Final Challenge Report

Team 16: Sameen Ahmad, Phillip Johnson, Trevor Johst, Maxwell Zetina-Jimenez, Ellen Zhang
Editor: Trevor Johst

1 Introduction

Author: Sameen Ahmad

The final challenge serves as a platform to showcase the culmination of our team's learning progress over the semester. Through the competition, we were able to apply our knowledge in computer vision, localization, and path planning to enable our autonomous race car to complete different tracks. The final challenge consists of two components: the Final Race, also known as Mario Circuit, and City Driving, or Luigi's Mansion. In the Final Race, teams are expected to program their cars to race along the Johnson Track while staying within one's lane and avoiding collisions with neighboring cars. In City Driving, teams are expected to navigate a model city environment while abiding by traffic laws such as staying on the right side of the lane, yielding to traffic lights and stop lights, and avoiding pedestrians. The robot must follow a path and travel towards certain points to collect shells. For each competition, penalties are assigned for lane breaches or traffic infractions.

For the Final Race, we utilized computer vision through a Probabilistic Hough Transform and PD control. The Probabilistic Hough Transform allows one to find line segments in a binary image efficiently, which we applied for line detection of the track's lanes. Then, we cropped the image to calculate the centroid of the robot's current lane and implement PD control to follow the point.

For City Driving, we used a state machine to determine the car's actions in different events implemented in the YASMIN package. We traverse a trajectory based on the selected points that was offset from the central lane. Then, we utilized a pure pursuit controller at low speeds to follow the trajectory, while stopping at the shell locations that were published. However, our car still struggles to stay within its lane while turning. We employed machine learning to create a stopping controller for stop signs and traffic lights, but were unable to test it sufficiently on our car.

If we had more time we would tune our controller to allow sharper turns, as well as reversing for turn-arounds. Also, we would integrate the stopping controller so our car is able to respond appropriately to traffic lights and stop signs. Over the course of the semester, our team was able to significantly develop our skills in various critical areas of robotics and look forward to applying our knowledge in the future.

2 Technical Approach

Author: Maxwell Zetina-Jimenez

The first part of the Final Challenge tasked us with staying in a lane on Johnson track and completing a lap at a maximum speed of 4 meters per second. We made use of skills from previous labs such as computer vision and controls. This challenge involved using edge detection and Hough Transforms to detect lanes, as well as homography to follow a real world point in the lane. Furthermore, we explored the advantages and disadvantages of pure pursuit and PD control, in this case ultimately having a solution with PD control where our error was the angle between the robot's heading and the middle of the lane. Our robot was able to successfully complete a lap around the track at 4 meters per second with a total time of 53 seconds.

The second part of the Final Challenge was to navigate Stata basement, visiting three unknown locations while staying in the right lane and adhering to traffic rules. Our team leveraged the work done from past labs but also explored different solutions. We employed our pure pursuit controller from the path planning lab to execute planned trajectories through both lanes. However, we also explored using a PD controller and a Stanley controller. We organized logic using a state machine to transition between tasks of reaching a goal, going to the next goal, and coming back to the starting location. These transitions required more complex dynamics like three-point turns to turn around, which we attempted to implement with Reeds-Shepps curves. Moreover, we experimented with the You Only Look Once (`YOLO`) machine learning model to detect stop signs and traffic lights, implementing controllers to adhere to traffic rules. Our team was able to visit all three goal points and return to the starting point but did so with traffic infractions.

2.1 Track Lane Follower

Author: Maxwell Zetina-Jimenez

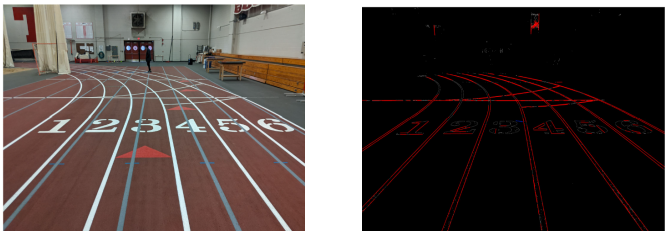
Completing a lap around the Johnson Track with our robot involved leveraging computer vision and controls to stay within a specific lane the whole lap. The robot's camera image was cropped and processed for line detection. The closest lines to the center of the image were considered the lane lines, and the midpoint between them was the reference point for our PD controller to follow. We were able to complete a full lap around the track in 53 seconds at a constant speed of 4 meters per second.

