FINAL CHALLENGE: Racing and City Driving Algorithms for AVs

Tags

TEAM 6 - Rulan Gu, Bradyn Lenning, Aimee Liu, Sruthi Parthasarathi, Tyrin-Ian Todd

6.4200 Robotics: Science and Systems

5.13.2024

I. Introduction

Author: Sruthi Parthasarathi

Autonomous vehicles (AVs) are becoming increasingly common in everyday life, especially as the algorithms needed for safe and efficient navigation become more and more robust to the various types of noise that exist in the real world.

In an effort to understand the different components that enable AVs to work, we tackled many several individual functionalities in the labs leading up to the final challenge, such as:

1. Obstacle Detection using Spatial Data (Lab 3)
2. Object Detection using Visual Data (Lab 4)
3. Localization in a Known Environment (Lab 5)
4. Path Planning (Lab 6)

Therefore, in the final challenge, we synthesize these tools to accomplish the following two autonomous tasks:

1. Mario Circuit: Drive around the Johnson race track while staying in a designated lane.

2. Luigi's Mansion: Navigate through a "city" in the Stata basement to three goal locations and return to the start while obeying traffic laws.

II. Mario Circuit

Author: Aimee Liu

The Mario Circuit tasks our robot to drive as quickly as possible around the MIT Johnson's Racetrack without breaking the following rules:

- Stay within the lanes of the track
- Avoid crossing other lanes on the track
- Do not crash into other cars on the racetrack

For this challenge, we needed to implement a method of determining the target point in which our car needs to drive to and afterwards implement a racing controller that steers the robot towards this point with every update. To do this we modified our color segmentation algorithms from Lab 4: Visual Servoing to instead track white lines and determine our lane and the target point we aim for as we follow the track. We then use our pure pursuit algorithm to follow this target point and steer itself along the track. We also used our safety controller from Lab 3: Wall Follower to not crash into other robots. With the full implementation of these features, our robot should be able to race along the track while following the rules of the Mario Circuit.

A. Determining the Target Point

Author: Aimee Liu

In order to follow the track and avoid lane infractions, we need to first determine which lane we are on. As people, we can visually identify the lanes by the white lines indicated on the track, and so it would be optimal to allow our robot to do the same by processing information from the robot's camera which sees the track, as shown in Fig. 1, and identify the lines of the track itself. However, the robot would also need to identify this pattern of colors as a line in order to determine the target point it aims for, therefore, we also apply a series of Hough Line Transforms on our processed image and filtering to identify our line and find our target.

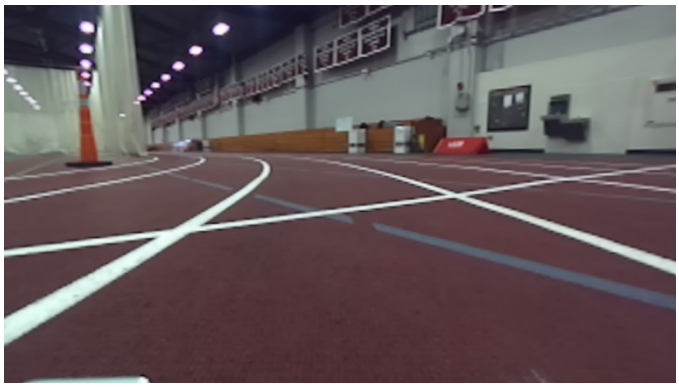

Fig. 1. Example source image frame from the camera. *The track the robot is currently on can be correctly identified by the white lines to the left and right of its vision. However, the robot itself may have difficulties with other lanes, intersecting lines, and gray lines, therefore our filtering must account for all these cases.*

Vision Processing

Author: Rulan Gu

The first step for this challenge is to create a vision algorithm that could find the lane lines in the camera image. This started with a vision pipeline that consisted of the following steps:

1. Block out the top and bottom of the image by drawing black rectangles across those areas
2. Apply an HSL filter that only kept white (high luminance) areas in the image
3. Use canny edge detection to find all edges in binary image after HSL
4. Use Hough line transform to find all the lines in the image

An example of this pipeline is shown below in Fig. 2. This pipeline would give us essentially all the lines along white track lines. To find the true lane our robot must follow requires further filtering of these possible lanes.

