Controls Guide

Last Updated May 2020
Kevin Kerliu, Armaan Kohli, Leart Krasniqi, Shyam Paidipati

Overview of Control Theory

Before we discuss the specifics of the team's work on the Autonomy Lab car, we first explain some of the main elements of control theory. We do this so someone with little to no exposure to control theory can be brought up to speed on the general goals of the controls team.

What is Control Theory?

According to Wikipedia, "control theory in control systems engineering is a subfield of mathematics that deals with the control of continuously operating dynamical systems in engineered processes and machines." Basically, control theory deals with making sure that a particular system (e.g. a circuit, a mass attached to a spring, or a car's cruise control) responds appropriately to various inputs. For example, in a cruise control system, we want to make sure that when the user sets a particular speed, the car remains (roughly) at that speed. The "various inputs" in this case can be the road conditions. For example, if the car attempts to climb a hill, more torque needs to come from the engine in order to keep the car speed at the user's desired level. Similarly, if the car begins going down a hill, the torque from the engine must be reduced in order to maintain the speed. This brings us to some of the key concepts in control theory.

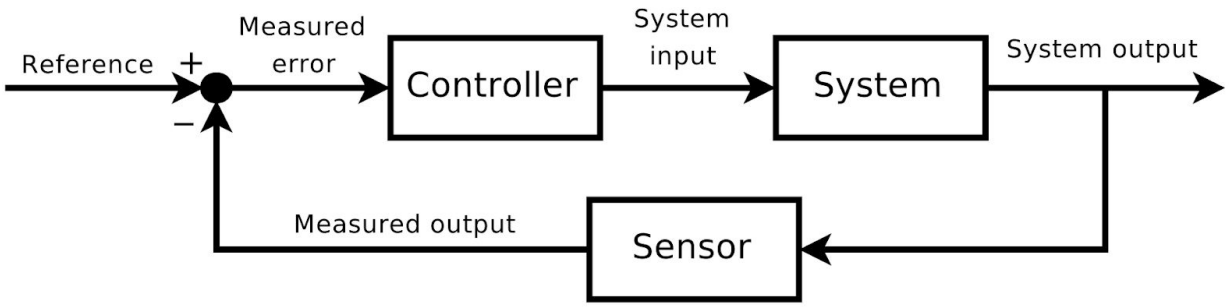
Key Concepts in Control Theory

The system that "performs the control" is called the *controller*. In our cruise control example, the cruise control module in the car is the controller. The fact that the controller is able to deal with the various road "inputs" means the controller is *robust*. In general, robustness means the controller is able to function properly even in the presence of uncertain disturbances in the input. We also generally desire *stability* in our controllers. In order for a controller to be stable, its output should not "blow up" if the input is bounded (i.e. a bounded input yields a bounded output). In the cruise control example, the controller would be classified as unstable if a bounded input, such as a hill with a 1% grade, resulted in the controller maxing out the throttle to the engine. Although this output is technically "bounded" (since the car can only go so fast), the controller is essentially asking for an infinite increase in torque. So, in the reference frame of the controller, the output is unbounded. Luckily, cruise control systems are designed to be stable, so changes in road conditions are rarely felt by the driver and passengers in the car. Another key concept is the concept of *state*. The state of the system basically means the

current relevant values in the system. These values can be modified by the controller to yield a desired state. For example, in the cruise control car system, the state may include the car speed and the throttle level.

