Cornell Autonomous Bicycle Project Team

Fall 2017 Report

Cornell Biorobotics and Locomotion Lab

Supervisor: Professor Andy Ruina

Cornell University

Team Members				
Name	Year	Major	Team Position	Credits Taken
Anya Chopra	Junior	CS		3
Aviv Blumfield	Senior	MechE		0
Bobby Villaluz	Freshman	CS		1
Conrad McCarthy	Junior	MechE		3
Connor Li	Freshman	MechE		1
Daniel Glus	Sophmore	CS		3
David Miron	Junior	ECE		3
Dylan Meehan	Sophomore	CS		3
Graham Merrifield	Senior	MechE		3
Jared Frank	Senior	CS		0
Jeremy Iver	Junior	MechE		3
Joshua Sones	Junior	CS	Navigation STL	3
Joshua Even	Junior	CS	Embedded STL/Vision	3
Jordan Stern	Sophomore	CS		3
Kane Tian	Freshman	CS		1
Linda (Haitian) Lu	Sophomore	MechE		3
Max Kester	Sophomore	MechE		3
Nicolas Barone	Sophomore	CS		3
Olav Imsdahl	Meng	MechE	Steer-By-Wire STL	3
Olivia Xiang	Sophomore	CS		3
Rohit Bandaru	Junior	CS		3
Sam Hong	Freshman	MechE		1
William Murphy	Senior	MechE	Team Lead	3
Woo Cheol Hyun	Freshman	CS		1

January 6, 2018

Contents

1 Introduction **3**

2 Dynamics and Controls **3**
2.1 Using MATLAB to Accurately Represent the Real World . . . 5
2.1.1 Improvement of the Score System . . . 5
2.1.2 Modeling Imperfections as Sensor Error . . . 6
2.1.3 Remodeling the Bike's Physical Properties . . . 7
2.1.4 Center of Gravity . . . 9
2.1.5 Steer Rate Settling Time . . . 10
2.1.6 MATLAB Simulation of the Bicycle's Updated Physical Properties . . . 11
2.2 Gain Optimization Using MATLAB . . . 13
2.2.1 Necessity for Optimization . . . 13
2.2.2 Optimizing Gains by Finding Balance Score Local Minima . . . 13
2.2.3 Observed Grid Search Gain Optimization Trends . . . 14
2.2.4 Creating a Continuous Gain Function . . . 15
2.3 Applying the Simulation to Real Life . . . 16
2.3.1 Adjustments and Preparation . . . 16
2.3.2 Results . . . 17
2.3.3 Next Steps . . . 20

3 Hardware **20**

4 Software **21**
4.1 Embedded Systems . . . 21
4.1.1 ROS Arduino Wrapper . . . 21
4.1.2 Serial Protocol . . . 22
4.1.3 Sensors . . . 23
4.1.4 Tests . . . 24
4.1.5 Matlab simulations . . . 32
4.2 Sensor Fusion . . . 33
4.2.1 Introduction . . . 33
4.2.2 Model . . . 33
4.2.3 Predict . . . 34
4.2.4 Update . . . 36
4.2.5 Tuning . . . 37
4.3 Navigation . . . 37
4.3.1 Algorithm . . . 38
4.3.2 Simulation . . . 38
4.3.3 Position Estimation . . . 39
4.3.4 ROS Communication . . . 40
4.3.5 Buck Testing . . . 41
4.3.6 Bike Testing . . . 42
4.3.7 Future Plans . . . 43
4.4 Computer Vision . . . 43
4.4.1 Obstacle Detection . . . 43
4.4.2 RTAB-Map . . . 44
4.4.3 Integrating Obstacle Avoidance . . . 44
4.4.4 Hardware . . . 45

4.4.5 Odometry and SLAM 46
4.4.6 Segmentation and Classification Algorithms 47

5 Future Work **48**

Appendices **48**

A Hardware **48**
A.1 Front Motor Gains Optimization 48
A.2 Landing Gear 48
A.3 Watchdog 48

B Software Appendix **48**
B.1 Embedded 48

C Navigation Testing Rig (Buck) **48**

1 Introduction

The Cornell Autonomous Bicycle Team develops a robotically-stabilized bicycle with the goal of balancing better than any other autonomous bicycle. This paper will present advancements made during the Fall 2017 semester, which represent the most impressive, up-to-date capabilities of our first prototype, the self-balancing bicycle. For the most current information and team affairs, please see our website at https://bike.engineering.cornell.edu/

After the spring 2016 semester, the prototype could self balance using steering control at 3.5 m/s, with Radio Control guidance for navigation. This semester, we improved our balance controller using different gains depending on the forward speed of the bicycle (which we can keep constant at a given speed), successfully balancing at 2 m/s. We also improved the MATLAB simulation of our bicycle and developed a Bayesian Optimization routine to better select balance controller gains. Furthermore, we developed a navigation algorithm to direct our bicycle based on GPS data. We have completed tests of the navigation algorithm on a simulated bike (a wooden board with the necessary navigation sensors). This required the associated sensor fusion and communication between sensors and algorithm to be developed. Next semester we hope to implement navigation control on our prototype to make our self balancing bicycle fully autonomous in navigation and balancing actions.

2 Dynamics and Controls

The balance controller of our bicycle is derived from the equation of motion of a point mass model of a bicycle. For a detailed derivation see Shihao Wang's 2014 report titled "Dynamic model derivation and controller design for an autonomous bicycle" available here. The following figure from that report illustrates a point mass model of a bicycle.

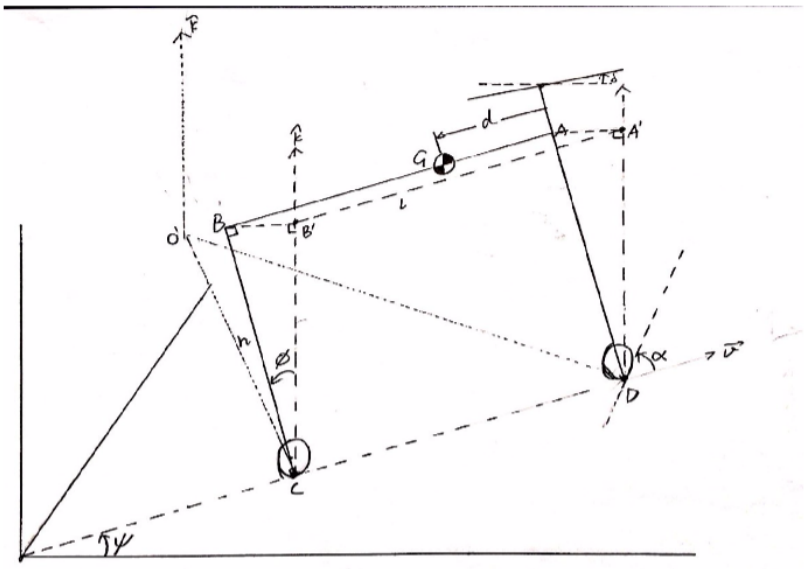


Figure 2.1: Point Mass Model of a Bicycle

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

$$\ddot{\phi} = \frac{g}{h}\sin(\phi) - \frac{v^2}{hl}\tan(\delta) - \frac{bv\dot{\delta}}{hl\cos^2(\delta)} - \frac{b\dot{v}}{hl}\tan(\delta) + \frac{v^2}{l^2}\tan^2(\delta)\tan(\phi) - \frac{bv\dot{\phi}}{hl}\tan(\delta)\tan(\phi) \tag{1}$$

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

ϕ = lean angle (rad)

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

$\ddot{\phi}$ = lean angular acceleration [rad/s^2]

δ = steering angle [rad]

$\dot{\delta}$ = steering angular rate [rad/s]

v = velocity [m/s]

b = distance from ground contact point of rear wheel (C) to center of mass (G) projected onto ground [m]

g = acceleration due to gravity [m/s^2]

h = height of the bicycle center of mass [m]

l = distance between front and rear wheel ground contact point [m]

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

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

This equation can be manipulated to develop the linear equation of our balance controller. Lean angle, lean angle rate, and steer angle are input to the balance controller which

outputs a steer angle rate.

$$\dot{\delta} = k_1 \cdot \phi + k_2 \cdot \dot{\phi} + k_3 \cdot (\delta - \delta_{ds}) \tag{3}$$

The variables in the linear controller are defined as above. Note δ_{ds} is desired steer angle input to balance controller. $\delta_{ds} = 0$ for balance controller testing.

The constants (gains) of the controller are k_1, k_2, and k_3. These are found using a grid search in MATLAB. For each set of gains, the bicycle is simulated for 10 meters. If the bicycle does not fall, a balance score and path score are assigned to the set of gains. A fall is defined as $\mid \phi \mid > \frac{\pi}{4}$. This optimization method is very inefficient, but conceptually simple and easy to implement.

2.1 Using MATLAB to Accurately Represent the Real World

2.1.1 Improvement of the Score System

Balance Score

In previous semesters, a score system was based off of the bike's lean angle ϕ, lean angle rate $\dot{\phi}$ (the time derivative of its lean angle), and steer angle δ. The equation used to compute the score is as follows:

$$score = \sqrt{\sum \phi + \sum \dot{\phi} + \sum \delta} \tag{4}$$

NOTE: Summation signs represent the aggregate total of each angular state throughout the bike's entire trajectory, for a simulated period of ten meters.

In order to optimize the gains, we must minimize the score. The reason for this being that a stable bike will yield a lean angle, lean angle rate, and steer angle that all converge quickly to zero. We realized, however, that the stability of the bike does not directly correlate to the aggregate totals of its lean angles and steer angles. For example, a bicycle that is traveling in a circle at a constant velocity will have a constant non-zero steer angle and lean angle. The turning bicycle can be considered as stable as a bicycle that is traveling in a straight line, despite the fact that its balance score would not converge to zero. Therefore, we decided to remove the ϕ and δ terms from our balance score so as not to discourage the bike from reaching stability about a turn. The new balance score is implemented as follows:

$$score = \sqrt{\sum \dot{\phi}^2} \tag{5}$$

Taking the square root of the sums of squares is a standard way of weighting multiple values of interest and normalizes the values appropriately. This new score seeks to minimize only the rate of change of the bike's lean angle ($\dot{\phi}$), effectively the bike's "wobble."

Path Score

In contrast with the balance score, we would also like some way to quantify the bike's ability to conform to a straight line path. Our MATLAB simulations include no navigation instructions to the bicycle. Thus, the bike will turn in any direction to stay balanced. Balancing the bike is our primary goal. Our secondary goal is to balance the bike on a straight-line path. The bicycle is simulated for a fixed distance, regardless of the direction it travels. Therefore, the bike's ability to stay on a straight-line path is directly related to the distance between its end point and the end point of the desired path, which can be quantified as:

$$navigationscore = \sqrt{(x_{desired} - x_{test})^2 + (y_{desired} - y_{test})^2} \tag{6}$$

Where x_{test} is the final x value of the bike at the end of its simulated trip and $x_{desired}$ is the input final x value of the intended, straight line, trip.

Analysis of Scores

The new balance score, equation 5, is the primary formula for quantifying the success of a chosen set of gains. The path score is useful but is not of immediate need. For now, we would like to ensure that the bike can successfully balance at low velocities by any means necessary. The final section of the paper discusses the trends that arise when using this scoring system to optimize the balance control gains. Our testing of the gains on the real bike will either support our decision to use this scoring system or motivate us to pursue other methods of scoring.

2.1.2 Modeling Imperfections as Sensor Error

During testing, we realized that there are many real world imperfections that are not accounted for in the MATLAB simulation. Although we cannot necessarily calculate these imperfections because they are unpredictable by nature, we can model them as errors in the bike's physical state at each time iteration. These errors can represent either errors in the GPS and IMU readings or physical imperfections such as a bumpy path, wind, or hardware malfunctions. In control theory analysis, our goal is to create a robust enough controller such that it can handle the worst possible conditions. Consequently, we input a sensor error feature into the MATLAB simulation in order to account for these conditions in optimizing our gains.

Initially, the error was created such that the bike reads its own lean angle with an offset of 1° from the actual value at each integer second. For example, if at time t = 3 seconds the lean angle is $\phi = 10^\circ$, the bike would read it as $\phi = 11^\circ$. Since the following physical states of the bike depend on the bike's reading of its current lean angle, this error propagates and causes a desired instability in the bike's trajectory. In order to make the impact of the error more drastic, we decided to rewrite the error such that it creates a continuous 10° offset before every odd integer time step and a continuous -10° offset before every even time step. For each 1/60 of a second time step between t = 0 seconds and t = 1 seconds, the lean angle is read as

follows:

$$\phi_{sensor} = \phi_{actual} + 1^{\circ} \tag{7}$$

Conversely, at each 1/60 of a second time step between t = 1 second and t = 2 seconds, the lean angle will read as follows:

$$\phi_{sensor} = \phi_{actual} - 1^{0} \tag{8}$$

This new sensor error algorithm effectively adds a net continuous 60° of error between each full second time step, causing the bike to have a much more profound wobble. The bike's lean angle never reaches 60°, however, since the balance controller accounts for the error at each 1/60 of a second interval. In order to more comprehensively test the bike's balancing capabilities, this algorithm can also be applied to the bike's lean angle rate $\dot{\phi}$ as well as its steer angle δ. A combination of a continuous 10° offset for all sensor errors is evident by the bike's oscillating yaw throughout its trajectory as seen in Figure 2.2:

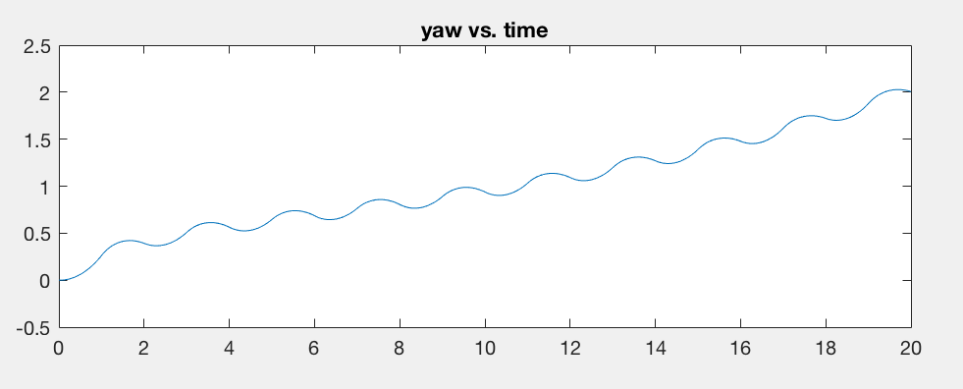


Figure 2.2: The yaw or orientation of the bicycle's body along a path as it accounts for 1 degree offset error in the lean angle, lean angle rate, and steer angle.

2.1.3 Remodeling the Bike's Physical Properties

Moment of Inertia

The Matlab model of the bicycle relies on a point mass model of the bicycle. (See link XXX here for an explanation of the point mass model) The height of the point mass is an important value to include in simulation. If the center of mass is located at the actual center of mass of the bike, then this is accurate in that this point is where the force of gravity is modeled to act through. However, placing the point mass at the physical center of mass of the bike does not guarantee that the point mass will oscillate (or fall) with the same dynamics as the physical bike. In order to find the appropriate height, we found the the natural frequency of the physical prototype would fall at. This can be shown through the simplified model of comparing a physical pendulum to an ideal point mass pendulum and implementing the small angle approximation.

