Final Challenge Report: Computer Vision, Monte-Carlo Localization, Pure Pursuit, Real-time Path Planning, and Other Algorithms for Autonomous Robotic Navigaton in Complex Environments

Team 7

Julianne Miana (Editor)
Binh Pham
Liane Xu
Michael Zeng

6.4200 Robotics: Science and Systems

May 14, 2024

1 Introduction - Michael Zeng

In the Robotics Science and Systems Final Challenge, our overarching objective was to integrate our learnings from prior labs in sensing, localization, planning, and controls, into one robotic system, in order to achieve a greater degree of robotic autonomy and intelligence. The Challenge therefore presented two tasks: driving the fastest lap possible around the Johnson Center Track, and driving through a city-like obstacle course in the Stata basement to reach landmarks while obeying traffic laws.

Achieving success in these tasks was primarily a challenge in integration and tuning. In prior labs, we've already developed implementations and mathematical intuition of core algorithms like Monte-Carlo Localization, Pure Pursuit path-following, Rapidly-exploring Random Tree (RRT) and Graph of Convex Sets (GCS) path planning, PID control, Image Processing methods using OpenCV, and more. Given the high speeds required on the Johnson Track and the high degree of complexity of the obstacle course in Stata basement, our focus in the final challenge was different; rather than building new algorithms, we focused, firstly, on finding the best way to integrate existing algorithms into one system,

and secondly, on modifying and adapting these algorithms to be robust in the final challenge conditions. Through many hours of testing and tuning, we were able to complete both of the Final Challenge tasks with a high degree of success.

2 Challenge Part A: Track

2.1 Overview - Michael Zeng

The first part of the Final Challenge is evaluated on both speed and accuracy of our car on the Johnson Center Track (See Fig. 1). We were awarded points for the car completing the track in a short amount of time, and penalized for instances and duration of the car driving out of its designated lane and colliding with cars in other lanes. Therefore, it was a priority to develop a vision system that is highly reliable and a controller that is stable and easy to tune for high speeds.

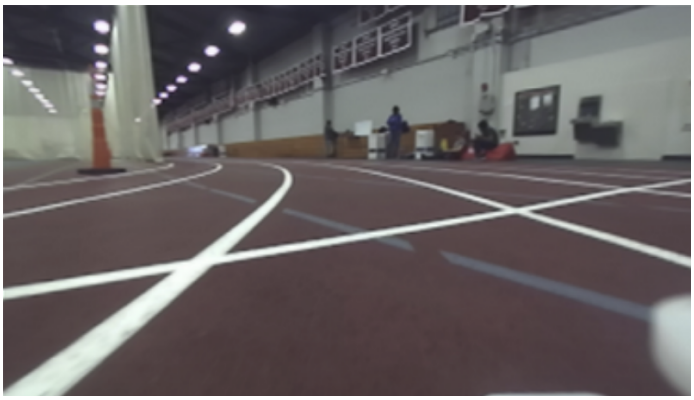


Figure 1: Racing on the Johnson Center Track from the car's Point of View

As a result, our system consists of two modules: first is an image processing module that utilizes masking and edge detection to reliably determine a "lookahead point" for the car. This lookahead point is then passed to the pure pursuit module which we carefully tuned to perform at our desired speed.

These systems combined have allowed us to reliably race around the track at the car's maximum speed of 4 m/s without incurring penalties.

2.2 Track Technical Approach - Julianne Miana

Our technical approach was straightforward: use our image processing module to detect lane lines and compute a lookahead point, and then pass the point to

the homography transformer and pure pursuit controller to set the speed and steering angle of our car. Our image processing module uses techniques such as masking, Hough Line transforms, and Canny edge detection, to consistently detect lane lines and compute a lookahead point by using geometry on our detected lines. Our homography transformer, unchanged from Lab 4, and our pure pursuit controller, a simplified version of our Lab 6 controller, then takes this point to calculate a steering angle for our car at our set speed.

2.3 Image Processing and Finding the Lookahead Point - Liane Xu

The goal of image processing for the track is to find an appropriate pursuit point. We ended up using a pursuit point that lies on a line that bisects the two lines that bound the car. The angle/slope of the bisecting line was biased toward the angle/slope of the line on the right side of the lane.

In more detail, we found the pursuit point through the following steps, using OpenCV (see Fig. 2):

1. Convert the incoming video from RGB to grayscale.
2. Apply a trapezoidal cropping mask to remove excess information from the image.
3. Threshold the grayscale image to find the white lane lines.
4. Use a Canny edge detector to get an edged image.
5. Apply a Hough line transform to the edged image. Search for the lane boundary lines – the slopes of the right and left lines each lie within a specific range.
6. In the case that a left and/or right line is not found, use the previous left and/or right line.
7. Draw a third "pursuit" line starting at the intersection of the left and right lines, with a slope that is

   $$\text{bias} * (\text{left line's slope}) + (1 - \text{bias}) * (\text{right line's slope}).$$

   On our car, bias = 0.4 to counteract the car's natural drift to its left as it runs.
8. Draw the pursuit point on the pursuit line, a specified distance away from the left/right lane intersection point. On our car, that distance was 150 pixels. This look ahead distance was found experimentally to work best for lane-following and prevent oscillations.

Figure 2: Sequence of images showing pursuit point image processing. The leftmost image shows steps 1 and 2. The center image shows step 4. The rightmost image shows steps 4-8, where the blue line is the left lane line, the red line is the right lane line, the pink line is the pursuit line, and the aqua dot is the pursuit point.