2.1.1 Lane Detection

Author: Maxwell Zetina-Jimenez

The first task in completing a lap around the Johnson Track was detecting the lane so that the robot could follow it with a controller. Lane detection involved first grabbing all of the lines in the image of the car, which was done using the Python `cv2` package. We did edge detection using `cv2.Canny` and a color threshold to filter out edges that were not white track lanes. We then took that edge-detection

image and used a Hough Transform (`cv2.HoughLinesP`) to get the lines in the image. An example result of this can be seen in Fig. 1, where the starting line of the track is processed for line detection.



(a) Johnson Track (b) Track line detection

Fig. 1: **Line detection on the Johnson Track.** Edge detection and a Hough Transform allowed for accurate line detection, which helped identify lanes.

However, because there are multiple lanes on the track, it was not enough to simply do line detection to determine a lane and follow it. To solve this problem and have our image focus solely on the current lane the robot was in, we cropped the image top and bottom to only look at a thin sliver of the image, namely the section of the current lane that was slightly ahead. As seen in Fig. 2, this cropping strategy filtered out lines that were far ahead and insignificant to the current position of the robot, as well as lines that were part of the robot's LiDAR sensor that appeared in the image.

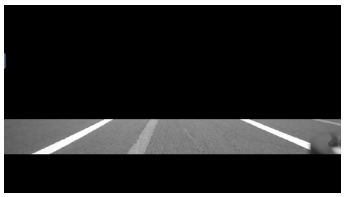


Fig. 2: **A cropped image of the track.** Ignoring the top and bottom of the image allowed the robot to focus on the lines of the correct lane and thus prevent lane infractions.

The reason for getting the lines of the lane was to be able to develop a controller that could stay in the lane and follow its curve. Thus, a reference point to follow was needed. As seen in Fig. 3, we used the middle of the lane as the reference point, which was computed by getting the midpoint pixel between the nearest left and right lines with respect to the vertical center line of the cropped image. This pixel was then passed on to our homography tansformer to determine a real world point for the robot to follow and try to be aligned with.

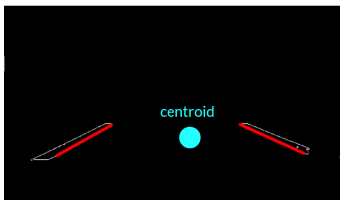


Fig. 3: **Lanes detected with centroid labeled.** After line detection and cropping, our controller followed the center of the lane to reduce error and prevent leaving the lane.

The only issue was that sometimes there would be horizontal lines on the track that crossed into the lane and were considered a closest lane line. This misinterpretation would skew the position of the midpoint, which resulted in the robot following a misplaced centroid that would lead it outside of the lane. To overcome this issue, we tried a different cropping technique to segment the horizontal lines into small chunks and then filter with a length threshold as seen in Fig. 4. However, this did not work all the time on curves, as the vertical stripping would interfere with the actual lane line and segment it. Because of this failure, we ultimately implemented a filter based on line slope that ignored lines with low slope — horizontal lines that were not part of the lane barriers.

(a) Vertical stripping of lane (b) Centroid after vertical stripping

Fig. 4: **Vertical stripping before image processing.** Segmenting horizontal lines allowed minimizing their influence on the position of the centroid (blue star) but also split the real vertical lanes on curves.

