Final Challenge Report: Utilizing Pure Pursuit, Computer Vision, and Path Planning to Navigate a Race Track and City Landscape

Team #9

Karen Guo
Yohan Guyomard
Michael Kraus
Jaclyn Thi
Carlos Villa

6.4200/16.405 - Robotics: Science Systems

May, 13th, 2024

1 Introduction (Karen Guo)

In recent years, the concept of autonomous driving has rapidly emerged as a transformative technology. By using a combination of sensors, cameras, light detection and ranging (LiDAR), and algorithms, vehicles are able to operate autonomously, without human intervention. This revolutionary technology can improve the safety, efficiency, and mobility for how we travel. Aside from cars, this can also be implemented in various vehicles, including drones, boats, and more. With such potential, we see various research projects in this area; for example, Tesla has integrated an advanced driver-assistance system (ADAS) into its cars. This includes semi-autonomous features, where the car will automatically steer, accelerate and break.

In this report, we detail how the concepts we have learned throughout the class allowed us to implement an autonomous car in a race track and city environment. Our challenge was to program our car to race around the Johnson Track (Mario Circuit) and to drive around the makeshift city in the Stata basement (Luigi's Mansion). On the track, our car must stay within a lane and complete a full revolution as fast as possible. In the basement, we must collect shells at different locations while following traffic laws, like stop lights, stop signs, pedes-

trian crosswalks, and driving on the right side.

Since the beginning of this semester, we have learned how to implement various controllers and computer vision and path planning algorithms for our car applicable to this problem: a wall follower, a cone detector, a line follower, car localization, and finally, path planning. By combining all these techniques, we have the building blocks to autonomously solve these problems. For the Mario Circuit, we employed a proportional-derivative (PD) controller and Hough transforms to drive inside a lane. From the camera data, we find the track lines using the Hough transformation and input a line to follow into our PD controllers. For Luigi's Mansion, we detected stop lights using color segmentation, and reached destination points using A* path planning, a pure-pursuit controller, and Monte-Carlo localization (MCL). Our safety controller was also continuously running in both challenges to ensure the safety of the car and people around it.

In order to assess the performance of our car, we kept the following values in mind. Because we were racing around the track, speed was a priority; we wanted to traverse a lap around the track as fast as possible. Additionally, we wanted our car to stay within its own lane when racing. In city driving, we wanted to ensure our car accurately completes its tasks. This meant correctly detecting stop signs and red lights, as well as going to the correct positions. Similarly, we also wanted our paths to be generated and traversed quickly.

To simplify our problem, we made a few assumptions about our environment. First, we assumed that the only obstacles we were dealing with in city driving are pedestrians, stop signs, and stop lights. This is a severe generalization of everything that would be present in a real city. In track racing, we assumed that no other cars will be in our lane; in the real world scenario, we would have to account for merging cars and more.

The remaining sections of our report will detail our technical approach for the two challenges, our experimental evaluation, what challenges we encountered, and what we have learned from this project.

2 Technical Approach - Race Track

2.1 Overview (Yohan Guyomard)

The objective of the race track challenge is to complete a lap as quickly as possible while incurring minimal infractions like crossing lanes or requiring manual adjustment. Thus, we determined that the winning strategy would be twofold: our implementation should be reliable and repeatable to avoid infractions, and "hugging" the inside of the lane reduces the effective perimeter of the lap and results in faster race times. Our approach consisted of a lane detection algorithm and a proportional-derivative controller. We opted for a rather simple controller

and devoted most of our efforts into developing a robust and consistent computer vision routine. Our robot is given the task of staying parallel and at some distance d to the left of the lane it started in. By settings its forward velocity to some fixed amount (up to 4 m/s), our controller is essentially "following" the inner lane of the track and completing laps.

2.2 Lane Detection using Computer Vision (Yohan Guyomard)

Our lane detection algorithm is the backbone of the entire track racing system. It is tasked with parameterizing a line on the robot's camera feed corresponding to the left lane of the track. In order to function properly, it must be robust to extraneous lines and symbols on the track (e.g. lane numbers), other lanes (i.e. only detect the lane it is currently in) amongst a few other edge cases, as shown in Fig. 1. We developed an algorithm which accomplishes this by exploiting key assumptions about the structure of the track and the camera's intrinsic parameters: the lanes are bright white and at a consistent angle on the camera feed.

Figure 1: **Example image of the track.** The robot's point of view is slightly lower in reality, which makes the problem easier (less lanes are visible), but our algorithm is robust to this too.

The first step of our algorithm filters out bright pixels from the rest of the image. This picks up on most of the prominent symbols on the track including the lanes and numbers. The result is generally noisy and filled with gaps, but applying an erosion and subsequent dilation helps "fill-in" the image mask, like in Fig. 2.

Figure 2: **Symbols masked out.** After the first step, background features and non-prominent symbols on the track are masked out.

Next, in Fig. 3 we apply a skeletonize filter which reduces the features in the image to a width of 1 without creating discontinuities. This is useful for the next step which detects lines.

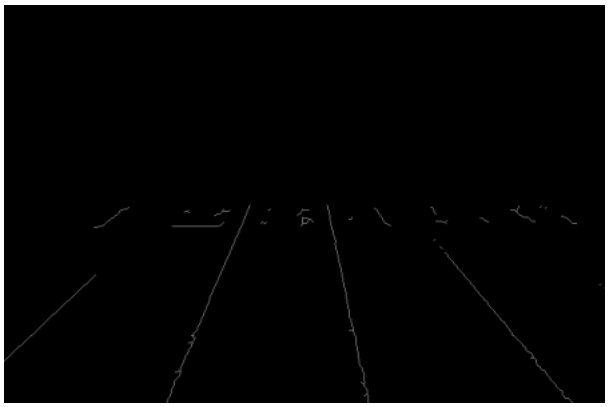


Figure 3: **Same symbols as before but skeletonized.** The lines remain somewhat intact, but features like lane numbers are almost eliminated.

We then apply a Hough transformation in Fig. 4, which parameterizes straight lines in an image.

Figure 4: **Hough transformation.** The unfiltered output of the Hough transform often gives false positives such as counting the same line multiple times.

We found that despite filtering the lanes to a width of 1 pixel, the Hough transform still yielded false positives where it identified multiple lines for where there was only one. Thus, we applied a line-merging algorithm which averages "similar" lines, which is shown in Fig. 5.

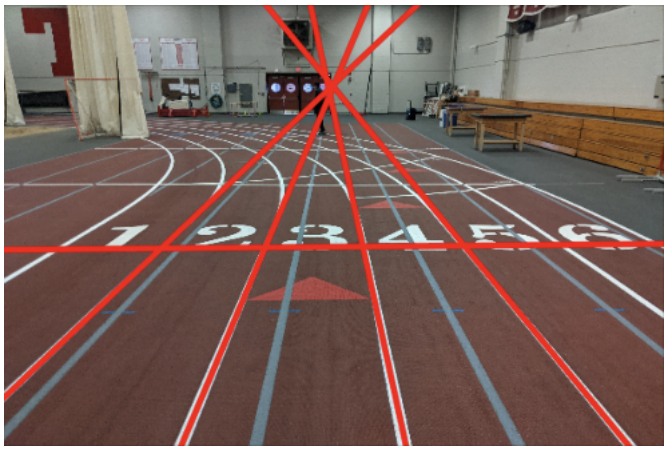


Figure 5: **Merged output of the Hough transformation.** Similar lines are averaged together which greatly reduces the amount of false positives.

