**EEC195: Autonomous Lane-Following Robot**

**Project Report**

**Contributors: Abdallah Hashem, Upamanyu Kashyap, Ralph Shehayed, Samuel Miller**

**Table of Contents**

|  |  |
| --- | --- |
| **Section Title** | **Page(s)** |
| Project Objectives | 2-3 |
| Hardware Documentation | 4-14 |
| Software Documentation | 15-22 |
| Design & Performance Summary | 23-24 |
| References & Acknowledgements | 25 |

**Project Objectives**

When we set out to create our robot, we had two primary objectives: To create a lane following robot that runs through a track as fast as possible and to add the capability for obstacle avoidance. Given the knowledge we gained from the line-following car with OpenMV, the strategy we developed for the first objective was to recreate and optimize this for a lane rather than a line, and to do it using a Raspberry Pi with the official Raspberry Pi camera. We chose to use a Raspberry Pi for its processing speed as well as its ability to be interfaced with other components easily. We picked the Pi over a Jetson Nano, which would have had too steep of a learning curve to properly utilize its more advanced capabilities within the given time frame. Implementing a faster car also meant developing a robust steering control algorithm, as well as a variable speed controller to better approach straights and turns in the track. This also meant optimizing the setup of the camera, as well as the images it collects. This was done by customizing the resolution and the amount of data collected in a given time interval, such that we have data that is accurate while retaining a fast update rate/processing speed. The second objective, obstacle avoidance, would be supplemental to the other work we do. It would build off of the components we already have initialized and interfaced, with the addition of an ultrasonic sensor to measure the distance to an object placed in the lane. This involves a software algorithm to translate the data taken from the sensor and determine if the car is being faced with an obstacle, and then inform it to stop accordingly.

However, an auxiliary objective became apparent as we started to go through the design process; Running the code and restarting the car became quite time consuming, given that the motor would continually spin after the code execution was interrupted. We wished for a way to switch from self-driving mode to manual mode to override and take control of the car in order to more easily debug the code. From this realization, we decided to implement a feature to drive the car with a video game controller as well as switch between manual and self-driving mode instantly. This involves interfacing a Playstation controller wirelessly via Bluetooth to the Pi, and programming methods to process and map each button press to a specific action with the car. Additionally, it requires an interrupt feature to detect when the user wishes to switch from manual to self-driving mode (and vice versa), at any point in the running of the code.

