The Art of Integration: Combining Modularized Algorithms to Navigate Busy Cityscapes and Fast Racetracks

Ayden Johnson Erin Menezes Gregory Pylypovych Jose Ramirez Gloria Zhu

I. Introduction (Ayden and Jose)

After having developed and tested all the individual parts of an autonomously driving vehicle, in this lab we finally put all the pieces together and tested them out on two challenges. In Luigi's Mansion (City Driving) the robot is tasked with navigating a city landscape, stopping at stop lights, waiting for pedestrians, and picking up shells. This challenge is a test of the robot's ability to use path planning, computer vision, and localization to get to its targets without breaking any traffic rules. In contrast, Mario's Circuit is a high-speed race around Johnson track, and focuses more heavily on the robot's computer vision, as well as its ability to react quickly and safely at high speeds.

The challenge in Luigi's Mansion reflects the tasks that autonomous vehicles must carry out in real life, such as self-driving cars obeying traffic laws and efficiently moving from one place in a city to another without causing damage to itself, people or other things. The types of code modules and techniques used for this challenge are nearly identical to those used by self-driving cars to carry out their functions. The challenge on Mario's Circuit is more reminiscent of tasks that a high-speed drone might need to fulfill, including moving through an environment as fast as possible and correctly identifying things in its surroundings, such as people in a search-and-rescue mission.

II. Technical Approach for Luigi's Mansion (Ayden)

The objective of the City Driving challenge is for the robot to move to three arbitrarily selected locations in the Stata Basement parking to pick up shells at each location. During the challenge the robot must also obey traffic laws by staying on the right side of the road, stopping at stop signs, abiding by traffic lights, and avoiding pedestrians. The modules needed for this challenge were the path planning module, the computer vision module, and the localization module. There was also a new module created to control what actions the robot takes during each stage of the challenge.

A. State Machine (Erin)

We knew that we already had the individual pieces written for this lab, but the big challenge was to integrate all these parts together. We decided we would manage all the tasks in this mission using a state machine (Fig. 1). This allowed us to focus on one task at a time, limit other nodes running to increase bandwidth, and have code that is easier to debug.

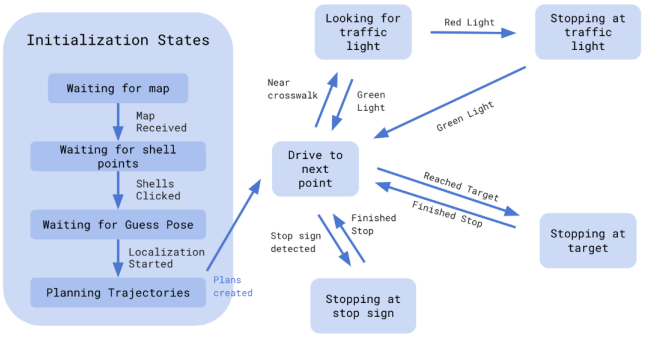

Fig. 1. This is the state machine that was implemented to manage the multiple tasks needed to navigate Luigi's Mansion while following the traffic rules. We start with a few states that call each other chronologically to initialize our trajectories. We then settle in a state where we try to drive to our next point in our trajectory. When our drive gets interrupted by stop signs or traffic lights, or we reach a target, we are able to switch our state variable to deal with this new piece.

The first few states occur once and in order while the other nodes boot up. The first state waits for the map to be initialized, which then calls the next state once the simulator map is up. The machine then waits for the three shell positions to be published from the RViz map, and from there, we generate our first guess pose and plan all trajectories between these points.

We then settle into the drive state, where we attempt to use pure pursuit to navigate to our next target point. As we approach crosswalks, we transition into a state where we are searching for a traffic light. If the traffic light is green, we switch back to the drive state, but if the traffic light is red, we have to stop to wait for the light to turn green again before switching back to the drive state. This is accomplished by yet another state that sets speed to zero while waiting for green. Next, if at any point we see a stop sign, we shift to a state that sets speed to zero for a few seconds to allow the car to stop, and then shifts back to the drive state to let the car continue on its path. Finally, once a target is found, we shift into another state where we wait for five seconds to collect the shell, and then continue on our way to the next shell.