Finally, since the camera's field of view (FOV) and distance to the ground are known constants, we can filter out lines that don't match a certain range of slopes. Our final result is shown in Fig. 6.

Figure 6: **Identified line.** Note that it doesn't just take the left-most lane, but rather considers the line that most likely corresponds to the left lane with respect to its slope.

2.3 Lane Following (Yohan Guyomard)

Our controls algorithm is a modified wall follower, as demonstrated in Lab 3. Therein, the robot is tasked with driving forward while remaining parallel and at a fixed distance d from the wall. We accomplished this with two Proportional-Derivative-Integral (PID) controllers, whose control outputs are summed and subsequently dictate the car's steering angle. Tuned correctly, this strategy achieves both goals very reliably despite its simplicity. In Lab 3, detecting a wall was achieved by mapping laser scan data into the robot's Cartesian frame (the data from the sensor is in polar coordinates) and running a linear regression to parameterize a line that corresponds to a wall relative to the robot. For this lab, our computer vision algorithm already computes a line corresponding to the left lane, analogous to a wall, but the effects of perspective distortion make it nontrivial to tune our PID controllers directly on this. So, we repurposed our Homography transformation from Lab 4 – this algorithm maps points on one plane (i.e. the camera's 2D video feed) to another (i.e. the ground). We sample two points on the line from the computer vision algorithm, use the homography transform to map those to "real world" coordinates, then linear regression finds the slope and distance of the lane relative to the robot. Although this approach is mathematically redundant, it greatly simplified the integration between three components we knew to work individually.

We found that a high proportional and derivative term gave the best results. Conversely, the integral term of our angle PID had a tendency to diverge in the long straight-aways, so we omitted it. Moreover, we limited the car's turning angles to 5 degrees to mitigate potential glitches in the computer vision algorithm.

2.4 Simulator (Yohan Guyomard)

For every lab thus far, we've been given a Robot Operating System (ROS) simulator of the Ray and Maria Stata Center basement to quickly iterate on our systems. This proved especially important for our team given the numerous hardware issues we experienced during the term. However, this challenge assumes a completely new environment for which there isn't a readily available simulator. Moreover, the existing simulator has no means of generating a virtual camera feed, which would render half of our implementation un-testable (since it is entirely computer vision). Thus, we created our own simulator of a track and field, complete with a 2D map for visualization and 3D renderer for the camera feed.

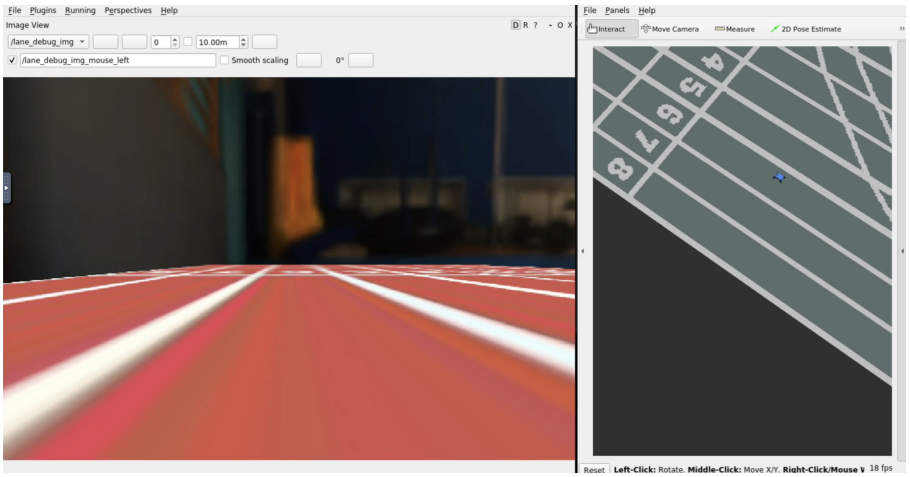


Figure 7: **Screen capture of our simulator in action.** The robot is placed in a virtual environment with a 2D map, as before, but a "fake" video feed is also generated reflecting what the robot might see.

Aside from the learning experience of making a ROS simulator and getting a glimpse into the work that the staff usually does for us, the simulator allowed us to iterate quickly without worrying about issues like faulty hardware or charging batteries to name a few. Notably, our implementation required less than an hour of work on the real-life robot before working as desired.

3 Technical Approach - City Driving

3.1 Overview (Carlos Villa)

City driving proved to be a very challenging problem to tackle. Our goal was to take in any points, plan paths to reach each of them in order, wait for a set amount of time at each one, and follow given rules of the road, such as staying in the right lane, avoiding collisions with pedestrians, and stopping at stop

lights and stop signs. To accomplish this we attempted to integrate most of the work we had done this semester into one system. At a high level, we had a state machine that received goal points and subsequently sent commands to a path planning node that we then sent to a trajectory following node to tell the racecar where, how, and when to drive. Additionally, we employed nodes to handle collision avoidance; we used our safety controller from Lab 3 and stop sign and stop light detectors to follow traffic laws. Due to the varying priorities of control for the robot, these additional nodes could send stop commands to the robot that would overrule any commands from our other nodes. Most of this was aided by localization which helps us maintain a consistent representation of our racecar's position throughout our nodes. A diagram of full integration city driving can be seen in Fig 8.

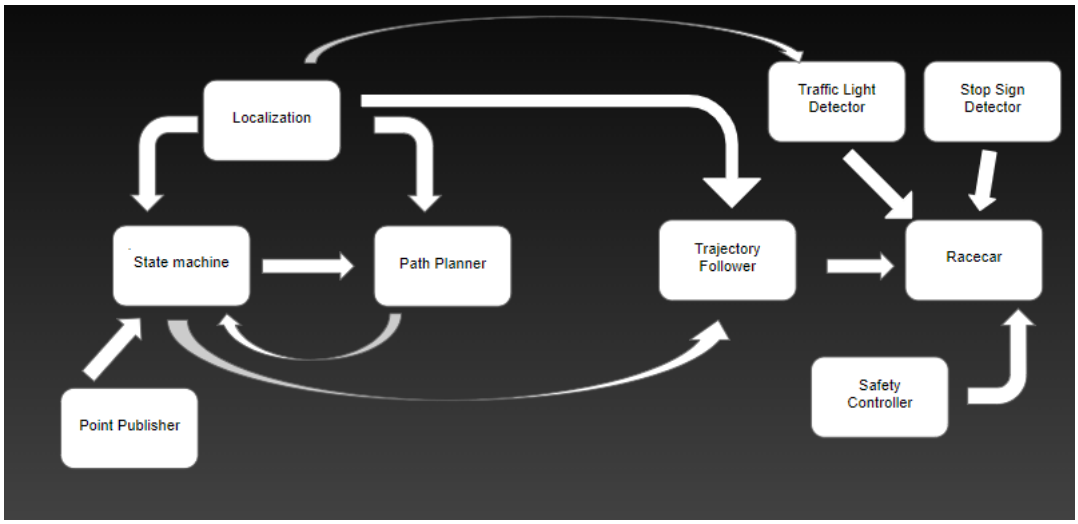


Figure 8: **High level control flow of Luigi's Mansion/City Driving technical approach.** After receiving a set of goal points our state machine coordinated our path planner and trajectory follower to give drive commands to our racecar. Independently, our safety controller, stop light detector, and traffic light detector determined appropriate times to give stop commands.