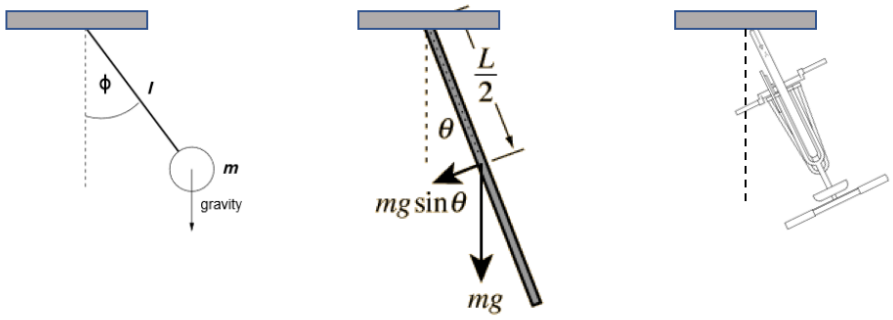


Figure 2.3: Displayed is the progression of models of the bike from a point mass on the left, a rod in the middle, to the actual bike on the far right. They are not modeled as inverted pendulums because it was easier to set up the experiment to have the bike oscillating upside down than right side up.

$$\omega_{n,physical} = \sqrt{\frac{mgl_{cm}}{I_s}} \qquad \omega_{n,ideal} = \sqrt{\frac{g}{l}} \tag{9}$$

m = mass of the pendulum
g = gravitational acceleration
l_{cm} = location of the center of mass of the physical pendulum = L/2 in figure 2.3
I_s = first moment of inertia of the physical pendulum
l = location of the point mass in the ideal pendulum

These equations can be set equal to each other in order to solve the location to put the point mass in relation to the length of the physical pendulum such that the natural frequencies are the same. If a rod is used for the physical pendulum, it can be shown that:

$$l = \frac{2L}{3} \tag{10}$$

This means that the point mass should be placed 2/3 along the length of the rod in order for them to oscillate with the same frequency when they are displaced. Keeping the larger picture in mind, we want the point mass model of the bike in the simulation to fall with the same frequency that the physical bike falls. In order to accomplish this, we needed to find what the natural frequency of the bike is. This was found by hanging the bike upside down and securing it such that the bottoms of the wheels were contacting the rotation point as they would if the bike was right side up. We then displaced the bike and timed the oscillations. This experiment is as shown in figure 2.4.

In order to find a frequency in the appropriate units, we had to multiply the natural frequency measured by 2 π:

$$\omega_n, ideal = \frac{1}{2\pi}\sqrt{\frac{g}{l}} \quad \Rightarrow \quad l = \frac{g}{(2\pi w_n)^2} \tag{11}$$

Figure 2.4
Displayed is the physical set up of the experiment performed. The bike as hung low in case of a fall and the wheels attached as closely to the pivot point as possible.

We find that the location to place the point mass is l = 0.5156 m above the ground. The point mass in the original simulation was placed at 0.9 meters above the ground so we expected this change to be significant. That being said, we still didn't know what the actual location of the center of gravity of the bike is. It is also important to note that we don't need to know where the center of mass is located other two directions (longitudinally along the bike or out of plane of the bike) because the bike can be assumed to always be falling about the the axis parallel to the wheel contacts with the ground and we can assume by symmetry that the bikes center of mass is zero (symmetric) in the direction perpendicular to the plane of the bikes frame.

2.1.4 Center of Gravity

We wanted to calculate an accurate center of gravity of the bike in the z direction, or the $\hat{k}$ direction in figure 2.1 in order to compare it to the effective height given the natural frequency of the bicycle. In order to find the center of gravity of the bike, an experimental method was utilized. We hung the bike from two different points from a chain attached to the ceiling rail in the lab. From there, we took photos of the bicycle from these different hanging points and used the chain as a reference line for the line of action of gravity through the bike. By superimposing the two images and extending the line of action of gravity, we found the center of gravity of the bike by finding the intersection of those two lines. Then, by using a known length in the

photos (the distance from the bottoms of each tire is 0.935 m) we could find the height of the center of gravity in the z direction by comparing its height to the length between the wheels. The superimposed images and the experimental setup are shown in figure 2.5.

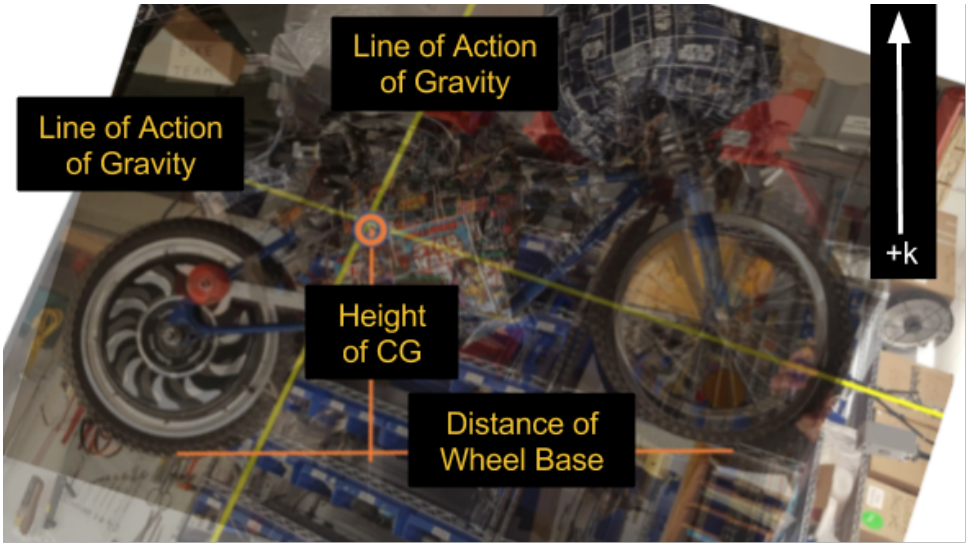


Figure 2.5: Displayed are two superimposed images of the bicycle as it hangs from two different points. The yellow lines are extensions of the chain by which the bike hung and the orange lines are reference lines in order to know the actual height of the center of mass.

This experiment provided a center of mass location of 0.4 m in the vertical direction. This agrees with what my physical intuition of the bike's size. Additionally, the effective point mass location based on the falling frequency of the bicycle is located about 0.1 m above this point. This results agrees with the analysis the ideal pendulum vs. the rod pendulum.

2.1.5 Steer Rate Settling Time

We wanted to find the maximum angular velocity the front wheel turns at on the prototype in a typical balance scenario. Taking a set angle, divided by the settling time required for the front wheel to achieve that angle gives us the maximum angular velocity that the front wheel can turn at on our physical prototype. There are two ways we have considered to find the settling time of the front motor. The first is to let the front wheel spin continuously and measure its rate of rotation. This will provided a steady state angular velocity. The second is the deflect the front wheel and measure the time it takes to return to the desired position.(figure 2.6) This will provide a "transient" response of the system and give a more valuable rate since the front steering wheel is most often in a transient state while balancing, due to the small angles required for a typical balance controller correction. We performed a test to diagnose "transient" settling time of different sets of front motor controller gains to see which performs best: with the smallest settling time and and overshoot weighted equally. We only show two trials, but after many trials there was a similar level of consistency, showing a precise characterization. We performed this test by rotating the front wheel by pi/2 radians from the straight forward position and letting go such that the front wheel returned to its desired position (0 radians

or straight forward). While the bike will never actually have to turn the front wheel by pi/2 radians, this provides an easily repeatable experiment. We divided pi/2 (angle traveled) by the settle time in order to find an average angular velocity for the front wheel in a transient response. Finding this average angular velocity is sufficient for the simulation and is still more helpful than finding a rotation rate to a steady state response. Imperfections my arise in that there was no friction between the wheel and ground when the test was performed, but when rolling, this friction force is actually quite low. These tests showed that the transient steer rate is 4.8 rad/s and this parameter was implemented in the simulation as a saturation to the steer rate. Whenever the simulation tries to make the steer rate higher than this value, it is capped and assigned the angular velocity of 4.8 rad/s both in clockwise and counterclockwise rotation. Given the consistency of the tests as shown in figure 2.6, we can confidently say that this is a helpful and sufficient parameter for the simulation.

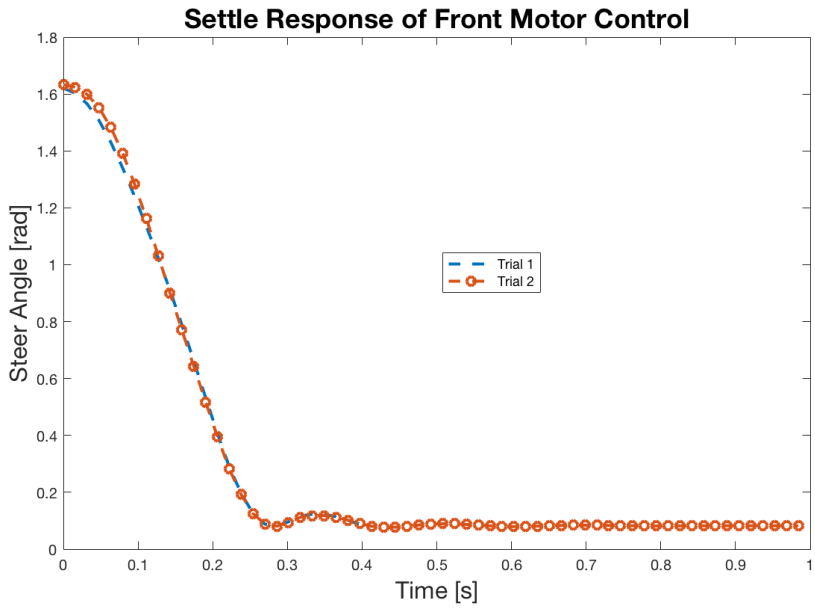


Figure 2.6: The settling response of a pi/2 deflection of the front motor. Using a 2 % settle time the maximum steer rate was found to be 4.8 rad/s. Two runs are shown to demonstrate the repeatability and consistency of the response to this angular offset of pi/2 radians.

It is important to note that this test was done on the front motor prior to it falling and being replaced by an older motor. It was observed that this motor likely had more internal friction which directly affected the rate at which it could settle to a desired position and this is the motor currently on the bike. To accommodate this, the max PWM for the front motor was increased from 100 PWM to 255PWM and the P and D gains adjusted to achieve a desirable response.

2.1.6 MATLAB Simulation of the Bicycle's Updated Physical Properties

The MATLAB simulation of the bicycle was updated with the newly found physical properties. We tested the success rate of the bicycle using both the updated physical properties and the

old values, using the same sets of gains, to compare the two. The new dimensions of the bicycle effectively makes it a shorter pendulum, which is more difficult to balance and thus should have a lower success rate. Comparing the success plots of the two sets of values shown in Figure 2.7 produces notable differences: With the new center of mass, the average success rate was 41.1375% and with the old center of mass, the average success rate was 48.4197%. The MATLAB simulation behaved as expected, with the new physical parameters producing a lower success rate than the old. This is promising as the simulation performed accordingly to a real life expectation.

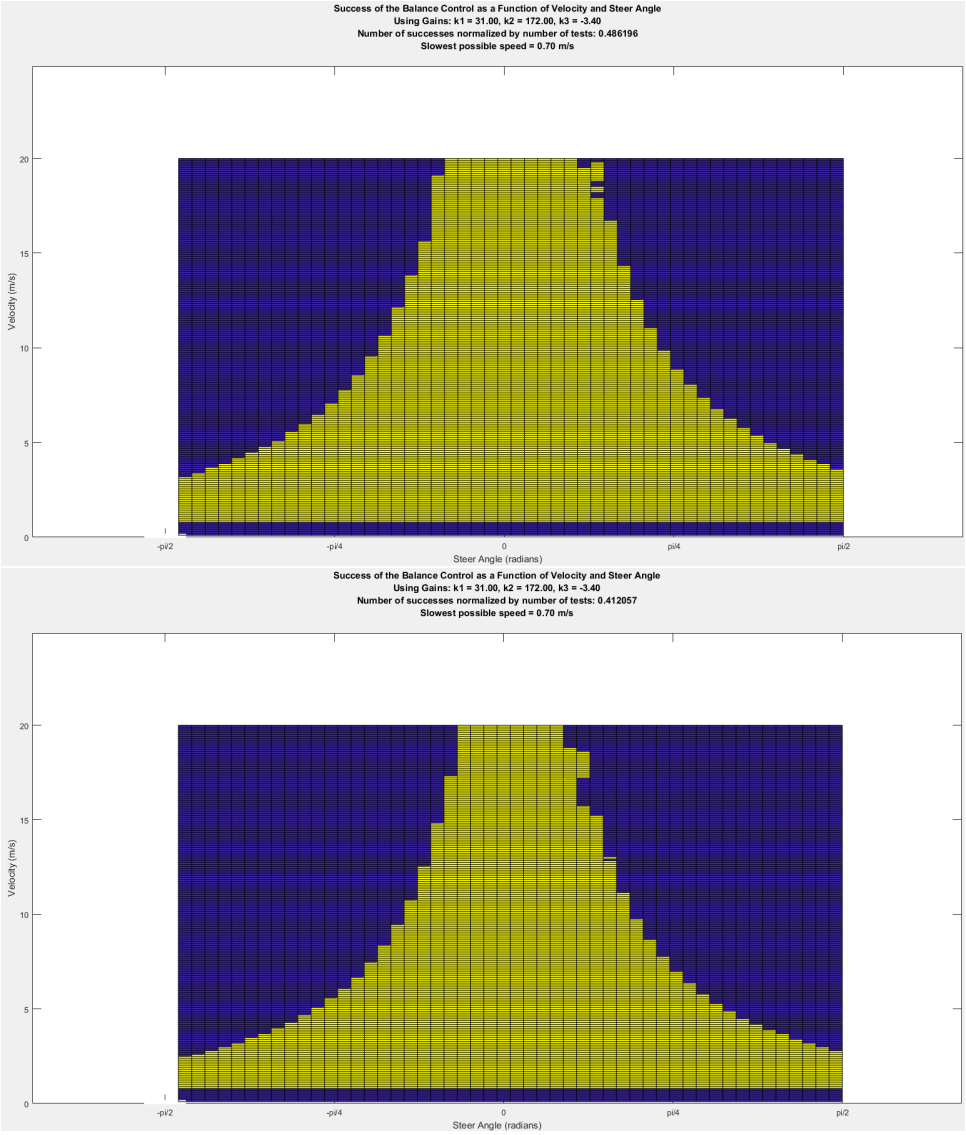


Figure 2.7: Sample success plot of the bicycle using the old (top) and new (bottom) center of mass, using gains optimized for 1 m/s and an accumulated lean angle error of 1 degree. The success plot displays when the bicycle balances successfully in yellow given an input velocity (y-axis) and initial steer angle (x-axis).

2.2 Gain Optimization Using MATLAB

2.2.1 Necessity for Optimization

To balance, the bicycle determines how to steer the front wheel in order to stay upright. Specifically, the lean angle (ϕ), lean angle rate ($\dot{\phi}$), and steer angle (δ) are used to determine a new steer angle rate ($\dot{\delta}$). The function for steer angle rate is expressed as follows:

$$\dot{\delta} = k_1\phi + k_2\dot{\phi} + k_3\delta \tag{12}$$

The constants k_1, k_2, and k_3 are weights, or gains, that determine the dominance of each term in our balance controller. Optimizing these gains allow the bicycle to balance "better", with a lower balance score. The sign of k3 must be opposite the sign of both k1 and k2. For example, if $\phi = 0$, $\dot{\phi} = 0$, and the front wheel is steered to the left, the front wheel should turn to the right for the bike to balance. $\dot{\delta}$ must be in the opposite direction of δ, so k_3 must be negative. By steering in the direction of a fall, the bicycle balances by returning its wheel base to under its center of gravity.

In previous semesters, the bicycle in simulation could not balance slower than 0.6 m/s. By updating the bicycle's center of gravity and max steer angle rate, the bicycle could balance at 0.25 m/s in simulation.

Rear wheel speed, v is a parameter in the equation of motion of a bicycle:

$$\ddot{\phi} = \frac{g}{h}\phi - \frac{v^2}{hl}\delta - \frac{bv}{hl}\dot{\delta} \tag{13}$$

Thus, rear wheel velocity affects the optimal gains. In general, the faster a bicycle travels, the easier it can balance. For a robust bicycle, optimal gains must be found at a range of speeds.

2.2.2 Optimizing Gains by Finding Balance Score Local Minima

Gains were optimized using a MATLAB simulation of the bicycle. The script balanceControlOptimizer simulates the bicycle over many sets (k_1, k_2, k_3) of gains. For example, the domains $a < k_1 < b$, $c < k_2 < d$, $e < k_3 < f$ can be input to balanceControllerOptimizer which would simulate the bicycle for each permutation of the three gains within there respective domains. The script finds the set of three gains which produce the lowest balance score. This method is called grid search optimization. Finding the optimal controller by enumerating all of the gains is possible but inefficient. This method takes cubic time in the ranges of the three gains.