2.4 Pure Pursuit Controller - Binh Pham

The lookahead point coordinate in the image is then passed to the homography transformer from Lab 4, which converts it into a real coordinate. The pure pursuit controller reads this point and pursues it by setting the steering angle to

$$\arctan \frac{L \sin \theta_{\text{target}}}{1/4 + L \cos \theta_{\text{target}}},$$

where L is the wheelbase length and θ_{target} is the angle to the target point from the car's frame.
The pure pursuit controller is a stripped down version of the pure pursuit controller from Lab 6. We no longer need to worry about adjusting lookahead distance and finding the lookahead point, since that is fulfilled by lane detection. We set the speed to be a constant 5m/s. This is more than the car's speed limit of 4m/s, but we set it higher than 4m/s in case the car could go a bit faster than 4m/s.

Another optimization we made was to constantly publish drive messages. Originally, we published a drive message only when the pure pursuit controller received a point. However, if lane detection is slow, there could be very brief moments where the car is not receiving drive commands. To make sure we are constantly driving, we update the steering angle as the pure pursuit receives points and publish drive messages with that steering angle and speed of 5m/s every 0.05 seconds.

2.5 Track - Experimental Evaluation - Julianne Miana

To evaluate our Mario Circuit program, we completed three laps around Johnson track on race day and used lap time, collisions, and lane breaches as our metrics. Table 1 below shows our results from race day. Across our three runs, our car was consistently fast and accurate. All trials except one produced a race split under 50 seconds. The trial over 50 seconds was still close at 51 seconds and

only took slightly longer due to a brief stop in our lap to avoid an unforeseen lane obstacle. Overall, our average race split was 49.67 seconds (0:49:40 sec), which was under the 50 seconds we were aiming for. Additionally, our car was accurate because across all trials, it had no collisions or lane breaches at all. Using the staff-given equations

$$\textbf{Score} = min(100 + (50 - \text{best\_race\_split}),\ 110) - \text{penalties},$$
$$\text{penalties} = 15 * \text{collisions} + 5 * \text{lane\_breaches} + \ 5 * \text{long\_breaches},$$

for Mario Circuit scoring, we achieved scores of 99 and 101 for an average of 100.33 across three runs.

Our team also participated in the elimination round and placed 1st in our first race and 2nd in our final race. This round further showcased our car's speed, accuracy, and consistency, as our car remained in its lane without veering off-course and sped past other cars to secure 1st and 2nd place positions. Overall, our team is satisfied with our car's performance as it showed our program's robustness and reliability in accomplishing the challenges involved in this task.

Table 1: Results of our Mario Circuit challenge. Overall, our car was consistent in its accuracy (no collisions and breaches) and speed (each race split was close to or under 50 seconds).

Round	**Race Split**	**Collisions**	**Lane Breaches**	**Long Breaches**	**Overall Score**
1	0:51	0	0	0	99
2	0:49	0	0	0	101
3	0:49	0	0	0	101
Average	0:49:40	0	0	0	100.33

3 Challenge Part B: City Driving

3.1 Overview - Michael Zeng, Binh Pham

The second part of the Final Challenge is evaluated on the car's ability to reach designated landmarks (called "shells") without incurring penalties for traffic violations, being too slow, or requiring manual intervention to assist the car. Traffic violations include running red lights, not fully stopping at stop signs, stopping at green lights, cutting into the opposing traffic lane, and hitting pedestrians. One particular difficulty in this part of the challenge is that the designated landmarks are chosen by course staff after the car's software has been finalized.

Given the number of considerations in this part of the challenge, we decided to prioritize simplicity and flexibility in our approach. Therefore, we based our approach around Monte-Carlo Localization, path-planning, and path-following;

this allowed us to utilize multiple algorithms from previous labs including our Monte-Carlo Localizer and Pure Pursuit path-follower, and gives us easy ability to modify paths in real time. In addition, this approach treats reaching the shells as the foremost concern, as the shells reward the most points in the challenge, while making a best effort to obey traffic laws along the way.

In detail, our approach involves a main loop that is running localization and path-following at as high of a rate as possible. The path being followed is manually pre-planned (called the "Master Path"), with small deviations from the Master Path to the shells planned in real time. U-turns to reach shells behind the car are also planned in real time. Traffic considerations (such as detecting and stopping for traffic lights, stop signs, and pedestrians) are integrated as interrupts and intermediate procedures to the main path-following loop, with detections occurring only in specific locations and at much lower rates to retain a high localization loop rate and accuracy.

This high level approach has allowed us to reach all shells along cityscape while incurring only occasional penalties for small deviations over the lane lines and manual interventions for localization inaccuracies.

Fig. 3 provides a general overview of our city driving.