3.2 State Machine (Carlos Villa)

Our state machine was the brains of Luigi's Mansion/City Driving; it regulated when other nodes would be used and defined the cars active driving. Once the state machine node was activated it had 5 states it operated within. The first state was an "off" state, wherein it waited for goal points to be given to the car, but otherwise didn't perform any other actions. When goal points were given to the car it transitioned to a "Start" state. At this point it formatted the 3 given

goal points as well as our current position as a 4th goal into usable messages to be passed to the path planner. After this, it sent the first of the goal points to the path planner and once it received a path back, it sent that path immediately to the trajectory follower to follow it. As the trajectory follower followed the given path, the state machine would be in the "drive" state. Once the state machine processed that the car had arrived at its goal, it would transition to the "start wait" state. In "start wait" the state machine would tell the path planner to start planning a path to our next goal, and then it would transition to the "wait" state. The "wait" state would wait for a set amount of time or until a path was received from the path planner, whichever was longer, before it transitioned into the "drive" state again by telling the trajectory follower to follow the new path. After all goal points were exhausted and the car returned back to start, it would once again transition to the "off" state where it awaited another set of goal points. An image of this state machine can be referenced in Fig. 9.

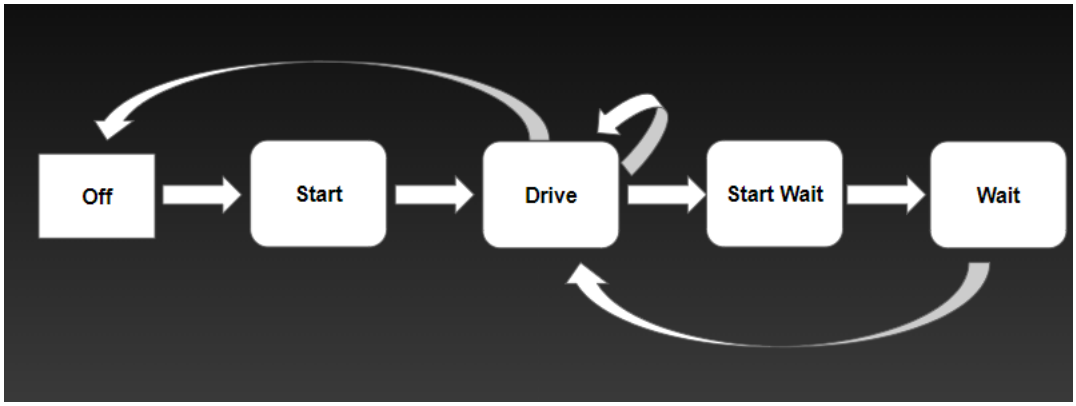


Figure 9: **High level flow of state machine for navigating Luigi's Mansion/City Driving.** The state machine transitioned between 5 different states to command and reflect the actions of the car while completing the Luigi's Mansion/City Driving portion of the final challenge.

3.3 Computer Vision (Jaclyn Thi)

We aimed to use computer vision to enable our racecar to stop for red stop lights and stop signs, each of which present their own challenges. Stop lights had known locations and, when red, must be stopped at until green. Stop signs had unknown locations until race day and required our car to come to a complete stop before continuing. Because these require different stop behaviors, we utilized different computer vision methods for each.

3.3.1 Stop Light Detection

Because stop light locations were known in advance, we aimed to utilize color segmentation (from Lab 4) to determine when the light was red and as well as Monte Carlo Localization (from Lab 5) to determine when our car was within a certain radius of a stop light.

If our car with pose (x, y) comes within a euclidean distance (1) of 1 meter within any stop light i's map coordinates and detects bright red, the city stopping controller would publish high priority stop commands until the red light turned off. Additionally, as seen in Fig. 11, we masked the top and bottom portions of the camera image to only scan for red lights at the height level of red stop lights. This would allow us to better detect red stop lights, rather than accidentally detecting other red objects in the environment.

$$d = \sqrt{(x - x_i)^2 + (y\_y_i)^2} \tag{1}$$

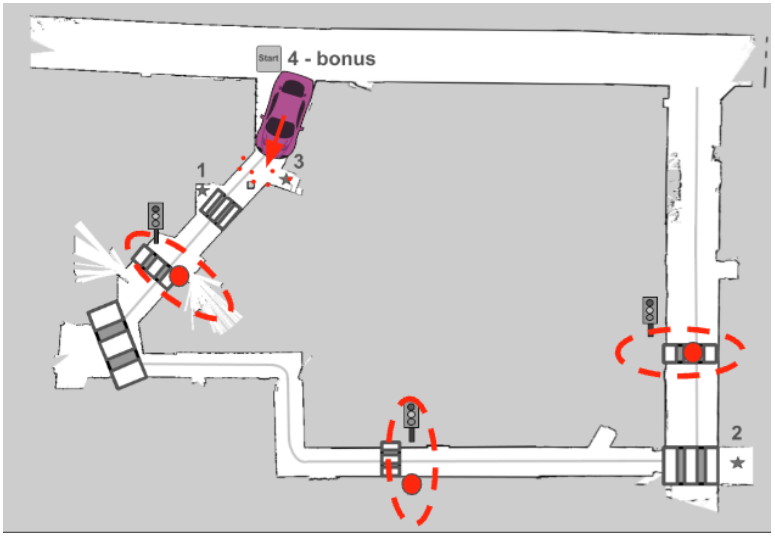


Figure 10: **High level view of car using localization to determine position relative to stop lights.** We first determine the ground truth pose of the car using our particle filter. We then compute the euclidean distance from the ground truth pose to each of the stop light positions, which we use to determine when the car should begin braking.

Figure 11: **Masked image used for detecting red stop lights using color segmentation.** Masking the upper and lower portions of the image allows for only objects at the height level of the red stop light to be checked for bright red. The green box indicates that a red light is detected in that area.

3.3.2 Stop Sign Detection

We utilized a combination of a machine learning model (provided by the course staff) and a homography transformer (from Lab 4) to detect and stop at stop signs. As seen in Fig. 12, the machine learning model returned a bounding box estimate on detecting a stop sign, from which we calculated an estimated pixel location (u, v) for the base of the stop sign. We calculated this by taking the midpoint of the bottom edge of the sign's bounding box and translating it downwards by 1.5 times the height of the box, which reflects the distance of the base of the sign with respect to the face.

The homography transformer then took the pixel estimate (u, v) and published the physical coordinates of the stop sign (x, y) relative to the robot frame. Using this, we calculated the euclidean distance to the detected stop sign and published a stop command for three seconds once the car is within 1 meter of the stop sign.

Afterwards, the car was to continue its trajectory and ignore all stop light detection messages for three seconds before being able to stop for a stop sign again. This was to prevent the car from repeatedly detecting the same stop sign and continuously brake.

