Cornell Autonomous Bicycle Project Team

Spring 2018 Report

Supervisors: Professor Andy Ruina (MAE) and Ross Knepper (CS)

Cornell University

Team Members					
Name	Year	Major	Team Position	Course	Credits Taken
Dylan Meehan	Sophomore	CS	Team Lead	CS 4998	3
Rohit Bandaru	Junior	CS	Vision Team	CS 4998	3
Olivia Xiang	Sophomore	CS	Vision Lead	CS 4998	3
Kane Tian	Freshman	CS	Vision Team	CS 4998	3
Woo Cheol Hyun	Freshman	CS	Nav Team	CS 4998	3
Daniel Glus	Sophomore	CS	Nav Lead	CS 4998	3
Sam Hong	Freshman	ME	Systems Team	MAE 1900	3
Max Kester	Sophomore	ME	Systems Lead	MAE 4900	3
Jordan Stern	Sophomore	CS	Nav Team	CS 4998	3
Zhidi Yang	MEng	ME	Controls Team	MAE 6900	3
Brian Bridge	MEng	ME	Controls Team	MAE 6900	4
Joshua Even	Junior	CS	Software Lead	CS 4998	3
Jonathan Plattner	Senior	CS	Navigation Team	CS 4998	3
Kyle Fenske	Junior	ORIE	Business Lead	MAE 1900	3
Andrew Jhu	Sophomore	IS	Business Team	MAE 1900	3
Joanna Zhang	Sophomore	IS	Business Team	ENGRC 3400	3

May 19, 2018

Contents

Part 1

Introduction

The Cornell Autonomous Bicycle Project Team works to design, build, and test a self-balancing, self-navigating autonomous bicycle. Our goal is to be "better" than any other autonomous bicycle. See https://bike.engineering.cornell.edu/index.html.

This semester, Spring 2018, we:

1. improved our position estimation, in part, by integrating wheel odometry into our localization algorithm.
2. developed new linear and nonlinear balance controllers. We tested balance controllers in simulation and on the physical bicycle.
3. implemented a computer vision system, on a testing platform (not the bicycle). We recorded localization data (useful for position estimation) and point cloud data (useful for obstacle detection).
4. refactored our embedded code to be more understandable and maintainable.
5. developed path smoothing techniques.

Part 2

Controls

Dylan Meehan, Brian Bridge, Zhidi Yang

2.1 Introduction

In Spring 2018, we explored new balance controller designs. The balance controller is responsible for achieving a desired lean angle and steer angle. See An Introduction to the Cornell Robotic Bicycle Project for a further description of our control structure, including the balance controller.

Specifically, we

1. implemented two linear balance controllers, in simulation and on the prototype bicycle.
   (a) One controller used gains found through a grid search of the gain space.
   (b) The other controller used gains found through an LQR (Linear Quadratic Regulator) method.
2. implemented multiple nonlinear (polynomial) balance controllers, in simulation and on the prototype bicycle. See Brian Bridge's report.
3. conducted detailed control theory analysis on our current linear controller, and defined a new method for "scoring" controller performance to be used for optimizing gains. See Brian Bridge's report.
4. derived a new dynamic model considering the angle between vertical axis and the front fork, and the moment of inertial of the bicycle, with modified processes of using acceleration rate of center of mass and Angular Momentum Balance, which are more rigorous than previous derivations. See Zhidi Yang's report.
5. designed, in simulation, two balance controllers using LQR (Linear Quadratic Regulator) based on the customized cost functions, with one using a fixed gains and the other one using gain scheduling generated from different equilibrium states of steady turn with different lean angles and rear wheel speeds, and employed an animation simulation in MATLAB. See Zhidi Yang's report.
6. developed, in simulation, a nonlinear balance controller based on sliding mode control with the new full non-linear dynamic model, and employed an animation simulation in MATLAB. See Zhidi Yang's report.
7. better defined the handoff between the navigation and balance controllers. See An Introduction to the Cornell Robotic Bicycle Project.

This semester, Brian Bridge and Zhidi Yang worked on the bicycle project as part of their (seperate) master's of engineering projects. Brian's report is availible here:
https://drive.google.com/open?id=1n9uH5CnOTcU4Kfdbw0Mm4SUgrc86hm6D.
Zhidi's report will be availbile in this folder:
https://drive.google.com/drive/folders/1YICQZkbT9YJg–DnE1lKGjRI25TkGEaP?usp=sharing

2.2 Nonlinear Equation of Motion

Dylan Meehan

To develop a balance controller, we use simulate a bicycle using MATLAB (https://github.com/CornellAutonomousBikeTeam/Matlab-Optimization). We would like the simulation to be accurate (enough). We describe the bicycle using its equations of motion. We switched from using a linear equation of motion for lean angular acceleration ($\ddot{\phi}$) to using nonlinear one. This change makes our the bicycle simulation more accurate. The nonlinear equation of motion is:

$$\ddot{\phi} = \frac{-v^2 tan(\delta)}{hl} + \frac{gsin(\phi)}{h} - \frac{b\dot{v}tan(\delta)}{hl} + \frac{v^2 tan^2(\delta)tan(\phi)}{l^2} - \frac{b\dot{\phi}vtan(\delta)tan(\phi)}{hl} - \frac{bv\dot{\delta}}{lcos^2(\delta)h} \quad (1)$$

Where:
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 = speed [m/s]

Equation 1 corrects typo errors in the Section 3 of Shihao Wang's report : "Dynamic model derivation and controller design for an autonomous bicycle". See that report for the full derivation. https://bike.engineering.cornell.edu/Dynamic %20Model%20Derivation%20 and%20Controller%20Design%20for%20an%20Autonomous%20Bicycle.pdf.

2.3 Balance Controller

Dylan Meehan

Most often, we use a linear controller for our balance controller. Specifically,

$$\dot{\delta_d} = k_1\phi + k_2\dot{\phi} + k_3\delta \quad (2)$$

Where:
ϕ = lean angle [rad].
$\dot{\phi}$ = lean angular rate [rad/s]
δ = steering angle [rad].
$\dot{\delta_d}$ = desired steer angle rate [rad/s^2]. The output of the balance controller.

To make a balance controller, we must find gains (constants), k_1, k_2, k_3.

2.3.1 Grid Search Gain Optimization

We believe our matlab simulation of the bicycle is realistic enough. That is, we think the controllers will behave about the same in simulation and in the real world.

Therefore, we can guess at a controller (a set of values $[k1, k2, k3]$), test that controller on a simulated bicycle, and judge how the controller perform. Simulating a bicycle is fast (0.05s), so we can test many different sets of gains ($[k1, k2, k3]$). Then, we can then choose the controller which performs the best.