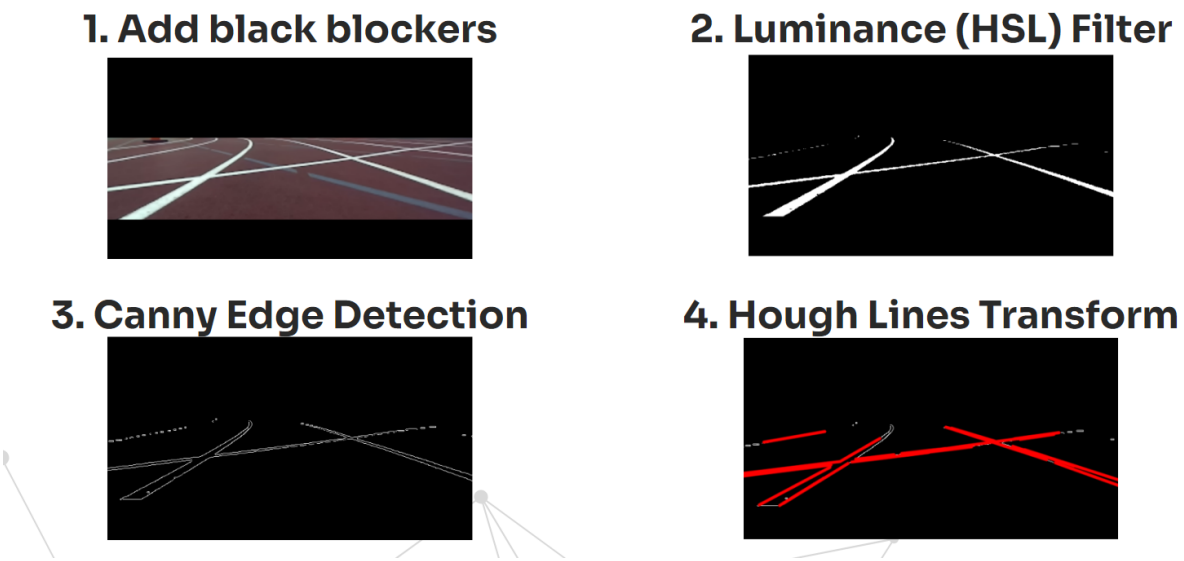

Fig. 2. Example vision pipeline process to detect the lines in the image, which are displayed in red lines on the 4th step. *Blockers are added to the top and bottom of the image to block extraneous and unnecessary information from the robot in the top left image. An HSL Filter is applied on the top right to filter out everything but the white lines. Canny edge detection on the bottom left determines the edges of these lines. Hough Lines Transforms turns this binary image into a set of possible lines in the bottom right.*

Line Filtering

Author: Rulan Gu

As shown in Fig. 2, there are multiple lines detected after the Hough Lines transform. Our next step is to find which lines are most likely to be the lane lines that we want. We need to find the left lane and the right lane. We find the lane lines using the following logic:

1. Filter out all lines with a slope between -0.15 and 0.15. These lines are too flat to likely represent our lane lines. Also, filter out completely vertical lines, which are also unlikely to represent lanes and may cause divide by zero issues in the code.
2. Separate the positively and negatively sloped lines. Positively sloped lines can only be the right lane line; negatively sloped lines can only be the left lane line.

3. For positively sloped lines, determine the one most likely to represent the right lane line. Start by looking at lines in the right half of the image and pick the one that's in the right half and has the leftmost intersection with the top blocker, but still intersects with it on the right half of the image. If this doesn't exist, consider the positively sloped lines on the left side and pick the one that has the leftmost intersection with the top blocker, but the intersection is still within the image's bounds. If that doesn't exist either, just take the steepest positively sloped line.
4. Same idea as above but reversed directions for filtering negatively sloped lines to find the left lane.

After this process, we are typically left with two lane lines, as shown in Fig. 3. However, sometimes, the algorithm is only able to find one lane line or isn't able to find any lane lines.

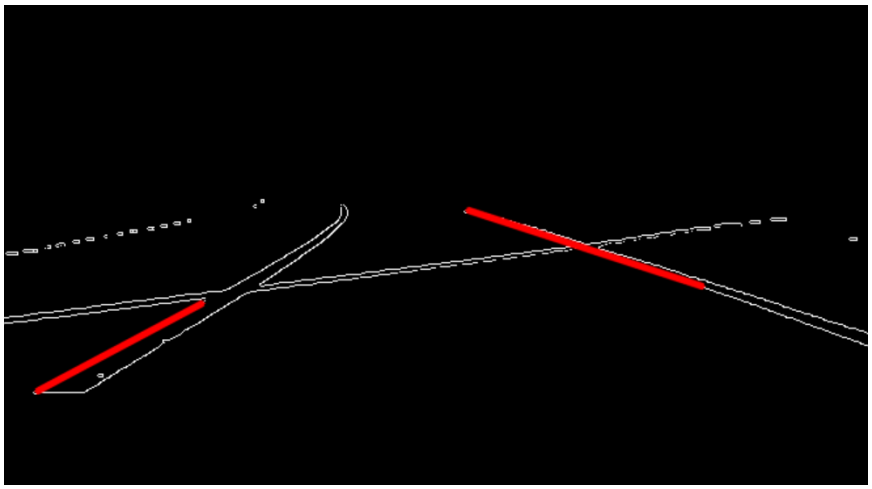

Fig. 3. Remaining lines after filtering for the a right and left lane line. *The identified lines in red are correct identifications of lane lines which avoid the intersecting line and all other lanes.*

Calculating Target Location

Author: Rulan Gu

Now that we've found the lane lines, we need to calculate a target location for the robot to actually try to drive to.

In the typical case in which both lane lines are detected, we draw a horizontal line slightly above the bottom blocker. Then, we find where that line intersections with each of the lane line. Next, we calculate the midpoint between those intersection points, which is our target point. A visualization of this process is shown in Fig. 4.