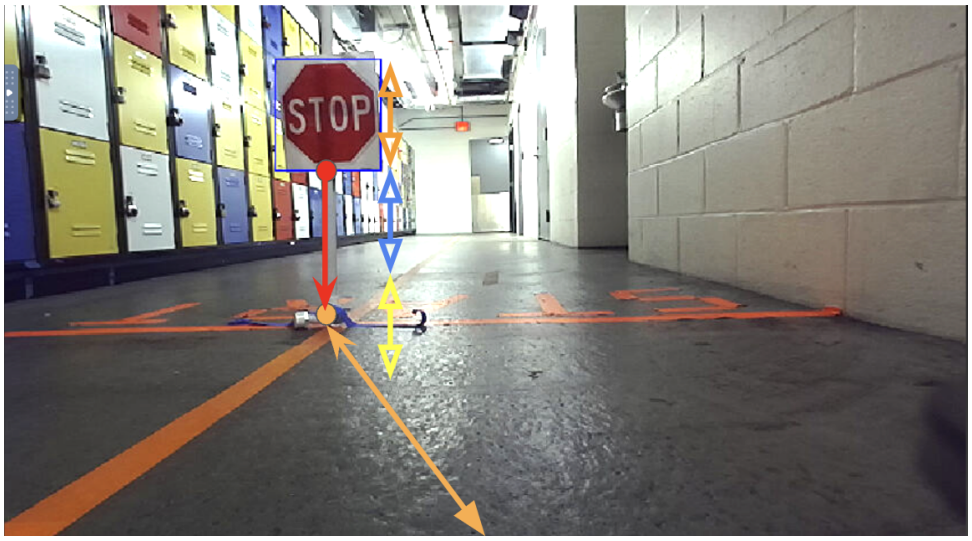


Figure 12: **Machine learning detection of stop signs and prediction of euclidean distance.** The machine learning model returns a bounding box for the face of the stop sign. We use this bounding box to predict where the base of the stop sign is in the image (orange point), and use the height of the bounding box and homography transformation to calculate the position of the stop sign relative to the robot (in orange).

3.4 Path Planning (Karen Guo)

3.4.1 Revised A* Algorithm

From Lab 6, we found that our A* algorithm generally had better paths than Rapidly-exploring Random Trees (RRT), but a slower runtime. Hence, we decided to update our A* algorithm by using map downsampling. To accomplish this, we max pooled the map with a relatively arbitrary pooling size of 5. This allowed us to find paths much faster on a smaller map. Once we found the path, we could then multiply them by the pooling size and transform it back into real world coordinates. We found this significantly increased our path generation speed, which is elaborated upon in Table 1 and Table 2 in the experimental evaluation section.

3.4.2 Three-Part Path

After improving our A* algorithm, we moved to the city driving challenge, where the main difficulty came from staying within the right side of the centerline of the road. We chose to tackle the path planning in three different parts, referred to as part A, part B, and part C. These parts consist of path planning from the start to the closest point on the centerline, following the centerline with an offset using the controller, and path planning from the closest point to the end. From the examples trajectory in Fig. 13, we can see that the green points $P(x_1, y_1)$ and $P(x_2, y_2)$ are our start and end points respectively. Part A refers to the path between the start point and the closest point on the centerline, while part C is the same except for the end point. Part B is represented by the red line segments between the two closest points to the start and end. Also before planning paths, we check to make sure that the closest index to the start is greater than the closest index to the end. Otherwise, we flip the indices on the centerline trajectory, and plan in the other direction.

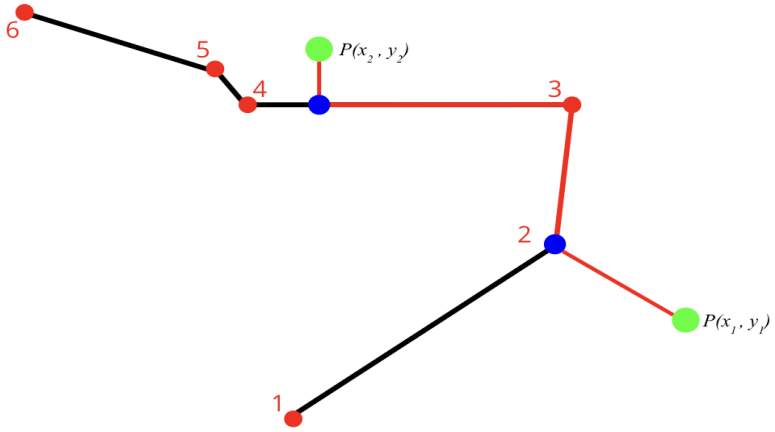


Figure 13: **An example centerline trajectory, start point, and end point, and the corresponding path.** The black line with red indices is the middle line trajectory. The start and end points are displayed by $P(x_1, y_1)$ and $P(x_2, y_2)$ respectively. The blue dots represent the closest points on the centerline to the start and end points.

In part A and part C, we calculated the closest points on the centerline using dot products. The centerline was given to us as a list of ten points, so simply checking the distance from the start and end would not be enough. We started by calculating which of the ten trajectory points were closest to the start and end. Then, if we were interested in the start point, we would look at the closest and next indices. This is because we did not want to path plan to points behind us, as that would hinder the controller and our speed. For the end point, this is flipped; we would look at the closest and prior indices. After finding those two indices, we calculated the vectors from the closest index to the start (v_1), and the vector from the closest index to the next index (v_2) using (2) and (3), and as shown in Fig. 14.

$$\vec{v}_1 = \text{Index2}(x_2, y_2) - P(x_1, y_1) \tag{2}$$

$$\vec{v}_2 = \text{Index2}(x_2, y_2) - \text{Index3}(x_3, y_3) \tag{3}$$

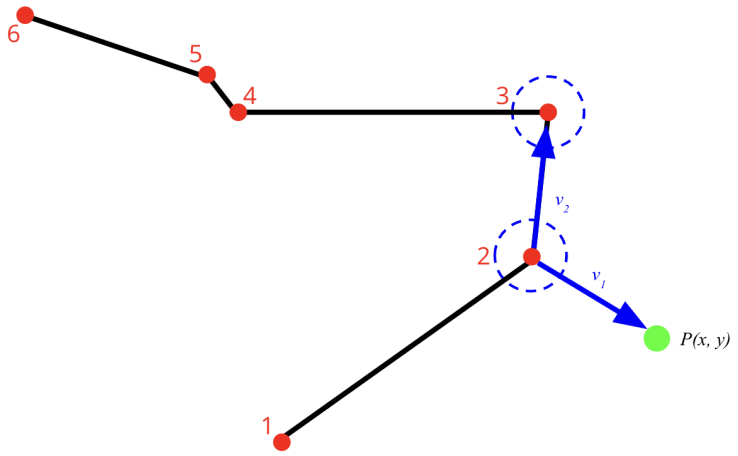
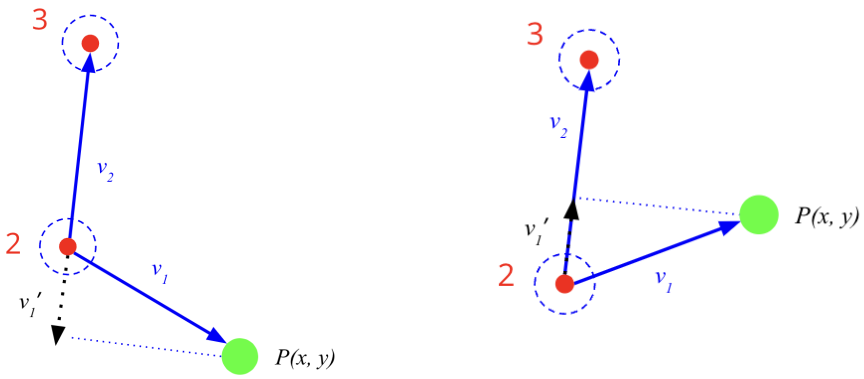


Figure 14: **Finding the relevant indices on the trajectory and calculating the vectors.** Index 2 is our closest index and index 3 is the next. We find the vector $\vec{v_2}$ from the closest index to the next index (since we are calculating the closest point for the start index), and the vector $\vec{v_1}$ from the closest index to the start point.