Golden Rule of Controls

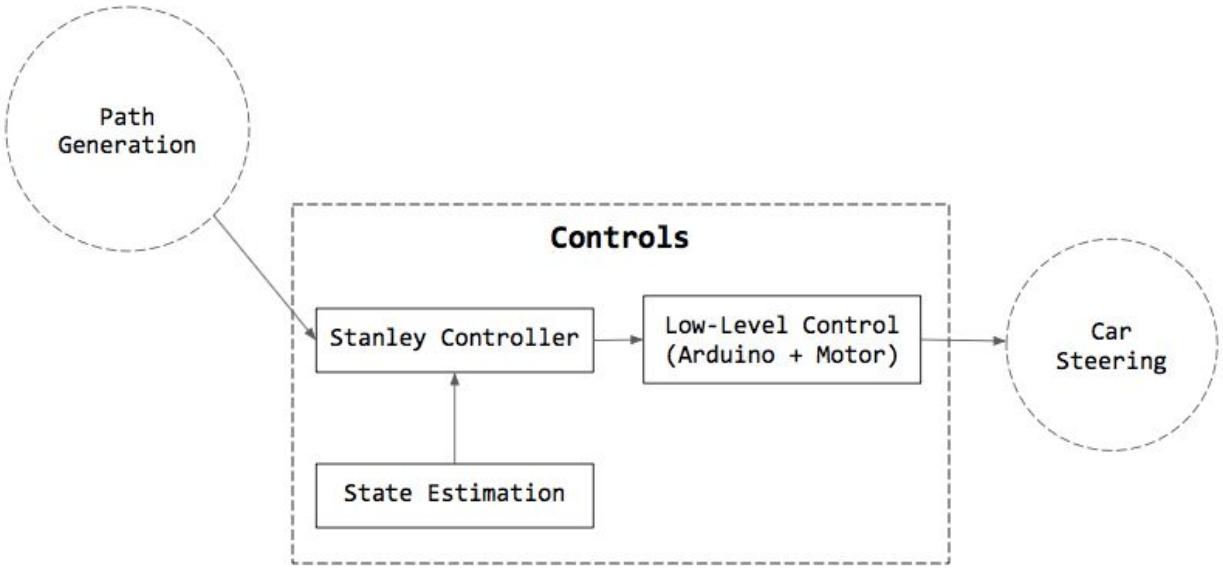
So, how exactly do we perform this control? Well, one of the most important concepts in control (so important that it has its own section dedicated to it) is the concept of *feedback*. Feedback refers to performing some control action, measuring the result of that action (i.e. the output state), and then adjusting the action to make sure the controller is performing as desired. The following diagram shows this:



Sometimes feedback control is also known as closed-loop control, which should be apparent from the diagram. Open-loop control does also exist. In an open-loop control system, there is no sensor that measures the output and provides feedback or “help” to the controller to tell it to adjust itself. In many situations, feedback is preferred over no feedback. For example, cruise control can be performed with an open-loop controller. When the user selects a speed, the controller can just lock the throttle at the current position. This is perfectly fine if the road does not differ much from the time the user set the speed. But, going uphill would result in a slower speed, and going downhill would result in a faster speed. In a feedback controller, however, the current speed of the car is constantly fed into the controller as well, allowing for the throttle control to be fine tuned to keep the car at the desired speed.

Application to the Autonomy Lab Car

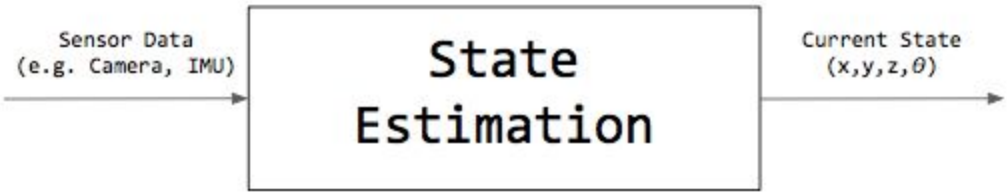
Okay, now that we understand what control theory is and what some of the key aspects are, let us talk about how control theory is applied to the car in the Autonomy Lab. In the most general sense, the car must be able to follow a certain path at a certain speed. Therefore, the controls team is tasked with making sure the car can do just that. Currently, the team is focused on the path-following portion of the control. The following diagram details the flow that the team is concerned with:



We will go into the details of the diagram shortly, but at a higher level, the controls team expects to receive data from the path generation team. With this data, invoke a path-following controller, the Stanley controller. The Stanley controller requires information about the current state of the car, so we also perform state estimation and feed that data into the controller as well. After the controller is invoked, we want to make sure the proper steering angle (which is the output of the control system, as the goal is to turn the car to follow a desired path) is accurately achieved in the mechanical steering action (i.e. properly setting the PWM so that the steering column is turned properly to make the desired turn). Once that “lower-level” (we call it that because that level is where the actual conversion of the software controller output meets the physical mechanical input of the steering column) control is performed, the car steers and follows the path.

You may be wondering, “I thought the golden rule of control theory was feedback. I see no feedback here!” Do not worry young one. The feedback is what holds all of the components together, and this diagram is just meant to show the overall flow of the control system. Without further adieu, let us explore the control blocks in more detail!

