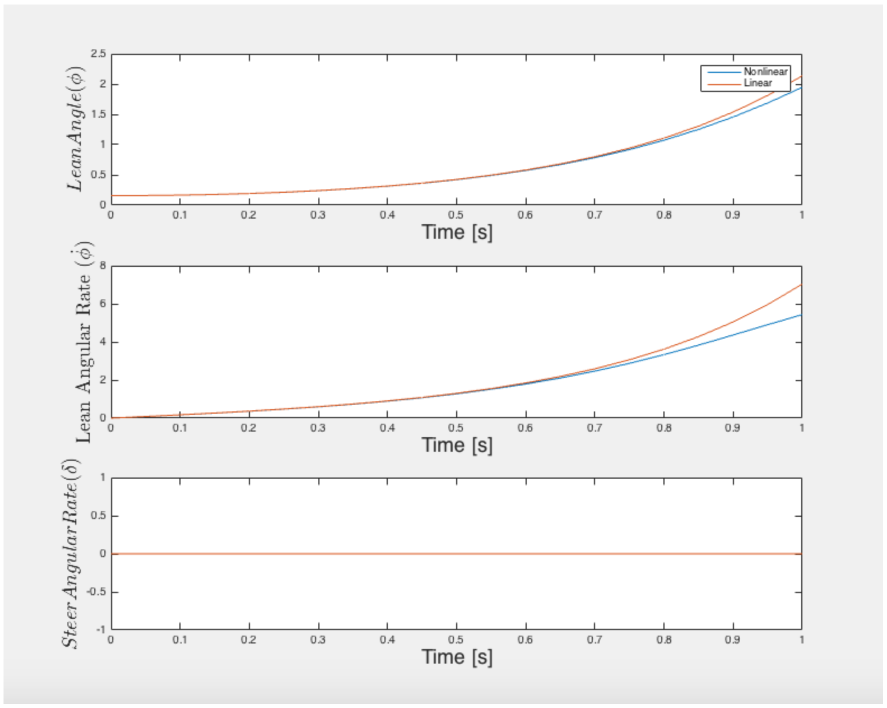


Figure 4.4: Linear vs. Nonlinear Bicycle Models

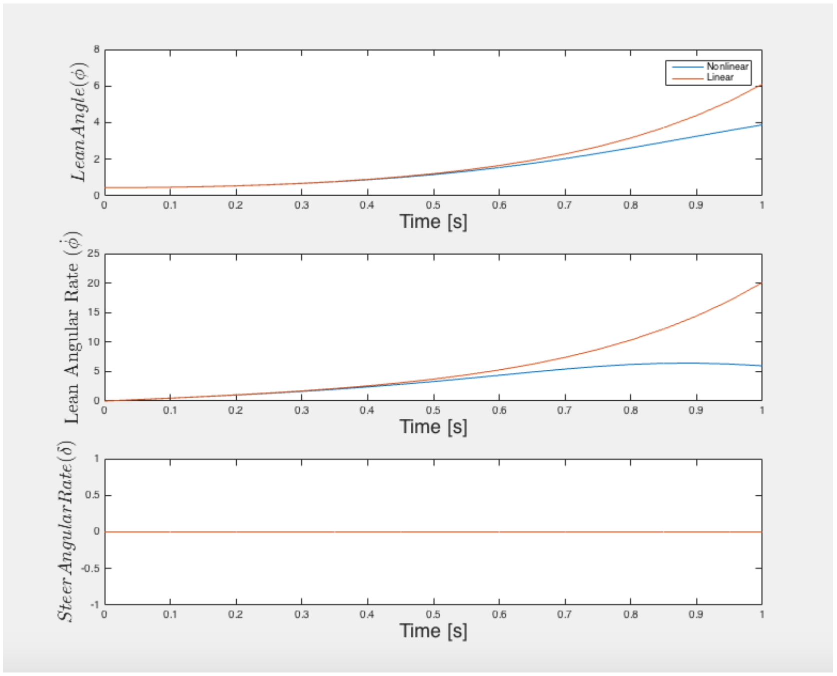


Figure 4.5: Linear vs. Nonlinear Bicycle Models

In figure 4, the initial conditons for the simulation were:

$$\phi_0 = \pi/20$$
$$\dot{\phi}_0 = 0$$
$$\delta_0 = 0$$

With a small angle initial condition, we see that the two solutions match fairly closely. In figure 5 , the initial conditions for the simulation were:

$$\phi_0 = \pi/7$$
$$\dot{\phi}_0 = 0$$
$$\delta_0 = 0$$

With a larger perturbation, the solutions from the linear and non-linear equations of motion differ more. This result is expected, as the linearized model is linearized about the unstable equilibrium point of $\phi = 0$.