After we have v_1 and v_2, we can project v_1 onto v_2 and get v_1' using the dot product in (4). We can also get the magnitude of this vector by omitting the multiplication of v_2 as shown in (5). In Fig. 15, we see two different cases of the projection magnitude. Either the magnitude will be ≤ 0 in Fig. 15a, which means that the start point is orthogonal or behind the closest index, or the magnitude is > 0 in Fig. 15b. In the case where the magnitude is ≤ 0, we know that the closest point has to be the closest index (in this case index 2). Otherwise, we can add the projected vector v_1' to the closest index and find the closest point.

$$\text{proj}_{\vec{v_2}}(\vec{v_1}) = \frac{\vec{v_1} \cdot \vec{v_2}}{\|\vec{v_2}\|^2} \cdot \vec{v_2} \tag{4}$$

$$\|\text{proj}_{\vec{v_2}}(\vec{v_1})\| = \frac{\vec{v_1} \cdot \vec{v_2}}{\|\vec{v_2}\|^2} \tag{5}$$



(a) Negative Projection

(b) Positive Projection

Figure 15: **Positive and negative dot product projection example.** The black vector v_1' is the projected v_1 vector onto v_2. In the negative projection we can see that the closest point to the start on the line segment is index 2. In the positive projection, we can add v_1' to index 2 to get the closest point. In both cases, our algorithm effectively finds the closest point.

We calculated part C the same way as part A, just with the prior index instead of the next. With the two closest points to the start and end, we then used A* to plan two paths for the start and end to each of their closest points. Once we had part A and C calculated, we took the indices of the line trajectory between the two closest points and labelled those as part B as shown by the red in Fig. 13. While the part B path does not fall on the right side of the centerline, our trajectory follower is responsible for adding an offset while following. An example of our final path can be seen in Fig. 16. Thus, our path planning is complete, and we fed these paths into our controller.

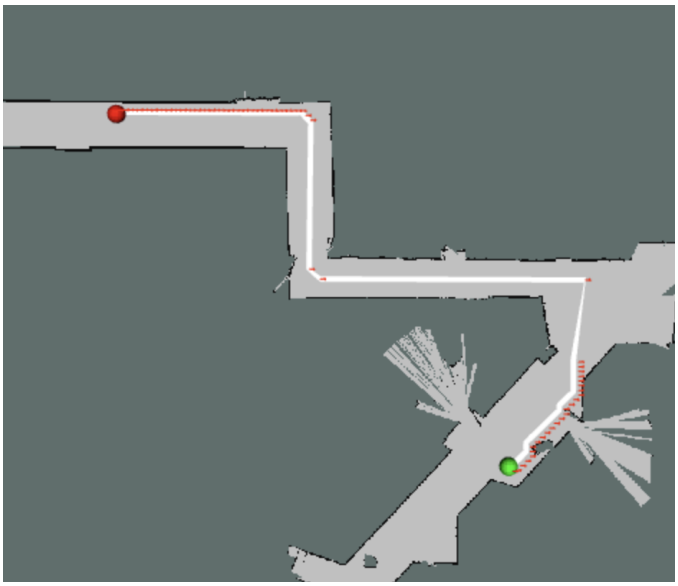


Figure 16: **Final generated path.** Path A and path C have a dotted red line as indication. Path B is the white line in between the red segments. We can see that we successfully generate a path from the start(green) to end (red) point.

3.5 Trajectory Following (Michael Kraus)

Once the path planning algorithm produces a path the trajectory follower is responsible for following planned sections of the path directly and keeping the car on the right side of the road when it is following the centerline. The general operation of the path follower is the same as in Lab 6 but some additional duties are assigned to the path follower in the final challenge:

1. Offset all lines by some set offset distance
2. "Trim" or "extend" the offset lines as needed to form a new complete path
3. Follow the new path

The following subsections will explain each step in greater depth.

3.5.1 Offsetting the Centerline.

The first new duty of the path follower is to offset the centerline trajectory by some set offset distance so that the car follows the centerline on the right-hand side, as displayed in Fig. 17.

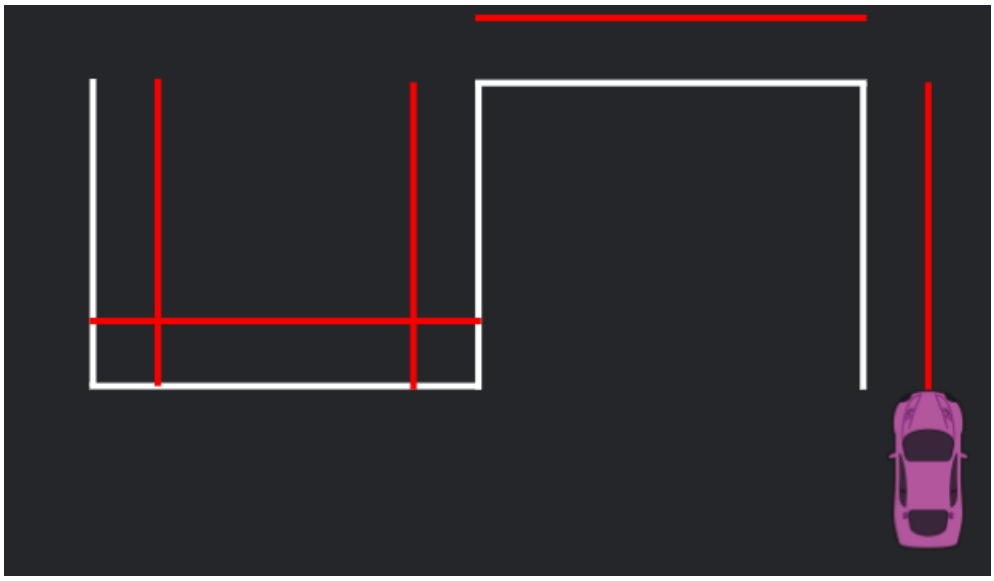


Figure 17: **Offsets on a trajectory.** If the path follower is given a trajectory (white) the first thing it will do is offset the segments by some distance to the right (red). This is to ensure that the car follows the centerline of the path on the right-hand side.

This is a straightforward process; the algorithm simply calculates the normal direction of each path segment with (6).

$$\vec{n} = [dy, -dx] \tag{6}$$

Note that with this formula, the normal direction will always point to the right of the segment. Then, to offset the segments, the start and end points of each segment are updated using (7).

$$newpoints = oldpoints + offsetdistance * \vec{n} \tag{7}$$

Once this is done the path will be in a state similar to Fig. 17.

3.5.2 Trimming and Extending Lines

The resulting path in Fig. 17 is unacceptable because the car does not have a continuous path to follow. To fix this, the segments must be trimmed to look like Fig. 18.