To minimize the time and computational power required to optimize the gains, trial and error in combination with basic calculus-based optimization can be used to determine the local minima of the balance score using a process visualized in Figure 2.8. During simulation testing, the optimization script would often return gains that were on the ends of each entered domain as the best gains. Shifting the domains in the direction of those gains and reentering them into the script would usually return the gains on the same ends of the new domain, while at the same time improving the returned balance score. This led us to the conclusion that the

optimizer script functioned similarly to a continuous, differentiable function of three variables; moving the input domains in the direction of the gains that produced the best balance score will continue to improve said balance score until a local minima is reached. In principle, this is the same as using the derivative of a function to determine where the local minimum/maximum is relative to the point of the derivative; if the derivative goes from negative at one point to positive at another point, a local minimum has to be between the two points. Once the optimization script returns gains that are between the bounds of the input domains, the gains have been optimized.

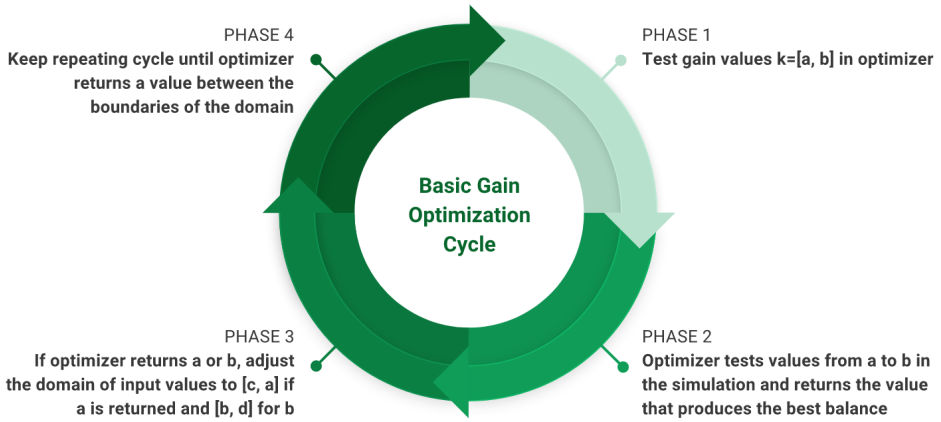


Figure 2.8: Flowchart of gain optimization process through determining local minima. This process can be applied to all three gain values simultaneously.

An obvious limitation to this optimization method is that it found local minima, not absolute minima. However, this method produced balance scores close to 0, the minimum possible value, see figure 2.9.

2.2.3 Observed Grid Search Gain Optimization Trends

Velocity (m/s)	k1 Best	k2 Best	k3 Best	Balance Score
1	31	172	-3.4	0.000001
0.5	61.2	383.7	-1.7	0.000001
0.4	76	455	-1.3	0.000001
0.3	102	591	-1	0.000000
0.25	123	768	-1	0.000042

Figure 2.9: Table of gains optimized to balance score local minima at sample velocities. Gains were optimized with an accumulated lean angle error of 1 degree per timestep (1/60th of a second). Note how the optimized gains appear directly related to velocity (when velocity is halved, k1 and k2 double while k3 is also halved).

While optimizing for the gains at each velocity given in Figure 2.9, some trends were observed from the relationship between the many input and output values that were used. The optimized

gains of $[k1 = 30.56, k2 = 164.17, k3 = -3.33]$ that were found for a velocity $v = 1[m/s]$ with the old iteration of the MATLAB simulation, that had a lower max steer rate of $\pi/3$. These gains are equal to the gains found for the same velocity with the most recent simulation iteration, implying that capping the max steer rate does not have a profound effect on the produced gains. This can be attributed to the lack of a need for the bicycle to modulate its steer angle quickly, because it never faces any sudden instabilities in the simulation. An increased max steer rate does, however, improve the bicycle's overall ability to balance in conditions when it has to steer quickly to balance.

A notable effect produced by lowering the velocity of the bicycle model and reoptimizing the gains is how all gain values increase consistently between each decrease in velocity. This is likely due to the need of a more aggressive balancing system to keep the bicycle upright when it is traveling slowly and therefore is more unstable. Additionally, as velocity decreased, balance score increased - the bike balanced worse. We could find any gains to balance the bicycle below 0.25m/s.

2.2.4 Creating a Continuous Gain Function

As the value of optimal gains change with forward velocity, we would like the bicycle to use different gains at different velocities. We would like to implement gain scheduling. TO create a continuous function of each gain, we used Lagrange interpolants to fit ideal gains found at discrete speeds using Bayesian Optimization.

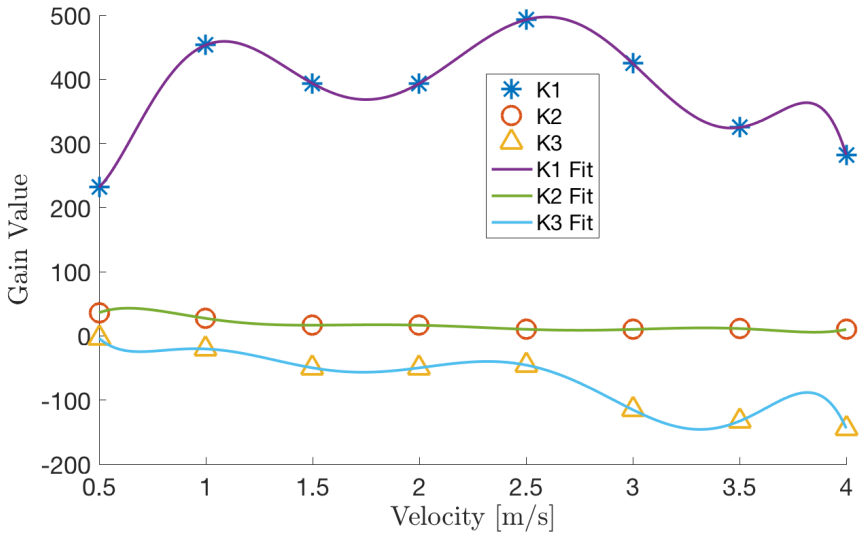


Figure 2.10: Lagrange interpolants were used to fit a function to the gains optimized through Bayesian Optimization

As seen by the polynomial function, there are oscillations near the end data points of Figure 2.10. This is a common phenomenon with this type of interpolation function but can be avoided by adding more data points so that the oscillations move away from the true end points of the bikes available velocity. Since the bikes max speed is about 3.5 m/s, the continuous functions are smooth between 0 and 3.5 if a data point at 4.0 is added. It is also important to

note that we treated each gain independent of each other and only a function of velocity itself. While appropriate for balance tests, there may be a best ratio of gains for the navigation of the bicycle that will serve to couple all three gains, but this is not being considered at the moment. This function will be implemented to have the gains updated to an optimal value for every speed during a test. From these performed simulations, the lean angle gain, k1, is dominant, the more the bike is trying to stay perfectly upright the better it will be balanced and still be able to travel in a straight path.

2.3 Applying the Simulation to Real Life

In order to test the accuracy of the MATLAB simulation in predicting the behavior of the bike, we tested the bike using various sets of optimized gains from the simulation. We intended not only to optimize the bike to drive at the slowest velocity possible, but to also prove the simulation's usefulness in optimizing bike's behavior. Previously the bike comfortably balanced at a velocity of 3.57 m/s. Our goal was to allow the bike to consistently balance at a velocity of 2 m/s and in a straight-line path.

2.3.1 Adjustments and Preparation

Rear and Front Motor Troubleshooting

We noticed the rear motor hall sensors sometimes provided inaccurate velocity data. When the wheel spins slowly, the hall sensors may not receive enough readings to accurately predict speed. The motor also cannot provide smooth motion to the rear wheel at speeds close to $0.5[m/s]$. This presents a problem in that we cannot create a continuous gain a function of velocity if the the velocity measurements from the bike are unreliable. We can work around this by setting a ceiling to the gain function so that gains above a velocity of 4 m/s are all the same. We can also filter out velocities that are outside of the 0.5 to 5 m/s range as these represent velocities that the bike cannot travel at. 0.5 m/s represents the lower limit that the wheel can rotate smoothly and we also know the maximum speed of the wheel to be about 3.5 m/s, thus, we can confidently filter out velocity measurements above our known maximum velocity.

Bike Velocity Discretization

Our simulation operates under the assumption that the bike is moving at a constant velocity throughout its test. Thus, to validate our simulation, the bike must move at a constant velocity. We adjusted the bike's velocity to be a step function, and not a linear function, of commanded speed from the RC remote

Below is a test of rear speed discretized in 0.5 m/s increments from 0 m/s to 4 m/s:

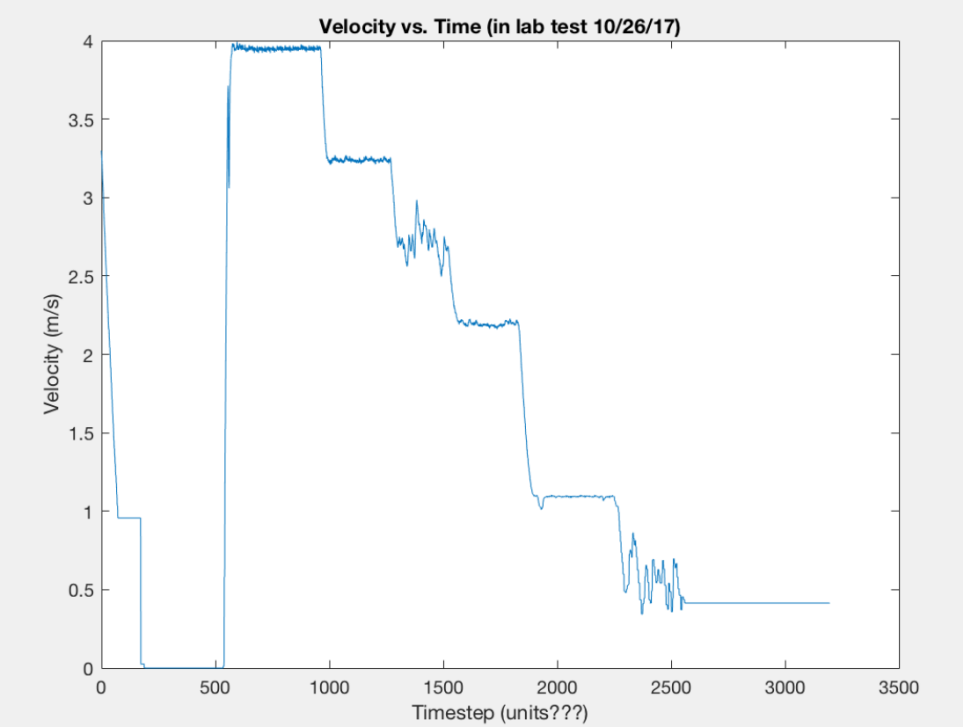


Figure 2.11: The bike's rear motor is programmed to move at discrete constant velocities in order to match the simulation.

According to figure 2.11, the bike's rear motor responds how we would expect except around 2.5 m/s and 0.5 m/s. It also seems to skip over the 1.5 m/s value. The distortions at 0.5 m/s occur because the motor cannot generate enough torque from such a low PWM to keep the rear wheel continuously moving. This can be seen in the wheel's "ticking" motion and should be considered when designing the bike to move at even slower velocities. We are not sure why the bike responds the way it does at 2.5 m/s. Fortunately, we have at least 4 different constant velocity values at which the bike can properly function, which is sufficient.

2.3.2 Results

Slowest Speed Achieved

We achieved a speed of 2 m/s which is a brisk walking pace quite. While balanced, the bike takes an arbitrary path. The link to this video can be found below:

Bike Balancing at 2 m/s

While this wasn't quite the 1 m/s we were hoping to achieve at the beginning of the semester, this is still significantly slower than the 3.5 m/s normal operating speed and is the slowest that the bike has been balanced to date.

During this test, we noticed that the bike was very difficult to control with the RC. This is to be expected because the MATLAB simulation tends to create a set of gains that are more sensitive to lean angle and lean angle rate, than they are to steer angle, as the velocity

of the bike decreases. While this trend bolsters the bike's ability to stay upright, it does not enforce the bike's ability to maintain a straight line path.

The yaw at 2 m/s shows a difference from the simulation result as displayed by figure 2.12 Where the simulation stays along 0 yaw, during the test the bike was turning and the yaw therefore does not match. This may be diminished by minimizing external differences like the flatness and smoothness of the pavement.

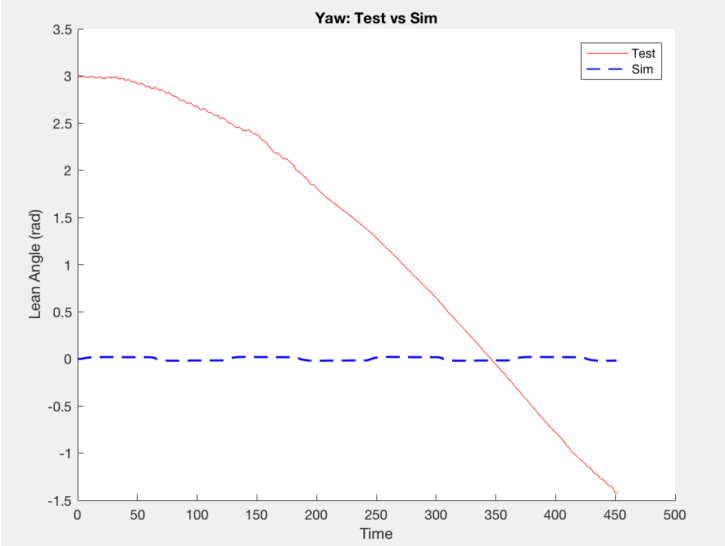


Figure 2.12: The yaw vs. time for this balance test at 2 m/s is graphed for both the physical test and the simulation of the bike with similar starting conditions.

Variable Gain Function Testing

The first variable gain tests used a step function to provide gains optimized at certain velocities over small rangers of the bikes actual velocity.

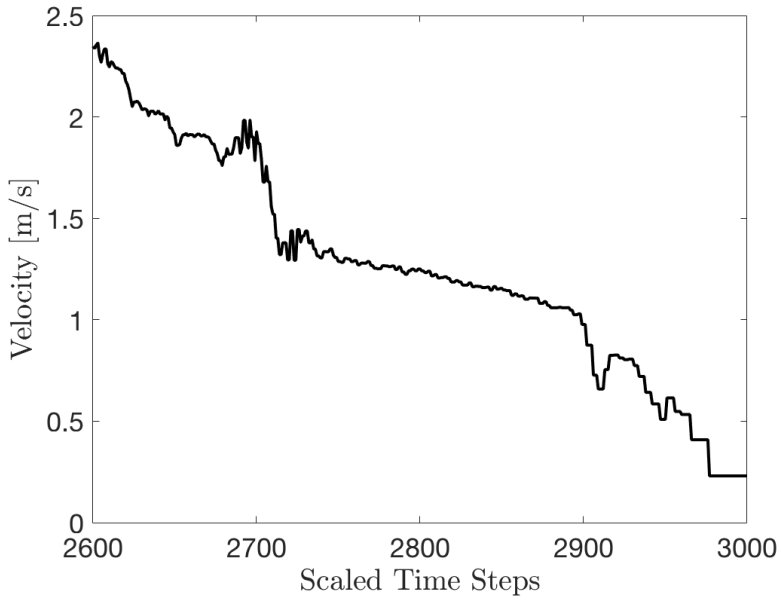


Figure 2.13: The velocity profile of one of the tests performed shows that the bike was able to maintain balance at a speed of about 1.25 m/s but falling once the bike dropped below 1 m/s.

