Final Challenge Report

HAzMAT (Team 12)

May 14, 2024

1 Introduction: Alonso, Heath

In combination, the two portions of the Final Challenge aim to accomplish a pertinent goal in the field of vehicular robotics: safe autonomous navigation. The first challenge, Mario Circuit, involved the creation of an algorithm to reliably detect and navigate between lane lanes. The second challenge, Luigi Mansion, involved the creation of not only algorithms, but also high level system architectures to obey the rules of the road and abide by traffic infrastructure while executing navigation from point to point. In combination, the two challenges are an amalgamation and extension of all the previous RSS Labs. Across the challenges, computer vision, controls, coordinate transforms, localization, path planning, and even machine learning comprised the building blocks used in the design of the implemented solutions.

The goal of the Mario Circuit challenge was to design an algorithm that allowed the car to race around a 200m track. The evaluation criteria was lap time and lane breaches. Given the real-world implication of accidental lane breaches, breaches were punished heavily. The applied solution, which will be discussed further in the technical approach section consisted of three distinct steps. First, using the onboard camera and image processing, a mask based on HSV values was created to pick out the lane lines. Second, using Hough Transforms, the lines were then processed and average lane lines were computed. Finally, a pure pursuit controller was implemented to chase after a point between the lines at a set distance.

The goal of the Luigi Mansion challenge was to implement system architecture that allows the race car to autonomously navigate between goal points while also abiding by common traffic markings such as stop signs, cross walks, and stoplights. The evaluation criteria was a combination of achieved way points and total solution time, with penalties imposed for moving violations. The applied solution, which will be discussed further in the technical approach section consisted of a variety of nodes with different levels of hierarchy for the outputted drive commands. At the base level was the drive command output of the path planning nodes which utilized pure pursuit to find a target point based on the creation of a Probabilistic Road Map (PRM). Above that were the nodes that dealt with stoplight and stop sign detection, which either outputted no drive command, or set the speed to zero. Last was the safety controller with the highest priority drive command output to ensure the protection of the race car and members of the community.

2 Technical Approach

2.1 Mario Circuit: Heath

The first step of the implemented algorithm was to take the image received by the robot's camera, and produce a mask to aid in lane detection. In order to ensure that the mask is only picking up the lanes, the top half and bottom quarter of the image were removed. The newly produced rectangle is then converted to a trapezoid, tapering to half the image width at the top, so that the only lanes that are shown in the mask are the lanes directly to the left and right of the robot. For determining the location of the lanes, color segmentation was used to create a mask with the following ranges in the HSV color spectrum:

$$\text{Upper Bound} = [179, 50, 255]$$

$$\text{Lower Bound} = [0, 0, 200]$$

The mask is created by thresholding the image such that pixel values falling within the specified upper and lower bounds are set to 1 (white), while all other pixels are set to 0 (black). A mask allows for the selective extraction of regions of interest effectively isolating the lanes within the image. Given the consistent color of the lanes this method allowed us to reduce the impact of noise and irrelevant features on the lane detection. An example of the mask created can be found in Figure 1.

Figure 1: A mask of the Mario Circuit lanes which does not include the top half and bottom quarter of the image. Additionally, the remaining rectangle is converted to a trapezoid, where is it skinnier on the top to remove nonadjacent lines.

Once the lane regions were identified, a Hough Transform was used to pick up the lines of the track. A Hough Transform is a technique used for detecting geometric shapes, particularly lines and curves, within an image. It works in the following way:

1. **Edge Detection:** Before application, edges are detected in the image by identifying pixels in the images where there is a significant change in intensity.

2. **Parameterization:** Each edge pixel in the image is interpreted as a point in the parameter space and are used to detect lines in slope-intercept form.

3. **Accumulation:** Each edge pixel contributes to the accumulator array in the Hough space but first needs to be converted from the image space to the Hough space.

$$\theta = \arctan\left(\frac{-1}{\frac{dy}{dx}}\right) \qquad\qquad r = x\cos\theta + y\sin\theta$$

4. **Voting:** Edge points that lie approximately on the same line in the image cast votes for corresponding lines in the parameter space. The lines with the most votes in the accumulator array represent the detected lines in the original image.

The Hough transform process results in lots of different lines (as seen in Figure 2), especially when cross lanes are apart of the mask (as seen in Figure 1). In order to determine the lines that are relevant to the calculation of the direct left and right lanes, angle ranges are applied to split up the lines into groups.

$$\text{Left Lanes} \in [0.4\text{rad}, 1.3\text{rad}] \qquad\qquad \text{Right Lanes} \in [1.7\text{rad}, 2.2\text{rad}]$$

Using the left and right hand lines, average ρ and θ values are determined which result in average left and right lines. These averaged lines are then used to determine the car trajectory.

In the implemented approach, all left-hand and right-hand lines detected within the are gathered. The average values of their corresponding polar coordinates, namely ρ and θ, are then computed resulting in the determination of an average line, representing a coherent direction amidst the detected left and right lines. Subsequently, the average line is leveraged to establish a point of focus for the navigation strategy. At a predetermined look ahead distance, approximately 2 meters ahead in the image space, a point is projected onto the roadway.