**Hardware Documentation**

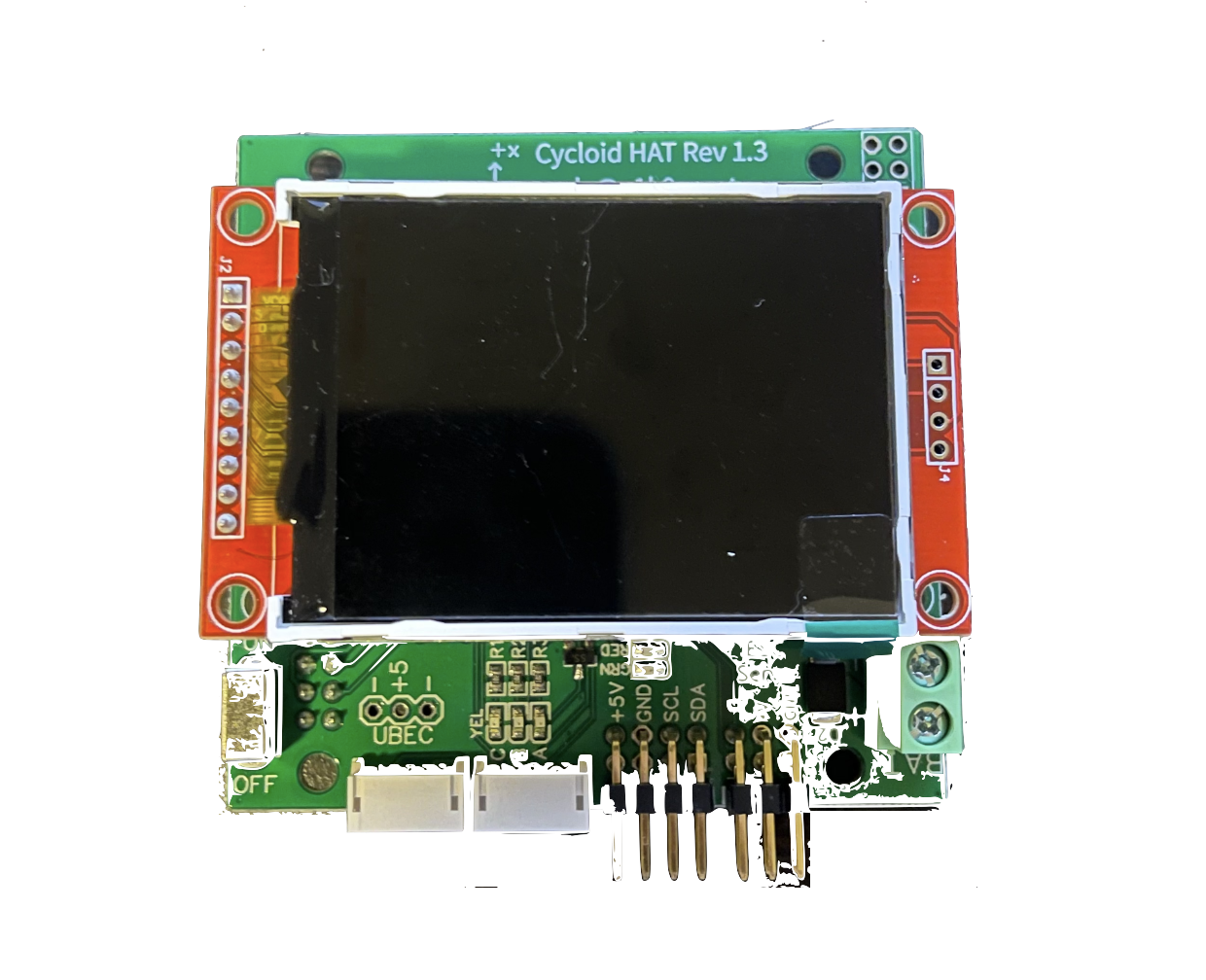
**Computing Hardware**

The computing hardware we picked was constrained by our objectives and time constraints. Initially, we wanted to use an Nvidia Jetson Nano, which would provide us with significantly more computing power than other options. However, using a Jetson involves a fairly steep learning curve, given that the user interface is much less intuitive than alternatives. It is for this reason that we elected to use a Raspberry Pi. The Pi gave us a lot more computing power than we had when using an OpenMV, yet it still had relative ease of use, given that we could install a version of Linux on it. Furthermore, the Pi allowed us to take advantage of pip, which is a very simple package installer integrated into the Pi’s terminal.

Even though the Pi has a lot of GPIO (General Purpose Input/Output) pins onboard, we needed additional hardware to allow us to control the vehicle. Controlling the DC Motor and Servo onboard the car requires the use of PWM (Pulse-Width Modulated) signals. These digital signals operate at a frequency of 100Hz, and the width (time interval) of the “high” part of the pulse determines the action that the car takes. The issue with the Raspberry Pi is that it is incapable of outputting accurate PWM signals on its GPIO pins. This means that the car’s ESC (Electronic Speed Controller) cannot understand the input actions we provide it. Therefore, we needed an alternative to provide more accurate data.

Fortunately, the Pi is capable of using I2C (pronounced I-squared-C, meaning Inter-Integrated Circuit). I2C is a communication protocol that allows different components to communicate with one another. I2C requires four pins. The first two are power, and reference to ground. The latter two are SDA and SCL, which stand for Serial Data and Serial Clock. Serial Data sends out signals and commands, and Serial Clock aids in timing of actions. We needed an additional board capable of driving PWM signals, and receiving commands through I2C.

In the first week of the quarter, half of the team made a trip to Oakland, California, where we attended a DIYRobocars event. This quarterly event is a robotics expo, where various businesses come to demonstrate their products. The event included an autonomous vehicle race on a lane track, which was the biggest draw for us to attend. At this event, we got the chance to speak to experts who were building very fast autonomous remote controller (RC) cars. Through this, we gained insight as to how most people program robots, and which tools they would use. When we talked to one of the competitors, Andy Sloane, he provided us with a lot of guidance and information we applied in our project. He also noted that he had the same issue with PWM signals. His solution was a custom PCB that he called “Cycloid.” This PCB is a form of a HAT (Hardware Attached on Top), which provides additional functionality through the use of the Pi’s onboard GPIO pins. Cycloid had multiple functions; Power supply, an LCD display, an IMU, a motor, and servo driver. Most importantly, all of these components communicated over I2C.



Andy was kind enough to send us two of these Cycloid boards at no cost. However, when attempting to program the HAT, we found that it was difficult to interface. The documentation wasn’t the most clear, and we didn’t have enough experience to figure out how to use it ourselves. After two weeks of attempting to use Cycloid, we pivoted to a new option: The Adafruit PCA-9685.

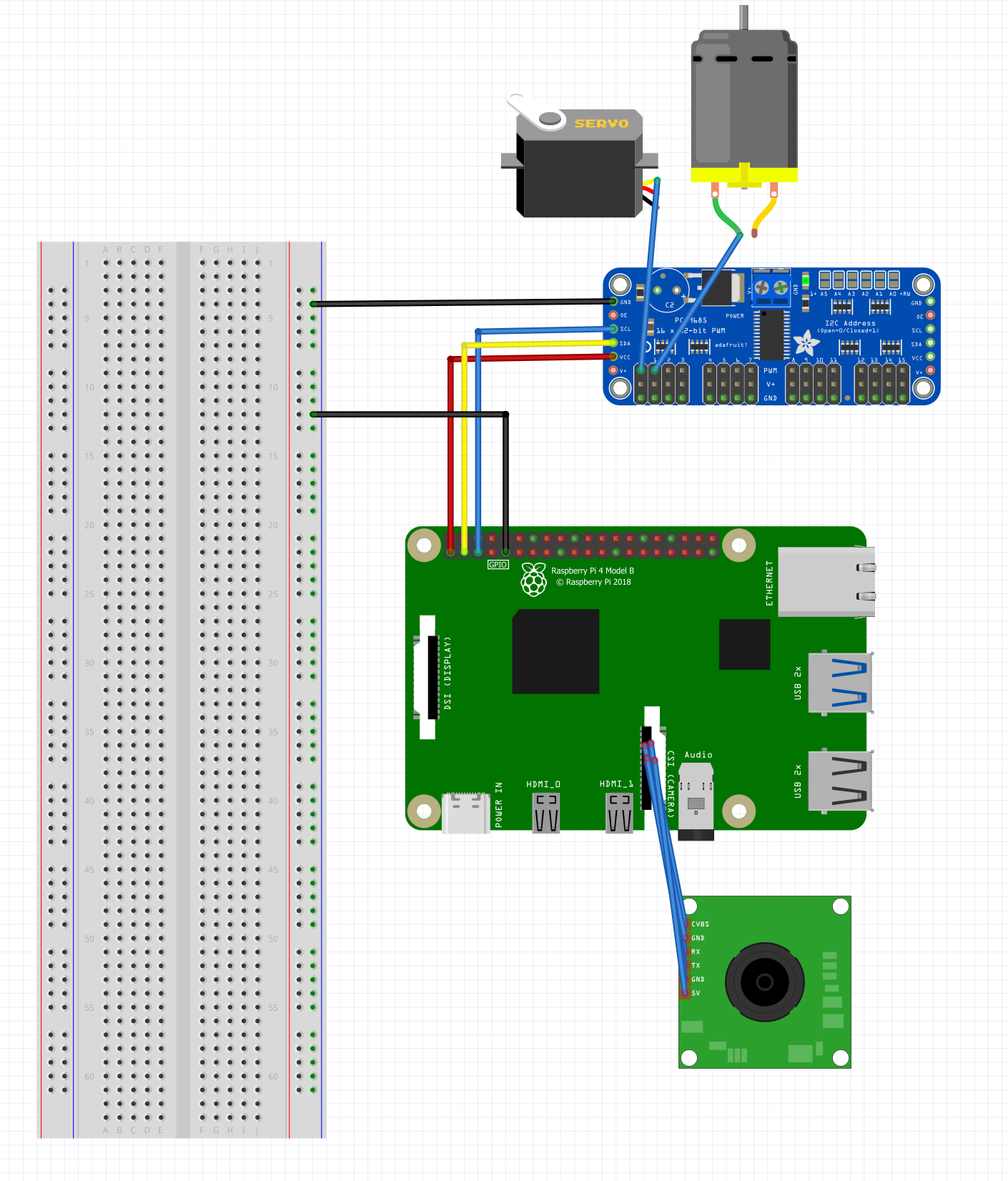
PCA-9685 is a 16 channel I2C based servo/motor driver. It came with well documented libraries that can be installed from adafruit using pip, and made the implementation significantly more simple, albeit less advanced than Cycloid. We took advantage of the Adafruit ServoKit library, which allowed us to map simple throttle and steering angle values into PWM signals that can be interpreted by the Traxxas ESC.