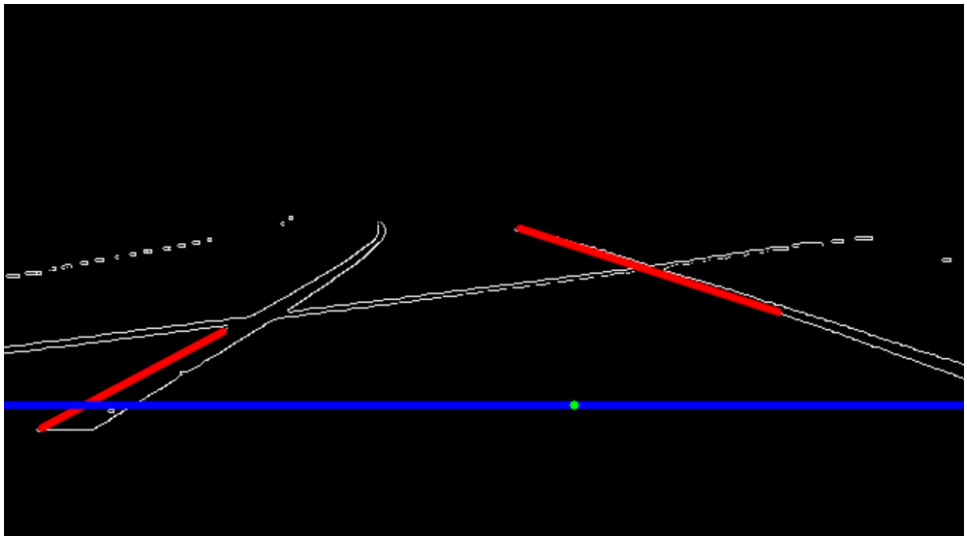

Fig. 4. Midpoint of a lane, and target point of our robot. *We calculate the intersections between the lane lines (red) and the horizontal blue line. Then, we take the midpoint of the two intersections, which is the target point (shown in green).*

In an edge case in which only one lane line is found, we apply a band-aid solution to get the robot to steer in the correct direction. We find where the detected lane line intersections with the top blocker (which is tuned to end at the horizon of the camera's point of view) and set the target point to (x = x-coordinate of the intersection, y = y-coordinate of the blue horizontal line).

In the case in which no lane lines are found, the target point is simply set to the last found target point.

Once we found the target point's pixel coordinates, we used a Homography transform to find the real-life location of the target point relative to the robot, which is used by the controller.

B. Racing Controller

Author: Rulan Gu

Once we have received our target point's real-life location from the algorithm above, we need to design a controller to steer the robot toward that target location. In order to do this, we decided to use a modified pure pursuit controller.

The origin for our Homography transform was around the center of the robot. We measured the distance from the origin to the rear axle of the bot so we could calculate the target point's relative distance from the center of the rear axle for pure pursuit. Thus, our formula for the robot's steering angle is shown in (1).

$$\delta = k * tan^{-1}(\frac{\frac{x}{y+d}}{lookahead}) - \theta \quad (1)$$

In (1):

- δ: target steering angle of the robot
- x, y: Target point location relative to homography origin
- d: Distance between homography origin and rear axle
- θ: Constant to correct for the natural drift of our robot. Set at 0.02 radians.
- k: Constant used for tuning
- $lookahead$: Distance from the center of the rear axle to the target point

III. Luigi's Mansion

Author: Sruthi Parthasarathi

Luigi's Mansion requires our racecar to drive along the MIT Stata Basement under more advanced constraints. The rules for this track is to:

- Remain on the right side of the lane while driving, unless completing a U-turn
- Stopping at stop signs and red traffic lights along the road
- Collect 'shells' at 3 assigned checkpoints on the map

For this challenge, our implementation consisted of a few parts. First, we revamped our A* algorithm from Lab 6: Path Planning for finding an optimal path between two points on the map to take into account the existence of two lanes

with restrictions on the directions of motion within each lane by transforming our map into a directed graph. This presented a need to update our path following algorithm as well, allowing the robot to carry out lane changes and U-Turns in practice.

We then built models to help detect stop signs and traffic lights, using a combination of computer vision models and simpler color segmentation from Lab 4: Visual Servoing. These were supplemented by a stopping protocol that allowed the car to respond accordingly to visual cues it observed in its physical environment.

In addition to detailing these algorithms, as the accuracy of our particle filter algorithms from Lab 5: Localization played a large role in completing this task, we will discuss a few modifications we tinkered with, although we chose to not integrate them due to computational inefficiencies in the robot.

Edited by Aimee Liu

A. Updated Path Planning

Author: Tyrin-Ian Todd

For path planning we used an updated version of our path planning algorithm from Lab 6, A*. Similar to Lab 6, the algorithm down-samples the grid into cell and then plans a path from the start point to the destination. However, in this challenge, we had an additional challenge with path planning, the algorithm had to plan a path that respected lane lines. One approach to this problem would be to turn the lane line into an obstacle on the map. Then the robot would never cross lane lines. However, doing so would not allow for the robot to switch lanes when necessary. Such as when the robot needs to turn around. To address this we instead interpreted the map as a directed graph. Where each cell in the map had edge orientations depending on their location in the map. I will talk about each case more in depth below. However, before I discuss how cells are oriented I will talk about how lane orientation is calculated.

Lane Orientation Calculation

To build the directed graph we must classify the lane orientation of each cell in the graph to one of the 8 cardinal directions as seen below in Fig. 5. In the code the lane line is represented as 10 joined line segments.

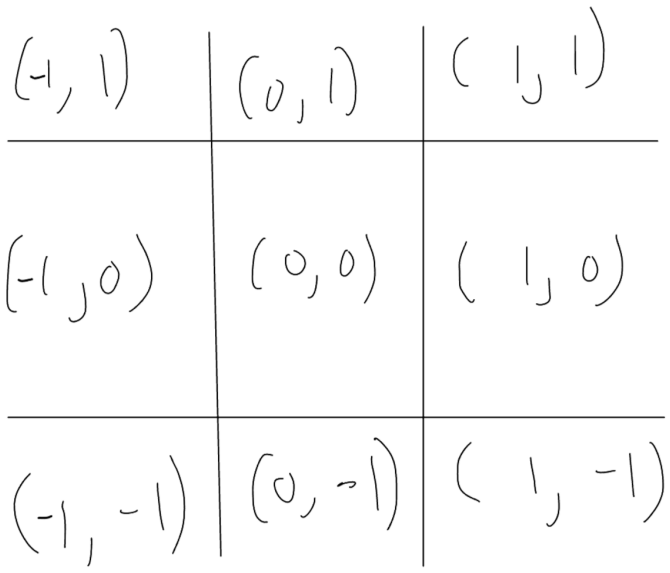

Fig. 5. The 8 cardinal directions in a grid as they are relative to the origin. *These directions are used to form a directed graph which the robot uses for path planning in city driving.*

Since the points are given in order the algorithm is able to generate a series of lane line vectors by taking the difference of the second point and the first point. On the map, this corresponds to a counter-clockwise orientation of the lane lines as seen below in Fig. 6.