This projected point, situated equidistantly between the left-hand and right-hand lines, serves as a navigation target. It encapsulates the optimal trajectory for the vehicle, derived from the collective insights of the detected lines. The pure pursuit controller is then employed to steer the car towards this point. By steering towards this strategically positioned point, precise and adaptive navigation within the circuit environment is ensured.

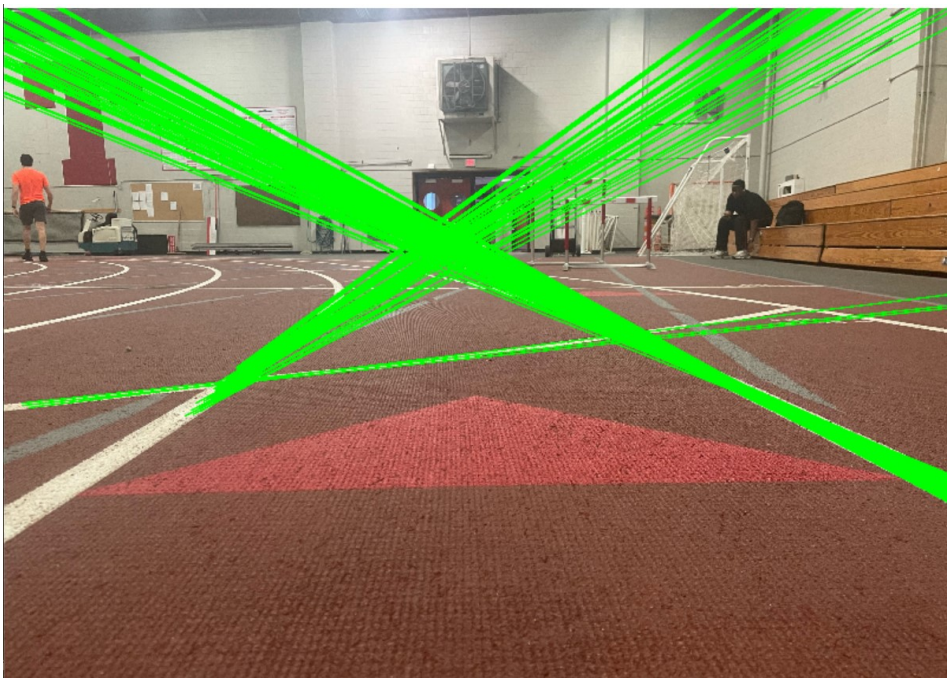


Figure 2: The lines produced by the Hough transform that must be categorized into related the lanes directly to the left and right of the car.

2.2 Luigi Mansion

2.2.1 Trajectory Planning: Thomas

Adhering to one side of the road is crucial for safe driving conditions in Luigi's Mansion. Therefore, adopting a strategy to combine the given midline of the road and the shell locations to form a trajectory was ideal for successfully navigating the landscape.

In order to accomplish this, many of the previously developed modules, including trajectory loading, planning, and following from the previous lab, were utilized. This allowed for consistent performance between modules and efficient time usage

At a high level, the trajectory planner works in three distinct stages. First, all the callbacks are initialized such that an event is not accidentally missed and, therefore, a callback is not triggered. Then, the map is initialized which triggers the map callback, generating a point distribution representing the map for path planning. Finally, shell locations are input, and trajectories are planned and merged from the midline to the shells to be fed into our pure pursuit algorithm. As stated, the first step of the algorithm is subscribing to the map topic. This allows the map callback to be triggered, a function present in our existing trajectory planner file. At a high level, this generates a distribution of 10,000

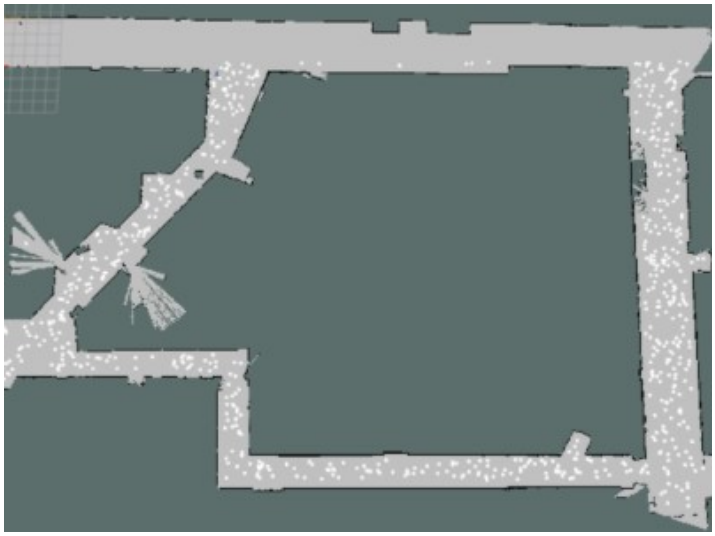


Figure 3: The particles are randomly spread out through the bottom 2/3 of the map, cutting out the unused sections.

randomly sampled points that span the map, and eliminates any that are not in valid locations such as too close to any obstacles or empty space. Using this Occupancy Grid, a Probabilistic Roadmap (PRM) graph for path planning is created and visualized in the simulation. This is returned to the main `city_drive_node file` in the form of `self.map` such that it can be used for path planning when shell locations are given.