State Estimation



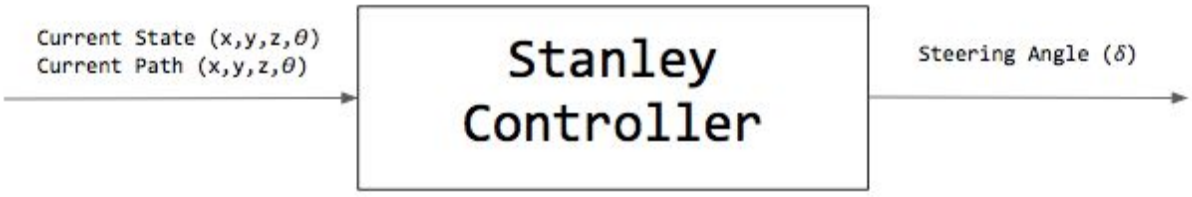
The state estimation block expects sensor data as input and outputs an estimate of the current state, which is the car's position and heading angle. Something that can estimate the state of a system is called an *observer*. There is a lot of math and theory around observers and state estimation (what even is a state?), but much of that is unnecessary to actually use a state observer model (though it is necessary to *design* one, but hopefully I'll have done that already so you don't even need to worry about it (maybe??, I guess not)).

So, basically, the state estimator that will in all likelihood work out for the best is what's called an Extended Kalman Filter (EKF). To understand what an EKF is in a more abbreviated way, checkout this: ___. It goes over the technical details in brevity.

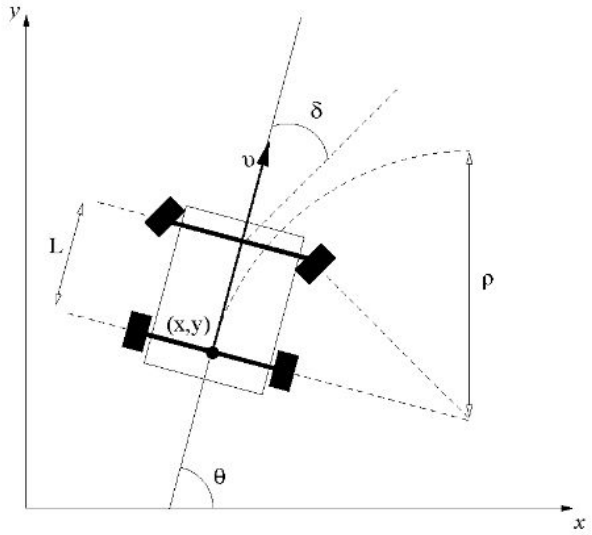
Practically, the state observer will take in sensor data from the encoders, camera and IMU and produce an estimate of the state (x, y, z, \theta) ie. the position of the scooter. This is useful because our controller (see below) is geometrical, meaning that it needs the position of the scooter in order to control. A ROS package like this_ will likely do the trick, so my work here is done! Just kidding, it's only just begun.

References: If you actually want a comprehensive understanding of this sort of thing, I highly recommend reading through Haykin's Adaptive Filter Theory for exposure to Kalman from an adaptive filtering setting (it's actually just an RLS adaptive filter, but what does that actually mean??). You should also take Fontaine's adaptive filter course. I also recommend reading Tucker Maclure's blog posts, and watching his lectures on youtube, they give a more 'intuitive' understanding of Kalman from a more practical perspective. This is also a nice little blog post that is vaguely helpful. I also highly recommend working through Kalman and Bayesian Filtering, it's a great resource that covers the basics nicely and goes into the more advanced topics related to Kalman Filtering via interactive jupyter notebooks. Also, generally speaking, if you find an old textbook on modern controls with a Russian-souding name, then it'll also be a great learning resource.