4.4 Controller Development

The goals of the controller are simple and reasonable. We want to balance the bicycle while following a certain path. The bicycle should be able to stay upright without external interference and should be able to recover from disturbances. The controller can be broken down into a balance controller and a navigation controller.

4.4.1 Balance Controller

The objectives of the balance controller are:

- Bicycle should be able to recover to an upright position quickly (short settling time)
- Bicycle should be able to withstand large disturbances (uneven ground, wind, etc.)
- The controller should be robust; that is, the bicycle should be able to balance regardless of bicycle parameter errors, sensor and actuator errors, and delays

To achieve these objectives, we start by using a multivariable proportional controller. We define our control input as:

$$u = \begin{bmatrix} k1 & k2 & k3 \end{bmatrix} \begin{bmatrix} \phi \\ \dot{\phi} \\ \delta \end{bmatrix}$$

Because we have a linearized bicycle model, we could calculate the values of the gains through the desired locations of poles. For our problem, the ideal linear system response would be fast and damped. The fast response allows the bicycle to return to its upright position quickly, and a damped system avoids overshoots and oscillations.

One approach for a good controller is to start with the ideal locations of the linear system poles. For this case, we used damping ratio $\zeta > 0.7$, and real component of pole $\sigma < -0.2$. This constraint gives a search space of about 174 controllers given our bicycle geometry.

We wrote a MATLAB simulation that, based on the assigned initial conditions and gains, calculates the response. We modeled our bicycle using geometrical properties (i.e., center of mass) of our actual bicycle. At first, our simulation made use of MATLAB's built-in ode45 numerical integrator. However, we also want to realistically model our bicycle's response on the computer. In real life, sensor data cannot be sent instantaneously or continuously to the computer for processing (which is what ode45 approximates). We eventually moved over to an Euler-integration method, where we can choose our integration time-step and see how the bicycle responds if its state is only updated a countable number of times per second. This way, we can see how slow our computer can run while still balancing the bicycle. In the simulations we run, we have a timestep of 1/100s.

We also have the problem of modeling the motor response. We cannot expect the motor to output a rate of rotation that is beyond its rated capability. So, we set an upper input limit of 10rad/s. After other members in our team collected data on the behavior of the motor at lower speeds, they discovered that low duty cycles fail to produce any motor movement. We can model this in our simulation as well, by setting a lower cutoff. For instance, if the desired $\dot{\delta}$ is less than 2rad/s, the variable u, which represents motor input, is set to 0; if $\dot{\delta}$ exceeds 10rad/s, u is set to 10. These represent saturation scenarios.

We observed in our controller tests (which are described further on) that the bicycle performed very poorly with the lower-end motor restriction (minimum turn rate) of 2rad/s that was loosely based on the tests performed by other members of the team. Since then, steps were taken to reduce friction in the steering, but the lowest steering rate the motor can muster is still unclear. We recommend that future team members obtain a clear picture of how quickly and how slowly the steering wheel can move, so that simulations can be accordingly modified. Ideally, the wheel should be able to turn at infinitesimal speeds (which is what the simulations described in this report assume).