During winter quarter, we used the oscilloscopes in our lab to determine the control scheme that the Traxxas used. The results of our measurements are shown in Table 1. These pulse widths are what we based our control scheme on, and they are what we used to program the speed and steering of the car.

|  |  |  |
| --- | --- | --- |
| Table 1: Control Scheme for Traxxas Rustler | | |
| Component | Action | Pulse Width (ms) |
| DC Motor | Drive Forward | (1.5-2] |
| Neutral | 1.5 |
| Reverse/Brake | [1-1.5) |
| Steering Servo | Full Left | (1.5-2] |
| Straight | 1.5 |
| Full Right | [1,1.5) |

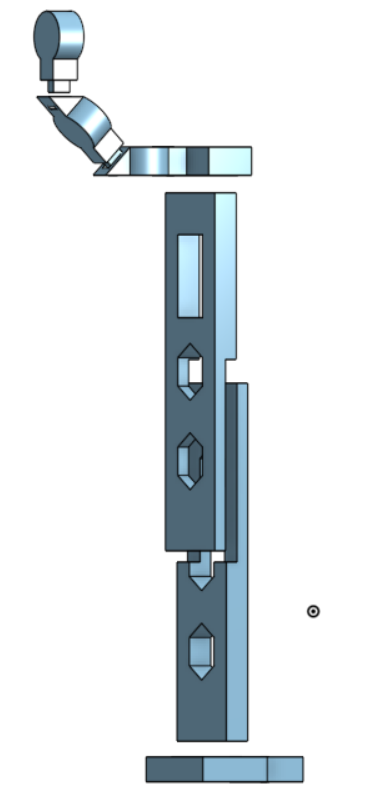
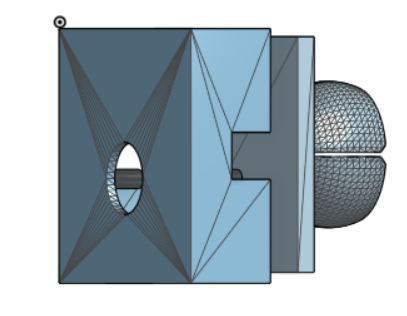
The next element we needed was a wireless controller. We ended up selecting a PlayStation 4 DualShock controller. This is because we had easy access to one, and it has a host of well documented libraries that we can take advantage of. Interfacing the controller required us to pair it to the Pi’s internal bluetooth module, and then use libraries to detect events and actions taken on the controller (explained in software documentation).

The last two elements of computing hardware we used were the official Raspberry Pi Camera, and an HC-SR04 ultrasonic sensor. The camera was, of course, the camera we used to capture images. Its integrated PiCamera library made it incredibly simple to interface, and required little effort from us. The HC-SR04 had a little more detail. It has four connections it needs; Power, reference to ground, a “trigger” pin and an “echo” pin. The sensor works by sending out sound waves, and measuring how much time it takes for these waves to return to the sensor. Knowing the speed of sound and the time taken to see a return signal, we can compute the distance from the sensor to an object. This is what the trigger and echo pins are for. The trigger pin tells the sensor when to send out a wave, and the echo pin measures the return wave. Software is then needed to compute the distance to the obstacle detected.

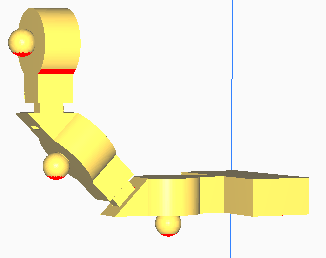


**Mounting Hardware**

All of the mounting hardware used on the vehicle was created by 3-D printing. There were four main components: The camera stand, Raspberry Pi stand, PCA9685 board stand, and the custom base to hold everything on the car.