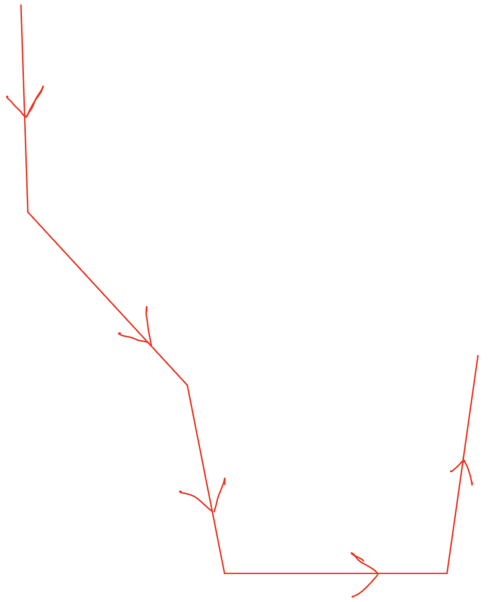

Fig. 6. Directed line representing the lane line of the city driving track. *This line in red is directed into a counter-clockwise orientation.*

To get the lane orientation of a cell the algorithm first finds the closest lane line and its associated vector using the point line formula in (2).

$$
\begin{aligned}
A &= y_2 - y_1 \\
B &= -(x_2 - x_1) \\
C &= A \cdot x_1 + B \cdot y_1 \\
\text{Point-Line Distance} &= \frac{|A \cdot p_x + B \cdot py - C|}{\sqrt{A^2 + B^2}}
\end{aligned} \tag{2}
$$

This formula calculates the distance from a point to a line where the point is (p_x, p_y) and the lines endpoints are (x_1, y_1) and (x_2, y_2).

For each cardinal direction the algorithm calculates the dot product between the normalized selected lane vector and the normalized cardinal direction. Whichever cardinal directions normalized dot product is closest to 1 is the cardinal lane orientation. You can see the result of this in Fig. 7.

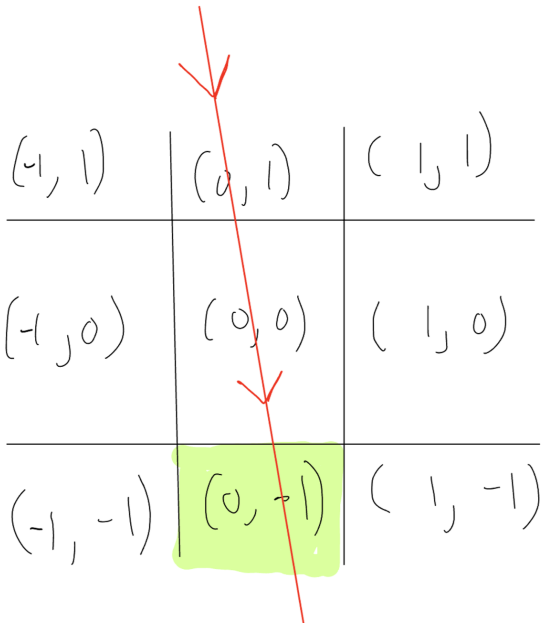

Fig. 7. Shows how one of the 8 cardinal directions are selected from a lane line. *The red line is the lane line and the grid square highlighted in green is the selected cardinal direction which is most parallel to the lane line.*

The algorithm then calculates the cross product of the lane line vector with a vector from the first point in the lane line to the cell. The cross product of these two vectors determines which side of the lane a cell is on. If positive, its on the left side (from the lane vector's reference frame) as seen below in Fig. 8. For cells on the left side the algorithm multiplies their calculated lane orientation by -1.

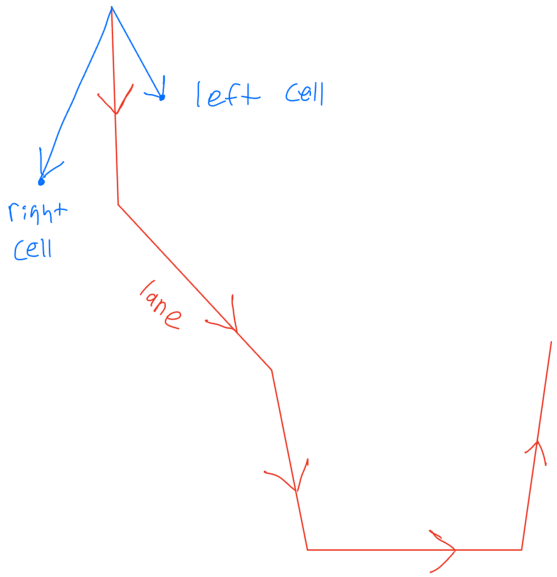

Fig. 8. Two cells on the map: one on the left side of the lane line and one on the right. *For each cell a vector is drawn from the top of the lane line segment to the cell. The cross product of this vector and the lane line vector is calculated to determine which side the cell is on.*