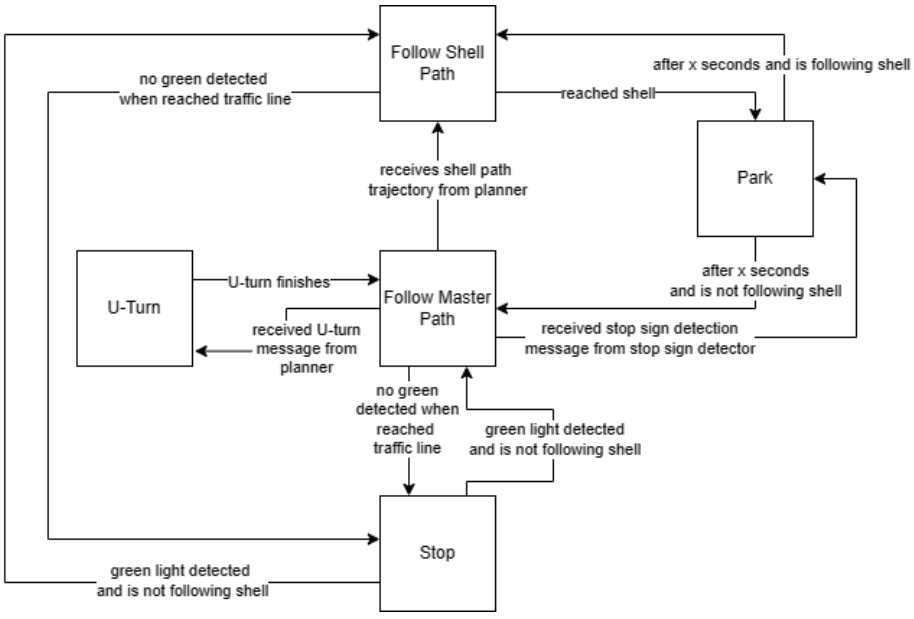


Figure 3: An oversimplified state diagram of our city driver. It starts at "Follow Master Path" and will be in that state for the majority of the time. We make an arbitrary distinction in this diagram between "Park" and "Stop". Park lasts a set amount of time, while "Stop" lasts until some condition.

3.2 Localization Tuning - Binh Pham

Our city driving is very dependent on Localization. Path following, path planning, and traffic light detection are all dependent on localization. Thus, to perform well in city driving, we needed a particle filter that performed well. We spent a lot of time tuning the localization and ended up with localization that performed well enough for us to drive the entire Luigi's Mansion city. We do not have the exact values with us as we forgot to pull our car's localization code before returning it (we only pulled our final challenge code); however, we will try our best to remember. When resampling particles, we added noise to their x and y values that was drawn from $\mathcal{N}\left(0, (0.5\, v_{\text{car}})^2\right)$. Since our distance estimates always came up short, we added 1.1 to the x and y velocities given by the odometry message to make the motion model more accurate.

3.3 Trajectory Planning - Binh Pham, Michael Zeng

3.3.1 Real-Time Planner

Our real-time planner is interconnected with the follower. If the planner does not say anything, the car will follow the Master Path. The Master Path is hand-drawn and based on the lane line in Stata. The car follows the right side of the Master Path to stay in the correct lane. More details on this are in Section 3.4.

The planner keeps track of which lane the car is in. The car always starts in the outer lane. The planner is in charge of managing U-turns. When it detects that the next shell is behind the car, it waits until the car is in a designated U-turn zone, then decides to initiate a pre-programmed U-turn routine. The U-turn zones are defined in hallways and open areas, away from traffic zones and pedestrian crossings to ensure total safety of the maneuver, as seen in Fig. 6. After the U-turn is conducted, the Master Path is reversed so that the car will now drive in the opposite direction.

When adding shell points, the planner keeps track of which side of the lane each shell is on using a custom-drawn occupancy grid, where we color inner and outer lanes differently, as seen in Fig. 4.

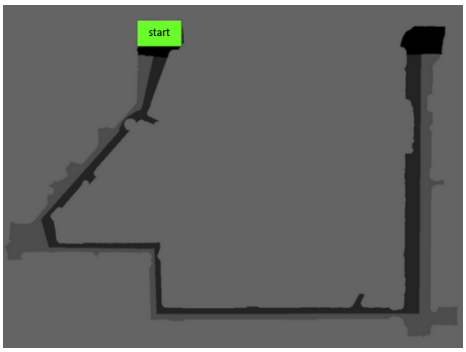


Figure 4: Custom Occupancy Grid that allows the car to determine which lane each shell is in. Values of 0 denote free space, 25's denote inner lane, 50's denote the lane line, 75's denote the outer lane, and 100's denote everything else.

We store both the shell sides and car side with 25 for inner lane and 75 for outer lane. We will see the use of these numbers when we plan early U-turns.

The planner attempts to plan an action every time it receives a new pose estimate for the car from the localization node. It can either plan U-turns or shell paths.

When the car is within 3 meters of the next shell, is on the lane side as the car, and the previous shell has been collected, it will attempt to plan a collision-free straight line path from the car to the shell. If it does find one, it will send that trajectory to the follower and mark that it has not collected the previous shell. An example of a straight line path to the shell is shown in Fig. 5, The path follower will then follow the path to the shell and park at the shell for five seconds (as specified in the challenge requirements). The path follower then resumes following the Master Path and publishes a True boolean message to "/shell_collected". The planner subscribes to this topic and when it receives that message, it will mark that the previous shell has been collected, making it able to plan for the next shell.

Figure 5: An example of a straight line shell path from the Master Path to a point near the shell. The straight white line near the shell is the straight line path to the shell and the C-shaped white line on the right is the Master Path. When the follower receives the shell path, it will start following it instead of the Master Path.

Our planner plans U-turns whenever the car is in the end zone, which are the lane ends of Luigi's Mansion as illustrated in orange in Fig. 6. Whenever the planner plans a U-turn it publishes a message (we used a Boolean but it could be any type) to the "/turnaround" topic. When the follower receives this message, it will make the U-turn if it can. More details on driving the U-turn will be in Section 3.4. If it can make the U-turn, it will publish True to the "/turn_outcome" topic and False otherwise. The planner subscribes to this topic, and when it receives True, it knows that the U-turn was successful and that the car is now on the other side of the lane.