Stanley Controller



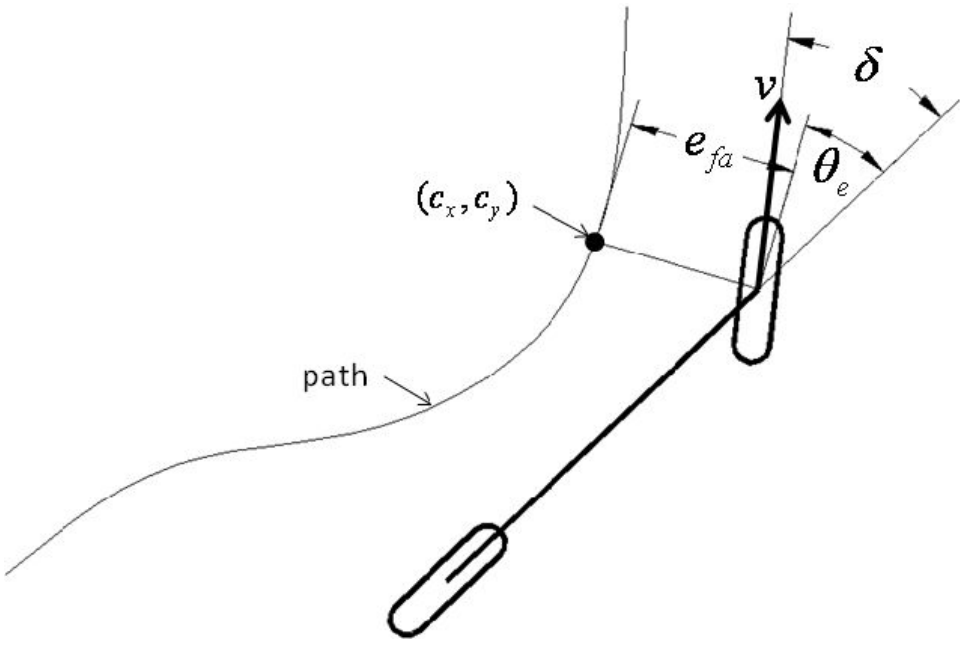
The Stanley controller takes the current state (from the state estimation block) and the current path (from the Path Generation system) as inputs, and provides the proper steering angle as the output. The Stanley controller is a geometric feedback controller, meaning that it relies solely on the geometric properties of the car's current state and the path that it is trying to follow (and, of course, uses feedback). In order to apply the Stanley controller, we need to have a model of the car (i.e. a mathematical way to represent the car). We do this through the *bicycle model*. The bicycle model essentially allows us to treat the car as a simple kinematic object, ignoring the axles holding the wheels together along with many other dynamics. A simple diagram details this model:



From the diagram, it becomes apparent why the state estimation block outputs (x,y,z,Θ) as the state of the car (the z is the height of the car, which is not vital to the Stanley controller, but is still a state variable that we may be interested in). Therefore, the entire motion of the car can be described by the following equations of motion (EOMs):

$$
\begin{aligned}
\dot{x} &= v\cos\theta \\
\dot{y} &= v\sin\theta \\
\dot{\theta} &= \frac{v}{L}\tan\delta
\end{aligned}
$$

Having a model of the car, we can move onto the actual Stanley controller. The goal of this geometric controller is to provide a steering angle, δ, which will allow the car to correctly follow the desired path. How does the controller do this you ask? Well, it works by trying to minimize the *heading error*, which is the the difference between the current heading angle (θ) and the desired heading angle. No heading error means the car is exactly parallel to the path. But, having the car just be parallel to the path is not enough, since the car can be parallel to the path but translated away from it (i.e. the car follows the direction of the path, but is not exactly on it). Therefore, the controller also looks to minimize the *cross-track error*, which is the distance from the car to the path. The steering angle is chosen so that both of these errors are minimized. The following diagram illustrates this: (Note that since the bicycle model does not take into account the axles holding the wheels or any real dynamics on the wheels themselves, we can abstract the 4-wheels away and just view the car as a 2-wheeled kinematic object, hence the name *bicycle model*)



The heading error is labeled as θ_e and the cross-track error is labeled as e_{fa}. Okay now that we know what we are trying to minimize, how exactly does it get done? Good question. This is where the math comes in. Instead of providing the derivation details (you can find them in the