After this, the program receives the given midline trajectory, consisting of ten points throughout the course that the robot should not cross. This triggers another callback, which begins by interpolating this path. Due to the nature of our algorithm, a fine distribution of points is necessary to pick a reasonable starting and ending location with respect to a placed shell. As such, an interpolation value of one meter was chosen such that the midline path was segmented into 116 points throughout the track.

After obtaining the interpolated track, an offset of 1.5 times the car's wheelbase was chosen such that the robot could closely follow the line during pure pursuit without overlapping it and also maintaining a reasonable distance from the wall. It begins by first calculating and normalizing the direction vectors between each point on the interpolated path. Then, it creates a vector perpendicular to that path and calculates the offset passed on, shifting each point by a factor of `self.wheelbase_length * perpendicular_vector`. Furthermore, for angles close to 90 degrees, an additional offset in the direction parallel to the path was desired such that the robot had more clearance to make the turn. This process was repeated for each pair of points along the path until all had been offset on either side of the path. These offset trajectories for both the right and left sides of the track were stored in PoseArray objects and merged to create a full loop around the track.

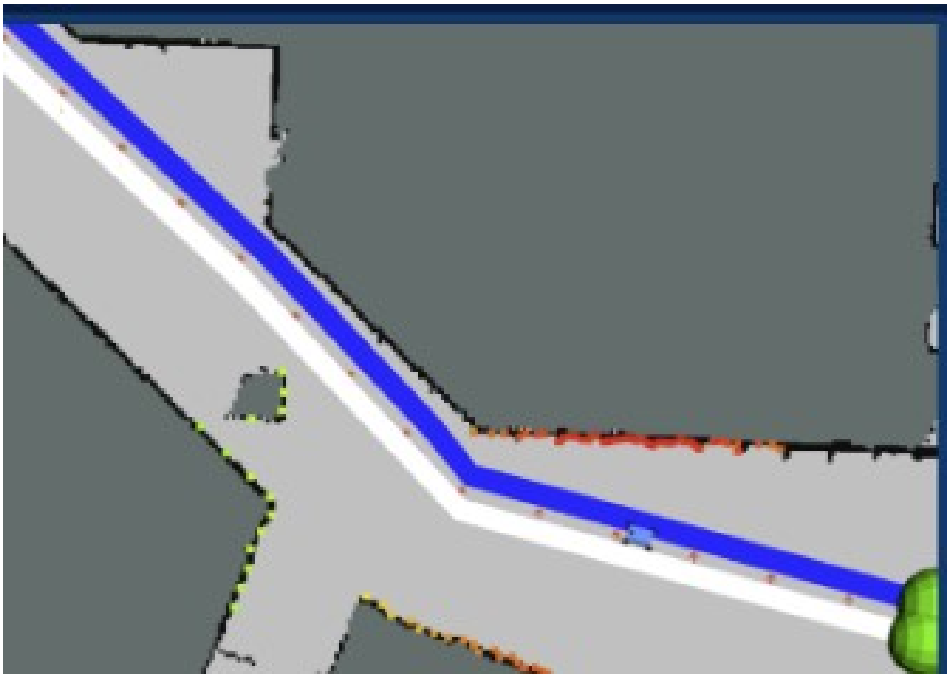
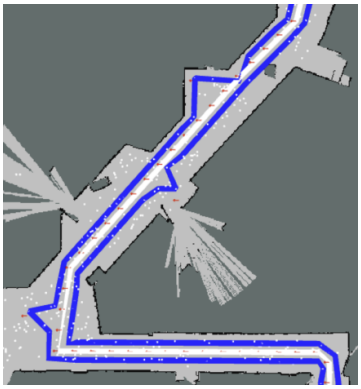


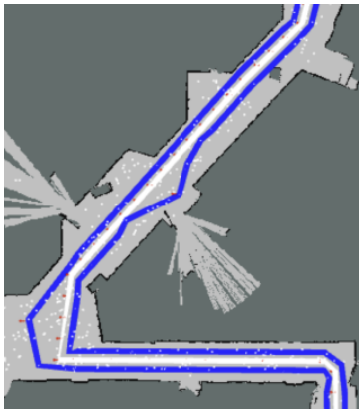
Figure 4: The midline trajectory is in white which got interpolated to make the shell paths easier to make.

Finally, once the offset trajectory was created, a process for inserting new trajectories based on shell positions was vital for an effective path. Upon receiving three clicks anywhere in the valid map space, a shell callback was initiated which stored the points as a numpy array and created a new shell path trajectory object. For each shell, utilizing the offset trajectory points for both sides of the midline, a path was created from the offset path to the shell and back to the offset path. This was done by first creating a helper function `self.find_intersection_points` in order to detect the closest points at a reasonable distance from the shell in order to path plan a smooth trajectory. This number was experimentally determined to be searching for points at a four-meter radius away from the shell. If it was less than four, the planned paths became very jagged, and if it was too far, they had the potential to cross the line. Once all valid points were obtained, it was important to consider only two points adjacent to each other and on the correct side of the mid-line. This was done by projecting the position of the shell onto the vector formed between each adjacent point and choosing the pair with the smallest projection value.