Our planner also plans U-turns whenever the car is in a turn zone, as seen in green in Fig. 6, and the next shell is on the other side and behind the car. To find whether the shell is behind the car, we first compute 'shell_index', the index of the closest Master Path segment to the shell and 'car_index', the index of the closest Master Path segment to the car. This is the same computation from Lab 6, except we never exclude segments from being considered the closest segment. The path planner never reverses the copy of the Master Path it has, only the

path follower will ever reverse its Master Path, so the shell indices never change even if the car switches lanes. This also means that a higher index always means it is closer to the end near the vending machines. Next, we define 'shell_side' as the side the shell is on and 'car_side' as the side the car is on. These have values of either 25 for the inner lane or 75 for the outer lane. Finally, to check if the shell is behind the car, we check if the boolean expressions 'car_index > shell_index' and 'car_side > shell_side' evaluate to the same value. The planner publishes the message telling the follower to make a U-turn.

We can also U-turn in a turn zone if the next shell is behind the car but on the same side. If the car and shell are both in the inner lane, we plan a U-turn if 'car_index < shell_index'. If the car and shell are both in the outer lane, we plan a U-turn if 'car_index > shell_index'.

That is the essence of our path planner, but we have other features to make it more robust. First, when a shell is placed, the point the car will end up going to is a little closer to the path. It will be 20cm away from the shell to be exact. This is so the car would not have to deviate away from the Master Path as much while also being able to collect the shell. We found this point by first finding the closest point on the Master Trajectory, which is done by projecting vectors, which is described in Lab 6 in the finding closest segment section. We drew a line from the shell to the closest point and picked a point on this line that was 20cm away from the shell, or on the closest point if the shell is too close already.

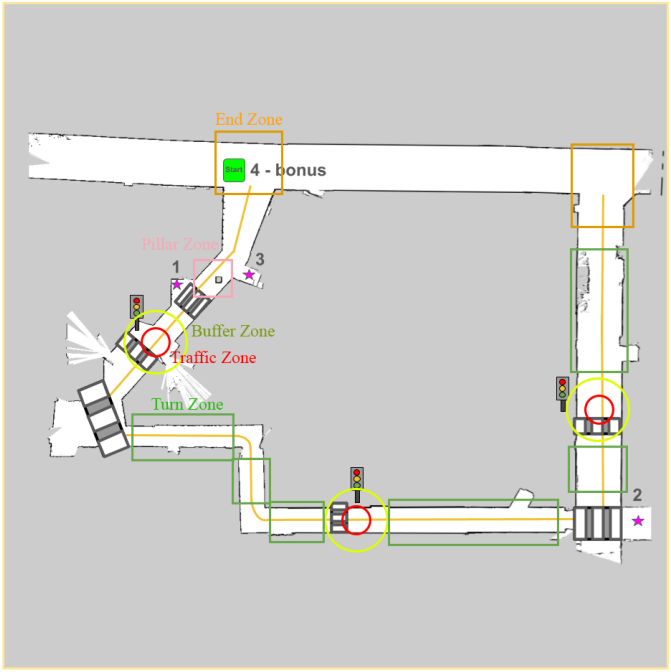


Figure 6: We defined various zones on the map. The orange zones are the end zones, the lane ends where the car will U-turn. The pink zone is the pillar zone, which is where the car needs to swerve the pillar. The yellow zones are the buffer zones, where the car starts slowing down before traffic lights. The red zones are the traffic zones, where the car will either stop or go depending on whether it saw green during the buffer zone. The green zones are the turn zones, where the car can safety make an early U-turn before reaching the end zones.

3.3.2 Better Real-Time Planner

We also developed a second real-time planning paradigm that attempts to improve the speed and efficiency of the planned path deviations. Although this method was not implemented on the physical robot due to its additional complexity and reduced robustness, we still think it is worth mentioning as it has potential to improve performance of our system. In this paradigm, the deviations from the Master Path are calculated as soon as the shell points are inputted by the course staff. This method plans U-turns in the same manner as the first planner discussed in Section 3.3.1. In total, four connected paths are computed and stored, with each end point corresponding to either the location of a shell or the ending location of the car. An example is show in Fig. 7.

Figure 7: This example demonstrates three connected paths (red, green, and blue) that join the car's starting position to two shells (with the third shell off-screen). The white line is the Master Path from which the red, green, and blue deviations are branched off of.

The core of this method is that it considers the minimum turn radius of the car and attempts to minimize the extra distance the car must travel to reach the shell. At a high level, it places a circle that goes near the shell, then finds collision-free tangent lines to connect back to the Master Path.

The first step in the algorithm is to compute the closest point on the Master Path to the shell; this is done using a linear search along the Master Path to find the nearest segment to the shell, then using a vector projection to compute the closest point. The center of the circle will then be placed on the line segment between the shell and the closest point on the Master Path, as illustrated in Fig. 8:

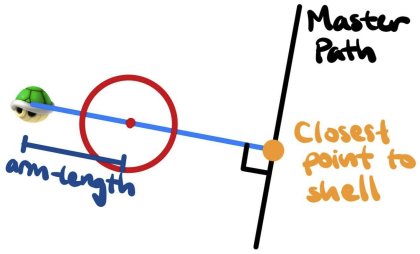


Figure 8: The center of the circle is placed an arm's length (0.5 m) from the shell, on the line segment between the shell and the closest point on the Master Path to the shell.