2.1.2 Lane Following

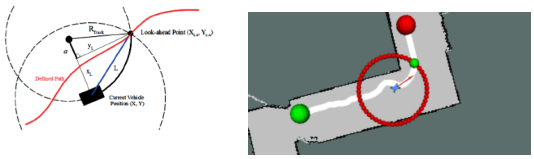
Author: Phillip Johnson

The initial approach to lane following was to use a pure pursuit controller that projected the detected centroid of a lane onto a lookahead circle that was a set distance ahead, similar to what is shown in Fig. 5a. The issue that we found with this is that balancing the speed to lookahead ratio proved to be incredibly difficult at high speeds. For example, a high lookahead is desirable on straight-line trajectories, but the curves require a lookahead that is much lower. However, one that is too low will overturn, and one that is too high will not follow the lane.

We attempted to fix this by projecting the slope of the lane onto the centroid and perform path following similar to what we did in lab 6 - a representation of this is shown in Fig. 5b. However, the slope of the line proved to be difficult to calculate due to the convergence of the lanes at a far vanishing point, as shown in 6

As a result, we decided to use a Proportional-Derivative (PD) controller. This controller works by actively minimizing the calculated error. The error equation that we used is shown in (1). It functions by taking the y (horizontal) distance from the centroid, projecting it out by 1.0 meter (this ensures that the angle is reasonable and standardizes the x distance of the centroid), and calculating the scaled angle difference.

The proportional control works to actively minimize that error by applying a control in the direction of the error (i.e. if the centroid is to the right of the car, proportional control will turn the car to the right). Derivative control



(a) Point based pure pursuit (b) Path based pure pursuit

Fig. 5: **Two different versions of pure pursuit.** Point based pure pursuit would calculate the steering angle to the next point on the trajectory. Path based pure pursuit would calculate the steering angle to a point a set lookahead distance along the path.

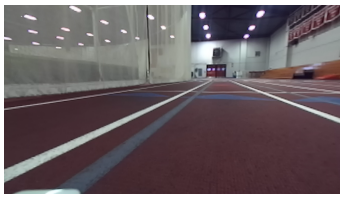


Fig. 6: **Straight parallel lines converging to a distant vanishing point.** For our track follower, the parallel lanes still converge at a distant point making it difficult to quantify their centroid line.

will calculate the change in error based on the applied proportional control. When added to proportional control, this will damp the system as, if the proportional control applies a turning angle that increases error, the derivative control will correct that error. The equations and gains we used for proportional control can be seen in (2) and (3), respectively.

Finally, we decided that implementing integral control was unnecessary as the overshoot did not seem to be an issue. Any issue we ran into with the controller was almost always due to incorrect centroid calculations, and the added complexity of tuning integral gains was deemed unnecessary.

$$error = \frac{1}{speed} \cdot arctan2\left(\frac{y_{error}}{1.0}\right) \quad (1)$$

$$Proportional\_Control = K_p \cdot error[K_p = 0.9] \quad (2)$$

$$Derivative\_Control = \left(\frac{K_p}{4.0}\right) \cdot (last\_error - error) \cdot dt \quad (3)$$

2.2 City Driving

Author: Ellen Zhang

The goal of Luigi's city-driving challenge is to drive around the Stata Basement and pick up three randomly placed shells. To do this, the racecar must operate localization to know where it is, use a controller that can follow the trajectory reliably, and be able to detect stop lights and stop signs and stop in time. We realized that there is a predictable flow of control from one state to another; to integrate this into our code, we developed a state machine utilizing the YASMIN python package for ROS2 as our infrastructure.

2.2.1 *Operational Logic*

Author: Ellen Zhang

Our operational logic for the state machine is explained below, and also shown in Fig. 7.