The camera stand ran through three iterations. The first was a fully articulated stand made up of ball-and-socket joints. The idea behind this design decision was to have the ability to change the height and angle of the camera with ease. However, while these joints made it easy to adjust the camera’s angle, they were not sufficiently sturdy. The outcome that this created was that the camera feed was not dependable, since the angle at which the camera pointed was inconsistent. The car would shake the joints too much, and without glue they would not hold their place. The second iteration was a 15-inch tall camera stand that pointed straight down. This provided the rigid structure we needed, but removed our ability to adjust the angle of the camera. So, we re-designed the mounting hardware a second time. This final iteration retained the 15-inch elevation from the second iteration, while also providing us three options to place the camera: straight down, 45 degrees, and 90 degrees. 

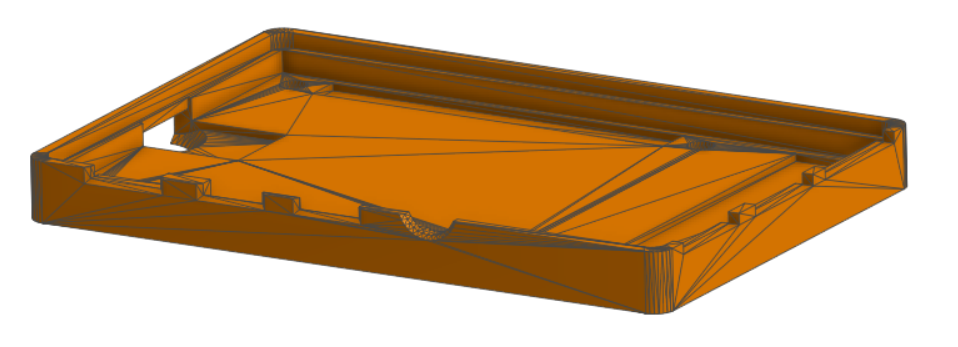
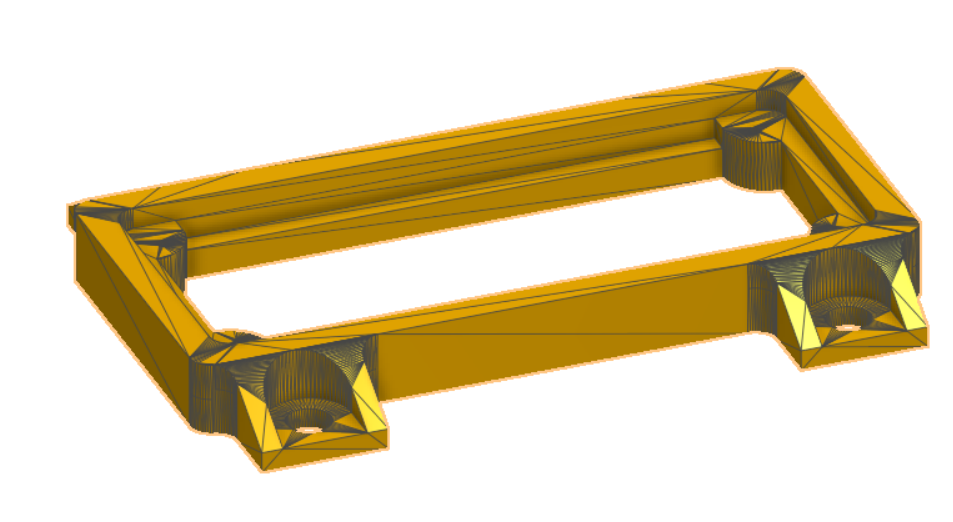
Camera Case

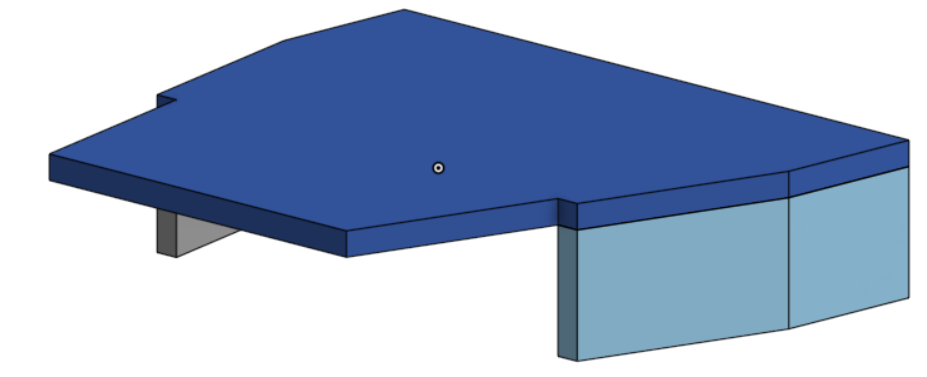


Camera Stand Without Ball and Socket Joints

Head of Camera Mount with Ball and Socket Joints

The Raspberry Pi stand was based on an open source file for a complete case. This design was edited so that the edges of the case were not as high. This allowed us to easily access the board while it was still connected to the car. The PCA9685 stand was an unedited open source file. It was a simple stand with screw holes that allowed the board to be tightened down. The last piece, the base for the other stands, was a simple custom made piece. It is a platform that covered the car battery, and allowed us to attach the raspberry pi and PCA9685 to the car and keep a streamlined appearance.