Next, the algorithm computes the points on the Master Path from which to branch off to the circle. To do this, the algorithm performs two opposing linear searches along the Master Path segments adjacent to the closest point to the shell. From each point in each linear search, the two possible tangent lines to the circle are drawn; the tangent line that falls closer to the shell is chosen. This tangent line is then checked for collisions; the first tangent lines in each linear search that are collision-free are taken as part of the modified path. The result of this operation is two tangent lines on each side of the circle connecting the circle to the Master Path, as illustrated in Fig. 9:

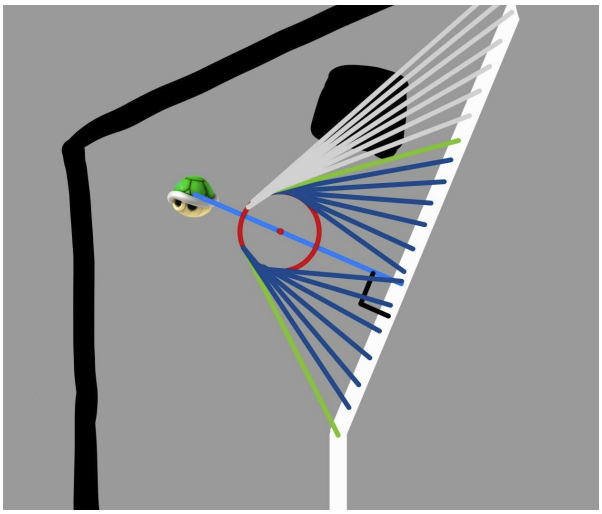


Figure 9: Visualization of the opposing linear searches. The white line is the Master Path. The black objects are dilated obstacles. The green tangent lines are the two tangent lines that are chosen; the grey tangent lines are not chosen because they are in collision (note that the blue lines are not computed in reality since the linear searches end once the green lines are found.)

With the tangent lines computed, the modified path is as shown in pink in Fig. 10:

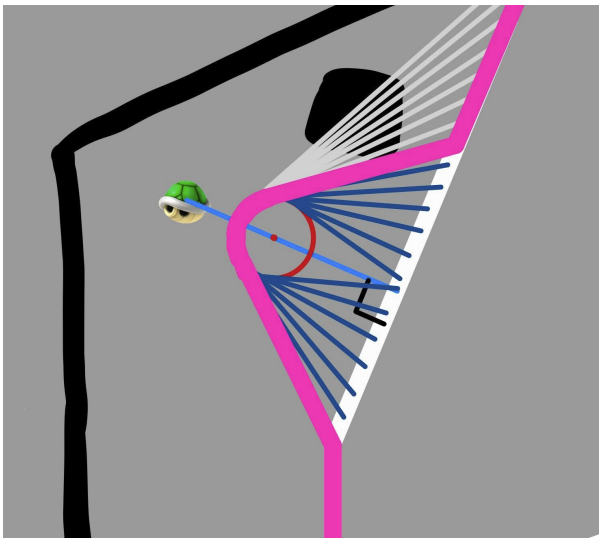


Figure 10: The modified path to the shell.

This procedure is repeated for each shell, with separated trajectories computed between shells, such that the planner is able to pause after reaching each shell.

There are many edge cases that must be handled in the software that have been glossed over for the sake of brevity; for example, what happens if the shell is close to the Master Path, such that the circle is on the opposite side of the Master Path as the shell? What if the circle encloses multiple segments in the path? One such example is illustrated in Fig. 11.

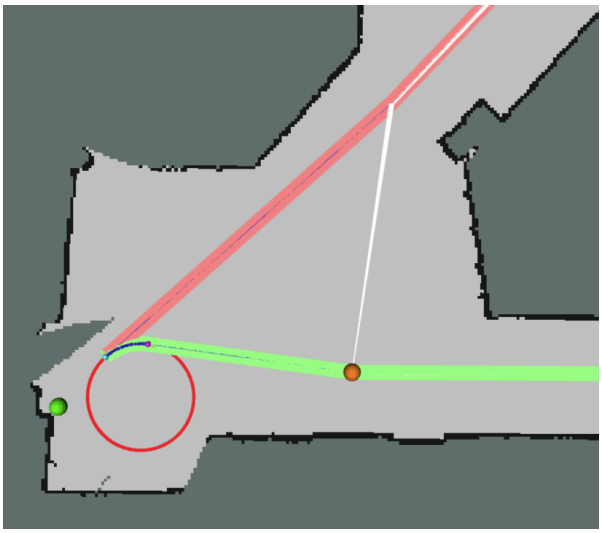


Figure 11: Example of an improperly handled edge case; the tangent line is drawn on the wrong side of the circle because of obstacles in the way. (The white line is the Master Path, the red and green lines are the modified paths computed.)

Since we used the simpler real-time planner described in Section 3.3.1 during the final challenge runs, we have not extensively evaluated this second path-planning method, but we have shown that it demonstrates promise in generating feasible paths that better optimize path length.