1) **Get shell locations**: By subscribing to the map and listening for clicked points, we store the locations of the three shells as well as our initial pose. After receiving all three shells, we move on to projecting our first shell's location.
2) **Project shell to the desired lane**: Given our car's current position, we know which lane it is by calculating the angle with respect to the central trajectory; if the angle formed is clockwise, it is on the right lane. We then project the shell location onto the desired lane to get the goal we wish to reach. If this projection is behind us, we must conduct a U-turn to switch lanes. Otherwise, we may proceed to plan the trajectory.
3) **Turn Around**: If the projected shell lies behind us, we must turn around and switch lanes. After doing so, we transition back to projecting the shell, but this time, we know that the projection will be in front of us.
4) **Offset line trajectory**: Now that we have the car's current position and our goal, as well as which lane we wish to be in, we simply find the portion of the lane we wish to follow. We precomputed the right and left lane trajectories and simply load them into our code, and calculate which portion is most relevant. Then, we navigate it.
5) **Navigation**: We now use our car's controller program to navigate the desired trajectory until we reach the goal. Once we reach the goal, we wait for 5 seconds to pick up the shell, before going to the next shell location. We will go more into this portion in the next section, trajectory following, because we had three options for the control.

The entire time, the stop light and stop sign detectors run in the background; if they detect the need to stop, they issue a higher-level drive command that overrides the current one. Similar logic follows for the safety controller program, which runs in the background and will issue an overriding stop command if a pedestrian walks in front of the car.

The overall state machine logic is sound and works well in simulation; Fig. 8 shows the racecar following both right

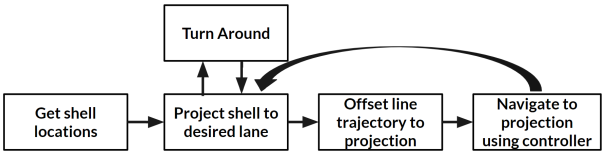


Fig. 7: **Flow of the State Machine** After receiving the shell locations, the robot then plans along the relevant lane and follows it to the projected goal, and then waits for 7 seconds before moving on to the next goal.

and left lane trajectories in simulation, from one shell to the next.

(a) Right lane trajectory (b) Left lane trajectory

Fig. 8: **Racecar following right and left lane trajectory in simulation, show in red.** The right lane trajectory goes from the start to first shell, while the left lane trajectory goes from the second shell to the third shell.

2.2.2 *Trajectory Following*

Author: Trevor Johst

In order to traverse the path between different objective points as outlined by the state machine, we had to have a robust controller. Throughout the testing phase, we tried variations of three different controllers: PD, Stanley, and pure pursuit. We found that for our offset lane trajectories, PD was difficult to tune on both corners and straight sections. The Stanley controller struggled on any portions of the trajectory that were not dynamically feasible. Pure pursuit required low speeds, and had to be balanced between oscillations and cutting corners.

For the PD controller, multiple error terms were tested. An error term as outlined in the track follower section was attempted, but proved to not work by itself. The most effective error term was a combination of both the cross-track error and the angle to a future point at a set lookahead distance. The errors were combined as

$$\mathrm{e} = \alpha \cdot \mathrm{e}_{cross\_track} + (1-\alpha) \cdot \mathrm{e}_{angle} \tag{4}$$

where α was a parameter between 0 and 1. We found that a value of 0.8 worked best. In the end we did not end up using the PD controller because some areas of the lane trajectory would cause unrecoverable deviations.

The Stanley controller was implemented as outlined in [1]. Without going into extreme detail about how the controller operates, the final steering angle is calculated as

$$\delta = (\psi - \psi_{ss}) + \arctan\left(\frac{ke}{k_{soft}+v}\right) + k_{yaw}(r_{meas} - r_{traj}) + k_{steer}(\delta_{cur} - \delta_{prev}) \tag{5}$$

where ψ is the current angle deviation from the trajectory, ψ_{ss} is a turn offset based on the momentum of the vehicle, e is the cross track error, v is the velocity, r is the yaw rate, and δ is the steering angle. All k values are constants that help adjust various sensitivities. One important note is that the last term is not possible to calculate exactly due to limitations on the car. The specified δ terms are supposed to be measured from the vehicle, but as a sensor does not exist to measure this we instead assume instant steering. This assumption definitely impacted tuning and made this term less reliable.