To choose the "best" controller, we must score each simulated bicycle. The balance score is a weighted sum of squared errors in lean angle (ϕ), lean angle rate ($\dot{\phi}$) and steer angle (δ).

$$score = \sqrt{\sum(\phi^2 + \frac{\dot{\phi}^2}{4} + \delta^2)} \tag{3}$$

A lower score implies a better controller. A bicycle is stable when $\phi = \dot{\phi} = \delta = 0$. Thus, a stable bike should have a score of 0. In general, a controller with a lower score will have the bicycle closer to stable for a longer period of time.

Given a simulation and a scoring system, we can we can exhaustively enumerate every controller in some range of values. Specifically, we tested each controller in the range $1 \leq k_1 \leq 150$, $1 \leq k_2 \leq 100$, $-50 \leq k_3 \leq -1$ for integer values of k_1, k_2, k_3. Each controller was simulated for 8 seconds with v=3m/s. The bicycle started from an initial lean angle of $\phi_0 = \pi/6$. A slightly different scoring system was used with $Score = \sqrt{\sum(\phi^2 + \dot{\phi}^2 + \delta^2)}$

The best gains for this simulation were [23,16,-2]. These gains performed well on the physical bicycle. See video: https://drive.google.com/file/d/1W0Flpho41YH2y_juHxQ04scaD6OkE6dM. To be able to steer the bicycle better using an RC remote, k_3 was quintupled. With gains of [23,16,-10] the bicycle could robotically balance and be steered with an RC remote: https://drive.google.com/file/d/1qG4s9Y585maeTts8ZiO8DnBWmBi4N0Ow.

For the full results of Grid Search Testing see:
https://github.com/CornellAutonomousBikeTeam/Matlab-Optimization/tree/master/GridSearchOptimizatio

2.3.2 LQR Gain Optimization

Linear Quadratic Regulator (LQR) is a method of designing a linear controller.
See https://www.mathworks.com/help/control/ref/lqr.html for more detail.

To describe out system, we use the following:
$\mathbf{x} = \begin{bmatrix} \phi \\ \dot{\phi} \\ \delta \end{bmatrix}$, the state vector.
$\mathbf{u} = [\dot{\delta}]$, the control variable.

$A = \begin{bmatrix} 0 & 1 & 0 \\ \frac{g}{h} & 0 & \frac{-v^2}{hl} \\ 0 & 0 & 0 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ \frac{-bv}{hl} \\ 1 \end{bmatrix}$ describe the system dynamics.

The system dynamics and controller structure come from Sections 3 and 4 of Shihao Wang's report : "Dynamic model derivation and controller design for an autonomous bicycle".
https://bike.engineering.cornell.edu/Dynamic %20Model%20Derivation%20and%20Controller%20Design%20for%20an%20Autonomous%20Bicycle.pdf.

The system dynamics are:

$$\begin{aligned} \dot{\mathbf{x}} &= A\mathbf{x} + B\mathbf{u} \\ \begin{bmatrix} \dot{\phi} \\ \ddot{\phi} \\ \dot{\delta} \end{bmatrix} &= \begin{bmatrix} 0 & 1 & 0 \\ \frac{g}{h} & 0 & \frac{-v^2}{hl} \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \phi \\ \dot{\phi} \\ \delta \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{-bv}{hl} \\ 1 \end{bmatrix} [\dot{\delta}] \end{aligned} \tag{4}$$

The feedback law for our linear controller is $\mathbf{u} = -K\mathbf{x}$. That is, the control variable ($\mathbf{u}$) is a linear function of state ($\mathbf{x}$). K is the gain matrix. $K = \begin{bmatrix} k_1 & k_2 & k_3 \end{bmatrix}$.

The balance controller (feedback law) is:

$$\begin{aligned} \mathbf{u} &= -K\mathbf{x} \\ [\dot{\delta}] &= \begin{bmatrix} k_1 & k_2 & k_3 \end{bmatrix} \begin{bmatrix} \phi \\ \dot{\phi} \\ \delta \end{bmatrix} \\ \dot{\delta} &= k_1\phi + k_2\dot{\phi} + k_3\delta \end{aligned} \tag{5}$$

The matlab function `lrq(A,B,Q,R)` calculates K. A and B are defined as above. Q weights errors in the state error. That is, Q can penalize $\phi \neq 0$, $\dot{\phi} \neq 0$, and $\delta \neq 0$. R weights actuator effort. R penalizes large front motor angular speeds ($\dot{\delta}$). The LQR command minimizes

$$\sum_{n=0}^{N} (\mathbf{x}Q\mathbf{x}^T + \mathbf{u}R\mathbf{u}^T) \tag{6}$$

We set

$$Q = c \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad R = [1]$$

c weights Q and R. If c is large, Q will be proportionally larger than R. In this case, `lrq(A,B,Q,R)` will put more emphasis on minimizing $\mathbf{x}$, the state error, than on minimizing $\mathbf{u}$, the actuator effort. If c is small, the reverse is true.

$Q[2,2] = 0.1$ to penalize errors in the lean rate ($\dot{\phi}$) of the bicycle. With this term set to zero, the bicycle experienced oscillations in the balance controller. These oscillations appeared only on the physical bicycle, not in the simulation. See
https://drive.google.com/file/d/1c-No4iEWizK_mmbRtGhBEkRRz4-w_ulA.
With $Q[2,2] = 0.1$, no oscillations were visible.

By changing c, we can change the ratio of Q to R. As in section 2.3.1, we can simulate many controllers using matlab. Thus, we can search for a good controller. Specifically, we searched through the 1 dimensional space of c values. For each values of c, we called `lrq(A,B,Q,R)` to produce the "optimal" gains (K). We then simulated the bicycle with these gains. This search method differed from the search in section 2.3.1, which searched though the 3 dimensional space of gains ($\begin{bmatrix} k_1 & k_2 & k_3 \end{bmatrix}$). We scored each controller using equation 3. The LQR simulation was much faster (about 2 orders of magnitude) than searching through the gain values directly.

`lrq(A,B,Q,R)` has no maximum threshold on actuator effort. In other words, the LQR algorithm allows $\mathbf{u} = \dot{\delta}$ to be arbitrarily large. On the real bike, the front wheel cannot spin arbitrarily fast. In our simulation, we limit $\mathbf{u} = \dot{\delta}$ at 4.8 [rad/s]. Using this limit checks that a controller can perform well given the hardware constraints of our motor.