3.4 Trajectory Following - Binh Pham

To follow the Master Path and shell paths, we modified our Pure Pursuit controller from Lab 6. Since the Master Path is based on the lane lines, we want to follow the right of it to stay in the correct lane. To accomplish this, we simply take the lookahead point on the Master Path and shift it 50cm right in the car's frame. However, we shift it differently in two cases. The first is when the car needs to turn. If we use the shift of 50cm right, the car will cut the corner and hit the wall. We define a turn as when the magnitude of the angle to the lookahead in the car's frame is greater than 30 degrees. In this case, we shift the point forward 50cm and 12.5cm left in the car's frame for a right turn and 12.5cm right for a left turn. With this different shift, the car makes a wider turn and is able to turn without collision. The second case where a different shift of the pure pursuit lookahead point is required is when passing by the pillar in one of the hallways. Since this area of the hallway is so narrow, when the lookahead point is within this pillar zone, we adjust the shift to instead be 50cm to the left of the lane, causing the car to momentarily swerve around the pillar, avoiding

collision. This behavior is illustrated in Fig. 12.

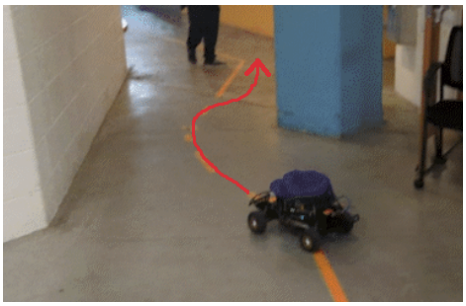
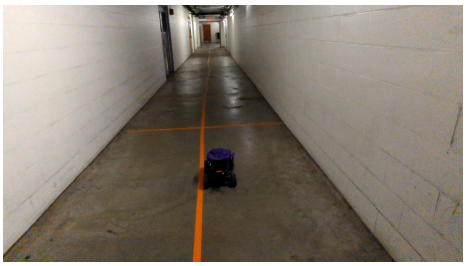


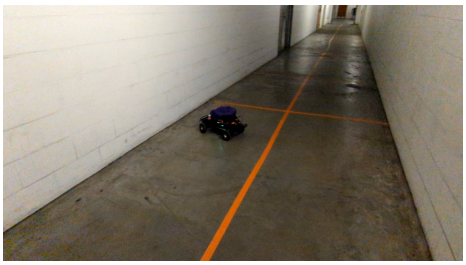
Figure 12: Near the pillar, the pure pursuit lookahead point is momentarily shifted left instead of right, to force the car to swerve around the pillar and avoid collision.

When the follower receives a straight-line path to a shell from the real-time planner, the follower will begin following that path instead of the Master Path. When the follower follows the straight line shell path, it does not apply any offsets. After it reaches a point near the end of the shell trajectory, it will stop for 5 seconds to collect the shell. Finally, it returns to following the Master Path. Previously, upon receiving the shell trajectory, we had the car reverse to align itself with the straight line trajectory. After parking, the car would reverse back to the beginning of the shell path to return to the Master Path. However, after being told that the RSS staff would be nice with their shell placements (the shells would not be placed in nooks or crannies that would require complex maneuvering for the car to reach), we removed these two reversing behaviors.

The follower also handles the driving of U-turns. When receiving a message to U-turn from the planner, it will check to make sure it is not currently following a shell or performing hard-coded driving like parking or reversing. Our U-turn is actually a 3-point turn so that we can turn around in tight spaces. If conditions are clear for the U-turn, the car will drive forward 1m/s with max steering angle to the left for 1 second, then reverse with max steering angle to the right for 1.25 seconds. Finally, it reverses its Master Path. Following the Master Path again will complete the final part of the 3-point turn. These stages are illustrated in Fig. 13. After completing the U-turn, the follower publishes a True message to "/turn_outcome" to let the planner know the car is now on the other lane.



(a) Car Start of U-Turn



(b) Car After Driving Forward-Left

(c) Car After Driving Back-Right

(d) Car After U-Turn Completion

Figure 13: The stages of the hard-coded U-Turn. Our U-Turn is in actuality a 3-point turn so that we can turn around in narrow hallways.

The follower is also part of traffic light and stop sign detection; however these will be described in more detail in the next two sections.

3.5 Traffic Lights - Julianne Miana and Liane Xu

Our team adopted a simplified approach to traffic light detection and combined it with driving logic to produce a reliable and robust traffic light detector. For light detection, our team opted to detect the green light only, as varying light conditions and noise (i.e. surrounding objects similar in color) made it harder to consistently detect the red light. For our driving logic, we defined and used traffic light detection zones to activate our light detection algorithm at the appropriate time as the car approaches each traffic light.

Our light detection algorithm processes each frame in the following way (see Fig. 14:

1. Create grayscale and HSV images from RGB image (taken by camera)
2. Make cropping mask using a customized shape. To find the optimal shape for cropping, we used frames from a video of our car following our planned path and reduced the image to just areas where the traffic light could show up in the camera frame.
3. Apply cropping mask to the HSV image
4. Create a green mask image by thresholding the HSV image to detect green light
5. Find contours in green mask image
6. For each contour, get its area and mean value of the pixels it encloses on the grayscale image. Without the mean and area conditions, bright pixels in the red light region could still be detected (see Fig. 15)
   (a) If $(50 < mean < 120)$ and $(100 < area < 1200)$[1], a green light is detected

Figure 14: Image processing sequence to find green lights. The leftmost image shows the cropped photo (step 2). The center image shows the result of applying the HSV green mask to the grayscale image (step 4). The rightmost image shows proposal of a green light (contour meets mean and area criteria), marked by a pink circle (step 6).

[1] Mean and area ranges were determined through sampled images from path-following video.