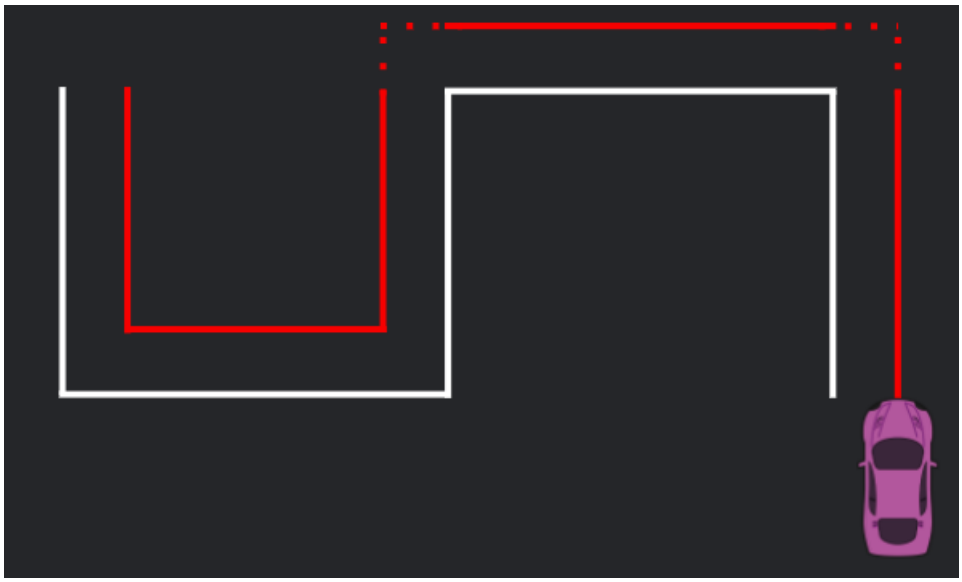


Figure 18: **An example of corners on a path trimmed/extended.** The corners of the new path should be trimmed or extended as necessary so that the path is fully connected without any gaps. This will ensure that the car has a smooth path to follow and will not cut or overshoot any corners.

To do this, the following method can be employed: Consider the two segments, $\vec{AB}$ and $\vec{CD}$, as shown in Fig. 19.

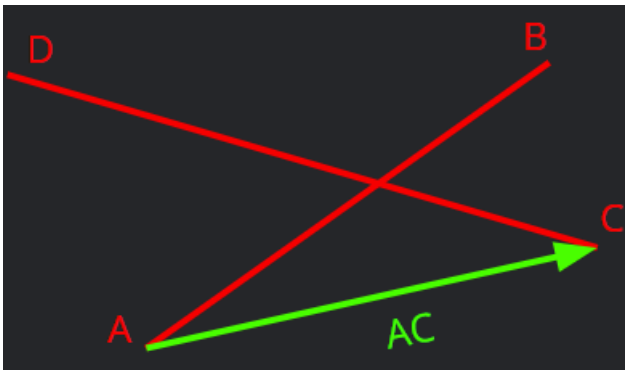


Figure 19: **Segments $\vec{AB}$ and $\vec{CD}$.** Segments $\vec{AB}$ and $\vec{CD}$ represent two segments on the trajectory to be followed. The segments need not intersect for our extending/trimming method to work.

If we parameterize each segment with parameters p_1 and p_2 we arrive at 8.

$$A + \vec{AB}p_1 = C + \vec{CD}p_2 \tag{8}$$

Where A and C are the coordinates at point A and C.
If we rearrange this we arrive at the system of equations shown in 9 and 10.

$$\vec{AC}_x = \vec{AB}_x p_1 - \vec{CD}_x p_2 \tag{9}$$

$$\vec{AC}_y = \vec{AB}_y p_1 - \vec{CD}_y p_2 \tag{10}$$

This system of equations can be quickly solved using numpy.linalg.solve(). Once all of the p_1 and p_2 parameters are solved, the position of every intersection and extension point can be easily calculated by using 8. While there are other more intuitive ways to calculate the intersection point of two lines, this method is easy to implement in a vectorized format. Given that a trajectory can have hundreds or even thousands of segments depending on the application, we chose to write all of our code in a vectorized manner to ensure that the algorithm quickly computes all of the intersection points at once.

3.6 Bidirectionality Feature

Once the segments are trimmed and extended, a path can be followed with any desired offset. However, the follower developed in Lab 6 only worked in one direction. A trajectory is stored as a list of waypoints in the computer. In Lab 6, the follower could only follow the waypoints in the order that they appeared in that list. Thus, some logic must be added to allow the follower to work in two directions.

This is easy to achieve. After the car finds the segment that it is nearest to, it simply has to calculate the vector between the first and last point of the nearest segment and dot that vector with its own heading vector. If the product is negative, the car is traversing the path in the reverse direction, and the points need to be flipped in storage as shown in Fig. 20. If the product is positive, the car is heading in the forward direction as shown in Fig. 21. Once this bidirectionality is added, the follower can follow any path at any desired offset in any direction.

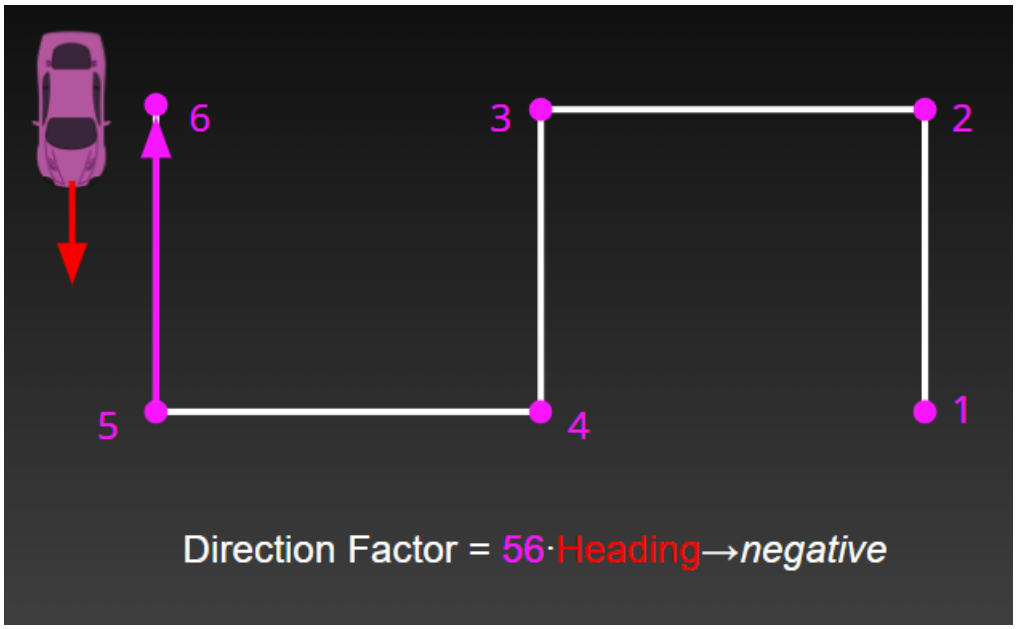


Figure 20: **Car heading and line segment vector with negative dot product on an example trajectory with indices.** The path follower will find the segment that is nearest to the car and calculate the vector of that segment (first point to second point). When this vector is dotted with the heading vector of the car, the result will be negative if the car is running the trajectory in reverse. In this case, the order in which the points are saved in memory must be flipped.