The optimal gains found using LQR search were [24,7,-8]. These gains performed well on the physical bicycle, see:
https://drive.google.com/file/d/1qF1xqRbgJhOg-1ifbnoOuajWSHukbjCi/view?usp=sharing

For the full results of LQR testing see: https://github.com/CornellAutonomousBikeTeam/Matlab-Optimization/tree/master/LQRSearchOptimization

2.3.3 Comparison

The below figure compares 3 controllers in simulation. A simulated bicycle, using each controller, is started from an initial lean angle of $\frac{\pi}{6}$ radians. All three controllers converge to a lean angle of 0 (stable riding). See Section 2.3.1 for more on the Grid Search gains. See Section 2.3.2 for more on the LQR gains. The Control gains were gains found in a previous semester. See https://drive.google.com/file/d/0BzMycrw3zvV7R0UyczZsTGdab2c/view. Thus, they serve as a control to the Grid Search and LQR gains.

Controller Name	Gains	Color on figure
Grid Search	[23,16,-2]	blue
LQR	[24, 7, -8]	red
Control	[71, 10, -20]	yellow

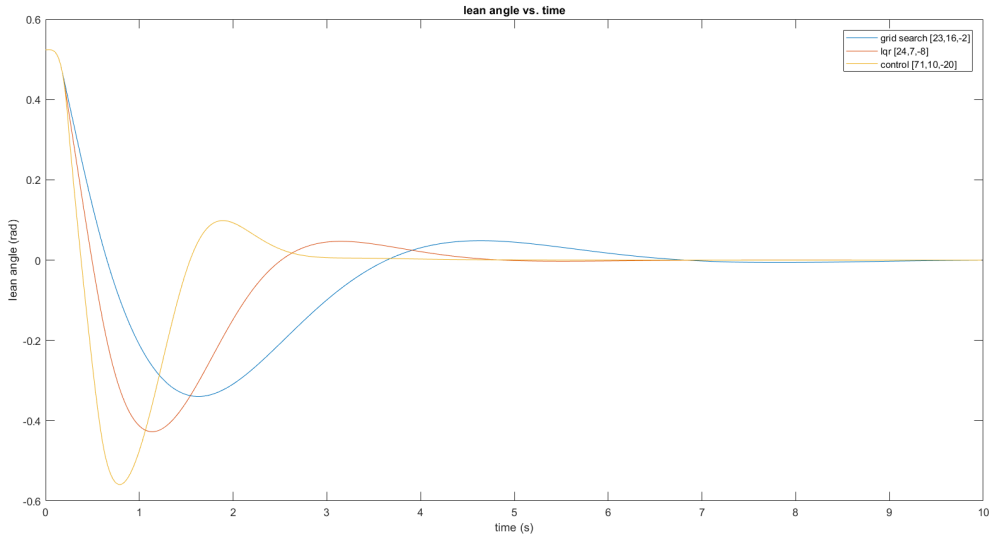


Figure 1: lean angle vs. time for 3 controllers

Both the "Grid Search" gains and the "LQR" gains are smaller than the "Control" gains. The "Grid Search" and "lqr" gains have k_1 values around 23 while the "control" gains have $k_1 = 71$. Generally, smaller gains, which still make a viable controller, are better.

All three sets of gains performed well on the physical bicycle. We tested each controller against a disturbance. All three controllers could keep the bicycle balanced. In the future, we should develop a way to quantify balance performance on the real bicycle.

Part 3

Navigation

3.1 Introduction

Daniel Glus

In Spring 2018, we tried to test our navigation algorithm on the real bike. Before this semester, we had only tested the algorithm in a simulator. We didn't successfully test the algorithm this semester on the real bike. In part, this was due to issues with the bike's localization code and sensors.

We also made progress on other projects:

- We iterated on localization and sensor fusion, and tested different ways to get bike state.
- We worked on allowing the navigation algorithm to handle different bike velocities.
- We added a path-smoothing feature to the algorithm to make it behave more predictably around turns.
- We improved the testing infrastructure, including new debugging functions and ROS calls.

3.2 Localization algorithm development

Jonathan Plattner

Accurate position measurements are best achieved by combining the readings from all of the bike's available sensors. These include the GPS, rear-motor speed (hall sensor), IMU heading, and front wheel steer angle. The purpose of the localization system is to combine these sensor readings in order to optimally measure the bike's position and velocity.

3.2.1 Single Kalman Filter

The previous system for localization utilized a Kalman filter with the 4-dimensional state space $(x, y, \dot{x}, \dot{y})$. The GPS provided measurements for x and y, and the state was updated by integrating $\dot{x}$ and $\dot{y}$. This smoothed out the noisy GPS readings, but would consistently overshoot turns.

3.2.2 Dual Kalman Filter

The filter input values $\dot{x}$ and $\dot{y}$ can be obtained from speed (v) and heading (ψ) via the following equations:

$$\dot{x} = v\cos(\psi), \dot{y} = v\sin(\psi)$$

These are nonlinear functions of ψ, but the update step for a Kalman filter is a matrix multiplication and thus can only model linear relationships between its state variables. So, it was impossible to incorporate steering and heading information directly into the existing filter. The solution was to create a second Kalman Filter with the two dimensional state space $(\psi, \dot{\psi})$. This filter is updated using the IMU heading, front wheel steer angle, and rear motor speed. The output of this filter (ψ) was used to compute the inputs to the original filter $(\dot{x}, \dot{y})$. The resulting dual-filter localization system performs much better during turns than the single-filter system.

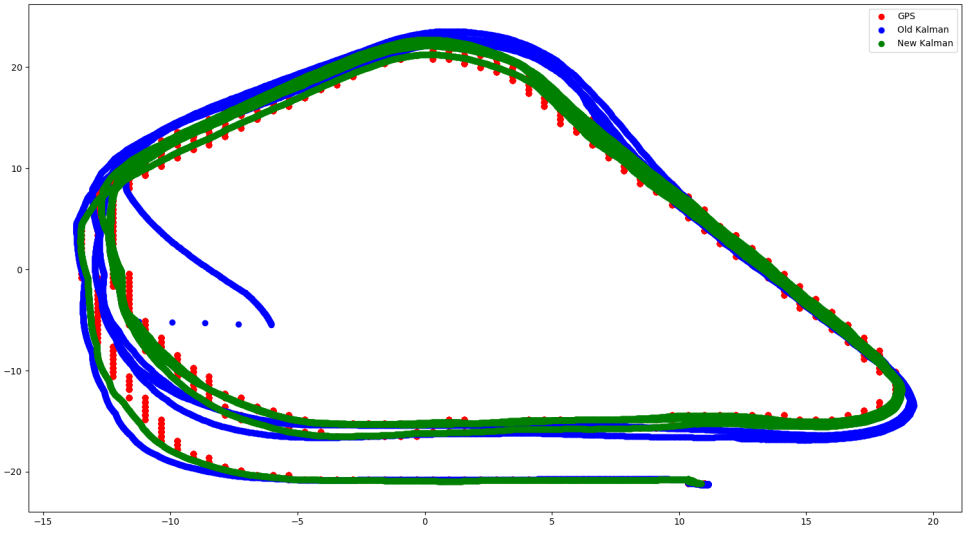


Figure 2: Position estimates using three different systems: red is GPS, blue is old (single-filter) Kalman, green is new (dual-filter) Kalman. The test included walking, 4 times, around a path on the engineering quad. The old (single) Kalman filter (blue) overshoots the top turn more than the new (double) Kalman filer (green).