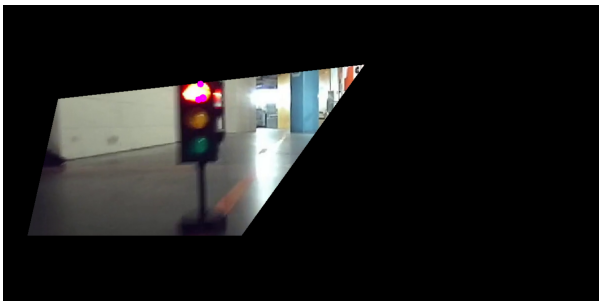


Figure 15: Image showing false positives for green light detection. Proposed green lights are shown in pink circles in the bright regions of the red light.

As a supplement to our green light detector, we incorporated straightforward driving logic that enables our car to check for a green light as it approaches a traffic light. A stop line and light detection zone were defined at specific radii from each traffic light to implement this logic. Fig. 16 illustrates this stop line and zone. The high-level logic followed by the car as it approaches a traffic light is as follows:

1. If in traffic light detection zone:
   (a) If GREEN LIGHT: Continue
   (b) Else: Wait at stop line until GREEN LIGHT

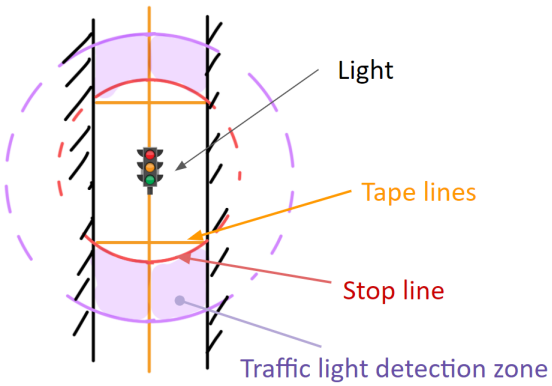


Figure 16: Diagram of light detection zone implemented for our driving logic.

To optimize our traffic light detector, we experimentally tuned our traffic light zone and stopping line radii, traffic light positions, and green mask minimum and maximum HSV values to maximize reliability.

3.6 Stop Signs - Liane Xu, Binh Pham

To find the stop signs, we used the provided YOLOV8 object detection model. We initially masked out the lower half of the image because the signs are mounted off the ground, but found that not using any masking worked better due to how slow the detection was running. Whenever a stop sign was detected, we published a True boolean message to "stopsign". The path follower subscribes to this topic and when it receives a message, it will park the car for two seconds.

3.7 Pedestrians - Binh Pham

We used our safety controller from Lab 3 to avoid hitting pedestrians. It sends higher priority drive messages, so it will always stop if it detects pedestrians or obstacles in front of the car.

3.8 City Driving - Experimental Evaluation - Julianne Miana, Binh Pham

To assess the performance of our city driving program, our team completed three runs through Luigi's Mansion. For our metrics, we used number of shells collected, whether we returned back to the starting area, completion time, and traffic violations to determine the effectiveness of our program.

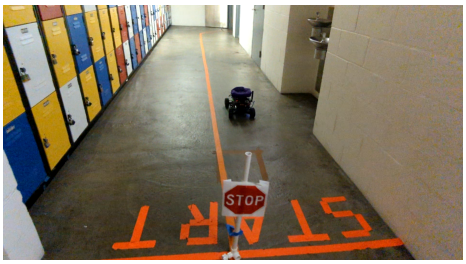
As we conducted our trials, we found a trend of improvement in our city driving performance, which was reflected in our increasing scores. For our first trial, we successfully collected all three shells and returned to start; however, due to a bug in our planner, we missed our U-turn, which forced our car to make the U-turn at the end zone. Although this took our car on a longer path, this also showed the reliability and robustness of our program as it still allowed our car to complete a path back to start, despite missing an early U-turn. For trial 2, the bug in our planner was fixed, which allowed our car to collect all shells using a shorter path and in a much shorter time. For trials 1 and 2, we each had 1 lane crossing, which prevented a perfect score in our second trial. Lastly, for trial 3, our localizer allowed our car to avoid the previous lane crossing from trial 2, which ultimately allowed our car to achieve the maximum possible score.

Besides the one-lane crossing in trials 1 and 2, both our localizer and controller were shown to be sufficiently robust as they consistently allowed our car to complete maneuvers, such as deviating from and returning to our Master Path to pick up shells and make U-turns to return to our starting position. This robustness is reflected in our car's consistent completion of the path to pick up all three shells and in its successful return to start. Our path planner was also very reliable as the car stopped within arm's length of every shell in each trial. Although the planner did not attempt to optimize path length, it was still surprisingly efficient as the pure pursuit controller smoothly and efficiently

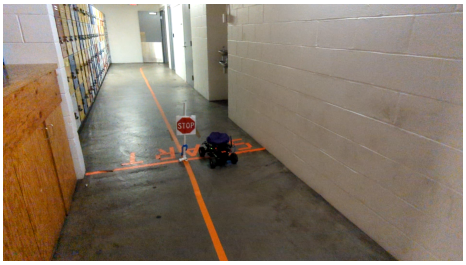
returned to the Master Path.

Our trial runs also demonstrated sufficient reliability for our traffic detectors. For example, our stop sign detector was effective, as our car achieved a full stop at the stop sign for each trial. However, the detector was not entirely consistent as the car stopped at varying distances after the stop sign in trials 1 and 2, as seen in Fig. 17. Our traffic light detector also worked well, as no red lights were ignored and green lights were consistently detected. Our trial runs did not involve any pedestrians, so we were not able to experimentally evaluate our pedestrian detector along with the other detectors. Given that it was consistent in our pre-competition tests, we are confident that our pedestrian detector would have also worked, had a pedestrian been involved in the trial runs.