A* Graph Edges

After calculating the cardinal lane orientation of a cell, the algorithm can determine which directions to use to find the neighbors for that cell. The cell's relative cardinal directions are forwards, left-forwards, right-forwards, left, right, left-backwards, right-backwards, and backwards to calculate the global cardinal direction we align forwards to the cardinal lane orientation of a cell. For general cells, the the algorithm uses the forward, left-forward, right-forward, left, and right direction as seen in Fig. 9. If a cell in near a lane line the algorithm uses the only the left and right directions as seen in Fig. 10. This forces the path to move to horizontally when on these cells and prevents it from "straddling" the lane line. There is also a section of the map where the corridor is too narrow and the robot is allowed temporarily ignore lane lines. For this section cells may use any of the 8 cardinal directions.

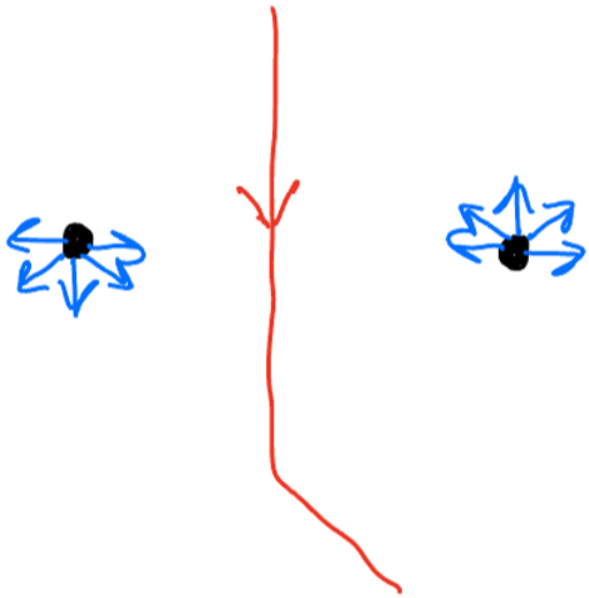

Fig. 9. Directions assigned to general cells. *The middle direction is aligned with the lane orientation with the other four directions being to the left/right of the middle.*

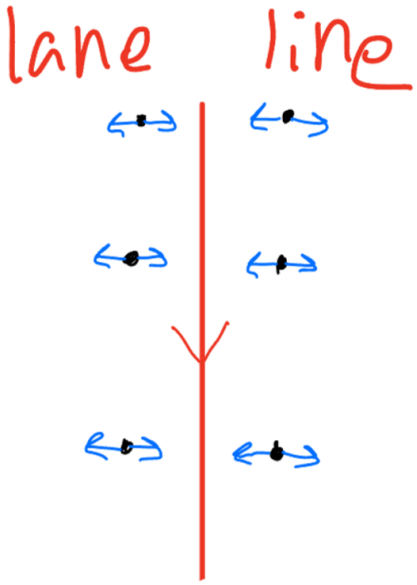

Fig. 10. Assigned directions for cells near the lane line. *Only the horizontal directions are considered restricting the path from traveling forwards when close to the lane line.*

Corner Edge Case

The setup of edges as outlined above doesn't account for an edge case around the corner as shown in Fig. 11. In this case the A* cuts corners by taking advantage of the perpendicular edges of cells that have perpendicular lane orientations. To handle this edge case the algorithm checks that the direction of travel of an edge is not anti-parallel to the the cardinal lane orientation of the neighbor it is traveling to.

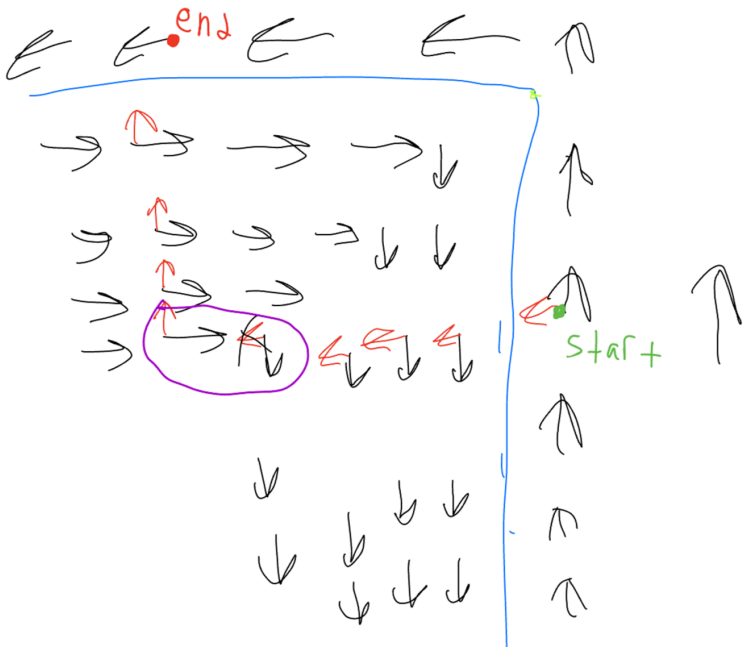

Fig. 11. Corner edge case. *As seen the expected path should go around the corner but instead the algorithm jumps across perpendicular edges to cut the corner. This corner edge case is mitigated by checking that the direction of travel of an edge is not anti-parallel to the the cardinal lane orientation of the neighbor it is traveling to.*

Edited by Aimee Liu

Runtime

While the calculation of the neighbor directions runs in constant time, the updated algorithm is pretty costly due to the scale of how many cells need their neighbors to be calculated. The added cost doesn't add too big of a difference to the runtime

in simulation. However, it uses 177% of compute on the robot's hardware and results in some longer paths taking over a minute to generate. To address this we precompute all the edges on the graph in simulation then just load the graph when working on the robot's hardware.