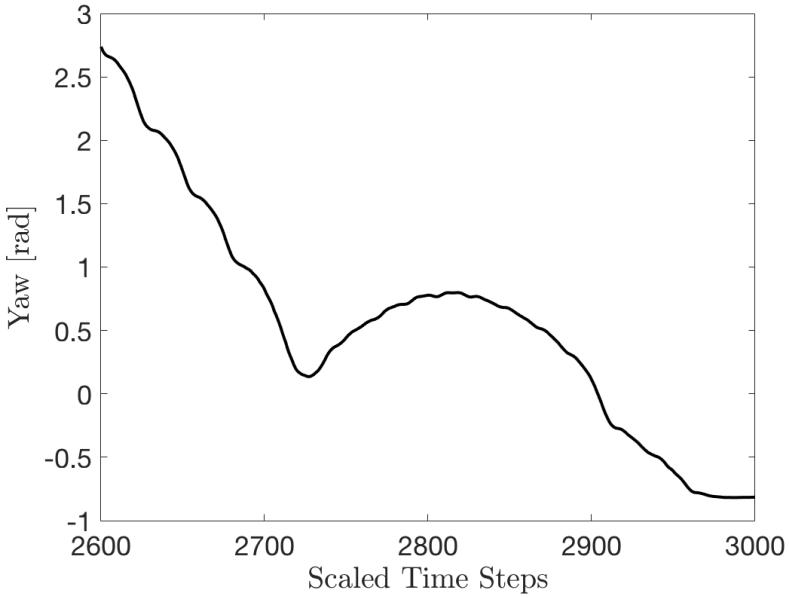


Figure 2.14: Matched in the time scale with the velocity figure 2.13, the yaw shows that the bike was turning while maintaining a balanced state in the 1.25 m/s range.

The link to the video that shows this test is provided below:

Bike Test with Variable Gains

2.3.3 Next Steps

Creating a set of gains that not only keep the bike upright but also allow it to be steered is difficult. To correct this, we placed floor on the value of the steer gain. The simulation will tend towards making the steer gain as small as possible in order to improve balance. To have control over the bike's direction, we need the steer gain to be nonnegligible.

3 Hardware

Figure 3.15 provides an overview of the bicycle's hardware.

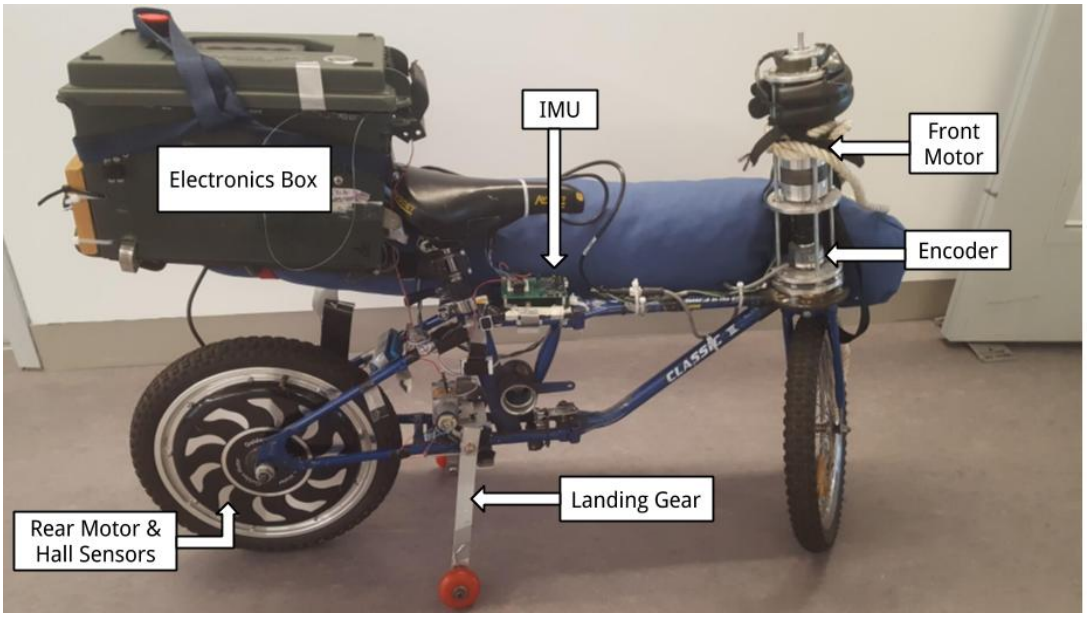


Figure 3.15: Hardware overview

The following components are used on the bicycle. Any components not labeled in figure 3.15 are in the Electronics Box. For a detail see Section 3 of the Autonomous Bicycle Team Spring 2017 report.

The following components are used on the bicycle:

actuator	**purpose**	**notes**
Front Motor	steer front wheel	a brushed DC motor
Rear Motor	maintain forward speed	brushless DC motor
Landing Gear	support bike when starting, stopping, or stationary	see Spring 2017 report

sensor	measured variable	notes
Encoder	front wheel angle (δ)	
Inertial Measurement Unit (IMU)	Lean Angle(ϕ), Lean Angle Rate($\dot{\phi}$)	Can also measure heading/yaw (ψ) and rate of change of heading ($\dot{\psi}$)
GPS	position (x, y), heading(ψ), speed(v)	2m position accuracy
Hall Sensors	Rear Wheel Rotations	Used to calculate speed (v)

computing	purpose	notes
Arduino Due	Run balance loop, read sensor inputs, output motor commands	
Printed Circuit Board (PCB)	Connect Arduino to sensors and actuators	
Raspberry Pi	Data logging, Navigation Control	Serial Communication with Arduino
RC Receiver	Receive signals from model aircraft remote control	

4 Software

ROS Overview

Communication between software modules on the bicycle are facilitated using Robot Operating System (ROS). ROS is a framework that allows us to write clear and robust code for communication between the different devices and sensors on the bike. Communication is facilitated by ROS nodes, which can publish and/or subscribe to topics published by other ROS nodes. Topics contain information of a specified data type, such as an array of integers or strings. For example, in the ROS Arduino Wrapper the "Bike" node publishes to the "bike_state" and "gps" topics, which have information from the sensors on the bike. These topics are utilized by nodes which handle position estimation and navigation, which publish to the "kalman_pub" and "nav_instr" topics respectively. Ultimately, the "Bike" node will subscribe to the "nav_instr" topic to inform the front wheel controller's steer angle during navigation.

4.1 Embedded Systems

4.1.1 ROS Arduino Wrapper

Jordan Stern and Robert Villaluz

The ROS_arduino_wrapper file holds code that is essential to the bike's balance and navigation. Given that it runs on the Arduino, the ROS wrapper collects and interprets data from sensors that are connected to the Arduino and uses them to inform the navigation algorithm and balance controller.

Sensors

The ROS_arduino_wrapper is used to collect and parse the data received from the sensors on the bike. These sensors include GPS, Hall sensors, IMU, and an Encoder. It is necessary to

parse data from these sensor so the Arduino can create abstractions of the sensor information in the form of ROS topics (see ROS Overview), which create topics that inform other parts of the bicycle.

Arduino ROS Topics

Two of the topics found in the ROS wrapper are gps and bike_state. gps stores data from the GPS and contains the bike's latitude, longitude, speed (m/s), and the age of the location data. bike_state contains data from the IMU, encoder and hall sensors. It stores the velocity and position of the front motor, velocity of the rear motor, battery voltage and the bike's orientation. Other software components, such as the bike's navigation algorithm, are able to access this data through the ROS topic.

Actuators

After parsing data from the sensors, the ROS wrapper receives data from the Raspberry Pi's nav_node. This informs the front wheel what the steering angle should be based on the results of executing the navigation algorithm.

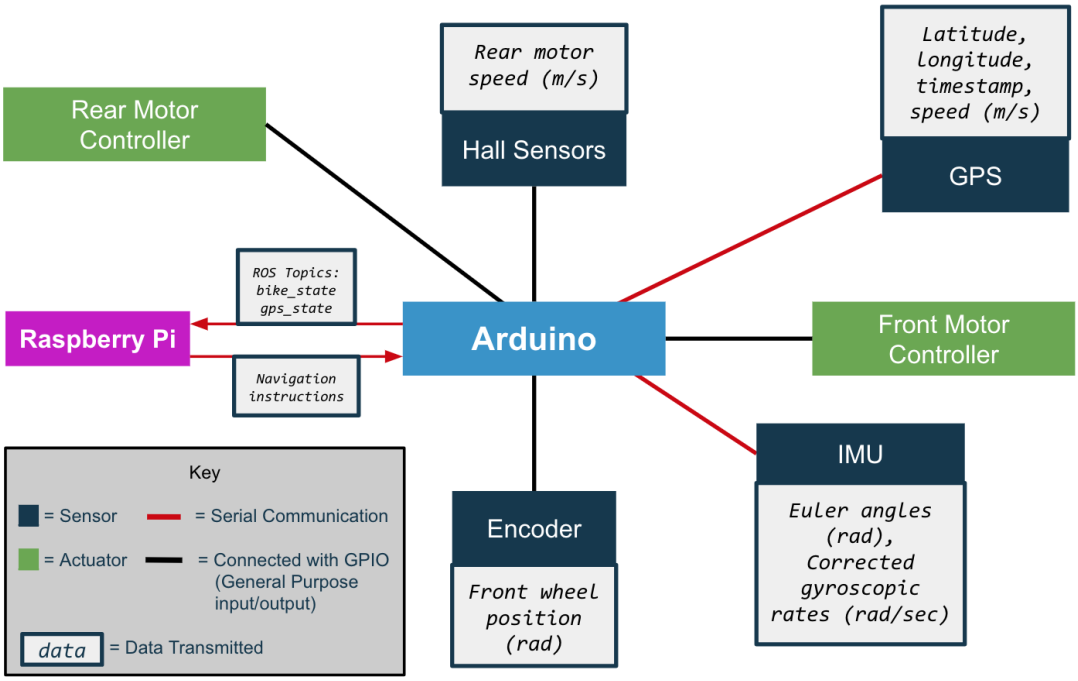


Figure 4.16: Overview of the bike's components that communicate with the ROS wrapper (which executes on the Arduino)

4.1.2 Serial Protocol

Robert Villaluz and Jordan Stern

Communication between the IMU and Arduino Due is facilitated over Serial. It is implemented in the ROS Arduino Wrapper (the file used to initiate and facilitate the Arduino's

communication with ROS and the bike's sensors).

New Serial Protocol

The new Serial protocol returns the bike's roll rate (radians/sec), roll angle (radians) and yaw angle (radians). It does this by parsing the raw data received from the IMU. The IMU is currently configured to return this data as ASCII text packets. However, the IMU is also capable of returning the data as byte packets. The raw data contains the bike's tared orientation as euler angles (which includes the roll and yaw angles) and the corrected gyroscopic rates (which includes the roll rate). Finally, it is intended that the Serial protocol selects the roll angle, yaw angle and roll rate values from the data and stores them in "bike_state". This data is then published as a ROS topic so that the Raspberry Pi can access it and provide navigation instructions.

Future Plans

The old SPI protocol functions correctly and utilizes binary commands to communicate with the IMU, as opposed to the ASCII commands used by the new Serial protocol. Although we designed the new Serial protocol with ASCII commands, as they simplify implementation, we intend to change the new protocol to use byte commands. Given that we can not rely on the IMU's documentation for ASCII commands, we will redesign the protocol with byte commands which, confirmed with the old SPI protocol, are known to be reliable.

4.1.3 Sensors

We have many sensors on the bike. We need to better understand the errors on each sensor and account for error when using this data. This data will allow for better simulations of the bike in Matlab and a better understanding of the system as a whole. Currently the simulations assign at error of 1 degree alternating with each time step. The IMU and steer angle are plus or minus one degree. Obviously real error is not that clean and is random. Therefore the simulated error is not accurate. The error is specified on the specification sheets for the hardware (IMU and front wheel encoder), but this does not account for different errors and interferences on the bike. For the most accurate results, we must manually test the bicycle and characterize its error. For the simulation, the key values to test are IMU lean angle, lean angle rate, front wheel encoder steer angle, and steer angle rate.

IMU

To understand IMU error, we must understand how the IMU works and where the error comes from. IMUs use magnetometers, accelerometers, and gyroscopes to determine position in 6DOF. These components are very sensitive to magnetic and electrical interference. Due to the density of electronics on the bicycle, this interference is very likely.

The team has worked with IMU error in the past, as described in the Fall 2016 Final Report. Electrical interference was tested for IMU data. With the front motor off there is no current through the wires that would cause electrical interference. They ran tests with the front motor on and off and found no noticeable difference, so the front motor can be turned on for all tests with no impact on IMU data.

Specifications There is data regarding the error from the specifications of the IMU. The specifications state 1 degree error, but this value needs to be verified with real tests on our system.

GPS
Joshua Even

Our GPS is a ublox NEO-6M, which has a published accuracy of $2.5m$. Given that our estimated path width is about $2m$, this is not accurate enough, and was part of our motivation for sensor fusion. The GPS is connected to our Arduino Due via a Serial Port, and is capable of updating at 10 Hz and transmitting data over Serial at a maximum rate of 115200 baud. The GPS can be configured by sending byte encodings to the GPS over serial, indicating what update and baud rate you would like to configure the GPS to send data. The GPS streams NMEA strings over serial that include information about position, heading, speed, and time stamp indicating time between updates. These strings are parsed using the open source tinyGPS module, which makes the data easily accessible. The time stamps published by the GPS were used to confirm that the rate we were receiving data was in fact 10 Hz.

4.1.4 Tests

GPS Accuracy

Aside from using the GPS for position, we also experimented with the accuracy of the GPS's velocity and heading readings for when we test on the buck and do not have access to the hall sensors for velocity. By using the IMU as a baseline we were able to conclude that although the GPS has lower granularity due to a slower update rate than the IMU, it's heading data is still accurate as depicted in Figure 4.

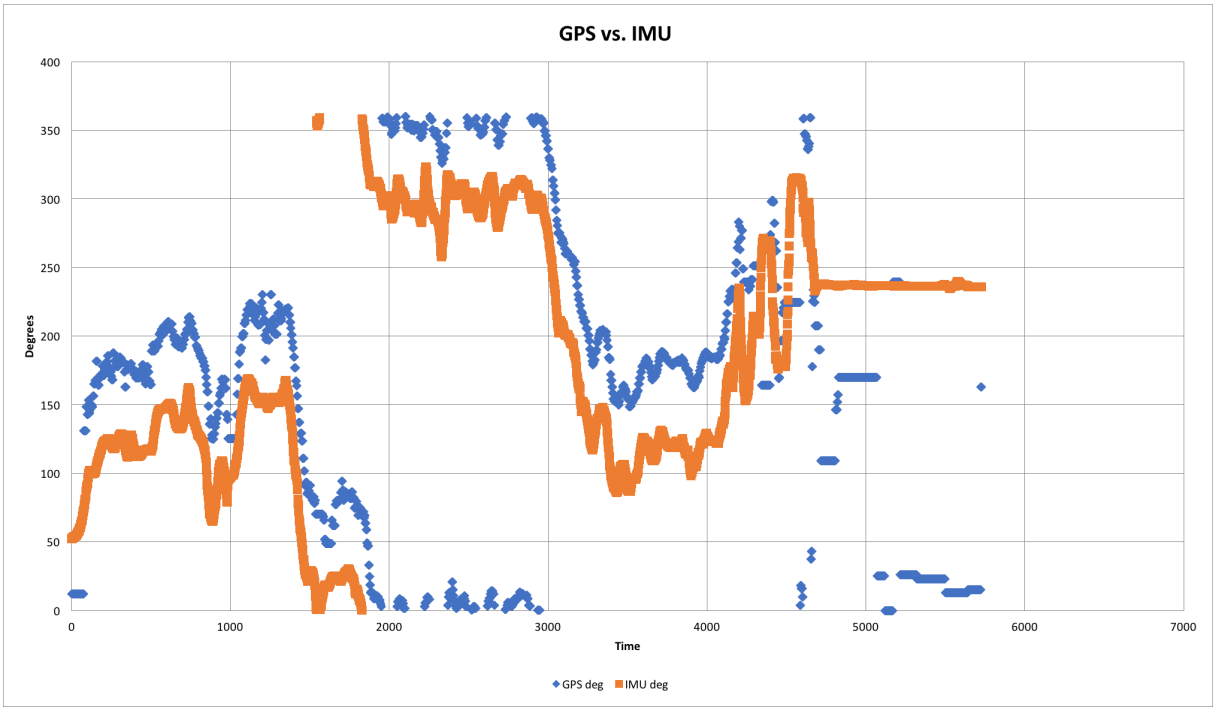


Figure 4.17: Note: There is a vertical offset between the GPS and IMU heading measurements

We also ran tests to confirm that GPS velocity was accurate for when we run tests using our buck, because unlike the bike, the buck does not have a hall sensor for measuring velocity. We walked at a constant pace for a set amount of time, and graphed vectors that indicated our speed and direction. After analyzing the results qualitatively and quantitatively we concluded that the GPS velocity data is also accurate.