We were able to rank the controllers in our search space using a MATLAB script that scored them based on the sum of the integrals of the lean, steer, and lean rate responses. The script tests each controllers' performances over different initial conditions (ICs), which represent different-sized disturbances. If the steer angle exceeds $\pi/3$ or the lean angle exceeds $\pi/4$, in any situation, a controller automatically fails. We want our controller to succeed at slow speeds, so we included the bicycle's constant forward velocity as an IC that could be changed

The set of initial conditions to which we subjected the test controllers were as follows:

$$\begin{bmatrix} \phi & \dot{\phi} & \delta & v \end{bmatrix} = \begin{bmatrix} \pi/6 & 0 & 0 & 3.57 \\ \pi/8 & 1 & 0 & 3.57 \\ \pi/8 & 0 & \pi/5 & 3.57 \\ \pi/10 & 0 & 0 & 2 \end{bmatrix}$$

We wanted to be sure that we were choosing the best controllers we could, in the approximate range of the search space we started with. So, we ran a simulation, testing approximately 100,000 controllers. The range we searched included the controllers in the first search space (or were at least approximately in range, since all our gains in the mass-scale search were integers). The top controllers we obtained from that search were better than the top controllers from the first search. We chose the top ten controllers, and subjected them to further testing, to see if they could meet our other design requirements.

4.4.2 Robust Controller

The next step to developing a desirable controller is to simulate the possible errors when implementing it. A robust controller should be able to withstand errors in bicycle parameters, sensor reading errors, and actuator errors. The modeling of each of the errors, as we used for testing on our top controllers, are as follows:

- bicycle parameter error: a constant value offset from the true values of l, b and h

- sensor error: a constant value offset from the true IMU and encoder data reading
- actuator error: a proportional multiplication of the commanded value plus an offset

We implemented these parameters, sensor and actuator errors to successful controllers of the previous balance controller tests. We used selected a few different values of proportional multipliers and DC offset to model each error. The bike parameter errors chosen vary from -0.3 to 0.5 meters; the sensor data range from -0.02 to +0.02 radians for imu, and -0.04 rad to +0.04 rad for encoder; finally, the actuator error varies from +-1% to 5 +-% of the original value. The results of the robustness tests are scored based on four criteria:

1. the mean of the total input squared
2. the integral of all states over time after simulatng with bike parameter error
3. the integral of all states over time after simulating with sensor error
4. the integral of all states over time after simulating with actuator error

The scores from each of these criteria are then weighed to calculate a final score. The mean of the input scored is weighed the least as stability is chosen as a bigger focus than minimizing actuation effort. The weights on items 2-4 from the above list are twice as high as the mean input squared. (item 1). The set of top performing balance controllers were reduced further based on the results of the robustness tests.

The two controllers that passed the robustness test are:

$$K1 = 71$$
$$K2 = 21$$
$$K3 = -20$$

and

$$K1 = 67$$
$$K2 = 22$$
$$K3 = -18$$

The top 10 controllers from the robustness test are plotted in the gain space below. The top two performers are marked in red.

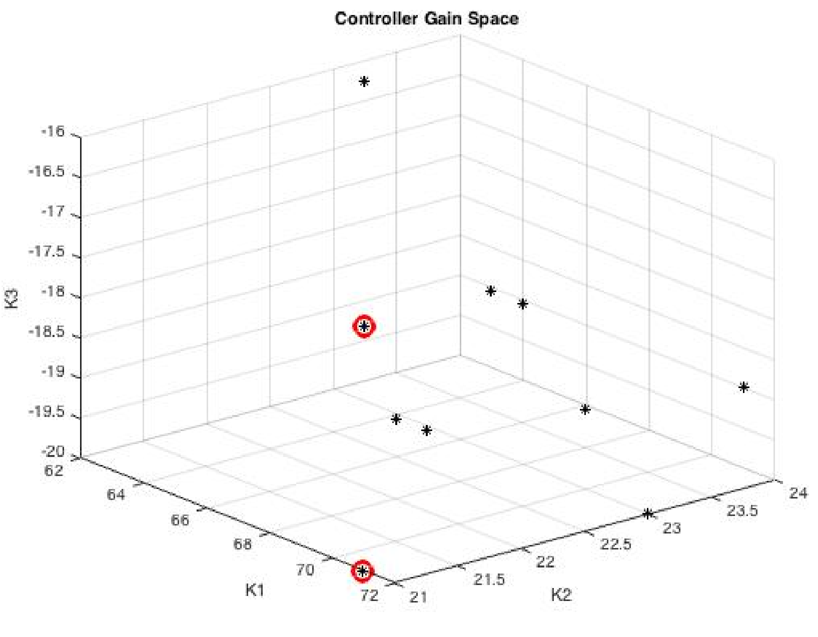


Figure 4.6: Top Controller Gain Space

4.4.3 Top Controller Characteristics

Through testing and scoring our initial set of 174 controllers and comparison with top controllers obtained from mass-scale testing, the top performer was:

$$K1 = 71$$
$$K2 = 21$$
$$K3 = -20$$

Applying this controller to the nonlinear bicycle model, the maximum lean angle it can recover from is: 41 degrees; the maximum lean rate it can recover from is: 2.5 rad/s; The maximum steering angle it can recover from is: 60 degrees. (These values are obtained using velocity of 3.57m/s and 0-initial conditions on all state variables except the one being tested). The controller is also fairly robust; it can endure bicycle parameter error of 0.75 meters DC offset and 0.25 rad DC offset sensor error.

A sample response from the top controller is shown below. The bicycle is modeled with a constant IMU error of -0.02 rad, and a constant encoder error of 0.04 rad, a initial lean angle of $\pi/8$ rad and lean rate of 0.2 rad/s.

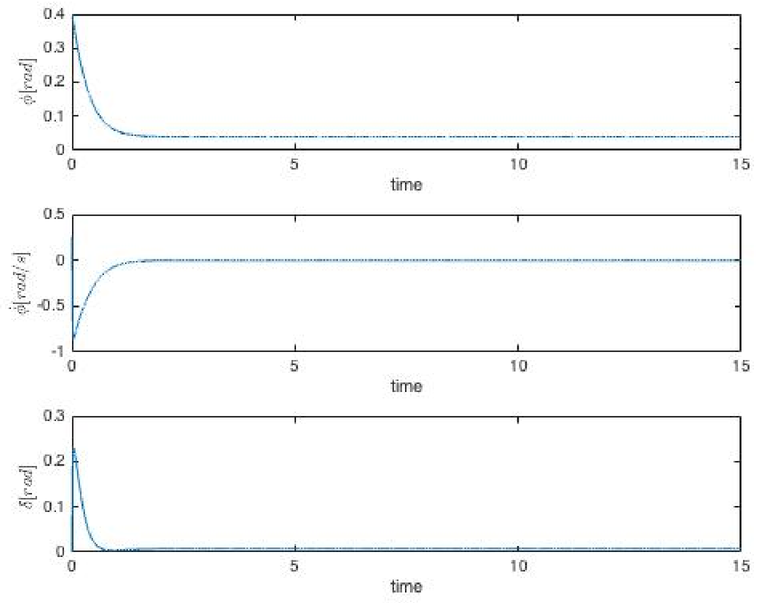


Figure 4.7: Top Balance Controller Sample Response

As shown above, the controller stabilizes the bicycle fairly quickly; but the lean angle settles to a non-zero value. This result indicates that the bicycle will continue to lean and drive in a circle due to the errors in either IMU or encoder.

4.5 Navigation Control

In addition to the balance controller, a navigation controller is also developed for the bicycle. The goal of the navigation controller is to direct the bike along a particular path. In order to control the position of the bicycle, a new bicycle model with a larger state is needed. The state now includes the x and y positions of the center of mass and the yaw angle (ψ).

The nonlinear bicycle model for navigation with constant velocity (v) is:

$$\ddot{\phi} = \frac{g}{h}\sin(\phi) - \tan(\delta)(\frac{v^2}{hl} + \frac{b\dot{v}}{hl} + \tan(\delta)(\frac{bv}{hl}\dot{\phi} - \frac{v^2}{l^2}\tan(\delta) - \frac{bv}{hl}\frac{\dot{\delta}}{(\cos(\delta))^2}) \quad (4.4)$$

$$\dot{\psi} = \frac{v}{l\tan(\delta)\cos(\phi)} \quad (4.5)$$

$$\dot{x} = v * \cos(\psi) \quad (4.6)$$

$$\dot{y} = v * \sin(\psi) \quad (4.7)$$

The algorithm for straight line navigation is as follows.

$$\delta_{target} = K_4\theta + K_5 d + K_6\dot{d} \quad (4.8)$$

where, K_4, K_5, and K_6 are controller gains, θ is the difference in the bicycle's current path angle and the desired path angle, d is the shortest distance between the bicycle's location and its desired path, and $\dot{d}$ is the rate of change of the shortest perpendicular distance between the bicycle and the desired path (d).