3.2.3 GPS Noise

The Kalman Filter assumes that each sensor measurement has Gaussian noise[1], centered around the true value. When this is not the case, a Kalman Filter is no longer the optimal state estimation algorithm. The noise in the GPS position readings is highly non-Gaussian. The values only appear on a grid with approximately a 0.5 meter resolution. So if the bike's true position is $x = 0.75$, the measured position will not be a Gaussian centered at 0.75, but instead will be a discrete distribution with a 50% chance of $x = 0.5$ and a 50% chance of $x = 1.0$. This non-Gaussian noise can be seen having a detrimental effect on the Kalman filter.

[1]Noise whose magnitude follows a normal distribution, or a "bell curve"

3.2.4 Future Work

The heading given by the IMU is not based on true north because its magnetometer is currently disabled. Thus, there is a random offset from true north that is different each time the sensor is turned on. This prevents IMU heading data from being used for localization, but would likely be fixed by enabling the magnetometer.

Rear motor speed given is currently computed by measuring the period between two Hall sensor ticks and taking finite differences. That speed is then integrated by the Kalman filter, but it would be better to directly update the filter's state whenever a Hall sensor tick is registered. This should be more accurate than differentiating and then integrating.

The filter could be improved to be more robust to GPS noise, but it seems likely that a long term solution will not use GPS at all for small-scale localization. Instead, the existing filter should be merged with the localization results achieved by the vision team.

3.3 Current localization algorithm

Daniel Glus

The algorithm has three stages:

1. Updating the speed (v), heading (ψ), and change in heading over time ($\dot{\psi}$) from the bike sensors
2. Updating the position from the GPS
3. Updating the position from the current beliefs about speed and heading

In the first stage, we update v, ψ, and $\dot{\psi}$. For speed and heading, we use the relevant sensor readings. If the bike currently believes a value is x, our sensor reports that the value is x', and our gain is k, then we run this statement:

$$x \leftarrow x + k(x' - x) \tag{$\star$}$$

$\dot{\psi}$ is instead calculated using the formula:

$$\dot{\psi} = \frac{v \tan \delta}{l}$$

where ψ is the bike's heading, v is the bike's speed, δ is the bike's steer angle, and l is the length of the bike's wheelbase.

At the moment, the sensor readings we use for this stage come from the GPS. This is a temporary measure. We want to use as many sensors as possible for redundancy. In the future, we expect to use the IMU for yaw and the Hall sensor for speed.

In the second stage, the position is updated from the GPS. We know that the points are only on a grid. That is, the coordinates can only change by a fixed amount at a time. So, first, we check if the GPS coordinate has changed by that fixed amount. If that's the case, meaning that we have just jumped to a new grid point, then we update the GPS-based position estimate using the gain formula given in ($\star$). Then, we update the overall position estimate using the same formula (where the GPS-based position estimate is taken to be the "sensor reading"). We also must cover the case where the GPS has suddenly corrected itself by a large distance. In

this case, handled in the second half of this stage, we apply the ($\star$) formula a second time. In the future, we could make a more aggressive correction, but the current process is acceptable due to the low reliability of the present GPS.

In the third and final stage, we update the heading by multiplying the time step Δt by the yaw dot. Then update the overall position estimate using the speed and heading values using the standard trigonometry-based formulas:

$$x \leftarrow x + \Delta t * v * \cos\psi$$
$$y \leftarrow y + \Delta t * v * \sin\psi$$

After that, the position estimate is handed off to the navigation algorithm to calculate a desired front wheel angle.

This algorithm was refined during the semester through many rounds of testing. However, it continues to experience some difficulty in tracking the position of the bike. In particular, while walking with the bike in the engineering quad with the current algorithm, we observed that the speed estimate was laggy. Specifically, for one test, we would move in the positive x direction for several seconds, then turn around and move in the opposite direction. The x-coordinate of the position estimate would continue to increase for some time (on the order of ten seconds) after we turned around. More dependence on the hardware sensors, once we get them to give accurate values, would probably help this situation.

3.4 Dynamic Velocity Adjustments

Daniel Glus

Improvements to the navigation algorithm were made on at least two fronts this semester. We implemented gain scheduling based on velocity, explained in this section, and path smoothing, described in the next section. Gain scheduling is the idea that the constants used in the nav algorithm probably change with the bike's velocity. For example, if we double the speed of the bike, some gains should change - say, we could correct angle deviations more aggressively. This project thus focuses on deriving, for the constants in the navigation algorithm, a dependence on the bike's velocity. A testing framework was set up that could quantify the performance of a specific version of the navigation algorithm using a given set of constants. Using the framework, many iterations of testing were performed. In each round, we would give the (simulated) bike a velocity and a set of initial constants, and converge to the "correct" constants. The results of this project were inconclusive due to running out of time, but we hope to come up with an approximate linear dependence in the future.

3.5 Path Smoothing

Woo Cheol Hyun

3.5.1 Purpose

The purpose of creating a path smoothing algorithm is to create a new set of waypoints given an original set of waypointts that the bike can follow exactly. In the navigation simulation,

the bike will cut corners and overshoot as it would be impossible for the bike to follow a path exactly. This is mostly due to the minimum turning radius of the bike. An example of the bike simulation following a path defined by 5 points is shown in below.