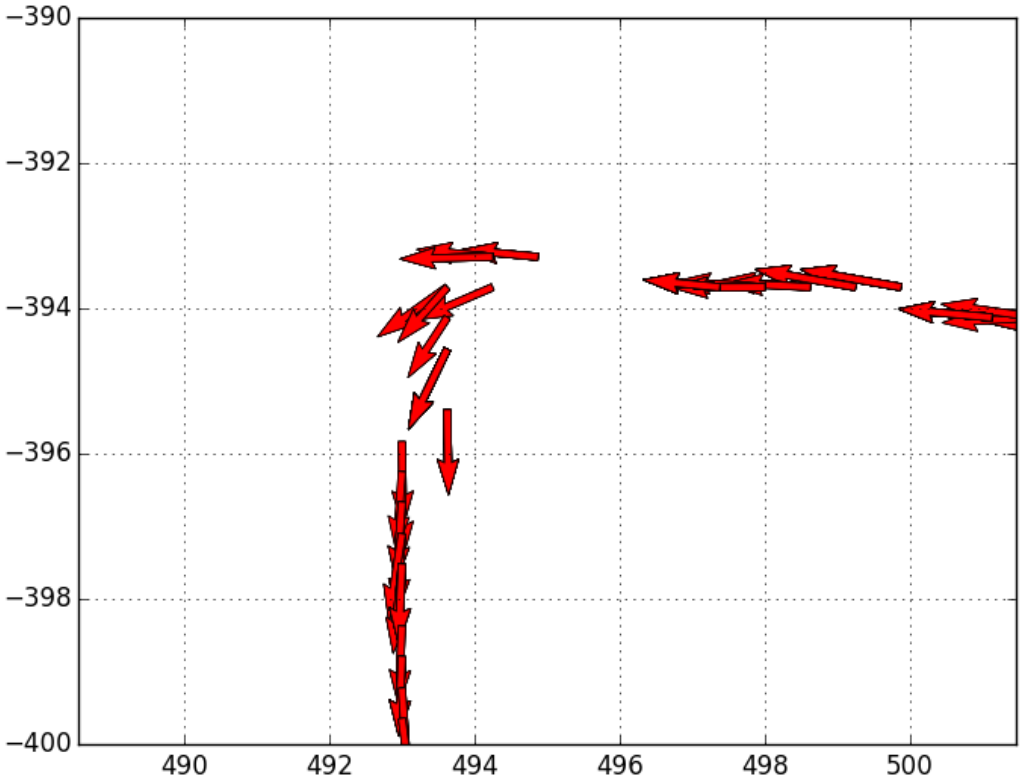
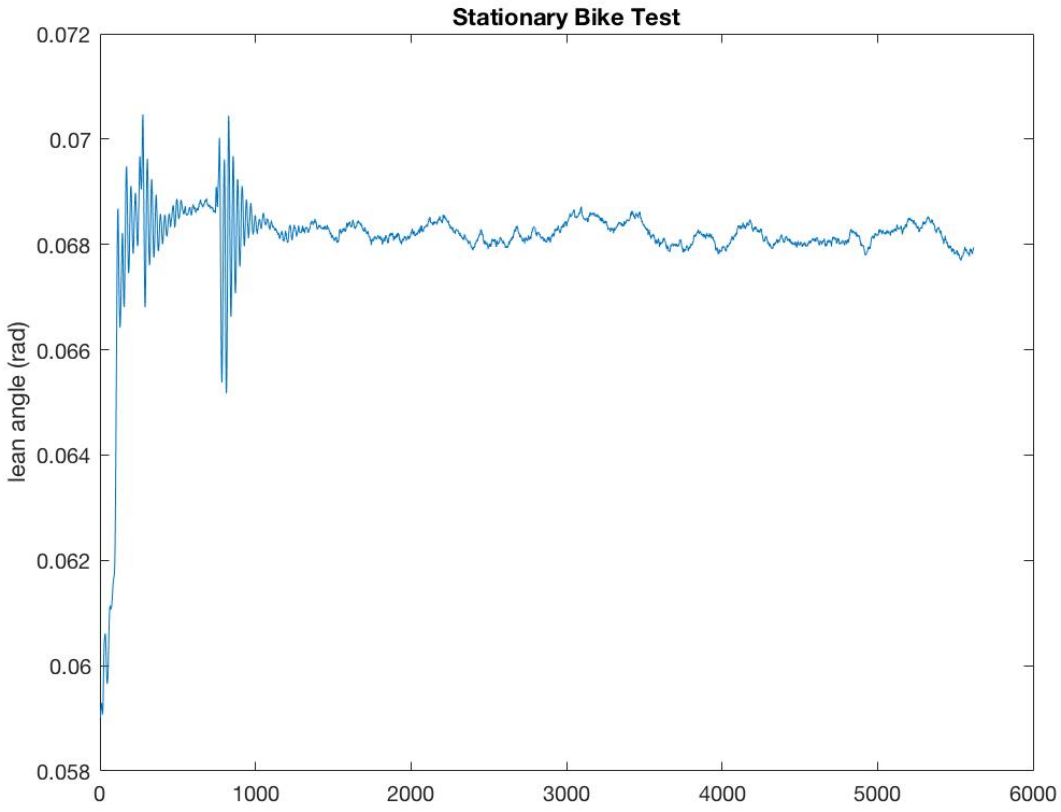


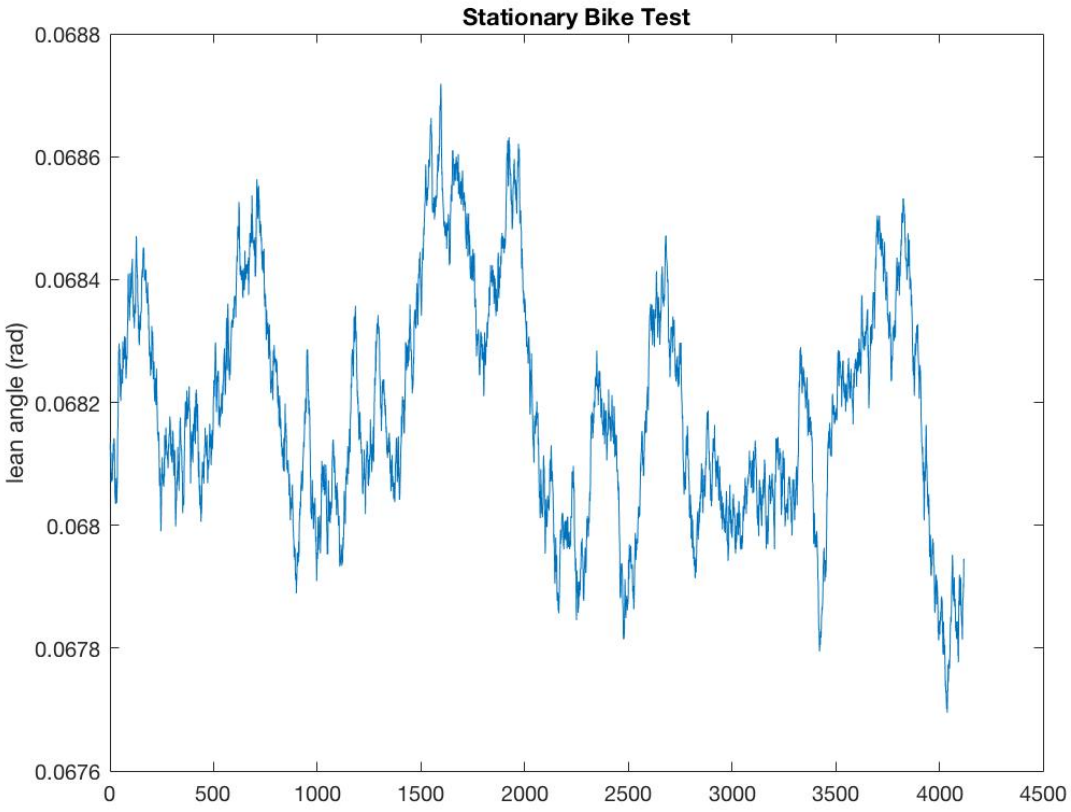
Figure 4.18: Velocity vectors while walking at a constant pace

Stationary Lean Angle Bike

To test error, the bike is left stationary with the front wheel straight, and the bike with zero lean at 90° with the ground. Everything is kept constant so we observe deviations from 0 in the sensor data. Each test was conducted for 3 minutes as to get a large amount of data points.



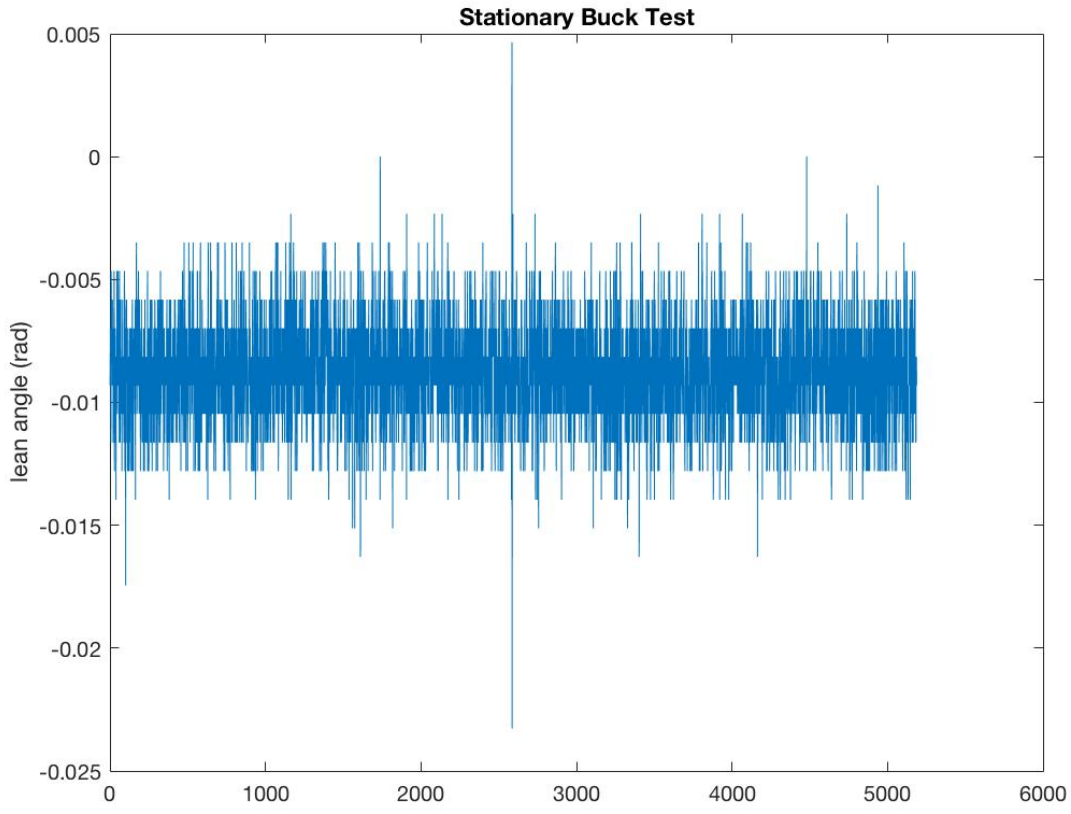
In this first test, the standard deviation was 0.0629°. However, there is noticeable jitter in the beginning of the test which can be removed as outliers.



Removing those points showed a standard deviation of 0.0109°. Many prior tests were conducted in the same way and consistent results. This standard deviation is very low, especially compared to the 1° error stated on the specifications and assumed in simulation. There is also no observable drift over time. This is a fairly long test, so a drift should be clearly seen if it is significant.

Stationary Test Buck

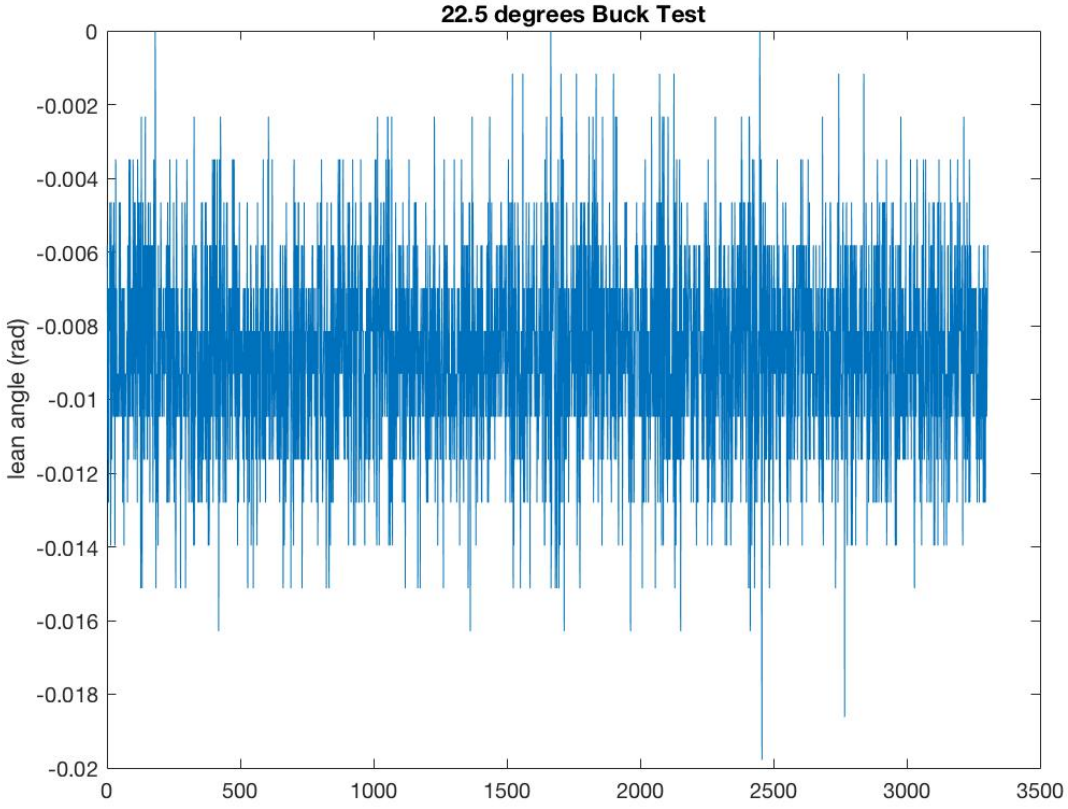
Many of the tests are easier performed on the buck. However, it is necessary to check whether the buck has significantly different error. This should not be the case as it was determined in prior semesters that the bike did not contribute significant interference error.



The standard deviation of this test was 0.1217°. This is slightly higher than the bike test, but can be attributed to a lower frequency of data points on the buck and outliers.