B. Node-Topic Diagram (Erin)

We then had to convert the state machine into something that ROS would understand, nodes and topics. We reorganized this system to define what nodes we would need, keeping in mind that we had already implemented many of the functionalities we required in previous labs. Thus, we assembled our nodes and topics in a way that utilized as many of the previous algorithms as possible (Fig. 2).

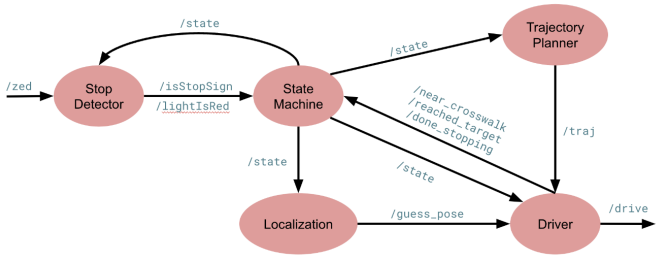

Fig. 2. The state machine from Figure 1 was converted into this node-topic diagram, and then implemented in ROS2.

We first used the state machine node to manage the states. This node subscribed to almost all the topics and published the current state to all the other nodes. The localization node was used to create guess poses of where the car was located in Stata basement, which fed into the driver state that commanded the car to move or stop depending on the state. Additionally, the trajectory planner state from the Path Planning lab was used to send trajectories to the driver's pure pursuit function. Finally, a new stop detector node was created to manage all the vision. This was the only node that subscribed to the camera, and identified stop signs and traffic lights.

Even though we were able to use many of the previous algorithms we had implemented in previous labs, we still had to make a few adjustments to accomplish the more specific tasks in this lab.

C. Path Planning

The path planning module underwent several modifications to better suit the robot for the City Driving challenge.

1) A Optimization (Ayden):* The A* algorithm we used to find the shortest path between two points was refactored to eliminate redundancies and improve the runtime of the module. Adjustments were also made to the processing of the ROS2 message that holds data on the map of the Stata Basement. A blurring filter was applied to the array of data so that obstacles indicated in the map (i.e. walls and other objects the robot can't move through) appeared larger than they actually were. This was so that the A* algorithm didn't draw a path too closely to any obstacles that the robot wouldn't be able to follow. The grid was also discretized into triangles rather than a square grid as we had in the previous lab, which created smoother path plans.

2) Trajectory Planning (Gloria and Erin): When planning our overall trajectory, we took two different approaches. First, we simply ran A* from our start location to shell 1, shell 1 to shell 2, etc. Although this approach technically worked, we found that it was often inefficient computationally and time-wise (more about this Exp. Eval.), and many stretches of the path were being recomputed over and over again with each trial. Thus, in our second approach, we took advantage of the fact that we were in a known map space with a known basic route. We manually plotted a sparse, optimized set of trajectory points along the hallway (Fig. 3). Then, to get to a shell, we would plan a "detour" path off our "hard-coded" path from computed "exit points." In order to determine these exit points off of our base trajectory, we cast a ray from the shell to the known route and walked backwards by around 1m. Then, we ran A* from the exit points to the shell locations with a step size of about .2m. (Fig. 4). This strategy allowed us to compute all our paths in significantly less time, as we were only running A* on paths of about 3m long maximum. Additionally, we were able to vary the resolution of our trajectory depending on what we were trying to do: when we were in the long stretches of hallway, we would be driving along our known route, which consisted of a sparse set of points usually many meters apart. Then, when we were targeting the shells, we would be driving along our detour routes, in which the points were much closer together, as we wanted to have high precision in locating and stopping at our targets. By utilizing this variable resolution, we were able to keep the compute time of our pure pursuit time minimal while reaching the precision levels we needed.

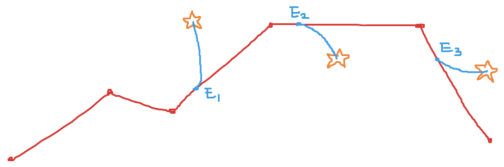

Fig. 3. Because the mission would traverse the same hallways in Stata Basement, we knew that we could essentially hard code a main path (in red). We then reached the shells (stars) through detours from the path (in blue).

Fig. 4. We used our optimized version of our A* algorithm to determine the detours (blue) from the path (red) to reach the target shell (star).