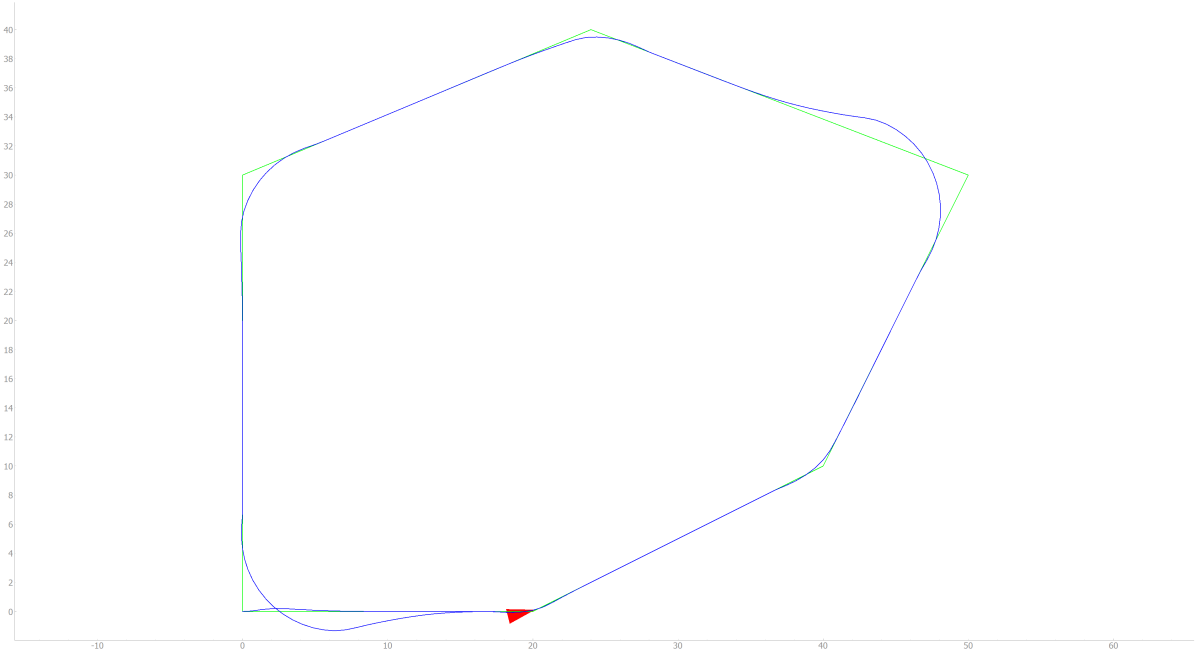


Figure 3: Green: original path. Blue: path of bike

A good path smoothing algorithm would achieve two goals:

1. Create a set of waypoints that the bike can follow exactly
2. Change the path as little as possible

The focus this semester was the first goal. Work on the second goal will be expanded upon in future semesters.

3.5.2 Bezier Curves

The path smoothing algorithm was implemented using quadratic Bezier curves. A quadratic Bezier curve is defined by 3 points.

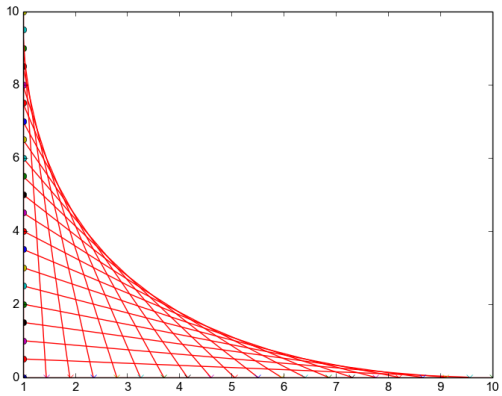
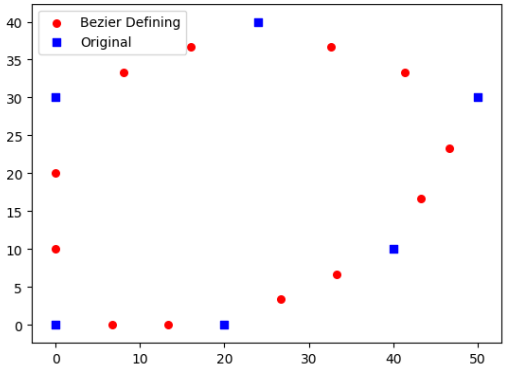


Figure 4: An example quadratic Bezier curve, defined with the points (0, 10), (0, 0), and (10, 0).

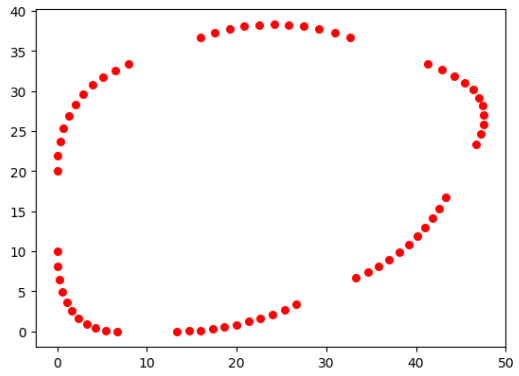
3.5.3 Implementation

First, we create the points that define the Bezier curves. Second, using the points, we create the Bezier curves. We improved this algorithm by changing the initial points.

The initial approach was to split each line segment into thirds. Each "corner" was set as the middle point of a Bezier curve. This is shown below. The original points, blue are part of the Bezier defining points.



The resulting Bezier curves are shown below.



With these points, the bike simulation does a much better job of following the path exactly, as shown below.