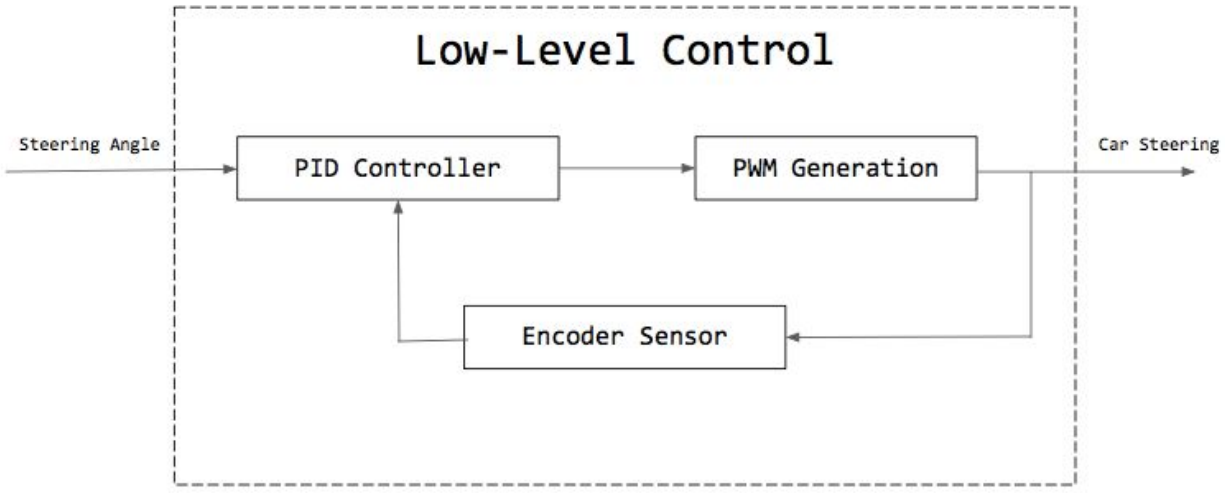
other Stanley controller papers, such as this one), we will just provide the results:

$$\theta_e = \theta - \theta_p$$
$$\delta = \theta_e + arctan\left(\frac{ke_{fa}}{v}\right)$$

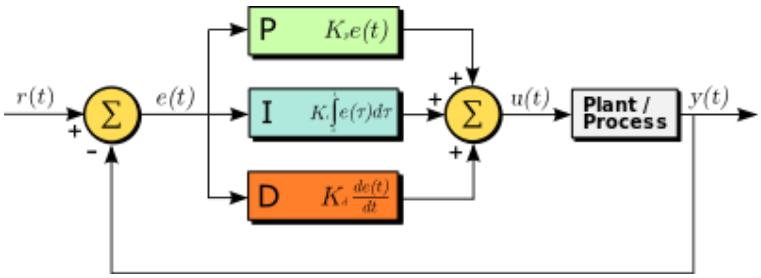
where θ_p is the heading angle of the point (c_x, c_y), k is a parameter that we can choose (varying this parameter is called *tuning* the controller), and v is the speed of the car.

Low-Level Control

The low-level control block takes in the generated steering angle from the Stanley controller block and ultimately outputs the signals to physically steer the car. There is actually another control loop within this block, which can be seen here (notice the lovely *feedback*):



The steering angle is fed into another controller known as a *Proportional-Integral-Derivative (PID) Controller*. This is one of the most common controllers due to its simplicity. The controller uses three parameters: K_p , K_i , and K_d. Each of these parameters are used as scaling factors to terms related to the error signal (i.e. the difference between the desired signal and the actual, observed signal). The terms are then added together to form the output of the controller. This is detailed in the following diagram (taken shamelessly from Wikipedia):



We have decided to incorporate this additional feedback loop within the controls scheme in order to fine-tune our implementation. Without any sort of low-level control, we would be trusting the translation of the steering angle (which is just a value on a computer) to the proper motor PWM (which is the signal sent to the motor controlling the steering) and ultimately to steering column's final state (which is what the car will *actually do*). Now, Arduino is great and all (did you know that Arduino means "little buddy" in Italian? Pretty neat, if you ask me), but we really *should* have some sort of control mechanism to make sure our little buddy and its pal (the motor) are doing what we need them to do. Now that you kind of get the gist of what we are doing, we will go into more detail about how to actually implement any/all of this stuff on the Autonomy Lab car/scooter.

Getting Started With the Mobility Scooter

Now that you have some understanding of the underlying control theory, we can move on to testing your implementations directly onto the mobility scooter. We will first start by just explicitly stating what commands to enter to get the scooter to drive autonomously. Then we will go into how to configure specific parts files and packages to test various new changes you have created. Finally, we will go over some of the tools ROS has to to help debug the output of the scooter after a test run.