Evaluation

Since we know A* will always generate an optimal path for this lab our evaluation metrics were based on runtime and how well the path followed the rules. We found, on average, our path planner was able to generate a path (on the robot's hardware) in approximately 45 seconds. We also found that our path itself never produced any lane violations.

B. Updated Path Following

Author: Bradyn Lenning

In the cramped hallways of Luigi's Mansion, it is important for the robot to be able to make sharp turns including U-turns at times. In order to accomplish this, a modified path follower was developed.

At first, a lot of work was put into adapting Model Predictive Control to do city driving. From the get-go it seemed logical as it was an advanced controller that can handle arbitrarily complex situations given the correct cost function. In previous labs, this cost function included costs for distance from the path to the lookahead goal and a cost for collisions in the projected path. Model Predictive Control proved to be powerful in sim, but at the cost of being very computationally expensive.

After sinking a lot of time into fine-tuning the cost function and changing how the controller was run to be more efficient, it started to become clear that a simple controller would likely work better. This realization came after seeing model predictive control frequently act in unexpected ways such as being stuck in a loop of backing up in the lane to avoid driving in the wrong direction.

This led to a minimalistic controller that simply finds the point on the path that is closest to the goal and within a radius of the car to be the lookahead point. This point is then translated to local coordinates of the car, and the steering angle is set to be the y coordinate of this point times a constant K.

In order to allow the car to make u turns, when the car sees a point closer than 0.5m in front it on the lidar, it backs up, continuing to turn towards the lookahead point. The car continues to back up until there is nothing within 0.75 in front of it. This allows the car to make U-turns in tight hallways. This system proved to be surprisingly robust and was used in the city driving challenge.

C. Traffic Light Detection

Author: Tyrin-Ian Todd

In order for the robot to follow the rules of the road we also added in traffic light detection. The traffic light detection has two parts. The first part uses color segmentation to find the light color in the image. The second part uses the result of the first part to determine what to do.

Getting the Color of the Light

To detect the color of the light the image is first segmented into a binary image based on thresholds within the HSL color space. We will get on binary image for red and one for green. Next the we find contours within the binary image. If any contours are found for the image which was segmented to green we will say that the light is green. This is because there is not a lot of green in Stata basement so if we see any solid green shapes within our color thresholds the light is most likely green. If no green contours are detected we will do the same process on the red HSL thresholds but we also calculate the variance of the contour locations. If there is low variance and we will say the light is red. If the variance is high or we didn't detect any red we say the light color is unknown. In Fig. 12 to 14, you can see what these cases look like.

Fig. 12. Case when a green the robot correctly identifies a green light. *The green circles are where green contours have been detected in the image and the red circles are where there are red contours. As seen there are green contours on the green light and there are some red contours in other locations of the image.*

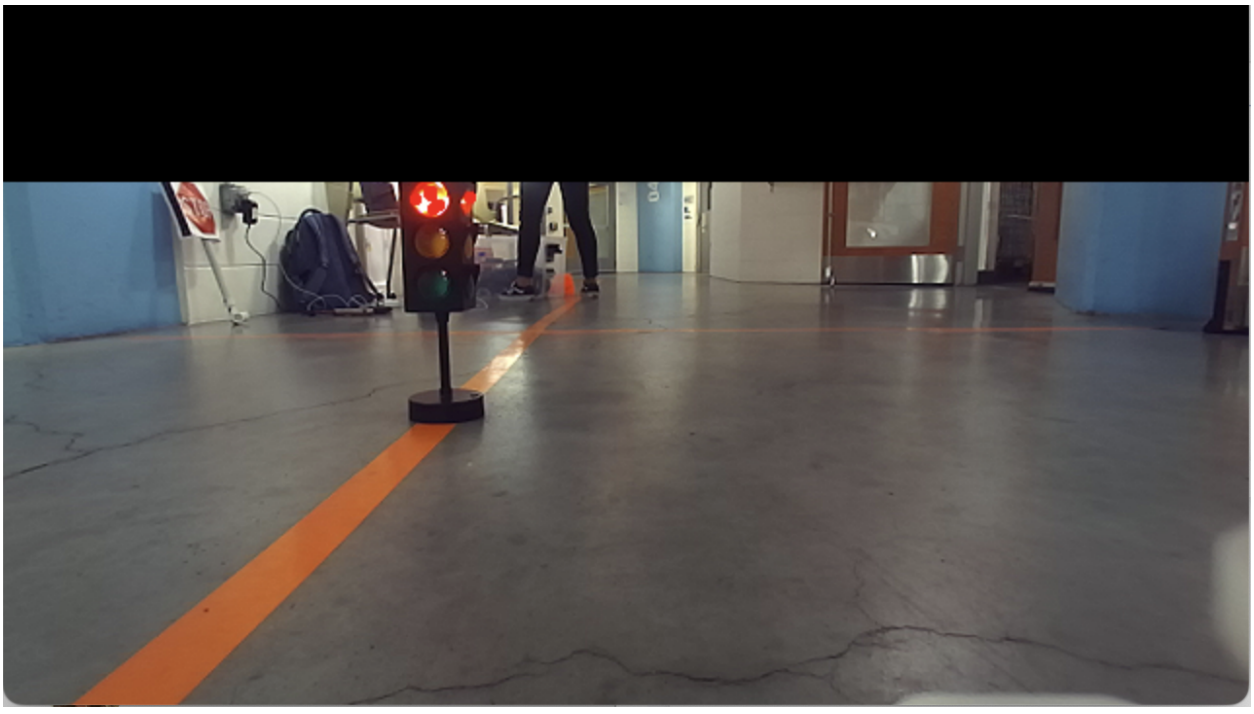

Fig. 13. Case when a red light is correctly identified in the image. *As seen there is a solid cluster of red contours where the traffic light is lit up red. Since the variance is low this image would correctly be classified as a red light.*