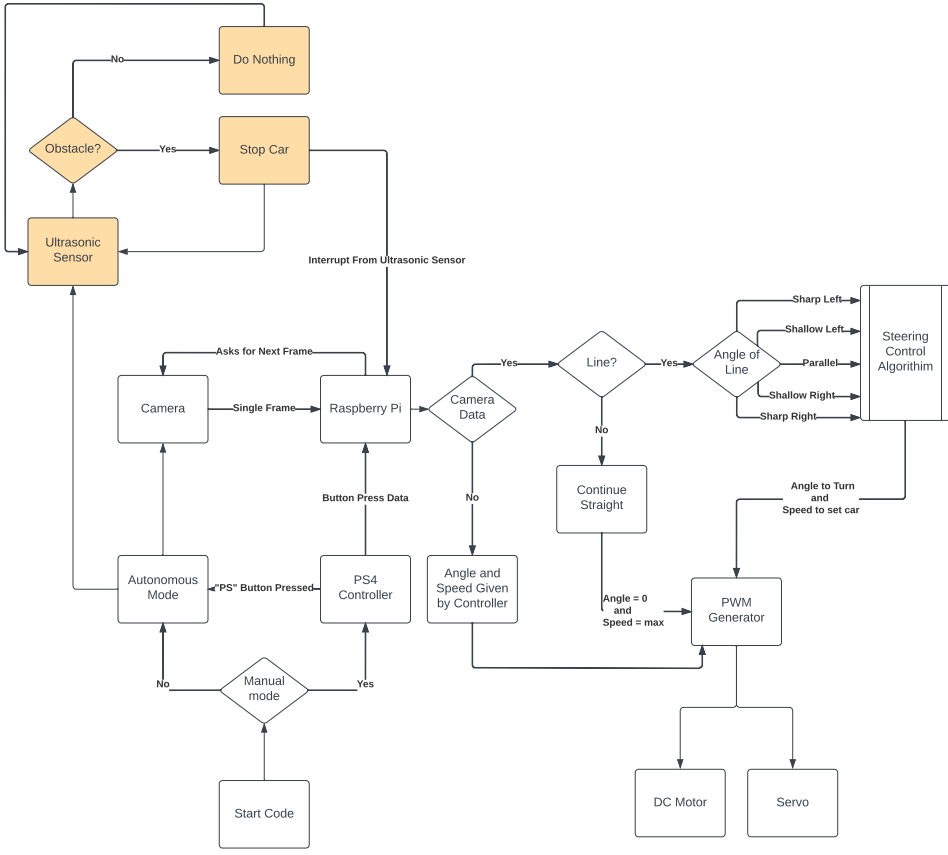




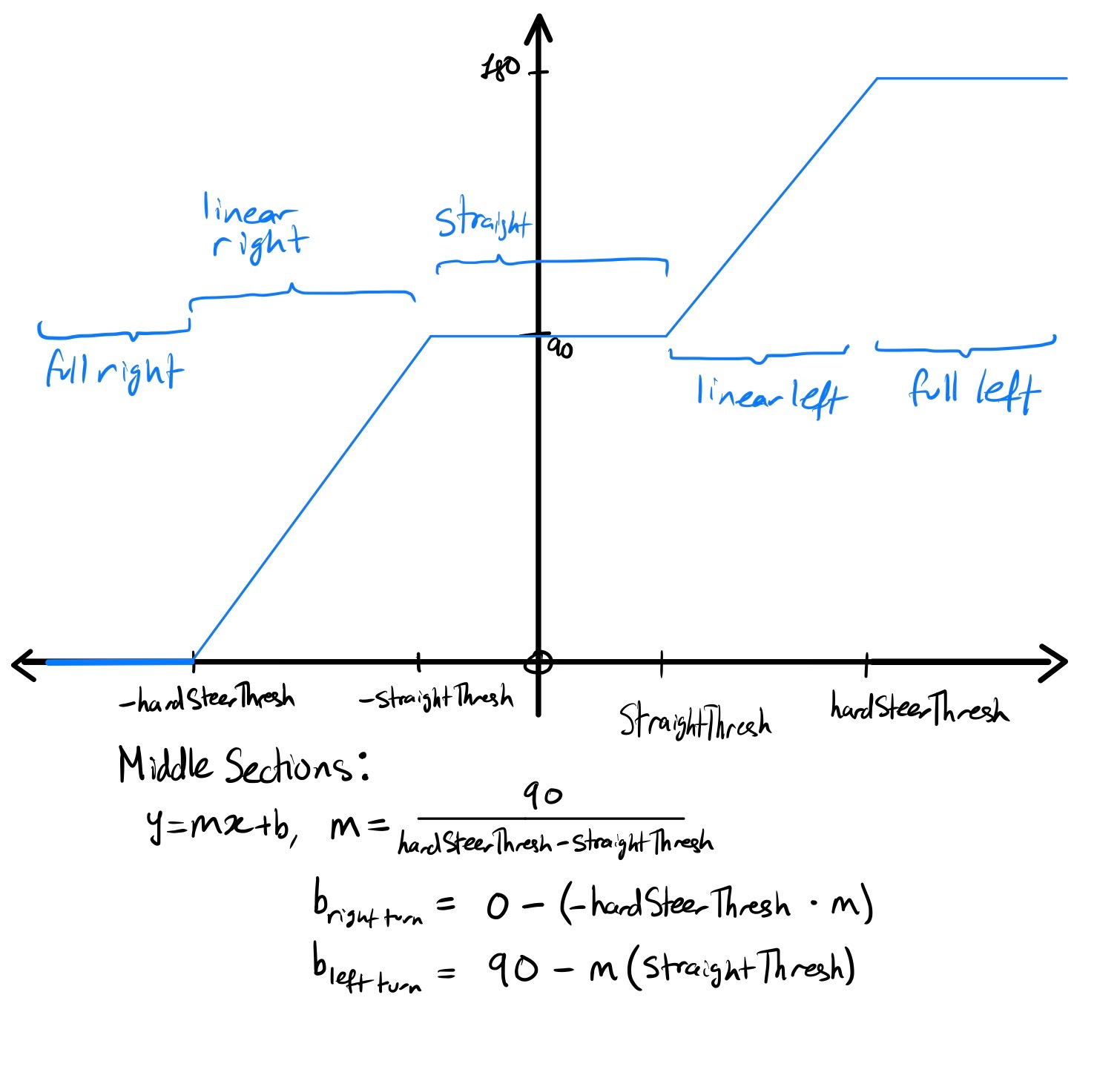
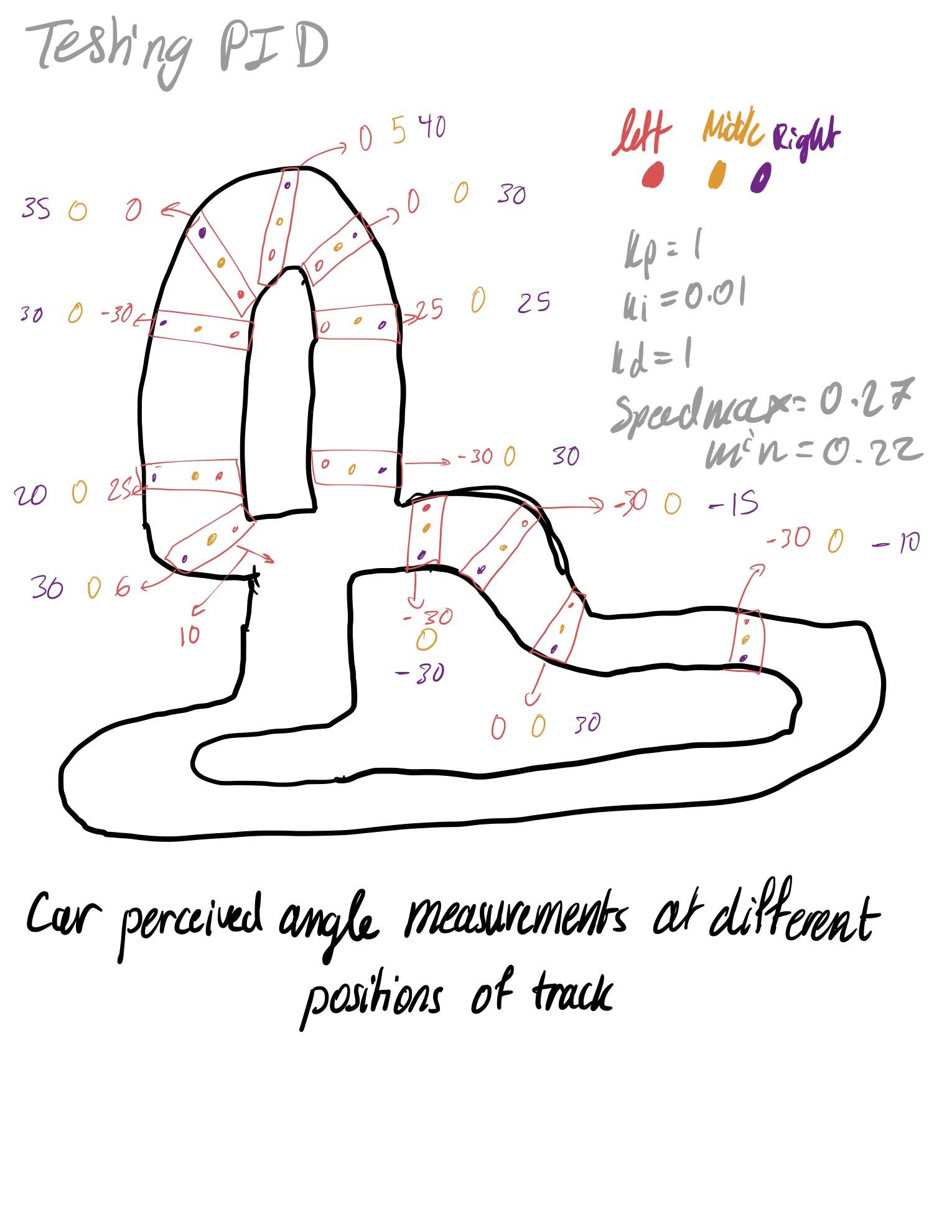
Base and Battery Cover

PCA9685 Stand

Raspberry Pi Stand



Control Logic Block Diagram

****

**Software Documentation**

We spent a tremendous amount of time making adjustments and additions to our algorithm, while optimizing it for higher performance. We started by interfacing the sensors, the servo and the motor with the Raspberry Pi. This was particularly time consuming since we ran into lots of issues while interfacing some of the components. Our approach was to research and utilize available libraries that enabled us to quickly interface components, while making sure that our code was legible and easy to dissect.

**Libraries**

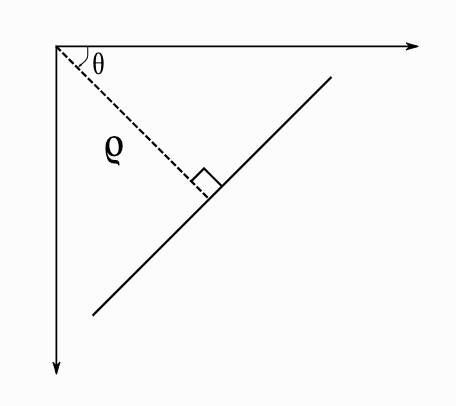
We have utilized the following libraries in order to achieve our goal of lane following