θ is calculated by taking the difference between yaw, ψ, and desired path angle. Both are measured from the x-axis in a global coordinate system. d and $\dot{d}$ are calculated using geometric relationships.

$$\theta = \psi - tan^{-1}(y/x) \quad (4.9)$$

$$d = \hat{k} \cdot (\hat{\lambda} \times (x\hat{i} + y\hat{j})) \tag{4.10}$$

$$\dot{d} = v * sin(\theta), \tag{4.11}$$

where v is the forward velocity of the bicycle in the $\hat{\lambda}$-direction (the direction in which the bicycle is currently moving).

The calculated desired steering angle is used to calculate the desired steering rate, as before.

$$\dot{\delta} = K_1\phi + K_2\dot{\phi} + K_3(\delta - \delta_{desired}) \tag{4.12}$$

If the calculated desired steering angle is greater than $\pi/3$, the value is manually set to $\pi/3$ for saturation.

This method should allow a self-stabilizing bicycle to successfully find any desired straight-line path. In the future, we may want the bicycle to navigate a course, perhaps consisting of path that change directions. In that case, we would model the path as a series of line segments, defined by "way-points," and have the bicycle treat each segment as a target straight-line path. Once it comes within a given distance of a "way-point" the bicycle should aim for the next line segment in the overall path.

We created a simulation that, given 3 control parameters, K_4, K_5, and K_6, simulates the bicycle's path-following behavior. We tested this simulation using the best stability controller (noted above). We did not have any formalized method for tuning this navigation controller, nor did we use any systematic scoring technique. Instead, we used intuition, based on plot information, to select a set of three parameters that would allow the bicycle to:

- reach and maintain a designated straight-line path quickly and with no oscillations, from any initial orientation and
- pass some harder navigation task: traveling through a series of way-points that did not make a straight line.

The first item was tested using the largest "disturbance" possible from a straight line: having the bicycle start facing the wrong way, 180-degrees from the desired heading. The two harder paths we chose were a square of side 50m and a 3-4-5 triangle scaled up 10 times. The set of gains that passed these tests were:

$$K4 = 0.9$$
$$K5 = 0.1$$
$$K6 = 0.1$$

We are sure that this can be tuned to perform even better, but what we have now has acceptable behavior–it can roughly follow these two paths that we have chosen, and with some more tuning (ideally on the actual, working bicycle), we believe this controller will allow the bicycle to accurately navigate a path, provided the way-points are chosen intelligently. The following plots show the bicycle's behavior; the first subplot shows the bicycle trajectory and the second shows its heading (this second plot is less relevant).

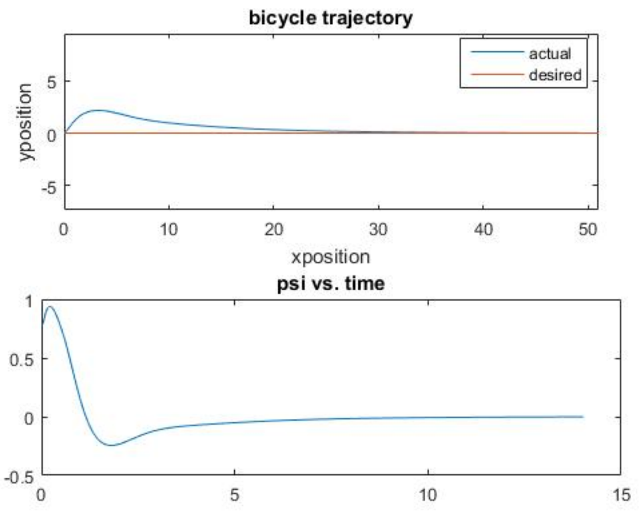


Figure 4.8: Straight-line Path Navigation (initial heading offset: $\pi/8$)