Gains Test

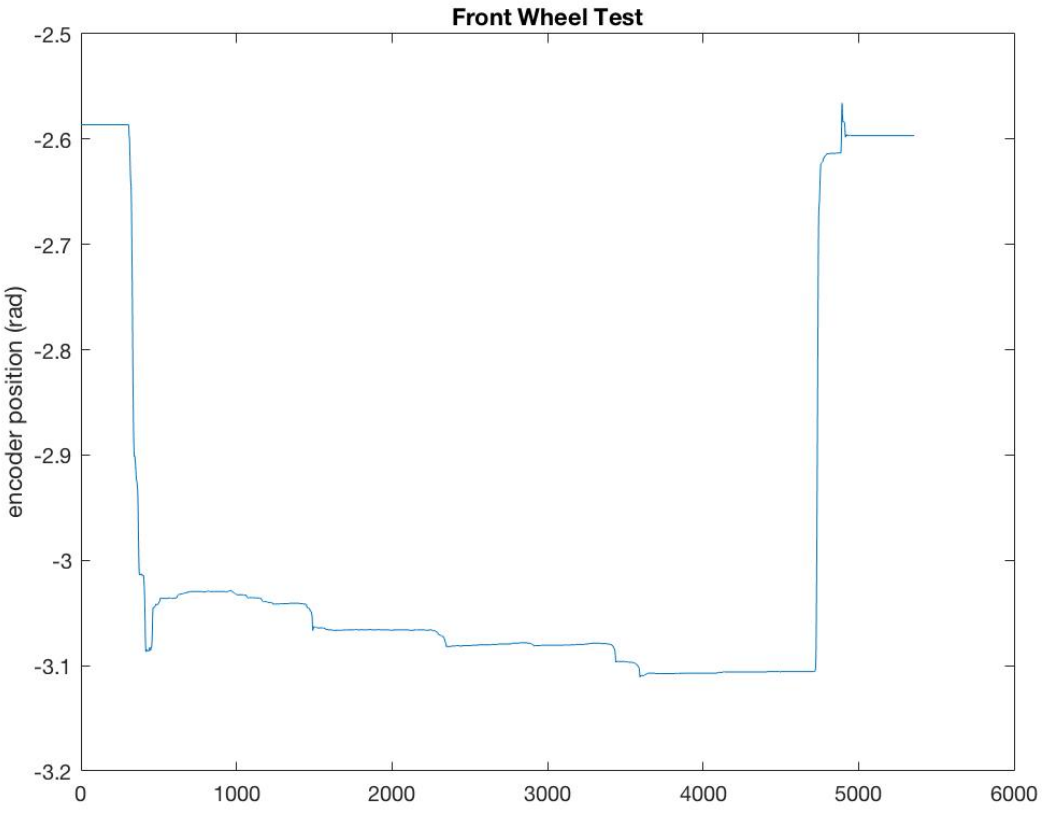
In addition to stationary 0 angle tests, we must see if the sensors behave differently at different angles. We tested to see if the sensor error is a function of the angle, i.e. is there a gain on the sensor error. To test this the buck was held at 22.5 °



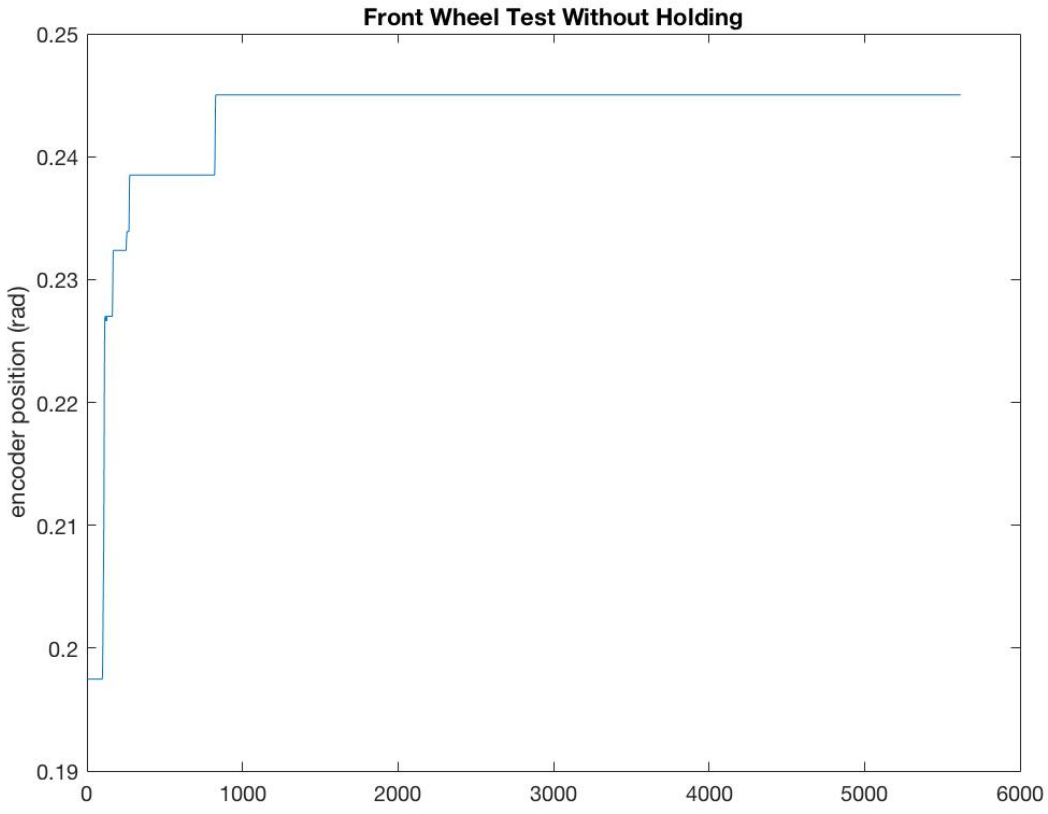
The standard deviation was 0.1471 °. Which is nearly identical to the 0 angle error. From this and other tests, it is clear that the IMU error does not have a significant gain.

Steer Angle

To test the front wheel encoder, a test was run with the bike stationary and the front wheel held at a straight angle.



In analysis, the first and last 500 points were removed, as the wheel was not held properly at the beginning and end of tests. Taking this into account, the standard deviation is 1.4747 °. However, looking at the the graph it appears that a lot of this error can be attributed to the wheel moving very slightly at discrete times. This is because the wheel was held in place by hand which is not the best way to approach this. Looking at the data from a stationary test with the wheel not held at all, it appears there is no error from the encoder when it is free standing.



Removing the first 1000 points, where the wheel moves at start up, the standard deviation is 2.0358e-13 °. This is completely negligible error.

4.1.5 Matlab simulations

The error is currently simulated as follows:

```
%SENSOR ERROR IN THE LEAN ANGLE
if rem(ceil(time),2) == 1
    phi = phi + pi/180;
elseif rem(ceil(time),2) == 0
    phi = phi - pi/180;
end
```

Instead of pi/180, we need to generate a truly randomized normal error based on the tests. The steer angle error is simulated identically.
The current simulation is meant to create a worse possible error. However, we would want a balance between worse case and realistically accurate error. Therefore in our tests we want to simulate the worse possible random error. This can be done by scaling up the standard deviation.

From our tests, we can determine a standard deviation for the error and use this error

to generate a randomized error with a normal distribution. We can keep the positive negative oscillation by taking the absolute value of this error.

```
%SENSOR ERROR IN THE LEAN ANGLE
sigma = // standard deviation of lean angle from stationary test
mu = 0;
randError = abs(normrnd(mu,sigma))
if rem(ceil(time),2) == 1
    phi = phi + randError;
elseif rem(ceil(time),2) == 0
    phi = phi - randError;
end
```

However, this is a very realistic simulation. We want the simulation to be worst case to some degree. This can be accomplished by replacing mu with 1 or a larger positive value, or by multiplying sigma by a constant factor. Either of these methods would artificially increase the magnitude of the random error.

4.2 Sensor Fusion

Joshua Even

4.2.1 Introduction

Prior navigation tests failed as a result of inaccurate position information. In order to augment the accuracy and rate that we receive information about the bike's position we implemented position state estimation by fusing readings from our GPS, IMU, and Hall sensors using a Kalman-like model. This model is both efficient and lightweight, as it only needs our current observations and previous state in order to estimate our current state. This is important given the computational limitations of the Raspberry Pi. There are three components to our state estimation: the model, the prediction phase, and the update phase. In these steps we fuse GPS position readings and results of odometry to better estimate the bike's position.

4.2.2 Model

$$\begin{aligned} x_k &= Ax_{k-1} \\ z_k &= Cx_k + v_k \end{aligned} \tag{14}$$

Where x_k is the current state, x_{k-1} is the previous state , and A is the matrix we use for estimating x-position, y-position, and velocity with respect to the x and y axes, which we will call $\dot{x}$ and $\dot{y}$ respectively.

z_k is the current observation of the system, and v_k is the current noise measurement. Note that C, which is generally used to apply weights to the state variables, is simply the identity matrix in our model.

The model is defined below:

$$x_k = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\dot{x} = v * cos(\theta)$$

$$\dot{y} = v * sin(\theta)$$

where θ is simply heading from the IMU and v is hall sensor velocity. Steps are being taken to estimate $\dot{\theta} = \frac{v*tan(\delta)}{l}$ where δ is the front wheel angle and l is the distance between the front and back wheel. $\dot{\theta}$ will be used to estimate heading in the same way we do odometry using Euler integration. However, this is not incorporated into our model yet.

4.2.3 Predict

$$P_k = AP_{k-1}A^T + Q \tag{15}$$

The prediction phase is fairly simple. Along with estimating x_k we also calculate P_k a 4x4 matrix which indicates our prediction error. P_k which Q, a vector which indicates covariance of our process noise. For our model, if $Q = I$, where I is the identity matrix, we observe that the filter perfectly fits the GPS position data. When Q is high, meaning Q values close to I, it is an indication that there is a lot of process noise so we should use our observed values exclusively. Lower values of Q, meaning Q values close to 0^*I, indicate that there is not a lot of process noise, and we can more heavily rely on our state estimates.

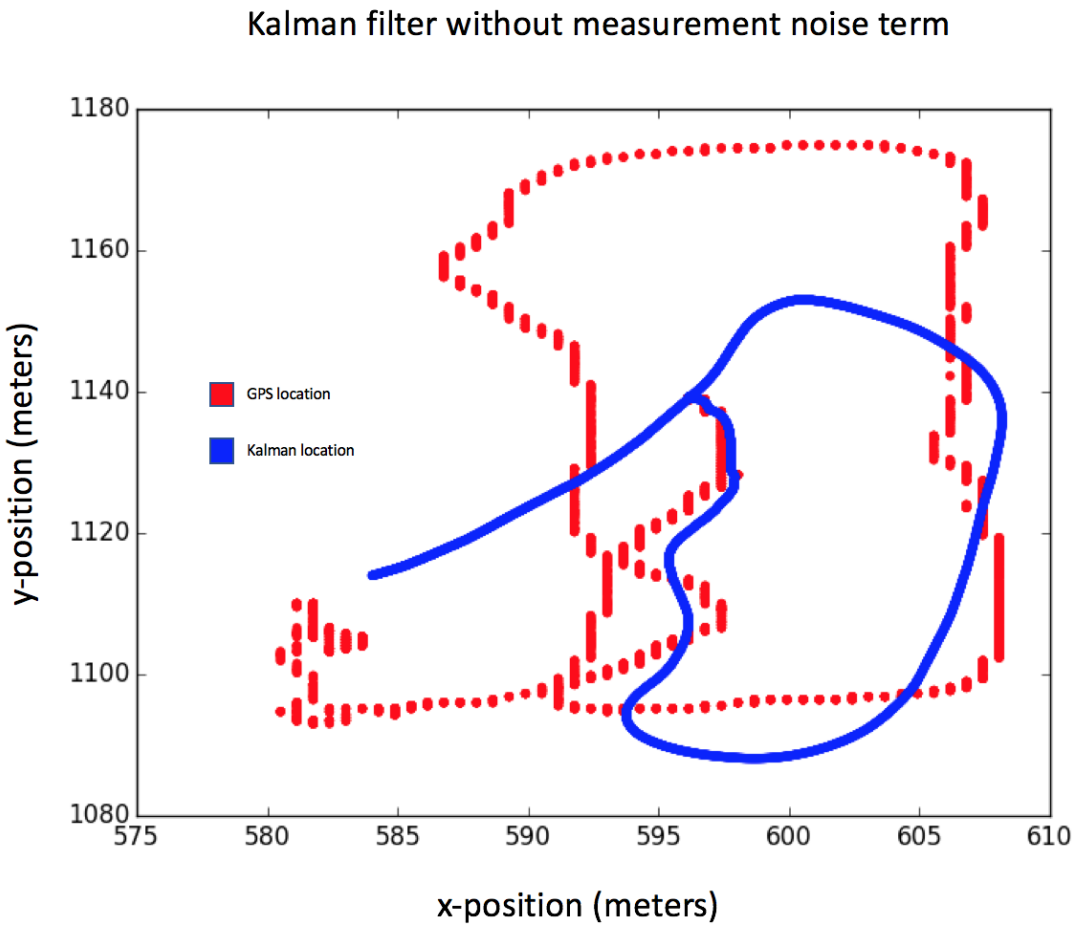


Figure 4.19: Position estimation assuming no measurement noise(meters)

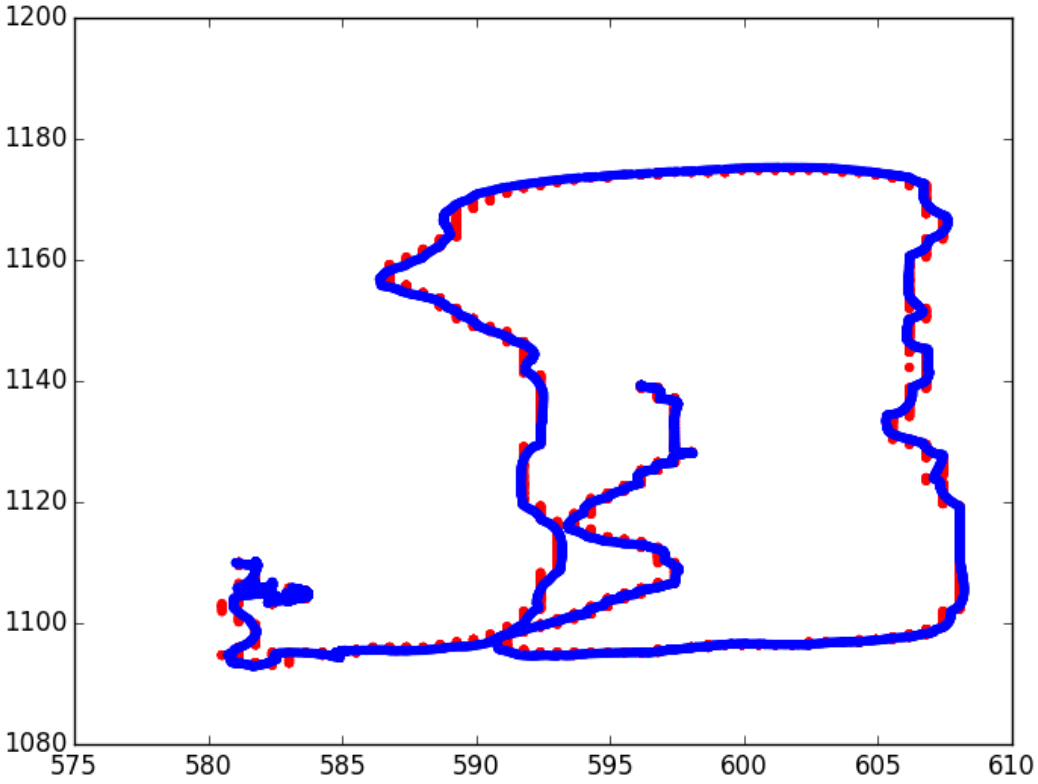


Figure 4.20: Position estimation assuming low measurement noise (meters)

4.2.4 Update

$$
\begin{aligned}
G_k &= P_k C^T (C P_k C^T + R)^{-1} \\
\hat{x_k} &\leftarrow \hat{x_k} + G_k(z_k - C\hat{x_k}) \\
P_k &\leftarrow (I - G_k C) P_k
\end{aligned}
\tag{16}
$$

During the update phase we calculate G_k which is a vector that indicates the "gain" value. A zero gain means no update of the prediction error, or $P_k = P_{k-1}$ while a gain of one means we have no prediction error. Generally, this value will fall somewhere in that range. R is a constant matrix, and indicates the noise we expect from each sensor. In this case, we use the published accuracy of our sensors for our R value, but these values will be updated to reflect new information we have found regarding our sensors. We then update x_k by applying a certain weight to our observations and our state estimation in order to achieve as accurate of a model as possible.

For more information on our overall model for sensor fusion consult this link from former Cornell professor and current Stanford professor Ashutosh Saxena https://www.cs.cornell.edu/courses/cs4758/2012sp/materials/MI63slides.pdf or the following link which explains the steps in developing a sensor fusion model https://home.wlu.edu/~levys/kalman_tutorial/.

4.2.5 Tuning

The filter was tuned by running tests where we knew our ground truth (position) and changing values in our model related to noise in order to fit the ground truth as closely as possible. Specifically, we would walk along a line on the track at Schoelkopf while walking at a constant speed in order to gather data on our GPS position, heading, and velocity. From here tuning the filter was easy because once we collected data from our walk we could retroactively test our sensor fusion since the model we used only needs to know about current observations and previous state. At this juncture our sensor fusion has given us position accuracy between $0.5m - 1.5m$ which is an improvement on the accuracy of the GPS on its own.