(a) Shell lookahead of 3 meters creates sharp turns which are unnatural for our robot.

(b) A lookahead of 5 meters caused smooth curves, but on 90 degree angles it would often cross through the midline.



(c) A lookahead distance of four meters combined the best of these two approaches and created a smooth path that respected the midline.

Figure 5: Comparison of different lookahead distances for point selection during path planning from offset trajectory to shells.

After proper starting and ending positions along the offset trajectory were determined, the `self.planner.plan_path` function was called utilizing the trajectory planner created earlier, given the starting and ending node as a PoseStamped object as well as the map in the form of the point distribution. From this information, distinct PoseArray objects were created for both the `start_to_shell` and `shell_to_end` paths.

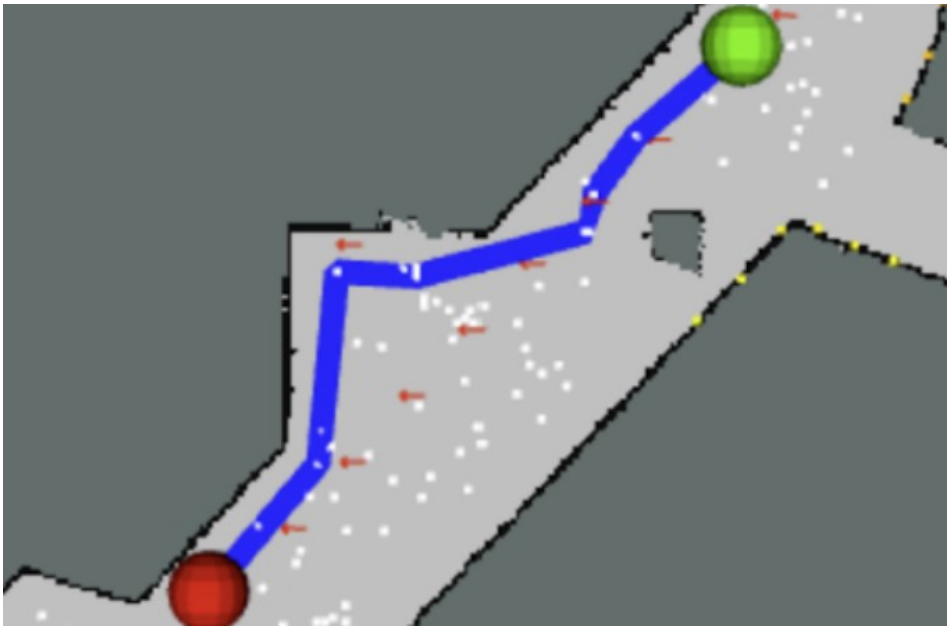


Figure 6: The final generated trajectory showing a full loop around the track with three shell locations path planned to using the map point distribution.

Finally, the positions of the starting and ending nodes were found in the context of the overall offset trajectory. A new pose array representing the updated trajectory was created by merging 1) the start of the offset trajectory to the starting node, 2) the start-to-shell path, 3) the shell-to-end path, and 4) the end node to the end of the offset trajectory. This process was repeated for each shell until a path was generated that demonstrated a route from the start to the end and passed by each of the shell locations.

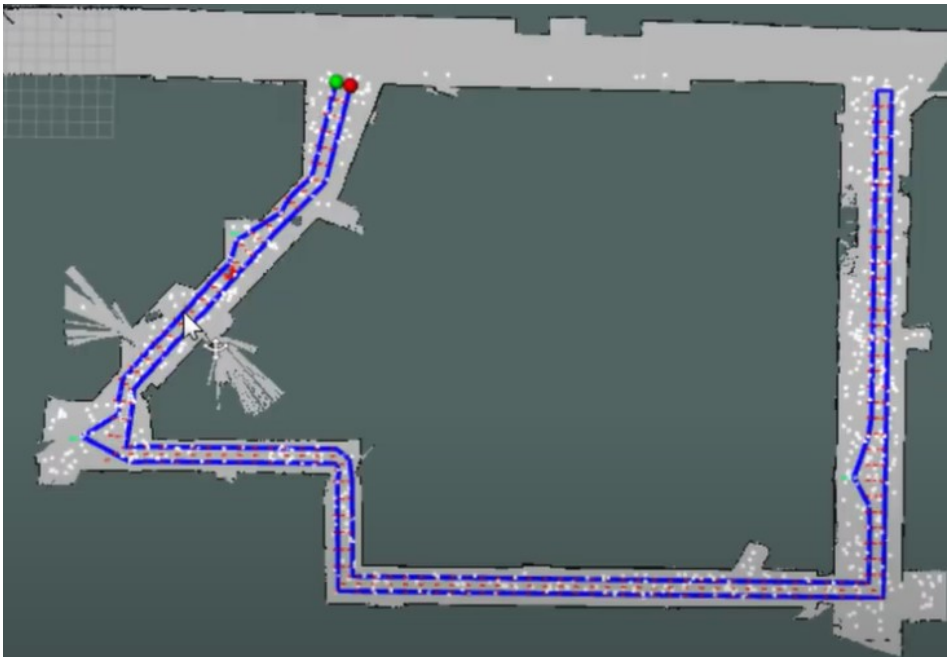


Figure 7: The final generated trajectory showing a full loop around the track with three shell locations path planned to using the map point distribution.