The original intention was to use the Stanley controller with our previously developed RRT* algorithm using both Dubins and Reeds-Shepps curves as shown in Fig. 9. As the Stanley controller assumes all angles along the trajectory are traversable, this would be an important step in operating the controller. In reality, we did not have time to test the state machine with manual path planning between states. This meant we were had to use fixed, pre-determined lane trajectories for travel. This simplified the state machine, but had the downside of making our trajectories not dynamically feasible.

Due to this, we finally settled on using the pure pursuit controller previously outlined in lab 6. This was a trade off as it still ended up cutting corners at times, but allowed us to test the full state machine before the deadline. To reduce the number of lane infractions, we lowered the speed of the controller and got the lookahead as low as possible. The result could have been further improved, if we updated the center lane trajectory that was provided to more accurately represent reality. There were a couple locations where the provided center lane trajectory would cut corners, and combined with pure pursuit it would often deviate significantly from the lane.

2.2.3 *Traffic Abidance*

Author: Sameen Ahmad

We used machine learning to detect stop signs and traffic lights to know when to stop appropriately. Unfortunately, we were unable to integrate it onto the car, but were able to test the algorithm in isolation. We were provided with a stop sign detector that used `YOLOv5`, a multi object detection algorithm. It works by processing the image with computer vision and displaying a bounding box around the desired object. We used a similar `YOLOv5` model that was trained on a data set of traffic lights to implement our traffic light controller.

Since the location of the stoplights were predetermined, we found their coordinate locations on the map and constantly calculated the minimum distance of the robot's pose to each stoplights. If a traffic light was detected to be within a 2.5 meter range from the car, the robot would begin to slow down. We chose to reduce the speed of the car when within a certain radius from any traffic light because the

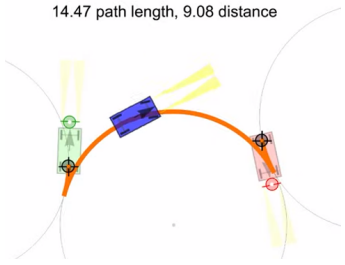


Fig. 9: **A Reeds-Shepp curve between two car poses.** The green pose is is the initial pose, blue is an intermediate pose, and red is the final pose. The initial and final poses point in opposite directions and reverse on the first and third segment.
Source: [2]

machine learning algorithm was able to detect traffic lights more consistently at slower speeds.

Once a traffic light was detected, we cropped the image of the traffic light to isolate the upper third portion containing the red light. Then, we performed color segmentation to detect when the light was on. If a light was detected it would start a buffer timer guaranteeing a stop for at least two seconds. If the light was detected again it would reset the buffer timer. This helped alleviate the poor detection by the model.

Additionally, we relied on our safety controller from previous labs to ensure that the car yielded when it came across pedestrians or other obstacles. When the car detected an object within a 0.5 meters radius through LIDAR, it would send a drive command to stop. We derived a function that took into account the car's current speed to determine when to begin sending the stopping commands so that it'd stop at the desired distance from obstacles.

3 Experimental Evaluation

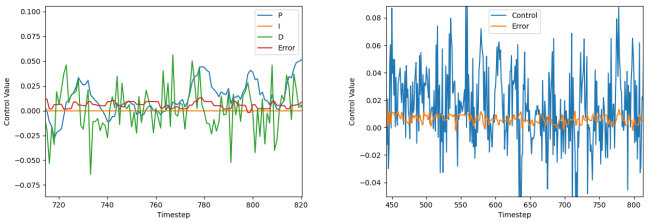
Author: Phillip Johnson

Evaluation is essential to controller success. We leveraged quantitative analysis and qualitative observations to ensure our controllers and stop sign detectors were adequate.