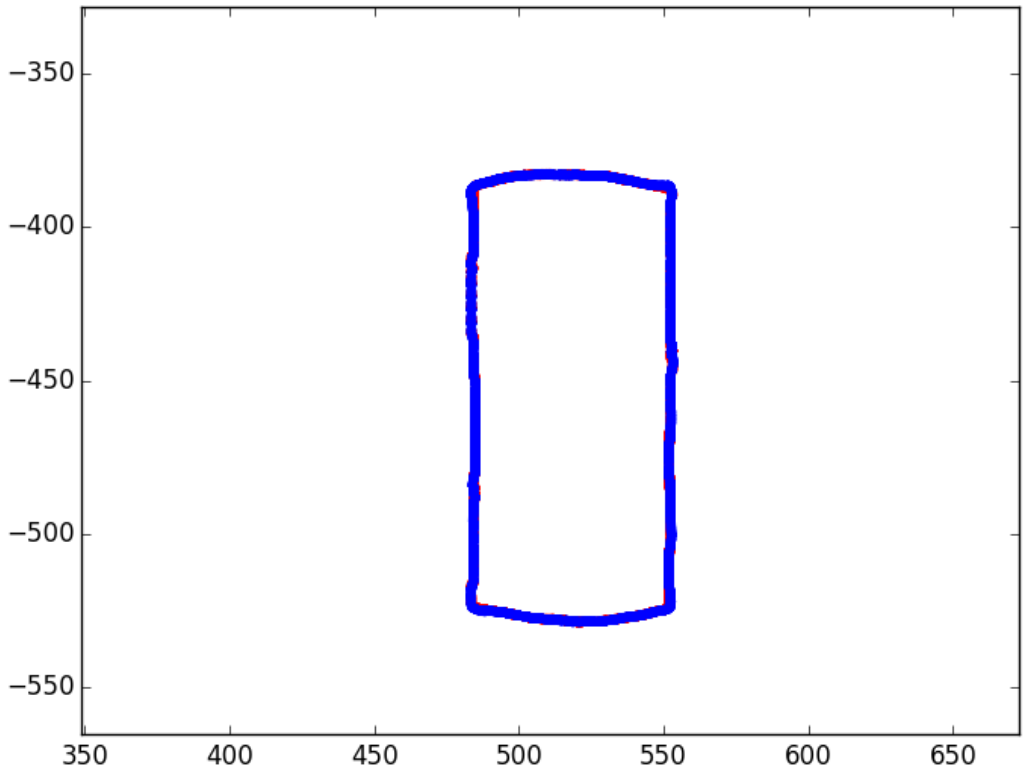


Figure 4.21: Position estimation results from a walk around the track at Schoelkopf field (meters)

4.3 Navigation

The navigation sub-team focuses its efforts on optimizing our path-tracking algorithm in simulation and applying the algorithm using real-world data from various sensors found on the bike. Our algorithm does not create the desired path itself; instead, it attempts to make the bike follow a predetermined path as closely as possible. After researching different types of path-tracking algorithms commonly used by autonomous vehicles, we implemented an algorithm that outputs a desired front wheel angle based on the sum of two proportional and two derivative

terms. After testing our algorithm vigorously in simulation, we have moved on to testing with real data from sensors on the buck and the bike. The navigation team has made a lot of progress and had some promising results, but there is still more work to be done before the bike can navigate completely on its own.

4.3.1 Algorithm

The starting point for the algorithm's development was the "proportional controller" from the spring 2017 semester, which, given cross-track error E_{ct}, heading error E_θ, and chosen gains K_{ct} and K_θ for each error measurement, produces a desired steering angle δ using this formula:

$$\delta = K_{ct}E_{ct} + K_\theta E_\theta$$

Cross-track error, E_{ct}, is the distance between the bicycle and the path. Heading error, E_θ is the angle between the heading of the bicycle and the desired direction of the path.This algorithm works well under two assumptions: the bike is close to the path, and the path is a straight line. We shouldn't assume either for a path-tracking algorithm, so we set out to improve the algorithm.

The current algorithm is a "PPDD" controller, meaning we use two proportional and two derivative terms. Given E_{ct} (cross-track error), E_θ (angle error), and their time derivatives, the algorithm outputs a desired steer angle.

$$\delta = K_1E_{ct} + K_2E_\theta + K_3\frac{d}{dt}[E_{ct}] + K_4\frac{d}{dt}[E_\theta]$$

K_1, K_2, K_3, and K_4 are gains. By adding derivative terms, this algorithm aims to avoid overshooting the path or oscillating about the path.

4.3.2 Simulation

In order to make the algorithm as robust as possible, we have run simulations on a variety of initial bike conditions and desired paths. These initial conditions include x,y position, lean angle (ϕ), heading (ψ), front wheel angle (δ), and lean rate ($\dot{\phi}$). Velocity was kept at a constant 3.57 m/s, but in the future hopefully we will be able to alter velocity as well. The algorithm was able to handle an array of starting conditions in simulation, excluding lean angles above $\frac{\pi}{4}$.

Our simulation works by first storing the initial bike object and target path in an object called a MapModel. The navigation algorithm uses the current bike state in the MapModel object to output a desired front wheel angle. Then, we use the bicycle dynamics equation to update the bike's state, and repeat using the new bike state. We plot the path and the simulated bike's movement using a PyQtGraph-based visualizer.

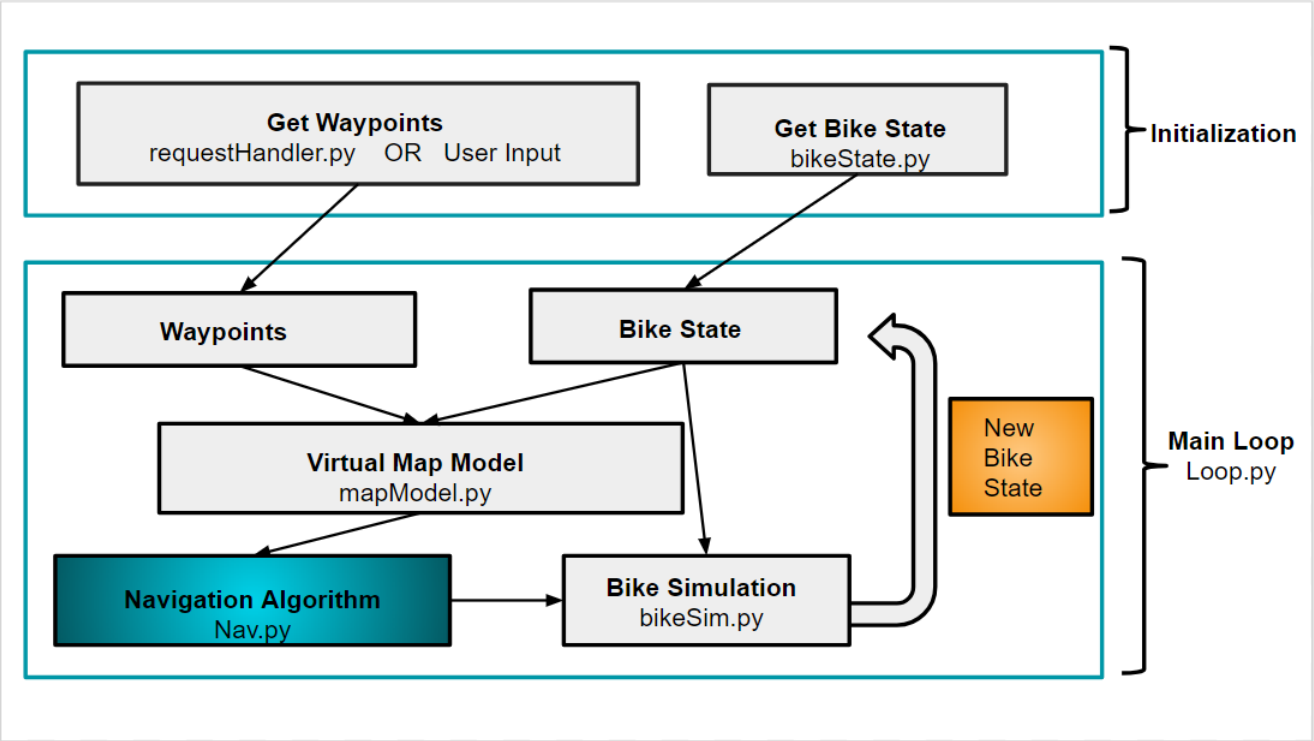


Figure 4.22: Current Class Relations Used in Simulation

4.3.3 Position Estimation

Navigation makes use of position estimation for two reasons. First, the algorithm works best with accurate sensor data, and position estimation is more reliable than GPS data alone (See Sensor Fusion). Second, the GPS updates position at a rate of around 10 Hz, which is much slower than the 63 Hz at which the main Arduino loop runs. This delay in GPS data has a negative effect on the navigation algorithm but can be fixed by using position estimation instead of raw GPS data.

To see the effect of delayed GPS updates, we implemented an artificial delay of position data readings in our simulation. We noticed an increase in oscillations due to the fact that the algorithm's desired front wheel angle output was not getting updated quickly enough.

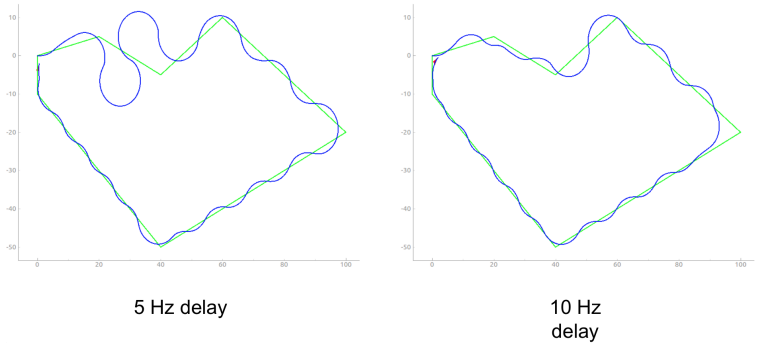


Figure 4.23: Effects of Simulated GPS Delay

Fortunately, our position estimation eliminates this delay by essentially "filling in the gaps" in raw GPS data. Without position estimation, the GPS would publish the same position data for about 6 loops of the Arduino code, and then there would be a significant gap in position at the next loop. With estimation, however, position data is spread fairly evenly out over these 6 loops.

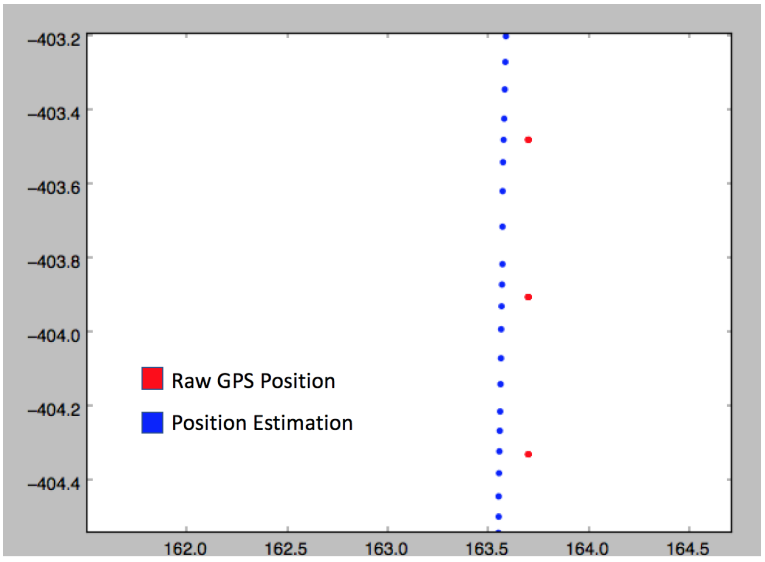


Figure 4.24: Smoothing Effects of State Estimation. Each red dot represents about 6 loops of the Arduino code. Each blue dot represents a single loop of the Arduino code.

4.3.4 ROS Communication

The navigation algorithm can be tested using various sensors either on the buck or the bike. Since the algorithm outputs a front wheel angle, we need to combine all of the data from our different sensors, input those to the algorithm, and send the desired steer angle output to the balance controller. This communication between sensors occurs on the Arduino and Raspberry Pi using ROS.

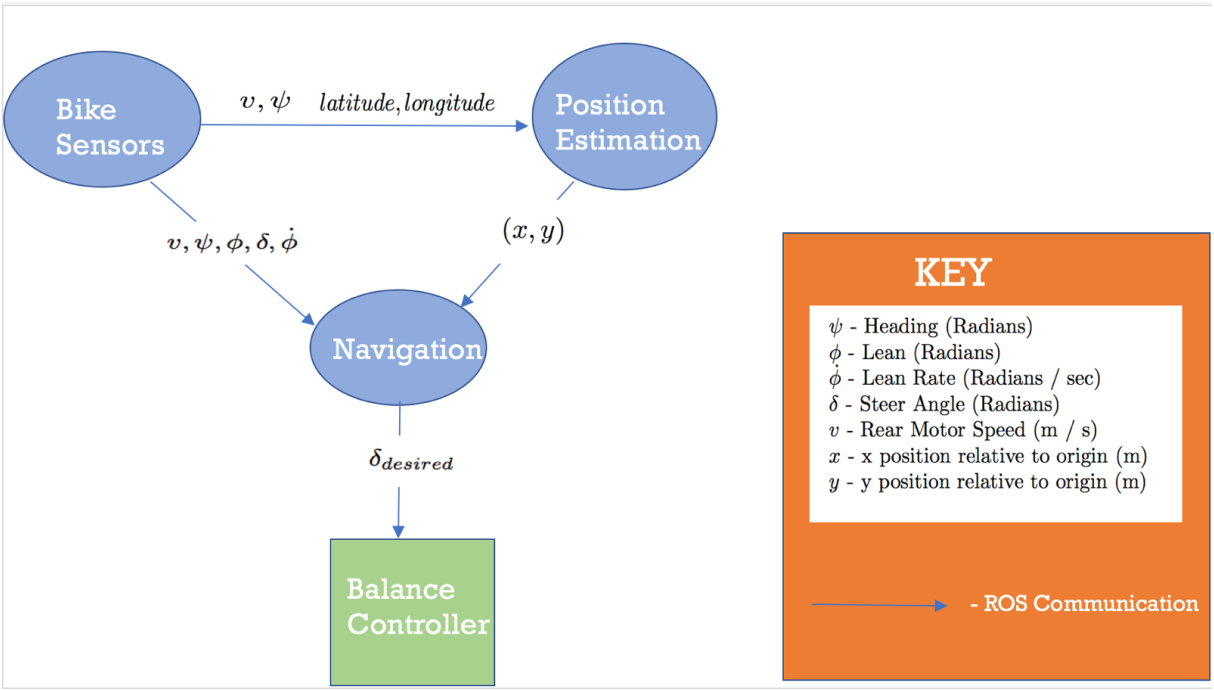


Figure 4.25: ROS Communication Diagram for Bike Navigation

The Bike topic on the Arduino publishes data from the IMU, encoder, and hall sensors. The Position Estimation topic, published from the Pi, subscribes to the GPS and Bike topics and runs the position estimation model to publish an estimated x-y position. Navigation, also on the Pi, subscribes to the Bike and Position Estimation topics and runs the navigation algorithm to publish a desired front wheel angle for the balance controller.

4.3.5 Buck Testing

To perform preliminary tests, we use a buck (testing rig - see Appendix C) due to its small small size and separation from potential motor and/or balance issues on the bike itself. When testing on the buck, we cannot use the encoder or hall sensors to obtain front wheel or bike velocity data, respectively. For bike velocity, we use data from the GPS instead. We set the current front wheel angle to 0, since it has no effect on the desired front wheel angle output of the algorithm. Occasionally when we experience issues with the IMU, we use heading from the GPS as well.

When we try walking along a path with the buck and reading the output of the nav algorithm, we obtain some interesting results. When we walk the path that we plot, we expect the navigation algorithm to output a desired steer angle close to 0 (if the bike is already on the path, it shouldn't need to turn). However, the navigation algorithm seems to always output a maximum steer angle of $\frac{\pi}{6}$ no matter where we are in relation to the path.