2.2.2 Pure Pursuit: Alex

Given that an autonomous vehicle knows how to localize itself in a given environment, if it is given a trajectory to follow, then it must be able to use its localization information to follow this path. For Luigi's Mansion, the robot is tasked with complying with city rules to gather three shells and return to the start, so after it receives a trajectory, it must follow it well so as to not crash into obstacles or pedestrians.

There are various methods to do this, but the method used for Luigi's Mansion was to manually transform the points from the `map` frame into the `base_link` frame. First, the robot takes in an Odometry message and extracts the quaternion $q = \begin{bmatrix} x & y & z & w \end{bmatrix}^T$ describing its orientation relative to the `map` frame. Then, the function `tf_transformations.euler_from_quaternion` takes in q and returns the Euler angles from which the yaw angle z is the desired one. The rotation matrix $R_r^w = \begin{bmatrix} cos(\theta_r^w) & -sin(\theta_r^w) \\ sin(\theta_r^w) & cos(\theta_r^w) \end{bmatrix}$ is formed, and is turned into a pose $T_r^w = \begin{bmatrix} cos(\theta_r^w) & -sin(\theta_r^w) & x_{robot} \\ sin(\theta_r^w) & cos(\theta_r^w) & y_{robot} \\ 0 & 0 & 1 \end{bmatrix}$. This pose is important because it will allow for the conversion between the `map` frame and the `base_link` frame which is necessary to facilitate pure pursuit. The following relation will do this:

$$T_r^w * \begin{bmatrix} x_{target}^w \\ y_{target}^w \\ 1 \end{bmatrix} = \begin{bmatrix} x_{target}^r \\ y_{target}^r \\ c \end{bmatrix} \tag{1}$$

Now, this process is done for every Odometry message that the robot produces, which allows for pure pursuit to then be done.

With the target in the robot's frame, it is easy to distinguish whether the target is to the left or to the right of the robot. Since the convention is that the positive x-axis lies perpendicular to the axles of the robot and forwards, right-hand rule says that the positive y-axis is left of the robot and parallel to its axles. Then, after completing this step, the code from the pure pursuit controller from Lab 6 can be reused to follow the trajectory.

2.2.3 Stop Sign and Traffic Light Compliance: Marcelo

Another key part of autonomous vehicles is complying with signage and traffic regulations, especially those that indicate that vehicles should stop at certain points. As such, the robot must be able to

recognize and react to stop signs and traffic lights. This is done in a two step process, recognition then reaction.

In the case of the stop sign, the robot must stop for at least 5 seconds in order to allow other vehicles and pedestrians to cross. This is achieved by identifying the sign via a neural network. The `stop_detector` node achieves this by subscribing and saving the latest image from the camera feed, then feeding this image through the neural network at a specified time period. If the neural network identifies a stop sign, it then identifies it and defines the bounding box around the sign.

Figure 8: A stop sign detected and framed in its bounding box

Once the sign is identified, the existence of a similar triangle between the bounding box in the image and the actual borders of the stop sign is utilized: This relationship allows us to find the distance from

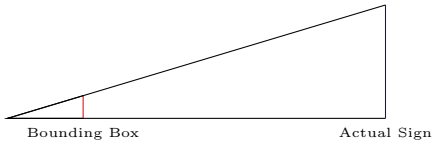


Figure 9: Similar triangles between bounding box in image and object

the object using the size of the bounding box in the image plane and the lenses focal length, which is obtained from the devices specifications, by using the following formula:

$$\text{Distance} = \frac{\text{Sign Height} \cdot \text{Focal Length}}{\text{Height in Camera}}$$

Should the sign be closer than a given stopping distance, the robot then brakes by using a drive command with higher priority than other driving commands. This node keep publishing the stop command until the desired time period has elapsed, where then no more messages are published and control is released to other nodes.

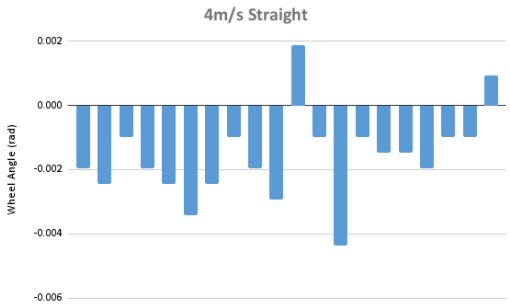
The stop light controller works in a similar way, but instead uses color segmentation in stead of a neural network. By removing unnecessary parts of the image, such as the road, the robot can search faster through this area for any items within the expected values for a red stop light. Should there be a match, the robot stops until no match is detected.