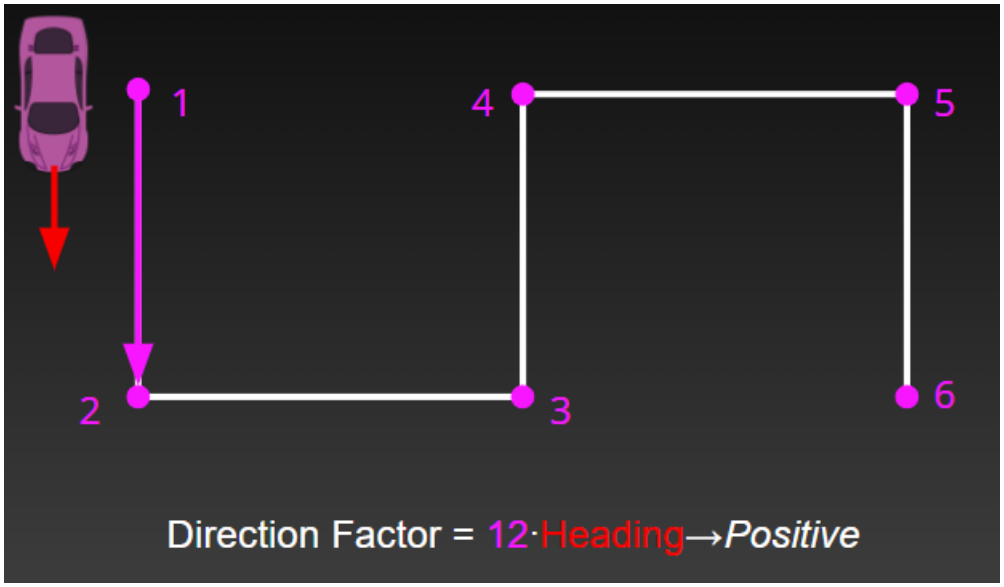


Figure 21: **Positive dot product between car heading and line segment vector**. When the nearest segment vector is dotted with the heading of the car, the result will be positive and the car will drive past the waypoints in the order that they are saved in memory; Thus, the rest of the path follower algorithm is allowed to run as usual

4 Experimental Evaluation

4.1 Overview

In order to assess our performance, we evaluated our controllers and algorithms based on metrics we stated in the introduction. For track racing, our main goals were speed and staying within the lane. For path planning, we followed the same values from Lab 6; we want our planner to be fast, accurate, and driveable. For our controller, we want the car to stay within the centerline and follow traffic constraints. While we were able to test our track racing, A* algorithm, and trajectory follower, we ran out of time to evaluate our computer vision and integrated system for city driving. If given more time, we would like to evaluate our three part path as detailed in the Path Planning section below, and our computer vision as detailed in the City Driving Traffic Stopping section.

4.2 Track Racing (Yohan Guyomard)

Overall, we were able to create a highly robust and repeatable autonomous race track driving system. Once tuned, we ran a dozen laps with our robot (some consecutive) and reported a consistent 56 second lap time with 0 infractions. We believe that with a faster car (ours was unable to reach the allotted 4 m/s), our system would be able to go even faster. Moreover, we experimented with less conservative lane distances; our robot would typically stay in the middle of the lane, or 0.5m to the right of its lane. However, by setting the distance to 0m (any smaller would result in an infraction), we were able to run a lap at 54 seconds.

4.3 Path Planning (Karen Guo)

4.3.1 A* Algorithm

We re-evaluated our A* algorithm with the same metrics in Lab 6, to compare the performance of the downsampled version. We valued computation time, driveability, and path length. We can see from Table 2 and Table 1 that our runtime is significantly faster than before. This makes sense as downsampling is expected to decrease runtime. We also notice that there are no real changes to distance, so A* is still generating near optimal paths. Thus, we believe that we have improved our A* algorithm from Lab 6, and that it would be more efficient and accurate than RRT as well.

Table 1: Old A* Computation Times and Path Lengths

Test Case	Computation Time (s)	Path Length (m)
(1) Straight Line	0.2979	12.0588
(2) Straight Line Behind	0.3609	12.0461
(3) Narrow Hallway	1.5725	71.8649
(4) Far Endpoint	3.2286	117.9590
(5) Sharp Turns	0.6104	25.2121

Table 2: New A* Computation Times and Path Lengths

Test Case	Computation Time (s)	Path Length (m)
(1) Straight Line	0.0227	12.2381
(2) Straight Line Behind	0.0266	12.1156
(3) Narrow Hallway	0.0738	71.2360
(4) Far Endpoint	0.1475	115.7855
(5) Sharp Turns	0.0356	26.0639

4.3.2 Three Part Path

Unfortunately, we ran out of time to evaluate our three-part path planning in simulation and real life because our main priority was to get the algorithms working before evaluating. However, if we were given more time, we would like to run a few tests for speed, and accuracy. This would include timing how long on average it takes for the path planning to plan out the paths to waypoints, how long it takes for the car to drive along those paths, and whether our solution is comparable to the optimal solution (to be generated using normal A*). Additionally, we would factor in traffic infractions and crossing the centerline, possibly using a similar approach to the staff scoring system, where infractions negatively affect scoring.

4.4 City Driving Path Following (Michael Kraus)

As discussed in section 3.3, the path planning algorithm from Lab 6 was modified to keep the car on the right side of the centerline. In principle, there are two variables that we can adjust:

1. Lookahead distance
2. Offset Distance

4.4.1 Lookahead Distance

In Lab 6, we found that the ideal look-ahead distance was 0.9 meters. We determined this by testing the car's ability to make 90-degree turns and respond to step inputs with a variety of lookahead distances. Since the final challenge more or less consists of turns similar to the turns that we tested our previous

controller on, we determined that there is no need to reevaluate the controller to find the ideal lookahead distance. **Moreover, for this challenge, we are using a lookahead distance of 0.9 meters.**

4.4.2 Offset Distance

However, the offset distance is a critical variable that we need to get set. If the offset distance is too small the car may cross over the centerline in sharp turns. If the offset distance is too large the car could potentially hit walls, pillars, etc. on outside (ie, right-hand) turns. To determine the best offset distance, we ran the car in both directions around the trajectory shown in Fig. 22.

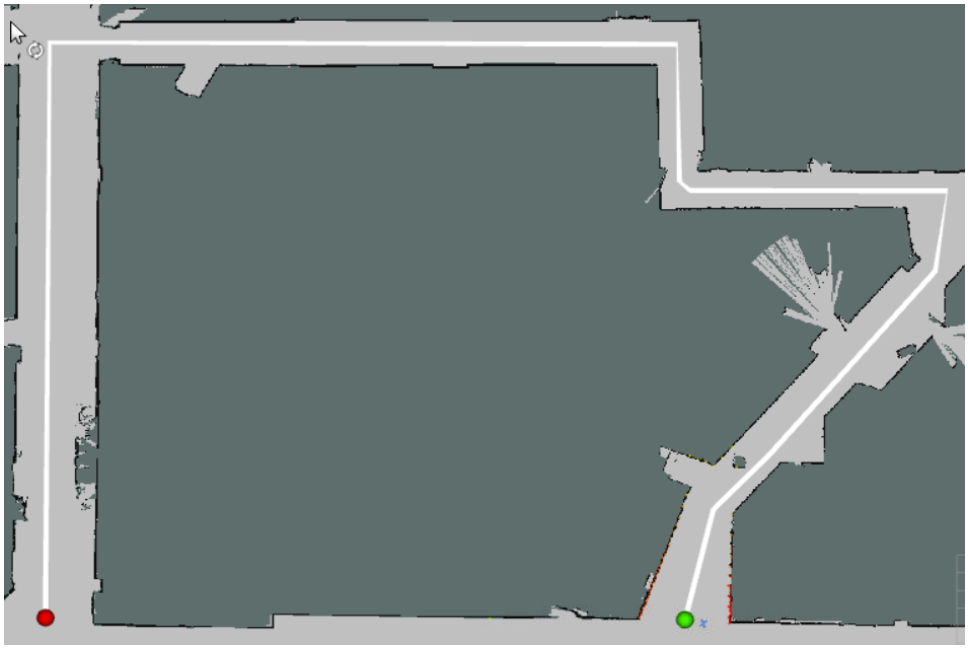


Figure 22: **The track used to test the car for the optimal offset distance.** The car was run in both directions to ensure that the car can make tight inside and outside turns.