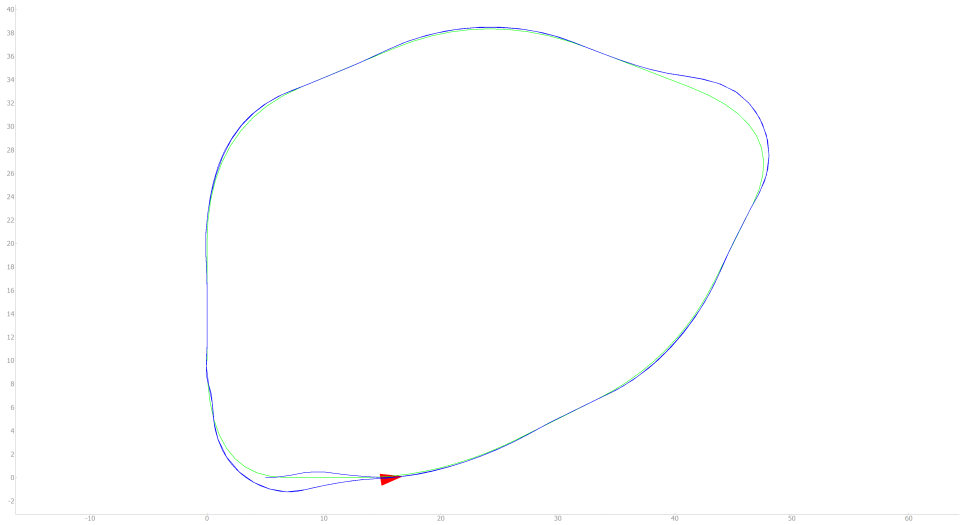


Figure 5: Green: result of path smoothing. Blue: path of bike

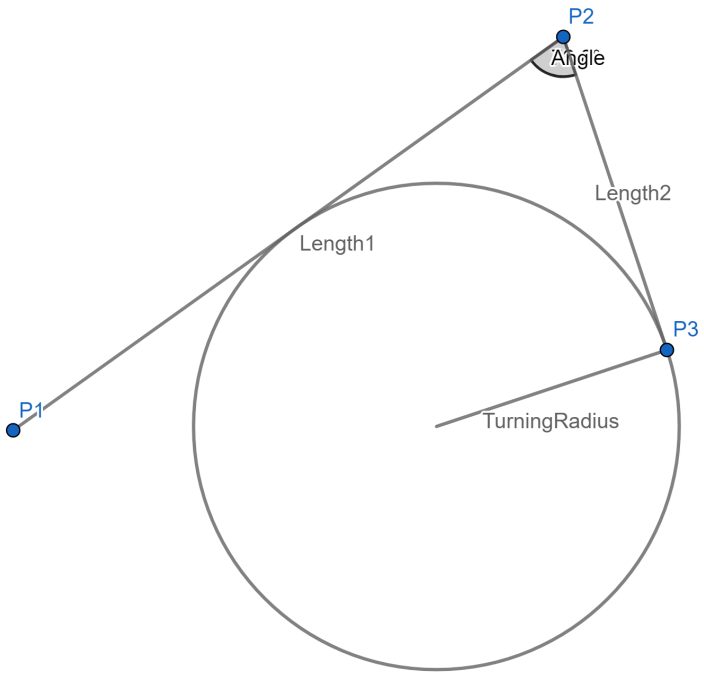
However, the simulated bike still cannot exactly follow the path. It has particular trouble with sharp turns.

To address this problem, we created a second algorithm was implemented to check that each curve respected the minimum turning radius of the bike. First, we create the Bezier points as in the first implementation. Then the turning radius at each vertex in the original path is calculated using the following formula.

The "turning radius" (R) of a set of 3 adjacent points was calculated with this formula:

$$R = \frac{min(length1, length2)}{2\cos(\frac{angle}{2})}$$

where the variables are those from this diagram:



After this, if any part of the curve had a radius of curvature[2] smaller than the minimum turning radius of the bike, the Bezier-defining points were spread further apart to create a smoother curve.

This algorithm ensures that the bike can physically follow the path. One drawback is that the resulting path may not be that close to the original. To solve this, we will manually inspect the generated paths. The simulation is shown below.

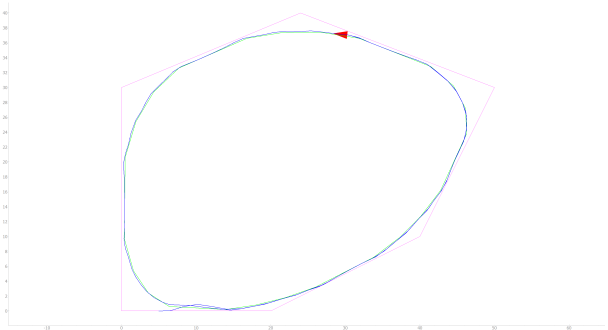


Figure 6: Magenta: original path. Green: result of path smoothing. Blue: path of bike

The current path smoothing algorithm outputs a set of waypoints the bicycle *could* follow perfectly. Future work will focus on keeping the smoothed path closer to the original.

[2]Radius of curvature: the inverse of curvature (κ)

Part 4

Vision

Rohit Bandaru, Kane Tian, Olivia Xiang

4.1 Introduction

The Vision Subteam has two goals: (1) mapping and obstacle detection, and (2) visual odometry. We want to detect obstacles to avoid a crash. With visual odometry we can generate more accurate position estimation then we can without. Better position estimation can lead to better navigation.

Our goals for this semester were to have our proof-of-concept vision system (operating on our vision buck) recognize obstacles in the vision system's path and know these obstacles' positions relative to the vision system's position. We made significant progress in both goals this semester.

4.2 Software

4.2.1 ROS

We used ROS (Robot Operating System) to create a node to receive obstacle and localization data. This node will be used in the future to help our navigation algorithm.

4.2.2 Rtabmap

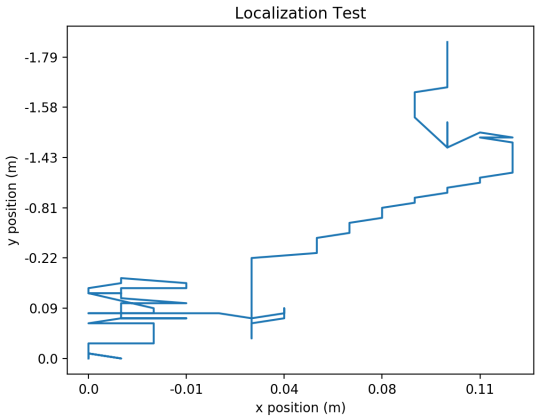
RTAB-Map (Real-Time Appearance-Based Mapping) is a 3D mapping library built on top of ROS. ROS is used for communication between different subsystems of a robot. Because our navigation system uses ROS, using RTAB-Map will make interfacing with the navigation system easier. We also found that RTAB-Map's creator has a forum that he responds to daily. We utilized this to great effect. RTAB-Map is highly configurable, which means we can tune the parameters to be optimal for our purposes.

4.3 Localization

We can use computer vision to localize the bike very accurately using the technique of visual odometry. Given two images taken in sequence, a human can tell how the camera moved between them. Visual odometry is a way for computers to accomplish this. Stereo cameras provide depth perception, which means more robust visual odometry. One drawback is susceptibility to drift, which makes the reported position differ from the truth over time.

We utilized two software approaches for visual odometry: the Zed SDK (Software Development Kit) and RTAB-Map. The Zed SDK includes a positional tracking feature, which keeps track of displacement in three dimensions. However, the Zed SDK is not open source, so the details of its implementation are undocumented. According to Stereolabs, its creator, it uses 3D point cloud SLAM techniques.

Figure 7: Localization test in Duffield Hall



From our positional tracking, we get a timestamp, a 3D displacement vector, and angles for heading. We tested the tracking by plotting data collected while moving the buck against time. RTAB-Map and the Zed SDK also come with built in visualizers that use Rviz, so we can see how the data changes in real time.

RTAB-Map allows us to use either the Zed's odometry through the SDK, or the RTAB-Map's odometry algorithm. We tried using both, but went with the Zed version due to faster performance. We valued performance because it's necessary for obstacle detection and mapping.

Moving forward, more testing is required to verify how effective visual odometry is. We also need to test in more diverse environments. We have found that performance is poor in indoor environments with a lot of reflections and glass, so we plan to also test outdoors. We must also do tests which happen over a longer distance with established ground truth, in order to test the drift of the visual odometry algorithms.