3 Experimental Evaluation

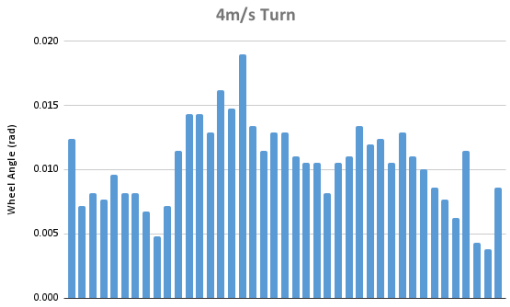
3.1 Mario's Circuit: Heath, Alonso

During the evaluation of the Mario circuit algorithm, the monitoring of wheel angles was conducted across various segments of the course to assess the vehicular performance and identify potential sources of error. As illustrated in Figure 5, a graphical representation of the wheel angles during a straight section of the circuit while driving at 2m/s is depicted. Observations reveal predominantly negative angular values, punctuated by a few positive outliers. This trend is attributed to an inherent leftward

drift exhibited by the vehicle. To mitigate this tendency, a corrective strategy involving the subtraction of a constant turn angle from each angle measurement was employed.



(a) Wheel angles during the straightaway of the circuit.

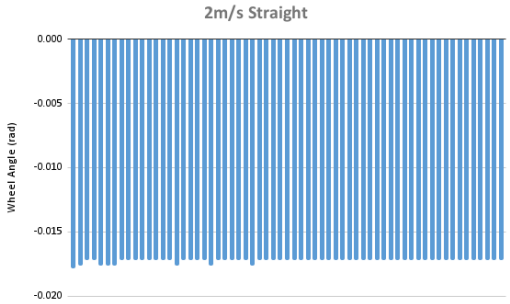


(b) Wheel angles during the turn of the Mario circuit course.

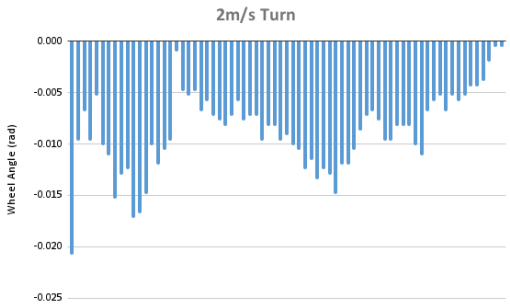
Figure 10: Wheel angles while driving the car at 4m/s.

The magnitude of the wheel angles are constantly changing due to the pure pursuit algorithm realigning the car between the lanes when necessary. Figure 5 shows the wheel angles while completing a turn on the circuit. Although deviations in the magnitude are observed, this is also expected due to the natural correction of the pure pursuit. Toward the end of the turn, the wheel angles start to decrease in an effort to straighten out the car.

Plots while driving the car at 2m/s can be found in Figure 6. Looking at the straightaway data, little to no deviation is found as at this slower speed the pure pursuit algorithm does a much better job at directing the car toward the calculated trajectory point. As for the turn, similar deviation can be found to the 4m/s case as while the speed is slower, a higher degree of accuracy is necessary in order to keep the car within the lanes during a turn.



(a) Wheel angles during the straightaway of the circuit.



(b) Wheel angles during the turn of the circuit.

Figure 11: Wheel angles while driving the car at 2m/s

While testing before the competition, we were able to record a lab time of 52 seconds with 0 lane infractions. When it came to race day, we experiencing some hardware problems which resulted in inconsistent power outputted to the car. This combined with racing in different lanes than we had tuned out parameters left us with a lab time of 55 seconds and 4 lane penalties.

3.2 Luigi's Mansion

3.2.1 Stop Sign and Traffic Light Evaluation: Marcelo

There were two main aspects to verify for the implemented detectors: detection in adversarial conditions and distance prediction accuracy.

In order to test the detectors robustness, its ability to find both the stop sign and stop light in multiple conditions was tested. The main parameters involved were signaling orientation and light conditions. In most cases, both the stop sign and traffic light were detected in a short period of time. However, the stop sign detector failed to find the sign in cases where it was located at either unusual angles, such as leaning downwards or tilted, as well as in darker ambient conditions. The opposite was true for the traffic light detector, which would suffer in brighter conditions due to a smaller difference between the color of an off and on red light. This was particularly difficult to account for when setting the HSV values for the mask

The second parameter evaluated was detector distance accuracy. In order to test this, the stop sign detector was set up to look for a stop sign in the optimal conditions (brightly lit environment and with the sign perpendicular to the robot's x axis). It was only necessary to verify the stop sign distance, since the traffic light detector used the same function.

The robot was placed at $0.1m$ from the stop sign, and each round it would be moved back by $0.1m$. If the robot detected the sign, an average of the first 10 estimated distances would be used. This was done in order to combat any variation due to noise in the stop sign detector model. This resulted in the following graph:

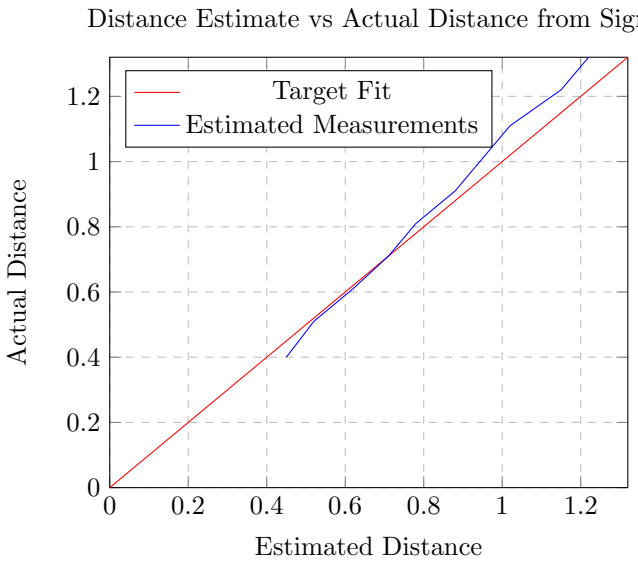


Figure 12: Plot showing deviation between estimated measurements and actual distance