3) Backing Up (Gloria and Erin): Our last step was to figure out how to return to the path from these exit points. We did not want to assume that we could make a shallow turn back to a future path point since there might be a wall or other obstacle. Thus, we used the existing detour path as our return trajectories to the main hard-coded one, and we navigated the car backwards along this path. Backing up

allows us to avoid having to over-complicate our maneuvers to get from the shell locations back to our planned path.

In order to publish a variable amount of trajectories to our pure pursuit follower, we implemented a custom Traj message type in which we could specify the number of trajectories within the message as well as the direction in which to drive along them (forwards and backwards). Then, we could finally focus on actually following the path using our pure pursuit algorithm from the last lab (Fig. 5).

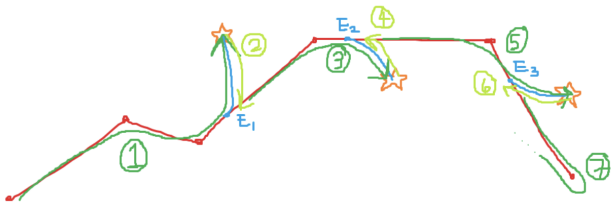

Fig. 5. All path plans for a generic route. The main path is in red, which is hard coded. The shell points (stars) are unique per round, so we only compute the blue and green detour paths from the main path to these shells.

D. Pure Pursuit (Gloria)

We made minimal changes to our pure pursuit module besides adding functionality in order for it to follow a trajectory backwards rather than forwards. Additionally, we switched to a geometric method for testing whether we had arrived at our target or not. This was because with the previous method, we found that we often stopped prematurely, as our stopping condition was if our localized pose was within a certain distance of the target. However, as our localization output would bounce back and forth along the x-axis of the robot, this meant that it would sometimes believe, if only for one epoch, that our robot was within stopping distance. Thus, to make this check more robust, we changed the condition to 1) being within a certain distance from the target for at least 1 second and 2) being closer to the target point than the second-to-last point in the trajectory. This greatly increased our positional accuracy when deciding when to stop at the target.

E. Localization (Gloria)

With our previous implementations of localization, we found that noise was difficult to deal with in the modules that made use of odometry. We wanted to keep the same levels of noise in the localization algorithm itself, as that made it robust to robot position changes, but we wanted to smooth out the resulting guess pose when implementing pure pursuit, proximity-to-crosswalk detection, etc. Thus, we applied a simple low pass filter to the output of our localization module in order to attenuate those noisy high-frequencies (Fig. 6).

Fig. 6. A graph showing the effects of filtering our localization outputs using a low pass filter.

F. Computer Vision (Erin)

Finally, this mission involved two tasks that required the use of the camera, so a stop detector node was implemented to manage both tasks. First, we had to identify stop signs, as well as how far we were from the sign. We used a machine learning algorithm that took in images and compared them to a large database of various images of stop signs to determine if there was a stop sign in the given image. If there was, the algorithm produced a bounding box around the sign.

The presence of the bounding box helped us determine if there was a sign, but we also needed to know how far away from the sign the vehicle was to know when to stop. We knew that the area of the bounding box of the sign would increase as we got closer to the sign, so we took empirical measurements to determine the box area at which we would need to stop the vehicle (Fig. 7).

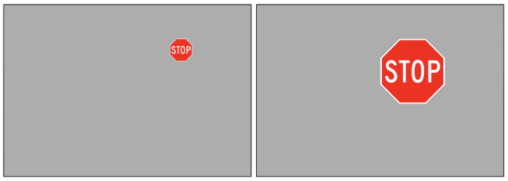

Fig. 7. The stop sign detection algorithm used machine learning to determine the presence and location of a stop sign in a given image. If a stop sign was present, it determined the bounding box around the sign. We used the area of this box to determine how far our car was from the sign, so that we could send a stop command to our driver node when we were within a predetermined stopping distance.

Secondly, we also used our stop detector node to determine the color of a traffic light. We used the given coordinates of the crosswalks with traffic lights along with our guess position of the car to only turn on the "search for traffic light" code when we were approaching these crosswalks. Thus, we were able to reduce the amount of code that was running at once, which ideally sped up our car's processes.

Once we were close to a crosswalk with a traffic light, we searched for a very specific color of green. This color was determined to be very unique across 150 test images taken of both green and red lights from various angles around a traffic light (Fig. 8).