4.4 Obstacle Detection / Occupancy Grid

For the purpose of obstacle detection, we decided to implement a bare-bones vision system using just RTAB-Map's built-in capabilities. After configuring RTAB-Map to our computer's environment, we followed RTAB-Map's RGB-D Handheld Mapping tutorial[3]. This allowed us to see a 2D occupancy grid, a map of 3D obstacles projected onto a surface, or the ground truth. To generate the obstacles, RTAB-Map has its own code to create obstacles from point clouds, which assign x, y, and z coordinates to the pixels in the field of vision of the ZED camera. These point clouds are generated by the ZED camera itself. The 2D occupancy grid is useful because it shows where the obstacles are relative to the camera.

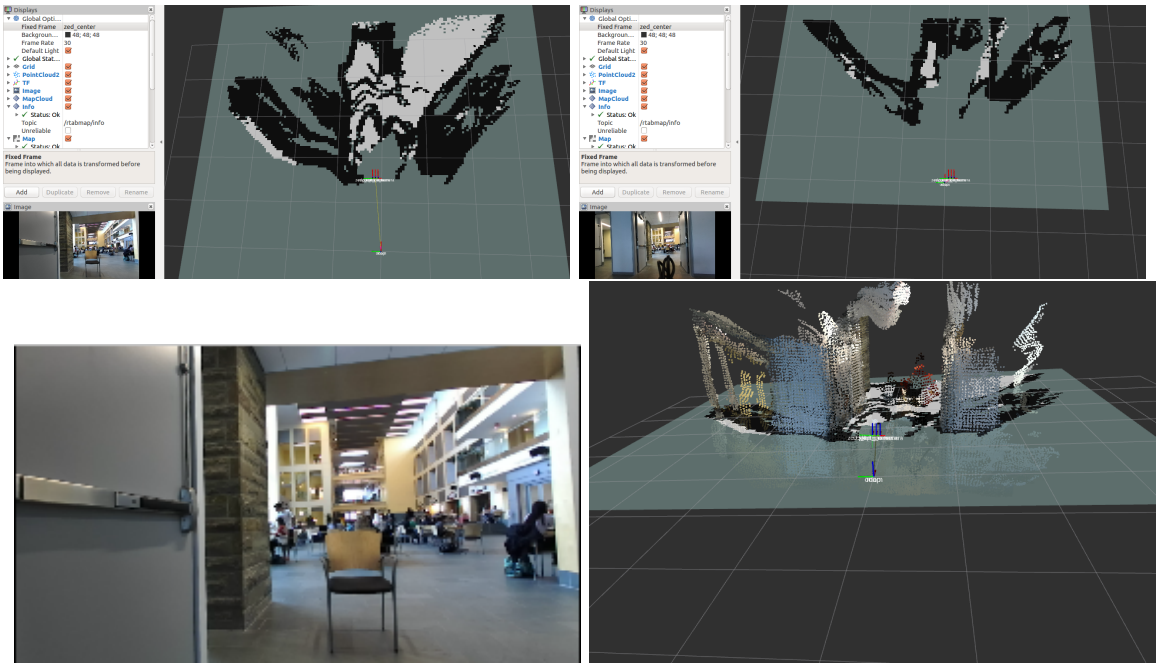
We ran a few tests in Duffield Hall to visualize the output of the 2D occupancy grid, and we had a difficult time determining what was considered an obstacle in the 2D occupancy grid. ZED's

[3] http://wiki.ros.org/rtabmap_ros/Tutorials/HandHeldMapping

odometry showed the camera crossing into what we thought were the projections of obstacles, which is impossible. This could be a problem with odometry in the ZED, although it is a highly-rated product so this is unlikely. We visualized the point cloud of the obstacles only and found that they mirrored what we saw, so we determined that we need to fine-tune the RTAB-Map parameters to achieve a clearer visualization and more nuanced obstacle detection. We will look into changing the ground-height parameter, resolution, ceiling-height parameter, and other parameters to improve the output.

We also successfully wrote a basic ROS subscriber node, which takes in data from the rostopic in RTAB-Map that publishes the 2D occupancy grid information. This demonstrated that we are capable of not only viewing the data from RTAB-Map, but also receiving and manipulating it, which is necessary if we want to fine-tune the vision system and integrate our system with Navigation.

Figure 8: Point Cloud Data



Part 5

Systems

5.1 Introduction

Max Kester

The Systems subteam is responsible for maintaining and upgrading the bike's physical mechanisms and hardware. Our primary goal was to ensure the software teams could test as frequently as possible without worrying about physical issues. Over the semester we solved multiple electrical and mechanical problems; ranging from cable stress and organization to securing our crash prevention system.

5.2 Back Kill Switch

Sam Hong

The servo is located on the back of the bike and is responsible for activating a kill switch that can shut off both of the bike's front and back motors remotely. It is controlled by the CH5 switch on the transmitter, and triggers the kills switch on the back of the bike by rotating its output arm in a clockwise movement.

When the kill switch is activated, because the power supply is separated from the PCB, both of the front and back motors will turn off but the board will remain on. Because the input power requirement of the TR624 receiver is 4.0 to 6.0 volts, four AA batteries are required to ensure that the servo properly works. Currently, the battery pack, servo and kill switch, and the receiver are all secured on to the back of the bike for ease of access. The batteries will drain if connected to the receiver, so they are unplugged to conserve battery life.

5.3 Second Brain

Max Kester

The Navigation and Vision teams have began to incorporate more components on the Bicycle, such as the Raspberry Pi, GPS unit, and eventually a TX1 and ZED camera. As a result, real estate in the toolbox 'brain' for additional circuitry was becoming limited. The systems subteam sought to design and build a second space to distribute and organize our electronics.

The primary design considerations were what materials to use, how to attach it to the bike, and size. Due to concerns over weight, wood and metals were ruled out. For the benefit of rapid prototyping, 3D printed ABS plastic was the final choice. Sizing constraints on RPL's printers meant that to achieve a desirable size, the box would have to be constructed as two parts to be adjoined.