As is seen in graph above, measurements hold acccurate from $0.5m$ to $0.7m$, with a minimum detection distance of $0.4m$ and a large deviation forming around $1m$. The minimum detection distance is due to the physical limitations of the camera, while the larger deviations are most likely due to an imperfect conversion from pixels to the actual distance in the homography transform. Further fine tuning would result in a better estimate at longer distances. However, since the estimated distances are closer than the actual distances, the values are conservative and thus safe to utilize. This is because the discrepancy would cause the robot to begin braking earlier, which is safer than allowing the robot to pass the point where it can no longer detect the sign.

3.2.2 Trajectory Follower Experimental Evaluation

Since the method used to follow trajectories was exactly the same as was used for Lab 6: Path Planning, the same metric for measuring the effectiveness and accuracy of pure pursuit was used. The distance between the nearest trajectory point to the robot and the robot's position was the main metric which follows the relation shown below:

$$d = \sqrt{(x_{traj} - x_{robot})^2 + (y_{traj} - y_{robot})^2} \tag{2}$$

With this, as the robot follows the trajectory, the data from this relation is collected and is plotted in a graph as follows. As the robot traverses, the nearest segment is initially far away from the robot, but

it decreases as the robot moves toward it. The subsequent repetition of peak and decrease in distance is due to the robot passing the midpoint of the segment it currently lies on, as the nearest segment to the robot will subsequently be the next segment on the trajectory. Even though the only change

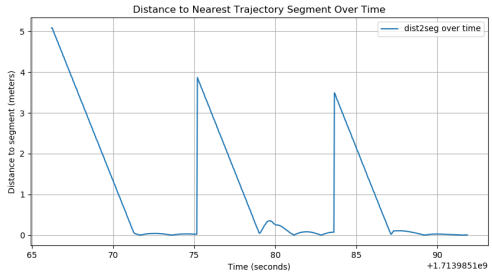


Figure 13: This shows the distance error to the current line segment the robot is navigating towards.

between Lab 6 and the final challenge was that manual transformation of points was done here, the same behavior holds with reading the transform using ROS2 functions because ROS2 does the math.

Additionally, if the robot had successfully ran the Luigi's mansion circuit, another metric that would have been used to test the quality of the solution was to create an efficiency metric. This metric would define efficiency e by taking the ratio of the total error —measured with the distance metric— of the run with the real robot over the run of the robot in simulation.

$$e = \sum_{t \in time} d_t^{robot} / \sum_{t \in time} d_t^{sim} \tag{3}$$

To successfully do this, the same shell points would have to have been used for both along with the same starting location.

4 Conclusion: Heath, Alex

This lab allowed our team to synthesize all of the different modules and code we had been working on resulting in our car being able to navigate real world driving environments. The end result was our caring being able complete a full lap in the Johnson track with a time of 55 seconds and 4 infractions. As for Luigi's mansion, we were less successful as we still have been able to fully complete the course using the car but, within simulation, we are able to be successful.

To conquer the Mario circuit we used many ideas used throughout the class including image preprocessing, color segmentation, homography transformation and pure pursuit. While testing pre-competition, we were able to integrate all of these ideas smoothly and recorded multiple 50 seconds laps with minimal infractions. On race day, we experienced some hardware issues with inconsistent power being outputted to our car which slightly reduced the accuracy of our pure pursuit controller. Additionally, the races we completed in lanes other than 4 seemed to require slightly different tuning. It was for these reasons that we believe our Mario Circuit performance was slightly worse on race day compared to testing. Nevertheless, we are pleased with our performance during competition and thought we came up with a great approach to solve the problem.

For Luigi's mansion, the successful combination of the eight different nodes that communicated with each other was the toughest part. The main ideas used in this challenge were localization, path planning, trajectory following, color segmentation, and controlling for safety. Each node worked individually in simulation, but once they were put together to launch, the communication between them and evaluating the priority of the result of each of them was nontrivial. The integration phase revealed a lot of bugs demonstrating the miscommunication between nodes, and the time crunch did not make it easier to debug, as ensuring that all the trajectory topics that needed subscription was very error-prone because RViz can sometimes be faulty without the code malfunctioning. This leads to tedious resetting of RViz topics and eats at valuable time for testing which was a big lesson learned.

Overall, the final challenge was such a great end to the class. While working on the individual modules, we were already able to do some pretty amazing things with our cars. Having the ability to integrate all of these modules and see our car navigate real world driving environments was truly a special

experience. Thank you to all of the course staff for your continuous help throughout the semester. You certainly made our experience much more enjoyable to provided invaluable pieces of advice and feedback. Thank you!

Figure 14: Team picture after the final challenge!