While walking along some particular straight line paths, however, we notice that at a certain point we start to see angles other than the maximum angle. When walking along wherever it output a desired steer angle of 0, we notice that this path seems to be a straight line that similar to the path we had originally plotted but rotated by some amount.

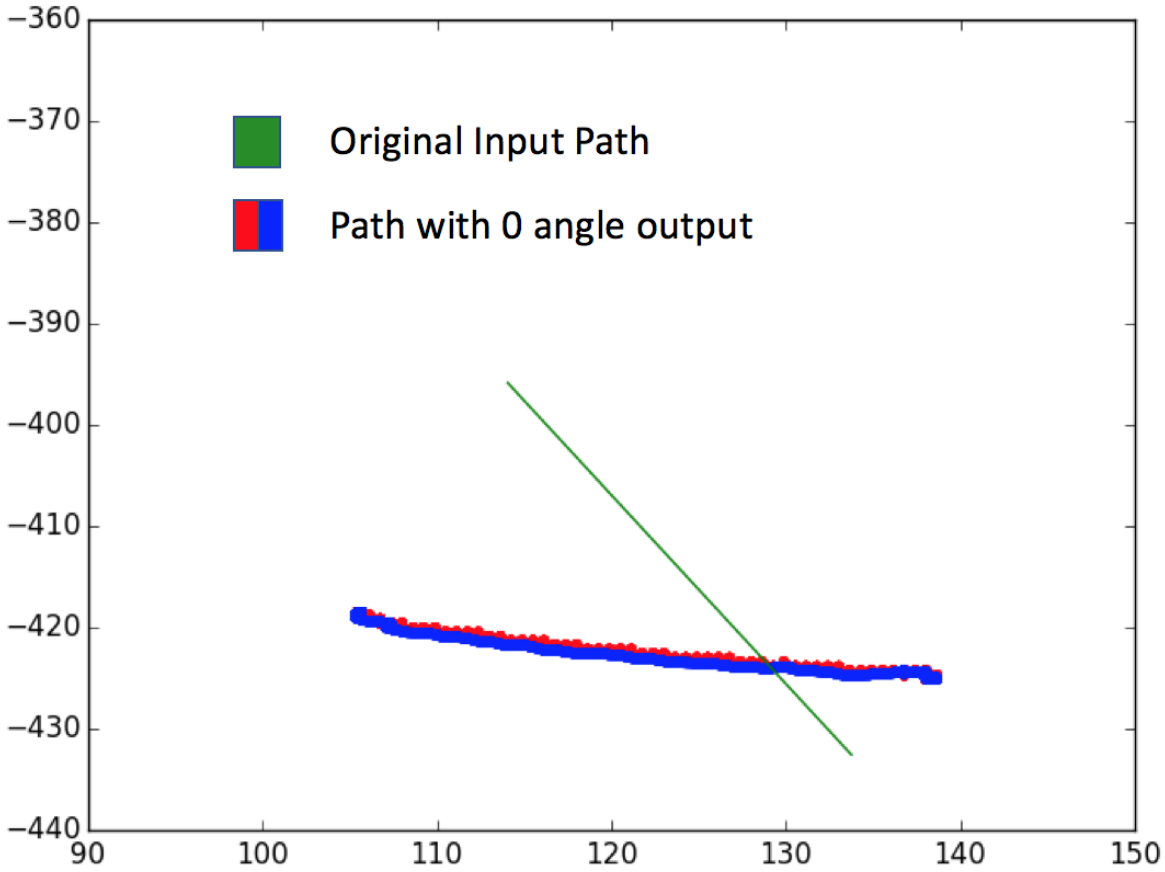


Figure 4.26: Plot of input path vs. path we walk showing 0 angle output.

We hypothesized that the path we input was somehow getting rotated by a fixed amount, and that we might be able to fix this in the short term by rotating our initial input path by that same amount. However, when we try to duplicate this test on other paths, and sometimes even on the same paths, we receive inconsistent results. It is difficult to find any place where the desired steer angle is different from max steer angle, and when we do, we see no clear pattern between tests.

4.3.6 Bike Testing

The navigation team has not done much testing on the actual bike. We have collected some data from various bike tests to look at the accuracy of the GPS and position estimation model. We do not feel that we are ready to start navigation tests on the bike, since our results from buck testing are not yet what we expect. We have, however, set up the ROS nodes needed to do

navigation tests in the future by sending the output of the algorithm to the balance controller.

4.3.7 Future Plans

Plans for the immediate future include debugging the issues we experienced while testing on the buck. One possibility is that algorithm fails due to inaccurate sensor data, so we plan to examine this possibility more closely in the future. One way to achieve this could be to incorporate real-time plotting so that we can examine our sensor data more easily while testing. Since the algorithm seems to work well in simulation, we also plan to compare more closely the outputs obtained while running the simulation and the outputs obtained in real-world tests to see where exactly the discrepancy occurs. Once we see better results from buck testing, we will start running tests on the bike.

There are a few different ideas for the long term future of the navigation team. As of now, the algorithm works in simulation for constant speeds around 3.57 m/s but not for speeds that are much higher or lower, or that do not remain constant. We plan to improve the algorithm by making it work for varying velocities. In addition, we plan on adding obstacle detection and avoidance into our algorithm once the vision subteam has made some more progress.

4.4 Computer Vision

Overview

The overarching goal of the Autonomous Bicycle team is to create a self-balancing, self-navigating bicycle. For a bicycle to be self-navigating, it must be able to detect and avoid any obstacles, stationary or moving, that would cause the bicycle to fall over. This is the ultimate goal of the Vision Subteam.

Our goal for this semester was to create a proof-of-concept vision system on a buck. Our goal for this vision system was for it to detect large objects in the buck's path when we wheeled it indoors.

4.4.1 Obstacle Detection

The Vision Subteam's primary goal for the bicycle is obstacle avoidance. In other words, the bicycle should avoid crashing into stationary obstacles (trees, buildings, signs, etc.) and moving obstacles (people, vehicles, animals, etc.) that would cause the bicycle to fall over. For the bicycle to accomplish obstacle avoidance, however, it must first be able to accurately detect obstacles in its path. Therefore, stationary obstacle detection became our focus for this semester.

We decided to use Robot Operating System (ROS) to create our vision system. There were two reasons for this. First, ROS has a wide array of existing libraries for computer vision. Second, since the Navigation Subteam already uses ROS to implement its navigation algorithm, there

would be little overhead for integrating vision into the bicycle. This would allow for a smoother transition from obstacle detection to obstacle avoidance, which will be incorporated into the navigation algorithm.

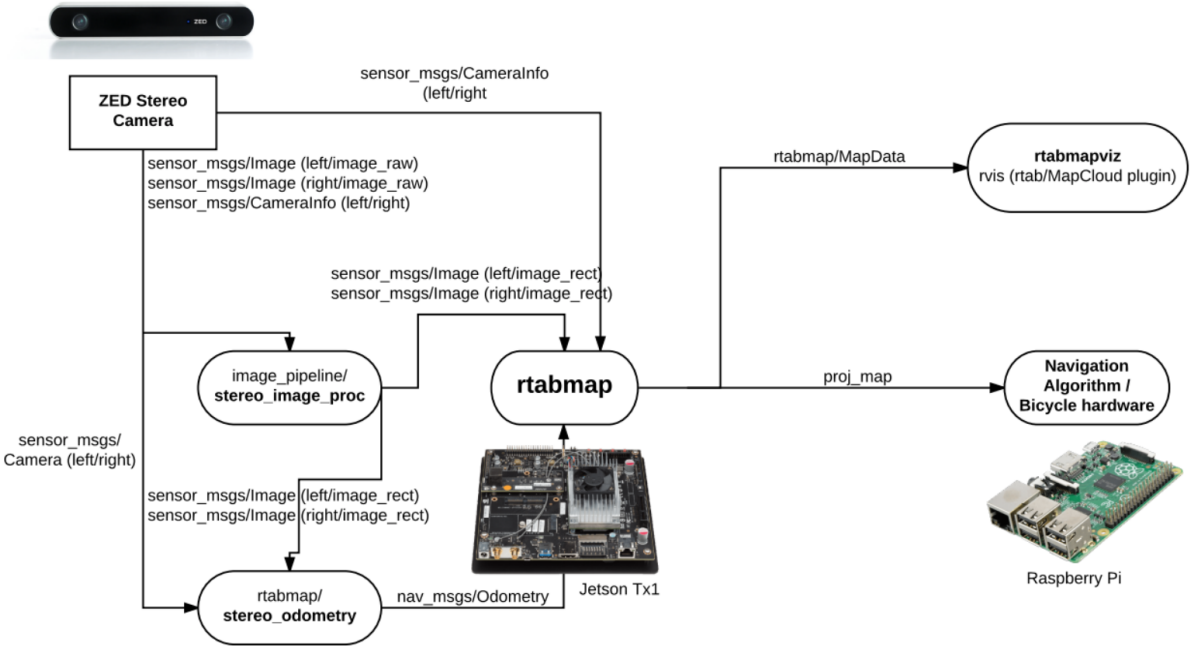
4.4.2 RTAB-Map

Our approach to obstacle detection was to use one of ROS's packages for obstacle avoidance called RTAB-Map (Real-Time Appearance-Based Mapping). RTAB-Map is compatible with stereo cameras (we use the ZED stereo camera). It is also capable of automatically generating 3D point clouds, which are maps of points in the camera's images with a depth component. It can automatically detect obstacles using these point clouds. RTAB-Map also has stereo outdoor mapping functionality. This means RTAB-map can create a 2D occupancy grid to map out obstacles in an area. This information would then be used in the navigation algorithm to steer the bicycle clear of obstacles.

In conjunction with RTAB-map, we use the visual tool RVIZ, a component of ROS. It allows us to visualize the 3D point cloud and see how the system interprets the camera images and detects obstacles.

4.4.3 Integrating Obstacle Avoidance

To accomplish obstacle avoidance using RTAB-Map, we utilize the stereo outdoor mapping functionality. Essentially, the ZED stereo camera communicates raw image data to the TX1 and the RTAB-Map nodes for processing. RTAB-Map then generates proj-map, a 2d occupancy grid that will show where mapped obstacles are located. We communicate this data to the navigation node so the bicycle can navigate away from the obstacles. Map data is also communicated to rtabmapviz through the MapData topic for it to be visualized through the TX1. (rtabmapviz is a wrapper around RVIZ with RTAB-Map options.)



4.4.4 Hardware

Vision Testing Rig (Buck) We need to test RTAB-Map in isolation before implementing computer vision on the bike. We used a cart with the TX1 computer and ZED camera, see Figure 4.27.

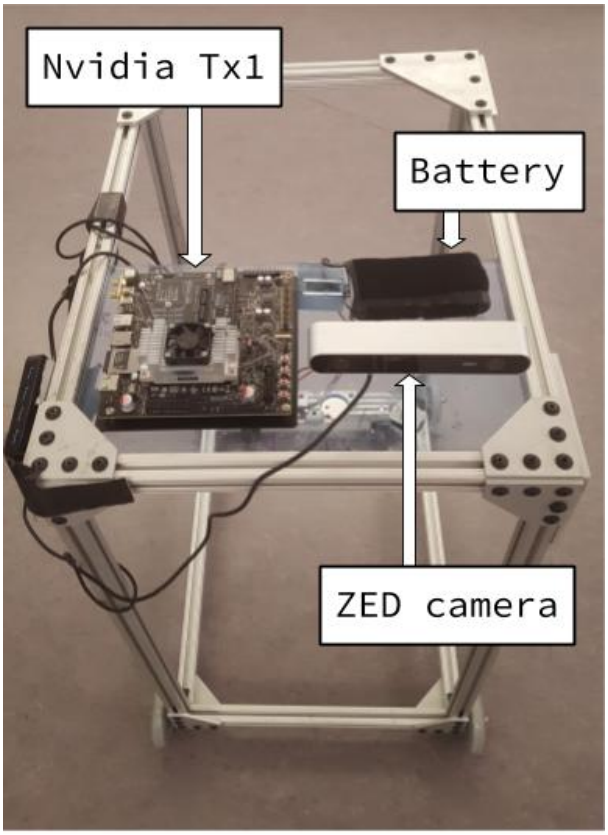


Figure 4.27: Picture of Buck

Camera Calibration Since the ZED camera is a stereo camera, which relies on two separate cameras to calculate depth, we needed to first calibrate the two cameras to get accurate readings of depth. Using the vision buck and ROS's camera-calibration tutorial, we attempted to recalibrate the ZED. However, we found that ZED is not compatible with this tutorial because it lacks necessary rostopics. On the other hand, we found that ZED is calibrated when purchased, and the SDK includes calibration tools in case we would need to recalibrate it.

Installation Installing dependencies for our software (ROS, ZED SDK) took a large portion of our time. Additionally, the Vision Subteam ran into other issues trying to accommodate the TX1 with the Vision Buck, which led to other software issues. These issues are detailed in the Installation Appendix .

Future Work

Two critical goals for the future are to optimize obstacle detection and integrate our obstacle detection into the navigation algorithm for obstacle avoidance.

4.4.5 Odometry and SLAM

We figured out how to get odometry data (position in 3 coordinates, orientation in quaternions) by using the built-in ZED wrapper for ROS. Using RVIZ, we were able to visualize the position

and orientation of the ZED camera. We are still determining vision odometry's viability for the bicycle's odometry.

We have also found a way to implement Simultaneous Localization and Mapping (SLAM) using RTAB-Map's stereo outdoor mapping functionality. Since we do not yet need the bicycle to know its position in a map of the environment, we are not prioritizing SLAM right now. We will look into SLAM further after we have refined obstacle avoidance.

4.4.6 Segmentation and Classification Algorithms

In the future, we would like to refine obstacle detection, instead of simply using RTAB-Map's built-in obstacle detection node.

One way to accomplish this goal is to refine the segmentation algorithms that RTAB-Map uses. Segmentation algorithms distinguish distinct objects in an image or point cloud. This is a necessary step in obstacle detection, since it enables us to find objects within a point cloud that could be labelled obstacles. Instead of using the default approach with RTAB-Map's built-in segmentation process, we can use a number of techniques to refine the segmentation process.

There are four types of segmentation techniques that we are pursuing: edge-based, region-based, model-fitting, and machine learning. In edge-based segmentation, the algorithm finds the edges that separate distinct objects in a point cloud. This method is the fastest of the three, but also the least accurate. Region-based segmentation finds seed points within a point cloud and then "grows" a uniform region from each seed, which segments the point cloud into distinct objects. This method is more robust than edge-based, and is the first technique we would try. The third method is model-fitting, which decomposes the point cloud into shapes, such as cylinders or cones. This method is fast and robust but not as established, especially for complex structures. The fourth type of technique relies on machine learning algorithms (K-means clustering or hierarchical clustering). Unfortunately, these techniques are time-intensive and may not be practical to run in real time.

The second way to accomplish our goal of refining obstacle detection would be to use classification algorithms. After we segment the point cloud into distinct objects, we could classify the objects based on their features using classification algorithms. This is advantageous because it allows us to detect objects that we would want to avoid and those that don't affect the bicycle's path. For example, we would definitely want to avoid objects classified as "trees" or anything large, but we would not want to avoid crumpled-up paper on the ground.

There are two types of approaches to classification: supervised learning or unsupervised learning. Supervised classification algorithms require a labeled dataset that the vision system is trained on. For example, we would need data to quantitatively describe what a tree would look like (based on the properties of the point cloud) if we want to train the system to quantitatively

describe the object as a tree. To do this, we would need many pictures of trees. This is unfeasible if we have many classes of objects to identify. Unsupervised learning, on the other hand, is feasible. The vision system would find structure within the point cloud and classify the objects into groups by itself.

5 Future Work

- Implement navigation algorithm on bicycle
- Improve position accuracy using sensor fusion
- Test and validate computer vision system
- Test landing gear start and stop sequence
- Improve balance controller - decrease lowest stable speed

Appendices

A Hardware

A.1 Front Motor Gains Optimization

A.2 Landing Gear

A.3 Watchdog

B Software Appendix

B.1 Embedded

C Navigation Testing Rig (Buck)