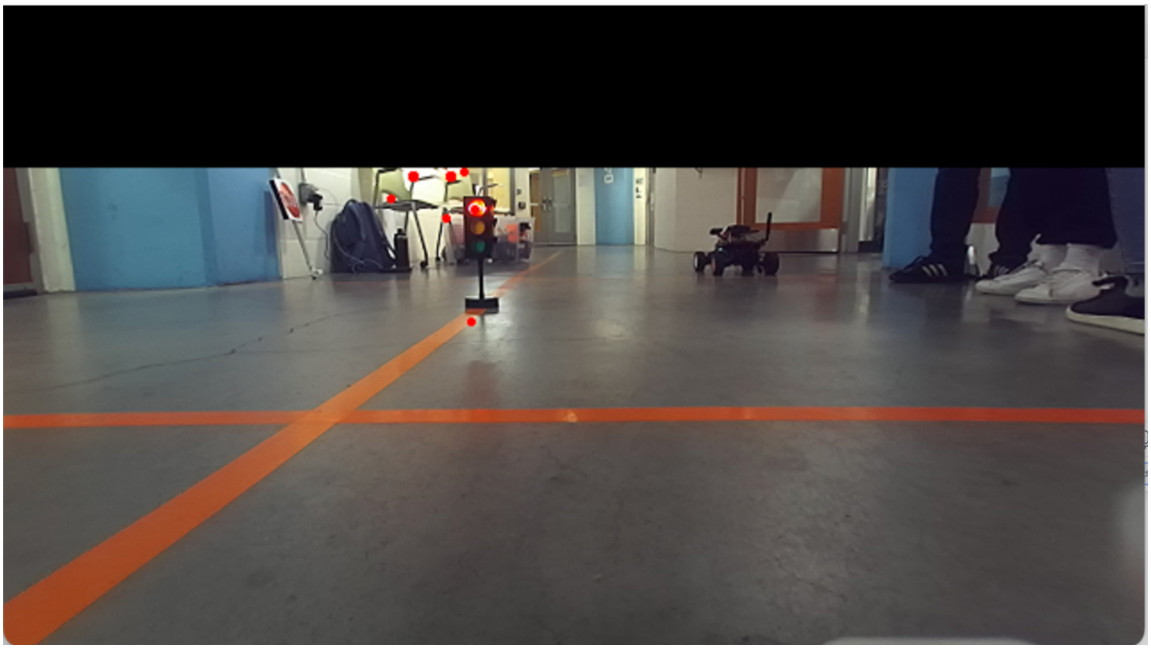

Fig. 14. Case where the detector would label the light color as unknown. *In this image while there is a cluster of contours on the red light there are also contours elsewhere in the image which causes the variance to be too high making the detector unconfident about the light color.*

Edited by Aimee Liu

Stopping the Car

The ROS node uses localization to determine if the robot is close to a traffic light, if it is it gets the light color. If the light color is green the node does nothing and allows trajectory following to continue following the path. If the light color is red. The node will send a stop command. If the light color is unknown the node will continue doing what it was doing before (if robot was moving keep moving, and if robot was stopped, stay stopped).

D. Stop Sign Detection

Author: Sruthi Parthasarathi

For the problem of detecting stop signs, we opted to use the trained model that was provided, which would take in an image as input and return a Boolean corresponding to whether or not it found a stop sign in the input, as well as a bounding box for the sign if so. The only additional logic that we had to implement was to ensure that the robot only stopped when it was sufficiently close to the sign, and that it did not continue to detect the sign once it resumed moving.

For the former, we chose to use the area of the bounding box as a proxy for the robot's distance from the sign. While we were not able to actually test the detector on the robot, future work in this regard would involve empirically determining the area threshold above which the robot should stop. As for the latter, we implemented a timer that would send stop commands to the car for a fixed amount of time, and store whether or not a sign had already been detected as a Boolean. This Boolean would only reset to False after the car both started moving and no longer detected a stop sign in its field of view, preventing the same sign from triggering the stop protocol multiple times in succession.

E. Incorporating Visual Signals in Localization

Author: Sruthi Parthasarathi

As mentioned in our overview of how we approached this second challenge, we sought to improve the localization algorithm we had designed in Lab 5. The algorithm previously only used data from the LIDAR scan in order to inform the probability of a particle being the true position, so we aimed to additionally incorporate the data from the ZED camera.

We went about this by first using color segmentation and logic similar to that of the lane detection in Mario Cart to detect the lane line. We then applied Homography in order to convert the location of the lane line from pixel coordinates to real-world coordinates, and computed the perpendicular distance from the car to the line. Finally, in order to incorporate this measurement into each particle's probability of being correct, in (3), we defined

$$p_{lane} = \frac{1}{1 + ae^{\frac{1}{|d-x|}}} \tag{3}$$

where p_{lane} is the probability of a particle that is x units away from the center lane line observing an actual distance of d using the ZED camera. As this probability lies between 0 and 1, we simply multiply it together with the particle probability calculated from the LIDAR scan data for each particle in the cloud.