5 Lessons Learned

5.1 Alonso

I found the Final Challenge to be the most rewarding and beneficial part of the class from both a technical learning and team work perspective. Technically, this lab required the synthesis of all the skills I learned over the course of the year from computer vision to path planning. What found particularly rewarding was how open ended the challenge was. It felt great to start from nothing, plan a solution, and then build it node by node. While we did not accomplish everything we had hoped for, the amount of learning I achieved was well worth the hours. In terms of teamwork, this lab also challenged us and required a high degree of skill. With the end of the year, everyone was busy with multiple other classes and their respective final projects. As such, it was difficult to get everyone working together at the same time, which at time caused some friction. The way that we were able to work through this was by over-communicating. It was much easier and better for team moral to know each others limitations ahead of time, which lead to a healthier work environment. At times, the Final Challenge was stressful due to the high volume of work but our team was able to remain composed under pressure.

5.2 Heath

This lab taught me tons about understanding when to declare a piece of software correct. When searching for errors it in incredibly important to shift one's attention on parts of the software that has a higher likelihood of contribution to this error. This is what we had trouble with in this lab. There were instances were we looked through a specific file for hours trying to find a bug only to find it in a completely different part of the code. When we looked back on the debugging process we concluded that it made sense for the bug to end up where it did and we likely should've thought more about the odds of the bug being in different locations before we narrowed our attention to just one place.

Additionally, we wildly underestimated the time it tasks to test systems with so many moving parts. This lab taught me that the more modules or working pieces software has, the duration for testing should be increased tenfold. The more moving parts the higher the chances for mistakes which can be much harder to track down. All in all, the final challenge was an incredibly rewarding experience and a great way to close out RSS!

5.3 Alex

The biggest lesson I am taking from this lab is that even if we feel very confident about our brainstorming process and on our ability to work as a team, it is an inherent quality of robotics to encounter many bugs during the development of the solution. Our sub-team that was in charge of working on Luigi's Mansion, Marcelo, Thomas, and myself, began brainstorming the path to success for it since the end of April. We had come up with the ideas we ended up implementing and having it work on simulation since then, and from my point of view, I know I could have allotted more time to this during the first days of the challenge. In a good way, I had developed trust in my teammates since we had been making solid progress and delivering better results for every subsequent lab since the outset of the class; however, this backfired on me because I ended up taking time to do other things non-RSS related, and when this past week came about, we began to realize that Luigi's Mansion, albeit it was reusing concepts we had already gotten to work, putting them together was nontrivial. I think that if I had managed my time better and evaluated my priorities, I could have spent more time working on this, and that is completely on me for not doing so. And because of this, roadblocks that we eventually found out and asked questions on were discovered too late, as it caused us to have to pull an all-nighter to almost get it to work. I also learned that programming a robot takes lots of iteration, and rapid prototyping of code is a very effective process. I feel like I am not as good as I can be at employing this, because I recognize that I tend to try to get everything right and double/triple check code instead of simply running it and debug as errors come in. I also realized that although I understood well the sections I worked on for previous labs, I did not have the best intuition for node structure and how to set up a solution, especially for such a complex problem, almost completely from scratch, and after this challenge, I feel like I have a much, much better grasp of this.

Although we may not have received the best results in Luigi's Mansion, I am still very grateful for this challenge because it allowed me to grow closer to my teammates and even if there were some tense moments, we were able to resolve them and learn from them.

5.4 Thomas

The most crucial skill I learned during this lab is to pay close attention to the underlying mechanisms of the algorithm. For example, one issue was that my generated trajectories were not being properly created or merged, which was a bug that persisted for days. The solution ended up being the way that Python creates and stores objects, and I happened to be making both the start and end trajectories point to the same object despite them being named differently. This shows that I need to pay closer attention to the language I am utilizing and the way it stores information, something covered in depth from 6.1010.

Furthermore, I learned a lot about how quickly a problem can be solved through dividing the labor into distinct modules for a creative solution. We all bounced ideas off each other during the ideation phase and were there for each other to be rubber duckies, but overall by sticking to our assigned module we were able to generate working code that integrated together very nicely. This class overall has improved my skills in choosing which tasks are appropriate to divide for the most efficient workflow and integration in the end.

5.5 Marcelo

Similarly to other labs, my main takeaway was coordination and planning. While we were able to coordinate times to work, we were not able to coordinate appropriate allocation of the robot to each sub-team, resulting in some parts of final project having a shorter development and testing period with respect to others. Another challenge was adapting our code from simulation to the actual robot, since many other factors came into play that were not previously considered. However, we had a lot of support from the teaching staff, which helped when certain parts of the lab were progressing at a

slower rate than anticipated. Through discussion with my team and with the teaching staff, many of the issues were resolved faster than I individually would have resolved them.

References

[1] MIT-RSS, "Visual_servoing," GitHub, 2024. [Online]. Available: https://github.com/mit-rss/visual_servoing.

[2] Pioneering Minds, "We're Being Sold A Fantasy About Self-Driving Cars," *Pioneering Minds*, 26-Jul-2019. [Online]. Available: https://www.pioneeringminds.com/were-being-sold-a-fantasy-about-self-driving-cars/.