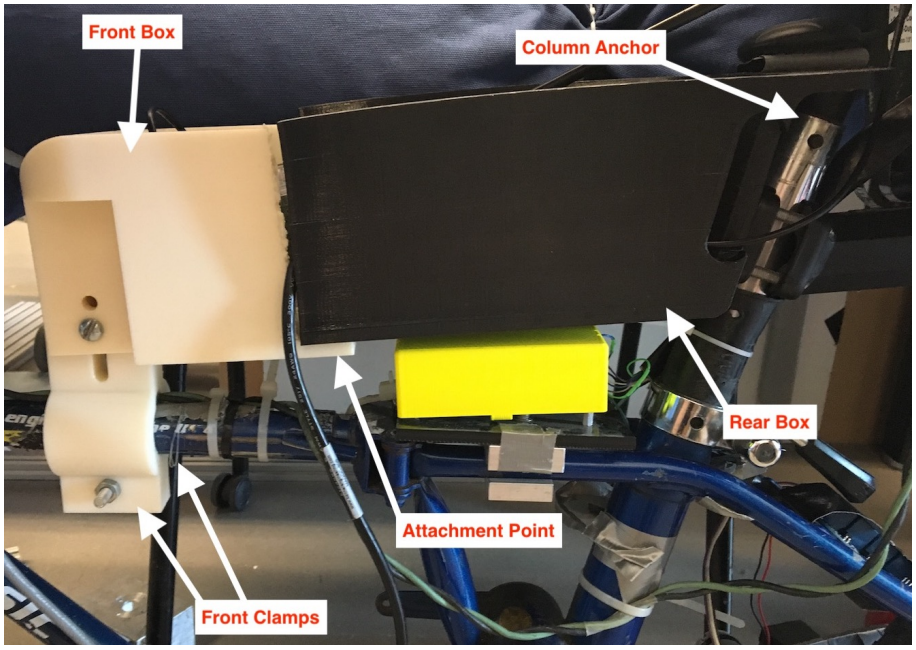


Figure 9: Brain Box Components

The final system consists of four components: the back part, the front part, and two front clamps. The back portion is attached to the seat column by an angled, tapered rod which when inserted tightly acts as an anchor to restrict translation in the horizontal plane. The front section is attached by semi-circular clamps which are screwed to each other around the horizontal shaft of the bike frame. If the brain needs to be removed, the procedure is (1) unscrew the clamps (2) remove box components from seat column.

We tested the final product. When the bike fell, the rear box sheared off the column anchor (see ??). Reducing the crash forces and improving the design can be done by (1) printing a solid, rather than lattice, structure in the column anchor, (2) creating supports for the airbags that minimize contact with the box, or (3) using a small metal component as the shaft anchor that can integrate differently with the design of the rear box.

Part 6

Business

6.1 Introduction

Kyle Fenske

In the Spring 2018, the business sub team heavily focused on outreach and sponsorship tools. Before the semester began, we had an outdated website with no specific material on sponsorship. Thus, we worked towards creating a platform where outside individuals can learn about the team.

This semester we made progress on/completed the following projects

- We applied to two Cornell Engineering Alumni Association awards and received 1700 dollars for one of the applications.
- We participated in Gift Day and crowd funded 500 dollars through the Cornell Alumni network
- We created a sponsorship brochure that outlines the goals and progress of our team as well as how companies can get involved and further our progress.
- We updated our website to serve as a more comprehensive tool that showcases all elements of our team, including current progress, testing videos, and recruitment information.

6.2 Future Goals

Kyle Fenske

The main goal of the business team for future semesters is to leverage the tools that this semester's team has created to reach out to companies to sponsor our team. As of now, our brochure outlines certain sponsorship tiers with monetary goals. At the same time, our team could also look for donations of supplies, such as an electric bicycle. At the start and end of each semester, the business team should update the website. The business team should always reach out to the other sub team leads to review the content and make sure it aligns with their vision for their sub team.

Part 7

Future Work

Everyone put your future work stuff here

7.1 Vision

7.1.1 Looking Forward

Our main goal for the future is to integrate our vision functionalities with the navigation algorithm and then the bicycle itself. The 2D occupancy grid can tell where the obstacles are, so the next step is for the navigation algorithm to use this information to navigate the bicycle in order to avoid these obstacles.

To reach our broader goal of implementing computer vision for bicycle navigation we have various technical challenges and sub goals which are outlined below.

7.1.2 Optimize Occupancy Grid and Localization

One short term goal is to better understand the output of the 2D occupancy grid in RTAB-Map. We would also like to fine-tune it based on the capabilities of RTAB-Map by adjusting parameters and perhaps also diagnosing the low frame rate (30 fps) of the ZED camera. (We decided to move on from the low frame rate issue because one of our advisors, Professor Knepper, believed the data our ZED was collecting was fine for our purpose.)

After fully understanding the 2D occupancy grid and achieving acceptable obstacle detection and mapping using the grid, we will then look to understand RTAB-Map under-the-hood and adapt it for our bicycle. To do this, we have many long-term goals for coming semesters.

7.1.3 Testing Procedure

In order to optimize our algorithms and see what works, we need a good testing procedure and a benchmark to compare performance. We would want to test the vision buck in an environment that obstacles that are consistent in time, variable in time, and easy to recognize. This environment should also mimic the environment that the bicycle will eventually be operated in. We also need to develop a testing protocol and a way to measure performance. This will allow us to iterate through different implementations effectively.

Moving on, we would like to integrate our system with the Navigation Subteam's system. We could do this in two ways: one way is to use a graph search to search the 2D occupancy grid for a path that minimizes the probability of a collision, and the other is for the Navigation Subteam to update their path based on the location of the obstacles. Much planning for this step will be necessary. We are also looking to incorporate an emergency stop system that will stop the bicycle if an obstacle gets too close. This is necessary to prevent the bicycle from crashing into a person or another moving obstacle.

7.1.4 Navigation Integration

In order to use our occupancy grid and localization functionalities on the actual bicycle, we need to integrate with the existing navigation algorithm. This is the most important goal, and also the most difficult. We see two main approaches to this. We could use the occupancy to upgrade the path that the navigation algorithm is trying to follow, or send the locations of obstacles to the navigation algorithm and have it avoid them.

7.1.5 Emergency Stop

The occupancy grid is built for long term path planning. However, it would not be effective in short term scenarios such as a person suddenly running in front of the bike. We need to implement an emergency stop functionality, so when the bicycle is about to hit an obstacle, it will stop navigation and focus on stopping to minimize damage.

7.1.6 Obstacle Classification

In the far future, we would like to classify the objects that the ZED camera sees to fine-tune the bicycle's response to obstacles. For example, the bicycle should swerve around a person, but we would not want it to adjust its path for small plants that may be picked up by our vision system. Also, this classification can be used to distinguish between stationary and mobile obstacles so a stationary object like a lamp post would be saved in a long term map, while a car would not be.

7.1.7 Long Term Maps

Saving the map created from RTAB-Map could also enable our bicycle to navigate around obstacles on the second pass more accurately. This would allow the bicycle to navigate really well in an environment that it has seen before. Simultaneous Localization and Mapping (SLAM), which RTAB-Map is used for, will also help with odometry in the future.

7.2 Systems

7.2.1 Brain Box

The brain will need to be reprinted with a stronger material connection.

7.2.2 Pillow Supports

A support system for the wine bags and pillows can be attached to could be beneficial. A structure that runs along the length of the bike on both sides would potentially: strengthen their security and placement, and absorb and distribute crash forces.

7.2.3 ZED Camera Placement

As the Vision team is closer to implementing the TX1 and ZED camera on the bike for testing, the Systems team will want to plan, design, and assemble a mechanism or structure for housing the vision units.