* **The PiCamera library** was used for the camera initialization function referred to in the code as CamInit.py. The library was used to switch on the camera and take a snapshot that would be later analyzed and used for the car’s control system to take decisions.
* **The CV2 library** was crucial to achieve lane following with a high level of accuracy. We used the library to perform image processing. It allowed us to filter out all the objects besides the track lanes, by converting frames to grayscale. After that, we relied on the HoughLinesP function in the library to detect line segments in the binary filtered snapshot taken by the camera. The function performs a probabilistic Hough transform.
* **The Numpy library** was used for the arctan2 calculation for the line segments outputted by the HoughLinesP functions. This was particularly important when calculating perceived angle of lines on the track. We chose Numpy instead of Python’s internal math library due to its higher speeds in matrix computations.
* **The Adafruit ServoKit library** was mainly used for controlling the servo and motor on the Traxxas. The library made it easy to specify the angle at which we would like to have the Traxxas wheels turn. We were able to do so by setting an arbitrary actuation range of 180. In practice, this means that an input of 90 degrees makes the car go straight, whereas zero degrees denotes a full right and 180 degrees denotes a full left turn. The function also maps out all the angles in between to the pulse width needed for the car to operate as desired. The library is also capable of controlling a motor. A throttle value of 0 denotes neutral, throttle = 1 for full forward speed, and throttle = -1 for full backward speed/brake. We can also input values between the ones mentioned to get to our desired speed.
* **The Board & BusIO** libraries are used to send signals to the correct I2C addresses on the boards
* **The PyPS4Controller** library was used to provide hooks for PlayStation 4 controller using python. We used library functions to create a custom class for the controller. In the class, we defined functions that would run based on the value perceived from the PS4 controller when paired over bluetooth and sending signals. We can read those signals using *MyController(interface="/dev/input/js0").listen()* which allows the code to listen to the controller inputs and run the code as in each controller button’s respective function definition.

**Image processing**

The main loop of the program iterates through different frames in the continuous camera capture. We convert each frame into an array to filter all the unnecessary objects and focus on the track. We’re able to do so using the *cv2.cvtColor(*) function. This function allows us to switch the image to grayscale which makes it easier to identify the white tape used for track. We then refined the image using Gaussian Blur (*cv2.GaussianBlur()* function in OpenCV). After blurring the image, we used the cv2.Canny() function, which implements the popular edge detection algorithm developed by John Canny. We chose to use this function due to its capability to find the intensity gradient of an image, have non-maximum suppression to remove any unwanted pixels which may not constitute an edge, and Hysteresis Thresholding which decides which edges are really edges and which are not. This is done by using a minimum and maximum threshold value to make a decision. The output of the *Canny()* function is the outline of the track.

Hough Transformation solved a major challenge we had in the design, which was detecting shapes on track (rectangles in our case). This decision was taken based on the way Hough Transformation works. A line can be represented in parametric form as, p = xcos(theta)+ysin(theta) where p is the perpendicular distance from origin to the line, and theta is the angle formed by this perpendicular line and horizontal axis measured in counter clockwise, as seen in image below. However, due to the need for fast computations to ensure the car reacts to changing track conditions quickly, we decided to use Probabilistic Hough Transform, which is an optimization of Hough Transform since it doesn’t take all points into consideration. Instead, it takes only a random subset of points that is efficient for line detection. This required us to decrease the thresholds for the transform to read the same amount of data. The function outputs the lines which we then use in order to add all small angles generated by each line outputted by the Hough transform. This allows us to understand how steep the curve is and react accordingly.



**P vs PD vs PID Control**

To get started with following a track, we aimed to first reach the limits of a proportional controller, then add a derivative controller, and lastly an integral controller. We were initially curious to see how the system would behave if we sent the Hough Transform’s angle output directly to the wheels of the Traxxas. Although the system works well in that manner on small curve adjustments, it wasn’t able to take a curve since in order to take a strong curve as prompted by the perceived angle, the car would need a wheel that can rotate 180 degrees (which is obviously not possible). We decided to create a function that maps the perceived angle to the angle of the wheels. We decided to go with a linear steering control mentioned above as *SteeringInit.py* . The module determines the piecewise function used to map perceived angles to steering based on a linear gradient between two thresholds; straightThresh & hardSteerThresh. In addition to that, we wanted the car to run on different speeds as it’s taking a curve versus going straight, so we added a speed controller in the function. We input the hardSteerThresh, straightThresh, maxSpeed, and minSpeed variables to the function which then outputs the linear steering slope and the speed at each respective decision.

Since the track is smooth with curves and straights, we mainly relied on derivative (D) control since it provided us with the most accurate measurement and decreased the oscillations acquired using the proportional (P) controller. Our Proportional-Derivative (PD) controller worked great since it was able to stay on track most of the time. We realized that our steady-state error was getting in the way of the car completing an infinite number of laps without missing a single one. We realized the need for the integral (I) controller to decrease the steady-state error. In addition, the car was able to respond to the deviations in track conditions, which is mostly due to uneven lighting.

**Linear Steering Control**

The outputs of the SteeringInit() go to the inputs of Steering() which decides whether the car should go right, left, straight, or any angle in between. We decided to run the logic based on the few variables mentioned above to make testing and optimization easier. The logic that determines whether the car goes straight is that if the absolute angle is less than straightThreshold and greater than zero. The reason we did that is because the Hough Transformation output angles range from -90 to 90 with zero denoting a straight line, negative angles denoting right turns, and positive angles denoting left turns. The variable straightThresh sets the boundary of what we consider as acceptable angles to maintain going straight. The right and left steering logic is that if the angle is less than hardSteerThresh, while also being greater than zero, the car needs a minor correction in either direction to stay in the middle of track. We use the linear function to map how much of the perceived angle should be corrected using the car wheels. The variable hardSteerTresh is the boundary of what we consider would require a turn to either fully right or fully left depending on the sign of the perceived angle and if it exceeds that threshold. Otherwise, the system would make minor corrections to keep the car on track.