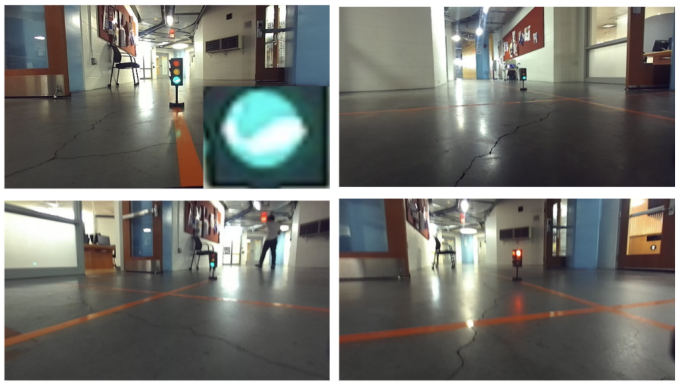

Fig. 8. Various test images of the traffic light in both its green and red states. These images were taken to help create and test a computer vision system that could correctly identify the color of the traffic light.

Using a specific range of HSV colors, we were able to determine if the green light was on or not, but we could not determine the presence of red since the color was not as unique compared to everything else in the image. Thus, our algorithm only searched for green, and when there was not green, we assumed the light was red.

III. Technical Approach for Mario's Circuit (Greg)

We've seen how to approach using our vision and lidar to successfully control a racecar in a setting mimicking urban traffic conditions. Despite navigating and responding to a complex environment composed of traffic signs, stoplights, and pedestrians, we have ignored a significant component of vehicle use in the world: high-speed travel in lane-systems on highways. In this section we demonstrate a solution to this problem on a toy example: racing around the lanes on the Johnson track.

The main components in a system designed for maneuvering forward at high velocity in a lane that potentially has turns are 1) localizing where you are in respect to the lane and 2) controlling to stay within at a reasonable distance from the lane while driving fast. We use computer vision for 1), and a PD controller based on some geometrically derived quantities for 2).

On a high level, we explain the considerations that go into high-speed control in simple lane-based environments, work through the components of our CV localization algorithm, as well as the reasoning behind our final controller.

A. Racetrack Specifics (Greg)

The racetrack is a 200m track composed of 6 lanes, each about a meter in width. The racetrack is a reddish-brown in color, but the lane lines dividing it are starkly white. However, the lane lines are not the only white features on the track, there are also large white numbers, as well as various diagonal and horizontal white lines. We wish to race around the 200m track in a designated lane counterclockwise. The racetrack is composed of two straight-line segments, as well as two curved segments.

Fig. 9. Sample images collected by racecar mounted zed camera along a diverse set of points around the track. Various misleading non-lane white features are visible in the images.

The challenges of localizing and controlling our racecar comes down to robustly recognizing the location of the lane from real-time camera images with a computer vision algorithm and controlling smoothly to stay inside the lane while driving at a high speed.

The computer vision challenges are 1)detecting white features corresponding to the lanes on the racetrack at all speeds 2) outputting a set of real coordinates corresponding to a lane we want our control algorithm to take in.

The main control challenge is creating a controller that can work smoothly both on the straight segments of the track as well as on the curved portions.

B. Properties of High-Speed Lane Detection (Greg)

Autonomous vehicles are expected to operate over a wide range of speeds (0-120mph) while still following vital traffic rules related to lanes. This means that autnomouous vehicles should be capable accurate sensing of lanes not just statically, but across the range of speeds they inhabit.

We were similarly curious in making a CV system that would work well over a wide range of speeds, both for scientific reasons, as well as the very practical reason of our racecar needing to localize itself with respect to the lane at speeds reaching up to 4.5 m/s.

To deal with this, we specifically collected our set of testing images for our CV by operating the racecar on the whole distribution of speeds we expected it to operate at, all from the car's point of view. We additionally ensured that our lane localization procedure was computationally tractable in order to allow us to be responsive in a environments where our robot's position is changing significant amounts between subsequent frames.

C. Localizing With Respect to the Left Lane Line (Greg)

In order to localize our robot in respect to the lane it is in, we use a CV algorithm that sequentially:

1) Filters and dilates the image to obtain a black and white mask that contains the lane features
2) Finds a series of pixels that form a backbone of the left lane line of our lane.

3) Use a homography transform to convert these points to real-world points in our racecar's reference frame.

We describe each of these steps in full.