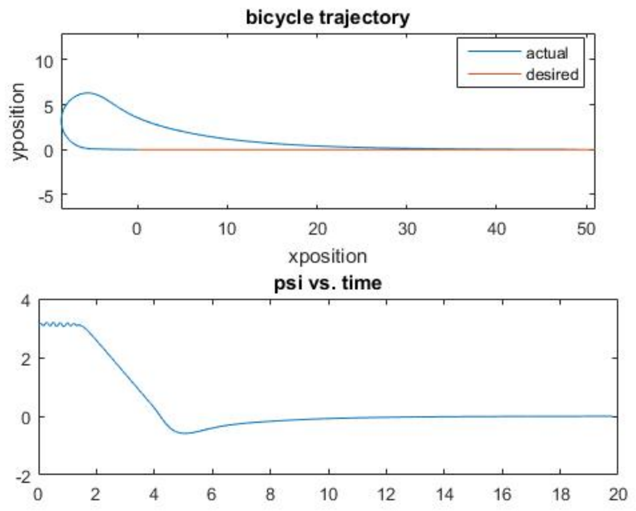


Figure 4.9: Straight-line Path Navigation (initial heading offset: π)

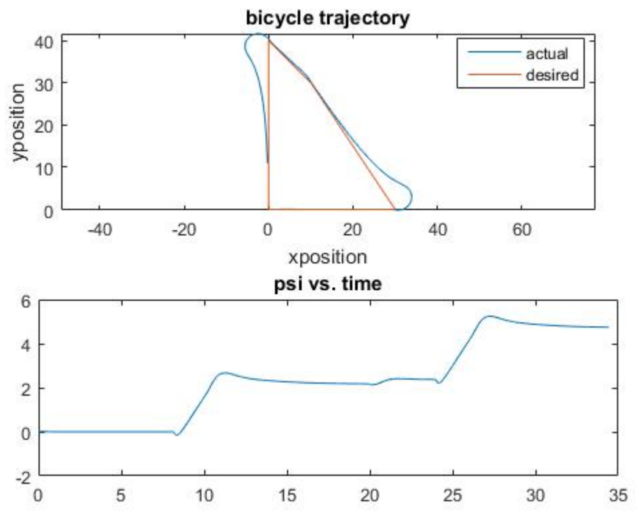


Figure 4.10: Triangular Path Navigation

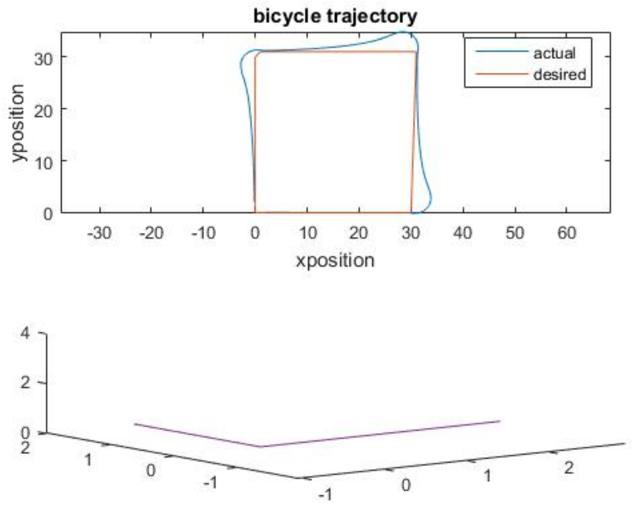


Figure 4.11: Square Path Navigation

4.6 Conclusion

The balance controller we have designed and customized to our bicycle appears to be robust and effectively and realistically stabilizes a bicycle in simulation. The navigation controller we have designed also appears to allow the bicycle to follow a variety paths, defined by a series of waypoints. With a working bicycle, these parameters will have to be tuned slightly, and should produce a self-stabilizing, autonomous bicycle. For reference, our report includes a zip file containing other zip files of MATLAB functions that can simulate bicycle behavior, evaluate/score/rank stability controllers, simulate modeling/sensor/actuation errors for robustness testing, simulate bicycle navigation, and animate bicycle behavior. All of this code (with the exception of the animation, which was written by Diego Olvera, Spring 2014) was written by Kate Zhou and Arundathi Sharma during Spring 2016. We look forward to seeing our controllers tested out on a real bicycle.