3.1 Track Lane Follower

Author: Phillip Johnson

We used both quantitative and qualitative evaluations when determining the functionality of the track lane follower. To initially tune the proportional gain, we would increase the gain until the car oscillated around a center line (we used lane 4 as there is a gray line directly down the center). Then, we would add derivative control and begin at a k_d value of $k_p/6$. To quantitatively determine the control attitude that we wanted, we would use statistics as shown in Fig. 10a. The main response that we wanted to see is that the proportional control increases when error increases and the derivative control damps the proportional control. This will lead to the response shown in Fig. 10b. The control will always be higher than the error so that the car can adjust quickly. More importantly, it will be in the direction of error which is what allows that error term to stay relatively constant.



(a) Control vs. error plot (b) PD control with error

Fig. 10: **Performance metrics for the track follower.** We can see that the proportional term handles direct error correction, while the error term handles overcompensation by the proportional term.

The error stays around +0.01. This is because our camera is to the left of the car's mid-line. Originally, this error was much higher, but we adjusted for the pixel offset by putting a tape measure in front of the car's mid-line and adjusting the offset until the mid-line pixel was above the tape measure. The before and after of this can be seen in Fig. 11a and Fig. 11b, respectively (the blue circle is the calculated center of the lane).

Additionally, the car is extremely left-turn biased at higher speeds. To account for this, we apply a constant turning angle offset of -0.045 at speed 4.0, and this offset decreases somewhat linearly for lower speeds. We determined this offset by publishing a constant speed in lane 4 and adjusting the offset until the car was able to drive in a straight-line for around 25m of distance.

Finally, we qualitatively measured performance by determining types of errors. For example, a sharp turn in a straight lane means that there is likely an error of calculating the centroid wrong which could be due to tracking incorrect lines. Oscillations likely mean that the derivative constant is too low. Understanding the qualitative differences between runs allowed us to correct our gains and fix lane detection issues.

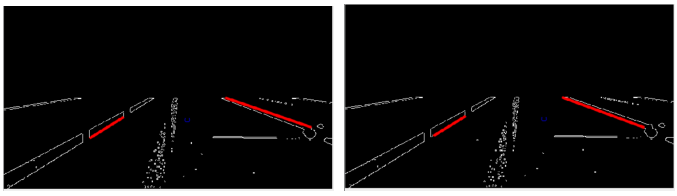
3.2 City Driving Trajectory Follower

Author: Trevor Johst

Due to time constraints, we were not able to acquire any quantitative metrics of our City Driving trajectory solution in real life. In Fig. 12, we show the simulation time vs. error of the car from trajectory as is navigates using Pure Pursuit, and we see that it maintains a relatively low error, never deviating above 0.20 meters from trajectory. If we had time to collect real life data, some useful metrics to record would be deviation from the intended trajectory, stopping distance at traffic stops, and some form of speed or time metric.

We can qualitatively assess the performance based on some of the successful runs that we did have. A common theme among runs was a large number of lane infractions. This was due to the aforementioned mismatch between the real lane and the desired trajectory, as well as the use of pure pursuit as our controller.

For the runs we had that were successful, times ranged from low two minutes to three minutes. This was in large part due to our low travel speed. A more robust controller or a more traversable trajectory would have likely allowed us to increase the speed and thus decrease the time.



(a) Without offset (b) With offset

Fig. 11: **A constant offset applied to the centroid.** The blue circle is the centroid between the red lane lines. Due to misalignment with our ZED camera, a constant offset had to be introduced to ensure the center of the line was accurate to reality.

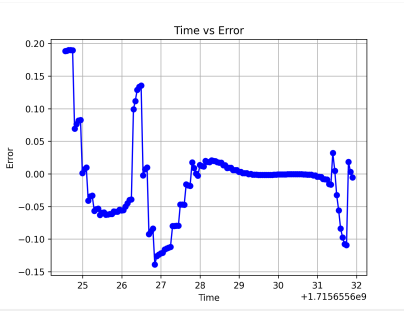


Fig. 12: **Error of racecar from planned trajectory in simulation.** The car follows the trajectory closesly in simulation, never deviating above 0.20 meters. Unfortunately, in real life we did fully troubleshoot, so the car would frequently cross lanes.