1) Filtering + Dilation: We filter the image for pixels with HSV values in the bounding box defined by $[0, 0, 150]$ and $[359, 180, 255]$. These correspond to colors that compose the majority of the lane lines we want to detect, so filtering for those values is an simple solution to detect the lanes. Because it is simpler to treat the lanes as filled-in solid objects in our image, we additionally apply a 3x3 dilation kernel that smooths any aberrations. The result now contains our left lane line as a smooth white curve on a black background, however there are still other white features in the image so we need to take further steps to detect the lane.

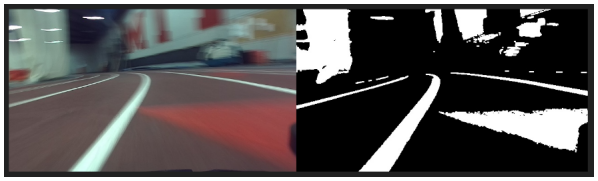

Fig. 10. Example of the filtering and dilation performed by our CV algorithm on an input images seen during a race.

2) Constructing a Pixel Backbone for the Left Lane: We use our knowledge of the track and the relative position of the left lane line with respect to a racecar in that lane to significantly simplify our process for contructing a backbone. We primarily use the assumptions that 1) the left lane line will originate somewhere on the bottom of the image, and if not there then somewhere in the lower part of the left and 2) it will shoot upwards.

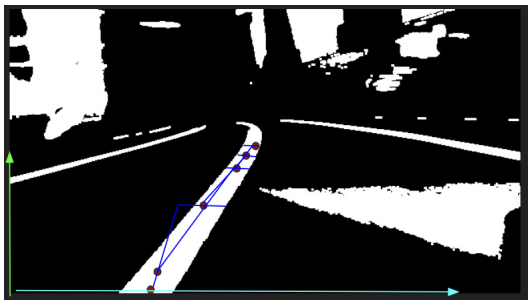

Fig. 11. Example of the filtering and dilation performed by our CV algorithm on an input images seen during a race.

We want to identify a backbone of points $p_0, \cdots, p_6$ describing the midpoints of the lane line as it recedes into the horizon.

We find p_0 by searching the bottom and then the left of the image for continuous streaks of 8 white pixels, which upon finding we take the midpoint of to be p_0. Due to the structure of the racetrack, we will always find the start of the left lane line somewhere here if the racecar is in a reasonable position.

To find the remaining p_i, we use a recursive algorithm where we repeatedly shoot upwards some Y_i number of pixels from p_{i-1} according to a fixed schedule, and then search left and right for continuous streaks of white points, taking the midpoint of the nearest one to be p_i. This algorithm is very computationally efficient, as it uses on average $O(100)$ queries to the image sequentially. It also guarantees that the backbone will be sufficiently spread out vertically to give a good description of the images.

3) Using homography to output real points: Finally, we take all the points in the pixel backbone $p_0, \cdots p_6$ and apply a matrix homography transformation H to get real world points $x_0, \cdots, x_6$ in the frame of our racecar that describe the left lane line.

D. Calculating following point (Greg)

For our control algorithm, we were surprised that we could use a monolithic controller for both the curved and straight parts of the racetrack. It sufficed to use our the backbone of points that comprised the left lane to geometrically derive a point ahead of us on the track to follow with a PD controller. We explain what point we are following in this section.

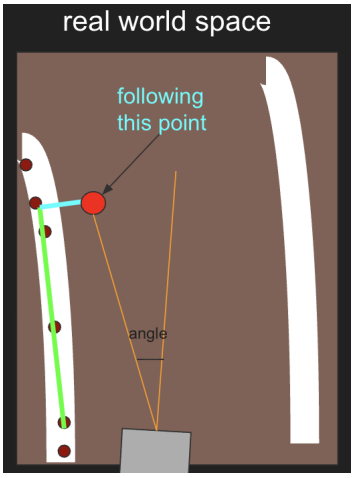

Fig. 12. Top-down depiction of using real-world points from our locatlization to find a point to follow.

We derive the point by calculating the equation of the line $Ax + By = 1$ with $A^2 + B^2 = 1$ passing through points x_1 and x_4, after which we took the normed vector $v = (A, B)$, and added a scaling of it to x_4 to get the point we want to follow of $s = x_4 + d \cdot v$, where d is the distance we want to stay from the left lane line.