**Sampling**

The biggest challenge is to make sure the car is sampling enough frames to make correct decisions at the correct time. If the sample rate was low, the car would react to the curves late and would lose the track, or would completely miss the lines when taking a new capture. We initially tried using time intervals in order to take consistent measurements, but it was extremely inconsistent which led us to utilize the number of samples as an interval. We also needed to optimize the code to make sure that the logic makes a quick decision before capturing the next frame. Optimizing the code became an eighth order problem, since we had to adjust: (1) camera angle on car, (2) Kp, (3) Ki, (4) Kd, (5) minSpeed value, (6) maxSpeed value, (7) hardSteerThresh, (8) straightThresh. They must all work in unison in order to follow the track at the desired speed. In the next section, we will further explore ways we tried to mitigate that issue.

**PlayStation 4 Controller**

Although the aim of the project was to create an autonomous lane following robot, the team decided to work on manual driving mode in order to aid with testing and to be able to switch back to manual mode when a crash is imminent, or to position it on track. We used a PlayStation 4 controller as the controller for the system. We mapped the buttons on the controller to correspond to different steering angles of the car in addition to driving forward and reversing. However, the PlayStation Logo button was mapped such that when pressed the car would run in autonomous lane following mode. The autonomous lane following mode would run infinitely until we send a KeyboardInterrupt (Ctrl-C on the Pi’s terminal) which would send the system back to manual mode.

**Obstacle Avoidance**

Although our team couldn’t completely integrate the obstacle avoidance with our main code in the time frame we had this quarter, we had obstacle avoidance code that would read the output from the ultrasonic sensor. In theory, when the autonomous code is running, the ultrasonic sensor would create an ISR (Interrupt Service Routine) interrupt to the main code that will make the car swerve to avoid the obstacle. However, this would require us to introduce multithreading, which is the ability to execute two or more simultaneous “threads” of code. This would require a full restructuring of how our functions currently behave.

The avoidance procedure would be based on how far the object is and the speed of the car. To simplify, let’s say the obstacle is on a straight portion of the track where the car is on high speed, then the car will swerve to left/right based on object position and counteract that swerve. However, if the obstacle is on a curve, then the car would be waiting until it’s at a closer distance to avoid it when coming back to the track. Our future work includes incorporating the ultrasonic sensor in our main code using ISR.

**Data Logging**

We saw the need for a data logger to log our data in an easy to see .csv file for debugging. This allows us to better understand the decisions the car is taking. It also allows us to see how well each P, I, and D terms work on track. We were hoping to incorporate a continuous video stream of the car on the track, but due to time constraints we were not able to incorporate that on time. However, we greatly benefited from the data logger data since it allowed us to optimize the car in a fast manner saving time and effort.

**Design & Performance Summary**

In order to achieve the desired performance, we started by deciding the desired speed at which we would like to run the track and adjusting other parameters in order to achieve that goal. We started by setting our max speed to 0.3 (~1.7ms pulse width) and min speed to 0.25 (~1.6ms pulse width) as our desired range of speed for the track. Then, we worked on adjusting the camera such that the shutter speed would be faster than the car speed so we can detect a curve and take the appropriate decision on time. We found that a camera angle around 45 degrees worked best in this situation. Although it would be great to have a larger frame size to be able to see more of the track, it would significantly slow down the logic. Thus, our robot would not be able to take snapshots at the desired frequency and the car would miss curves more frequently. Based on continuous trial and error, we decided that a frame size of 852 x 480 produced best results since the logic speed was fast and we were able to acquire curve steepness on time to make a decision. We then mapped the different angles around the track in order to better understand the thresholds needed for the car to take appropriate actions at different positions. This was the driving force in having a hard steering threshold that allows the car to take hard turns when needed, and the straight threshold for minor adjustments to stay within the track.

**Logic Speed**

The main bottleneck that we experienced with the way the code is set up, is the speed of the logic to make a decision before the next snapshot is taken. When the code runs slowly, the car completely misses important data about the track since it is analyzing a frame that could have passed a long time ago. We worked hard to ensure that the logic runs at much higher speed to support our desired speed objective. Our original code for lane following runs one iteration in 0.017 seconds which would be good for lower speeds. However, we were aiming for a much higher logic speed. After optimizations, the current version of the code runs one iteration in 0.004 seconds, which is 4.25x faster than before. This increase in speed saw significant improvement in the results that we saw on the track.

**Challenges**

The track had uneven conditions which gave unreliable data to the system and caused it to make wrong/undesirable decisions. Unreliable data, such as bad lighting, made the car miss parts of curves, or small white objects on the track which the camera perceived as line segments- contributed harmfully to perceived angle. We aimed to reduce the noise by filtering the camera input through color and gaussian blur before analyzing the frame, however this did not always suffice. We also investigated using a different Operating System- ROS (Robot OS) in order to create a digital map of the track, which the car could use to localize itself at any given point. In theory, this would provide a much higher degree of capability, since the car could veer slightly off track, but return just as quickly and continue. However, we realized quickly that the level of complexity in ROS would set us back too far, and we would not be able to present a fully functioning product by the quarters’ end.

**References & Acknowledgements**

Dr. Lance Halsted

Andy Sloane

Aidan Callahan

github. (2020). *GitHub*. Retrieved from https://github.com/ArturSpirin/pyPS4Controller

github. (2021). *GitHub*. Retrieved from <https://github.com/a1k0n/cycloid>

Thingiverse. (2018). *Thingiverse*. Retrieved from <https://www.thingiverse.com/thing:3017729>