3.3 Stop Sign and Stoplight Detector

Author: Sameen Ahmad

To evaluate the performance of our stop sign and stoplight detector, we published the bounding box that was drawn around each detected object as shown in Fig. 13. We tested the algorithm with stop signs and traffic lights at various locations along the map, and qualitatively found consistent results of detection at the appropriate location in front of the devices. We also utilized a logger to track the state of each detector, letting us know when it is able to see either a traffic light or stop sign.

We found that in implementation, the ZED 2 camera would require a large amount of compute while running. This would not only slow down our other nodes, but would also slow down the detection. To actually implement real time detection of both stop signs and stop lights would require either logic to slow down when near them, or significant optimization for many components of the robot.

4 Conclusion

Author: Sameen Ahmad

For the final challenge, we applied computer vision, path planning, and localization to develop algorithms that were suitable for the Final Race and Circuit Driving competitions. We implemented a Probabilistic Hough Transform and PD control so that our car was capable of detecting the lane it was in while racing on the Johnson track. In City Driving, we projected published goal points to a trajectory that was offset from the central line and followed it using pure pursuit. We also utilized machine learning to respond appropriately to stop signs and traffic lights. Additionally, we explored the Stanley Controller and Reeds-Shepps curve to handle reversing. These components were integrated through a state machine that switched between various states as prompted by external events.

(a) Stop sign detection (b) Traffic light detection

Fig. 13: **A stop sign and traffic light being detected with ML.** The bounding box around the traffic light is then used to color segment for stop detection.

Our experimental evaluation in simulation and our physical implementation demonstrate the feasibility of our algorithms for either challenge. In simulation, our race car precisely and consistently generates and follows the trajectory. Our physical implementation shows deviation from expectation, especially through lane infractions and struggling to turn, revealing areas for future improvement.

Overall, the final challenge offered a valuable opportunity to display the culmination of our prowess in various areas essential to robotics. We look forward to building upon the foundations that were laid through this course as we apply our leanings through our academic and professional careers.

5 Lessons Learned

5.1 Sameen Ahmad

I learned the importance of version control while collaborating. We ran into several issues where changes in our program were unexpectedly not saved and reverted to earlier versions that possessed errors. This caused our team to spend significant time debugging algorithms that performed well in earlier labs. I was also exposed to how to apply machine learning for object detection.

5.2 Phillip Johnson

One thing I learned is that simple controllers are often better than ones which require intense complexity. I feel like our team would be stuck trying to add complexity to controllers to handle edge cases when good controllers are able to handle edge cases without unnecessary complexity.

5.3 Trevor Johst

5.4 Maxwell Zetina-Jimenez

This final challenge was eventful. It was great to see many components come together for an integrated product. To accomplish this, however, it involved careful organization. I learned about the importance and advantages of modularity in system design. Modularity made it easier to debug and integrate components. It is a great technique to employ as systems grow in size and complexity. I also learned that teamwork becomes even more important with larger projects. Sometimes we would make progress, then try a fix, and then try to go back to our previous but would find it difficult to remember what we had. Then we would bounce between forward progress and backward progress. Logging and great organization become even more necessary in a larger project like this.

5.5 Ellen Zhang

For the final challenge, I believe we may have saved more time if we had planned and started earlier, particularly for the Luigi's mansion part (Fig. 14), as much of the first week was spent trying to figure out what was going on. This would have helped a lot, to be able to do more testing, as a large bottleneck towards the end was the amount of time and effort it requires to launch the car in real life and debug it. Overall, however, I learned a lot, both technically and as a teammate, and enjoyed the experience.

REFERENCES

[1] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun, "Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing," in *2007 American Control Conference*, 2007, pp. 2296–2301.

[2] May 2023. [Online]. Available: https://www.youtube.com/watch?v=fAqh$_c y7ePIt = 32s$

Fig. 14: Luigi