E. PD Control (Greg)

We also found that a simple PD angle controller sufficed to follow the above point, resulting in smooth high-speed motion that stayed within our designated lane.

As the backbone of points $x_0, \cdots x_6$ was in our racecar's reference frame, the point s is also within the racecar's reference frame, meaning that we can calculate the angle θ between s and the point $(1, 0)$ in the forward direction, and then use a PD controller for our steering angle with the goal of controlling θ to be 0.

We go into detail about our specific parameter choices and tuning the controller in the experiments section.

IV. Experimental Evaluation for Luigi's Mansion(Gloria and Erin)

Because there were so many moving parts in this lab, and especially since we were using all our sensors at the same time (lidar, zed camera, odometry, etc.), integration was a challenge. We were able to get our trajectory following system to work successfully in simulation using our own version of localization, but we were unable to cross the sim2real gap to deploy this same code on the vehicle as successfully.

We were never able to get our own localization code working during the localization lab, so we had been implementing the TA's solution in the path planning lab, which worked successfully. Thus, we assumed that we would be able to use it as successfully in this lab as well. However, we found that it struggled with bandwidth in this specific application, and we were unable to get it to run alongside the camera. Along those lines, we struggled to integrate our stop detector with the rest of the system. We had tested this node separately on the vehicle and it had run successfully, but there were problems when adding it to the rest of the nodes.

Because of these problems in deploying the real vehicle, we found that evaluating our personal code in simulation would be more accurate, as our own localization solution worked well in simulation, just not in the real world.

We first looked at the effects of hard-coding the main path and only using A* for the small detours between the path and each shell versus doing A* between each shell. As Figure 5 summarizes, we found that the hard-coded path method proved to be almost 10-15x faster on average than the full A* method due to shorter segments planned with A*, and fewer points on the main trajectory.

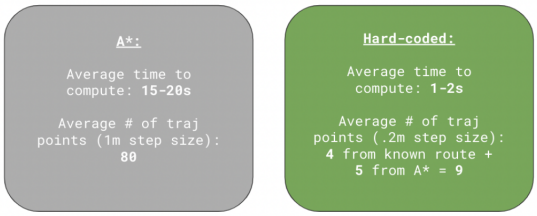

Fig. 13. A comparison in run time and number of trajectory points between using A* to compute a full path between shells (grey) and using A* for smaller detour paths with a hard-coded main path (green).

We then evaluated our traffic light detector to test its robustness in various lighting conditions and at various angles. We found that across all 150 test images of both green and red lights, we were able to identify the correct color light 100% of the time.

Finally, we wanted to confirm that the stopped distance between the vehicle and the target point was as small as possible, or at least reasonably close to the target. We expected our variance parameter in the Gaussian distribution of our particles for localization to affect this value, as well as our lookahead distance. To do this, we created a trajectory of more than fifty points for the car to follow in simulation, and determined the average distance between the car and each point. We found this average stop distance to be 0.32m, which seems reasonable as a stop distance from a target point, at least in a use case as in this lab.

V. Experimental Evaluation for Mario's Circuit (Greg, Jose, and Erin)

We walk through how we evaluated our racecar on Mario's Curcuit, as well as our specific hyperparameter choices to ensure good performance.

We wanted to evaluate our approach to Mario's Circuit using two metrics:

1) The time it took for our car to go 200 meters around each lane since each lane has a different curvature
2) A comparison between our steering angle and the car's actual angle to the set point to evaluate our controller design

However, very early in our data collection process, our car "bumped" into an obstacle on the track, and would not turn on again. Thus, we were unable to collect any data.

Therefore, we imagine that our car would traverse 200 meters faster in the outer lanes than the inner lanes due to shallower turns. Furthermore, we expect that our controller performed quite well, at least compared to other teams in the class as our car was one of the fastest around the track, despite being in lane 2. We also had zero infractions, which perhaps shows that our combined computer vision and controller algorithms worked well together.

We outline what controllers we found to work well as well as what kinds of modifications to the CV system made it more robust below.

A. Tuning our PD controller

We noticed that a K_p value of 0.2 multiplied by our angle difference in radians was enough to oscillate around our lane, as well as effectively steer enough to handle the curved segments of the track.