Running the Mobility Scooter

When you first use the Alienware laptop, or any device connected to the scooter, you must first change your workspace to the *catkin_ws.* This workspace has already been created for you, or if you are using a seperate device, must be created in order to utilize ROS. If you are not familiar with ROS at this point, a basic comprehensive guide on ROS has been made by Kevin Kerliu that will walk you through the fundamentals of navigating and using ROS. After you have changed workspaces, you now have access to simple ros commands such as *roslaunch, roscd, rostopic,* ect. Run the command:

```
roslaunch scooter_nav scooter_nav.launch
```

to launch the package that configures the entire scooter to start. This process may take a few moments. The final screens that show up will be the RVIZ window that shows the scooter and all of its sensors in a blank map. RVIZ is a ros tool that allows for visualization of data coming through the sensors as well as background algorithms running to perform calculations for localization and path planning. With RVIZ open, you can set a goal for the scooter to move to, look at different sensor data streams, localize the scooter in a new environment and more. The left hand side of the RVIZ window gives access to all of the ros topics that are being published and subscribed to and can be visualized on the map if you select them. To exit out of the program, simply hit Ctrl + C in the terminal.

Editing Files and Configurations

It is assumed at this point that you are familiar with basic ROS concepts such as topics, services, and nodes. Considering how this is a Controls guide, we will go through files and packages that relate to changing the control scheme of the scooter. One of the fundamental files is *cmd_to_ctrl*. This package is responsible for sending the velocity and heading angle to the arduino for translation. Any changes here will modify how the scooter interprets incoming data from sensors and other nodes within the network. One way this file has been changed in

the past is by writing different subscribers to get various topics such as heading angle, the desired heading angle, and goal coordinate to feed into the Stanley Controller method implementation. The output is then fed into the arduino through the *cmd_to_ctrl* topic. Here, you can write your own implementations of controllers. A tip if you decide to write your own files is to make sure that the appropriate publishers and subscribers are created. A common mistake is to add the file to the network and run it to see that your file has not changed the behavior of the scooter at all. This is most likely due critical information not being published to your file or your file not publishing its calculations for the network to use. Another tip is to look at the network chart of the system. This will allow you to find where information is coming from and being passed to. The chart will show which nodes are publishing to topics and which nodes are subscribing to certain topics. Using this tool will help you configure any new files you would want to add to the network.

ROS Debugging Tools

This section will go through several ROS tools to help you debug output when launching ROS. The first few tools will help you understand the configuration of the entire network by retrieving metadata of ros structures. The first few commands are:

```
rostopic list [topic_name]
rostopic info [topic_name]
rostopic echo [topic_name]
```

The first command lists all of the topics active when you launch a roslaunch file. It will display every topic being either published to or subscribed to. The second command will list metadata about that topic, such as the data type it communicates with, the frequency which it is published to, ect. The last command displays every message in real time of the topic passed to it. This tool is useful when you are running the scooter and want to see values such as the control steering angle change in real time as the scooter moves. A whole list of command line tools in ROS is available at their wiki. It is recommended that you understand how to use these tools as they will make debugging efficient.

Further tools are programs that have already been installed for you. Rqtplot is a useful tool to graph the results of a scooter run. Plotting two topics with respect to time and to each other can help determine at what values undesired behavior occurs. Rosbag is another tool to help capture data for analysis. Rosbag allows you to record the data from every topic as you make a run on the scooter. This recording can be played back later on the scooter to replicate the exact run or be used to analyze specific topics at various times during the run. A rosbag to csv program has already been installed on the Alienware to help with easy conversion and extraction of data. All of the information on ROS commands can be read on their wiki with further insight.

Current Status

This section is meant to detail the current status of the Controls team (as of the date at the top of this document). We will go into each of the subsystems individually.

State Estimation

State estimation relies on encoders, visual data and IMU data to work properly. Because we haven't had access to the lab to actually get data from the encoders or the IMU or the visual system, we haven't been able to implement any sort of state observer. [I'll go into more detail soon]

Stanley Controller

Currently, the Stanley Controller has been implemented in the getHeadingAngle() function under `cmd_to_ctrl/src/cmd_to_ctrl.cpp` . In this function, both errors are calculated from the desired path point and the current state (which is currently only retrieved from the ZED camera). As of right now, there seems to be a saturation error, which sends the steering angle to its maximum value almost as soon as the controller is activated. We are currently in the processes of fixing this error. After this issue is resolved, we plan on moving onto treatment of the Low-Level controller.

Low-Level Control

A map of steering angles in software to the actual steering column angle is in the process of being made. This is so that we know what the *actual* steering angle is (like, in reality) depending on the *proposed* steering angle (like, the one that the computer/software thinks the steering angle is). Once the Stanley controller is fully fixed, we plan on tuning a PID controller to ensure that the car achieves the steering angles that we want it to achieve.