For each run, we tracked the distance of the car from the centerline. Our results are shown in Fig. 23 and Fig. 24. Note that each spike in the graphs represents a turn on the map. In both graphs, we never saw a car that crossed the centerline. *However, these distance are all taken from the center of the car, which is about 15cm wide.* While "crossing the line" was defined as the centerline of the car crossing the line, we wanted to minimize the possibility of *any* part of the car crossing the centerline. **Thus, our team selected 0.3175 meters as our offset distance.** This offset distance gives our car an added buffer and ensures that the car will at most only slightly cross the line (perhaps it would result in

a tire on the line in most cases).

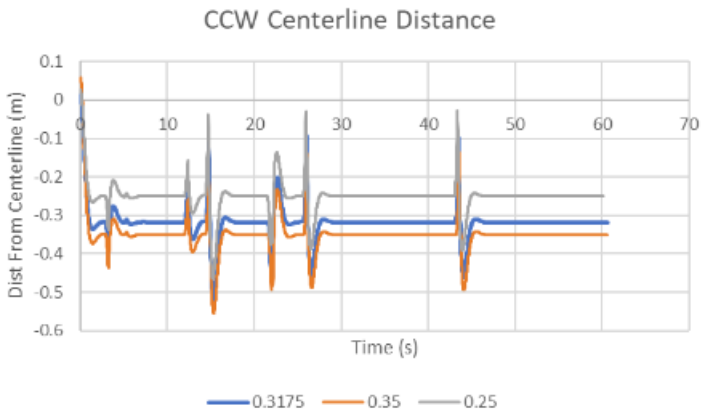


Figure 23: **Counter Clockwise Centerline (CCW) Distance.** With all of the values we tested, the car never crosses the centerline. However, it is important to note that these distances are taken from the center of the car. Thus our team believes that an offset of 0.3175m is the best choice to minimize the possibility of *any* part of the car crossing the centerline.

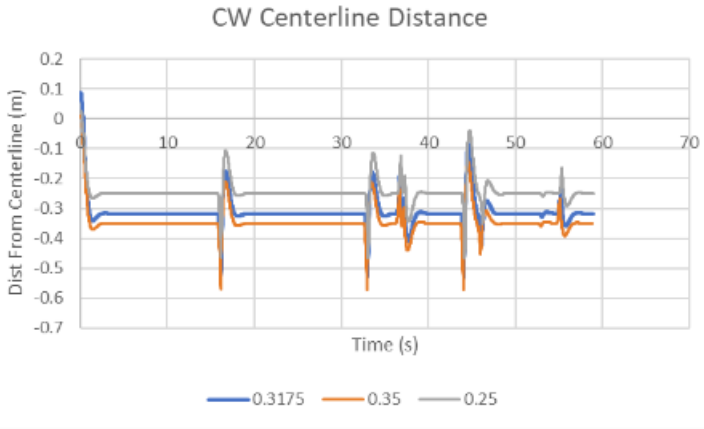


Figure 24: **Clockwise Centerline (CW) Distance.** With all of the values we tested, the car never crosses the centerline. Again, it is important to note that these distances are taken from the center of the car so we selected an offset of 0.3175m to minimize the possibility of *any* part of the car crossing the centerline.

4.5 City Driving Traffic Stopping (Jaclyn Thi)

Due to hardware and time constraints, we were unable to experimentally evaluate our racecar's ability to stop at stop signs and red stop lights. For example, due to frequently having a faulty wire which prevented the car from turning on or that could not connect with our LiDAR, which is necessary for localization. We also repeatedly found issues with connecting our joystick, which prevented us from running code on the actual car.

In the future, if we were to resolve the hardware issues faced we would plan to design a variety of test cases for our racecar approaching both stop signs and red stop lights at different angles and speeds. We would measure the stopping distance of our racecar from each stop sign and stop light and calculate the error between the actual stopping distance and the target stopping distance, which is 1 meter.

5 Conclusion (Carlos Villa)

The final challenge had our team firing on all cylinders as we attempted to implement successful lane following for track racing/Mario Circuit and city driving/Luigi's Mansion. To tackle these challenges we drew on our experiences and knowledge from the semester until now. While work on different components of the systems were allocated to different members of the team, it was ultimately a joint team effort that brought us to our results.

For track racing, we developed computer vision algorithms that could accurately detect a lane line to follow and we repurposed our old wall follower from Lab 3 which consisted of two PID controllers. With inconsistently and/or not at all working hardware, we still managed to get a consistent lane follower working on the robot for raceday. On raceday we achieved a best 200m lap split of 54 seconds. This achievement was due in large part to very creative and extraordinary efforts to simulate the track and test our solutions in simulation before we got them on the car.

On the city driving front, we adapted our path planning algorithms and pure pursuit controller from Lab 6 which were used by a new state machine that we developed to coordinate city driving. Additionally, we developed a stop light detector using knowledge of color segmentation from Lab 4, adapted a machine learning based stop light detector from the staff, and utilized our safety controller from Lab 3. Most of these were additionally aided by localization from Lab 5. While the individual parts were generally seen to be working at the times code could be put on the robot, it was difficult to do any real world evaluation of our solution as hardware limitations stopped us from reliably moving our code onto the robot or using the robot at all.

All in all we walked away with a track follower that worked well in simulation and real life, and a city driver that showed promise in simulation and isolated testing, but ultimately did not get a chance to shine on the robot. Our team had to come together and use all of the semester's knowledge to achieve what we did. While it was disappointing not seeing the city driver on the racecar it was exciting to see the success of the track follower and exciting to see all of our knowledge utilized.

6 Lessons Learned

6.1 Yohan

The biggest take away of this final challenge was learning to deal with faulty hardware. Unfortunately, the vast majority of our time on this project was spent investigating and fixing issues like the car not turning on, disconnecting from the network and other miscellaneous glitches. Thus, I learned the importance of running everything locally first before attempting to deploy to the physical system; it's always faster to iterate on a script in my downloads folder than deploying and recompiling on the actual robot!

6.2 Michael

As far as path following goes, this lab went relatively smoothly. Finding the intersections of lines was not as easy as I thought it would be, especially when

trying to write the code in a vectorized manner. On the hardware side, we had several power cables and several car batteries - some bad some good. While troubleshooting the car I got to practice some of my problem isolation skills to determine if a battery, cable, or both were bad.

6.3 Jaclyn

I learned that tuning HSV values is important, especially when there are similar colors in the environment that one would not want to detect. I also learned that masking can help narrow the robot's field of vision to better detect only red stop lights, and the importance of communicating with and checking in on one's team.

6.4 Karen

For technical lessons, I learned that implementing a path planning in conjunction with a line follower is more complicated than I thought. While I had originally believed that we could reuse parts that we had already coded for this, there were various edge cases that we weren't initially aware of. Thus, it would have probably been easier if we altered where the path planning could plan rather than taking our current approach. Also, I witnessed how hard integration can be when dealing with so many nodes. For communication/non-technical lessons, I realized I am not particularly efficient when working on less sleep. Hence, I aim to have better time management and to be well rested when working on big projects in the future.

6.5 Carlos

I learned about the difficulties of mass integration. Our implementation of the city driver had a lot of moving parts that relied on each other, so it was important that they could work together well. It's hard to achieve this in an ideal world but even harder when facing the types of hardware challenges we encountered when trying to move from simulation to real life. While we weren't able to get the system integrated on the hardware, I learned the strength of the team we had developed as everyone stepped up in various ways to try to move the project forward.