We found that a K_d value of 1.6 worked well for following straight curved segments. We found it to work *really* well on curved segments, consistently staying at the very left of the lane with almost no room to the left lane line, almost drifting. This saved significant amounts of time, as on the curved segment the closer you are to the center of the curvature the less distance you have to cover. However, despite being extremely consistent on turns, it was less consistent on straight line segments, occasionally moving out of bounds. We suspect there is a simple fix for this that would preserve our extremely advantageous control on the curved segments, but in order to guarantee our performance in the challenge we opted to instead amp up our K_d value to 6, which made it smooth on straight segments, albeit now more mediocre on curved segments.

We found other crucial design choices to be our lookahead distance (determined by the index i of the point x_i from which we determine our following point) as well as the setpoint

distance d from x_i where we set s. We settled on $i = 4$, and noted that further i gave stabler control due to being less sensitive to the racecar's position, however were less consistently detected by our CV algorithm. We found that d is a parameter that needs to be often tuned depending on the racecar's battery level. We used values in the range $[0.165, 0.3]$, opting for larger values when the racecar had lower battery.

B. Adapting to the CV algorithm breaking

Occasionally in certain moments external factors will line up that cause our CV algorithm to not be able to find a good match for a left lane line. In these cases we pass an "lane unknown" message to our controller. Rather than using whatever pathological guess our CV might have for the lane based on very distorted sensory input, we instead choose to simply repeat the sensor signal from the previous message. We hypothesize our CV algorithm never breaks for significant periods of time as we experimentally found this work well.

VI. Conclusion (Jose)

For the final project, we put together all the previous labs to compete in Luigi's Mansion and Mario's Circuit. For Luigi's Mansion, this was done through the usage of localization, path planning, the safety controller, and computer vision. These modules were used to develop a program that would arrive quickly at the desired destinations while still following traffic laws. As for Mario's Circuit, we mostly used a more advanced version of our previous computer vision programs. In effect, we learned how to put together our previous work, and improve on the bases that we had been given or had made ourselves.

A. Future Work (Jose)

As for future developments, the inclusion of a state machine sets the stage for much more complex work. By separating how the robot reacts into different states, and setting rules for how states change, we can create dictate how the robot reacts to certain events, and make clear how the robot must respond. Apart from that, learning ROS will give an advantage with other projects. By using ROS to communicate between different areas of the robot, we can ensure that every node can pass information to each other, regardless of what base coding language they use. Additionally, computer vision is very popular currently, especially when combined with AI, so having learned that in this class will give a head start on other potential projects.

B. Lessons Learned (Ayden)

I think the biggest lesson for me was to be more patient with editing and optimizing the path planning code, because several of my initial changes ended up making the code work slower than before, so I had to take a step back and reconsider what parts of the code actually needed optimizing.

C. Lessons Learned (Jose)

It was very nice to see all that we had learned in the previous labs come together for the big final project. I especially enjoyed linking together old nodes like the safety controller with the new code, and making sure that it still worked. Also enjoyed debugging the stop light code, even if it was annoying at the time, it is still a good feeling to know that it worked at the end. Also learned that trying to test on the stata basement at the end was the same thing that everyone else tried to do, so it didn't work out the best for us.

D. Lessons Learned (Erin)

This final lab felt like a nice conclusion to the class, where we got to integrate multiple nodes into a larger ROS system. I enjoyed thinking about the mansion lab on a higher system level, especially from sketching state diagrams on a white board to converting it into a node-topic diagram, to finally seeing how we were able to implement each node together using code that we had already written. As someone who has not worked with so much code before, it was cool to see how more experienced members of the team organized the code into functions and separate files rather than sticking everything into one file like I would have done. It was also a good way to involve multiple teammates into coding one file since each person could code a function.

E. Lessons Learned (Gloria)

I really liked working on this last lab! It was cool thinking about the system at a higher level, and being able to (mostly) assume that all the modules would work as planned. Thinking about the state transitions and exit conditions was an interesting foray into designing the overall software architecture of our system, rather than getting into the weeds of A* implementation or something. Also, it was cool just seeing all the labs we worked on come together to form the entire system.

F. Lessons Learned (Greg)

I learned how important it is to debug all the components of a system step by step, as well as being extremely aware of how many robotic components the systems I'm designing are relying on, and hence what distribution of things could go wrong or vary day to day. With that in mind, I'm more conscious that ever of distribution shift and the importance of designing decision-making systems that actually adapt to it on the go and can be deployed at large scales for economic productivity reliably.