In practice, this algorithm was difficult to integrate into the robot as it required computing each particle's theoretical distance to the lane line for each particle in the cloud, and the non-discretized nature of particle locations, along with the large space from which they could be sampled, made it difficult to have these values be efficiently precomputed and stored. As a result, we chose to stick to the original localization implementation for the actual challenge.

IV. Analysis

Author: Aimee Liu

Our project culminated into a Final Challenge on May 11th, 2024, where our robot competed with various other teams to best complete the goals given by the Mario Circuit and Luigi's Mansion courses. The metrics of the competition can also be used as a quantitative evaluation of the performance of our robot for the tasks given.

Mario Circuit

The Mario Circuit challenge measured the time it took for our robot to run a lap of the Johnson's Track, taking the best of 3 different trials. Our team completed two of the three trials due to time constraints. For our first run, our robot drove a lap in 1:06 minutes with 4 lane infractions. On our second run, our robot took 1:45 minutes to run a lap with 4 lane infractions.

Our robot was running at a speed of about 2 m/s, which is less than the maximum speed that the robot is capable of. This was due to the additional tradeoffs where higher speeds result in greater inaccuracies with our model, as our robot is either unable to visualize and identify lines fast enough or change its steering angle as robustly.

To make improvements, we can better our filtering to identify our lanes with more precision. Of the 94 test images of the track given, our line detection and filtering identified the lane 95.7% of the time, however further filtering can increase this

accuracy to prevent misidentification of lanes. Additionally, we can further tune our pure pursuit controller to optimize the steering of our robot car. Understanding how much or how little the robot should steer make monumental differences in the accuracy of our model, therefore it would be ideal to use more time for further testing and debugging with the physical robot.

Luigi's Mansion

The Luigi's Mansion circuit measured how many shells the robot is able to collect throughout the course of the Stata basement while avoiding lane violations by staying to the right of the road and traffic infractions by stopping at stop signs and red lights. This run is repeated 3 times and the best is taken for the competition. Our team completed one of the three trials due to time constraints and performed our best trial asynchronously. In our competition day trial, our car made it to the 3rd shell with no lane violation but many traffic infractions by running red lights. Our best asynchronous trial, we completed the challenge with all three shells and the goal point, with 50% traffic infractions, 1 lane violation, and 1 manual assist.

Our robot had difficulties identifying traffic lights and stop signs despite our implementation, therefore it may be helpful to further debug and finer tune these modules to see how these features are being identified by the robot as it runs through the course. It is also possible that some features simply take too long to run or are too slow for the robot to update appropriately. Finding bottlenecks in our code would allow the robot to more robustly control its movement.

V. Conclusion

Author: Aimee Liu

Throughout this semester, we have implemented 4 different modules that each build together to form a fully functional AV. This includes obstacle detection (Lab 3), object detection (Lab 4), localization (Lab 5), and path planning (Lab 6). These modules proved useful when synthesizing for a single automated task, such as racing along a track or driving along a city. We were able to use object detection to identify lane lines and pure pursuit from path planning to find and follow a lane on a track. We were then able to use object detection to identify traffic lights, localization to identify our position on a map, and path planning to get our robot to various checkpoints along the map.

Further work can be done to optimize our code and better improve our robot for these specific situations, such as improving our object detection and filtering to better identify traffic lights or lane lines. We can also further test and debug to more finely tune our robot in robustly controlling its movements, or find different bottlenecks in our code for our robot to improve the update time and subsequent performance of our robot.

Our robot has been able to grow from an expensive toy car to a fully automated AV with the many features we have implemented over the course of this semester. It has been an incredible journey for our team to build and create our own robot and have it culminate into this final challenge which implements everything we had made.

Lessons Learned

Each member of the team contributed heavily to the success of the project and the overall performance of the robot throughout the semester. As a result, we each learned various lessons in both the technical and communicative aspects of the lab.

Author: Aimee Liu

This course has given me a much greater appreciation for robotics and implementing different functions for AVs. Many features that humans find natural or simple become incredibly complex for a robot to understand. I find that it is always better to not underestimate a problem and make sure each part of the task given if fully thought out to manage time wisely. This project has been the source of many sleepless nights but rewarding accomplishments.

Author: Rulan Gu

I learned a lot over the course of this class about ROS and different components of an autonomous vehicle. It has been a stressful but fun experience overall. This project was really fun because it challenged us to use our previously built projects to solve interesting problems.

Author: Bradyn Lenning

I have always enjoyed taking robotics classes at MIT, and RSS is no exception. Seeing the robot working in sim was always really exciting, and I spent way more time than I needed to exploring concepts such as Model Predictive Control. Though integration was always rough, it was worth it seeing how well the bot did in both parts.

Author: Sruthi Parthasarathi

I really enjoyed this course because it was one of my first forays into the realm of software and hardware integration. As someone that has always enjoyed thinking about algorithms in theory, it was a unique experience to see them brought to life. In addition, the problems themselves were really interesting to think about, and I enjoyed coming up with creative modifications to traditional solutions.

Author: Tyrin Todd

During this final project I learned about how different modules that we built work together to perform. I enjoyed learning about path planning and code optimization for make our robot work through debugging and testing. I thought it was very interesting how each feature we've made in previous labs came back to help with this final challenge.