(a) Trial 1



(b) Trial 2



(c) Trial 3

Figure 17: The position of our car after it detected and stopped for the stop sign in each of our three city driving trials.

Overall, our city driving program was sufficiently robust and reliable to accomplish each challenge of the city driving task. Our program collected all three shells and only incurred one penalty for two runs: a lane crossing. Since our approach was to prioritize collecting all shells first and then minimizing penalties, we are satisfied with our city driving program, as it accomplished exactly that. It kept our penalties to a minimum (the maximum number of violations for any one run was 1 with the final run incurring no penalties) while collecting all shells and returning to our starting position, which earned us bonus points for all trials. Table 2 presents our results for each trial and shows an improvement in overall score, as we fixed a bug in our planner. The overall score is calculated

based on the staff-given equations

$$\textbf{Score} = \text{location\_score} - \text{penalties},$$
$$\text{location\_score} = 40 * \text{L1} + 30 * \text{L2} + 30 * \text{L3} + 20 * (\text{L1} \wedge \text{L2} \wedge \text{L3} \wedge \text{Start}),$$
$$\text{penalties} = min(5 * \text{traffic\_infractions},\ 30) + 10 * \text{manual\_assist} + min(\text{seconds\_over\_baseline},\ 30),$$

where baseline was 2 minutes and L1 to L3 are binary variables that represent the shells picked up.

Table 2: Results of our City Driving trials. Our best score was in Trial 3, where we achieved the maximum score possible.

Trial	**Shells**	**Time**	**Penalties**	**Overall Score**
1	3 + Bonus	3:15	1 Lane Fault	95
2	3 + Bonus	1:22	1 Lane Fault	125
3	3 + Bonus	1:22	None	130

4 Conclusion - Liane Xu

In this lab, we adapted what we learned from previous labs and implemented new ideas to make our car drive around Johnson Track and navigate obstacles around Stata Basement.

For the Johnson Track (Mario Ciruit) portion of the challenge, we used pure pursuit, following a point that fell along the line bisecting the left and right lines. To find the left and right lines of a given lane, we made a custom mask and filtered through lines proposed by the Hough Transform. We tuned the location of the pursuit point to minimize oscillations and overcome our car's natural drift. Overall, we created an algorithm that allowed our car to perform consistently and at the full 4 m/s speed.

For the Stata Basement (Luigi's Mansion) portion of the challenge, we combined localization, pure pursuit, RRT, our safety controller, image processing, and machine learning to drive our car to specified points (unknown ahead of time) while obeying traffic lights, stop signs, and avoiding pedestrians. Overall, we found this portion of the challenge to be more difficult than the Johnson Track because there were more interdependent parts. Almost all of our traffic detectors relied on localization to some extent, so we spent a lot of time tuning that. We also spent a lot of time trying out different methods for traffic light detection before coming to the conclusion that it was the easiest to detect "green" or "not green" rather than "green" and "red". In the end, we were able to obtain full points.

Given more time, there are many improvements we would apply to our Luigi's Mansion stack. First, we would implement a path planner that takes into account the angle the car stops at the star/shell locations so the car can follow a smoother trajectory from the star/shell location back to the main trajectory. We would also get the stop sign detector to work faster so the car consistently stops in front of the sign rather than after passing the sign. We would also implement a more robust traffic light detector that thresholds the green better, or senses red lights as well as green lights.

This lab also provided us with non-technical takeaways. We learned to trust our abilities when things are looking down. We also learned about how to determine what issues to prioritize to manage our time best while maximizing the quality of our work. Lastly, we feel through the course of the semester, our team was able to build a sense of trust that helped us succeed through this difficult challenge.

5 Appendix

- Track Run 1 Video Footage
- Track Run 2 Video Footage
- Track Run 3 Video Footage
- City Driving Run 1 Video Footage
- City Driving Run 2 Video Footage
- City Driving Run 3 Video Footage

6 Lessons Learned

6.1 Julianne Miana

Given that this final challenge needed several interdependent working components, an important lesson for our team was starting and testing each module or component early. This allowed us to debug and tune previous implementations to improve our car's performance in each challenge. Despite making progress early on, however, we still had to stay up the whole night before competition day to work on our city driving. Another lesson that I found critical was trying multiple approaches to each challenge. For both the Johnson track and city driving challenges, our team attempted different approaches which allowed us to pinpoint areas that needed improvement and further optimization. In terms of teamwork, working on different modules also allowed us to streamline progress as a team. Lastly, we learned that integration is another challenge in itself because we needed to make sure individual modules or components not only worked well individually but also with other interdependent components to ensure good system performance.

6.2 Binh Pham

Localization tuning was painful, but I got through it and our localization now works super well. Even though our car did not need to go through the full city driving course on race day, our car's localization was robust enough that even though it missed an early U-turn in one trial, it returned to the start area, making all the localization tuning worth it. Integrating traffic lights was also difficult, as we did it right before race day and it caused some other problems. I am grateful for my teammates who worked on computer vision because I did not have enough experience with vision to work on it for the final challenge. Seeing the car following the Johnson track without ever going out of its lane was amazing. The traffic light was also really cool, though it could be improved a bit. During this challenge, especially during this time when everyone is very busy, regular communication was key to coordinate meeting times to get as much time working on the robot as possible. In the end, we still did not have enough time and ended up pulling an all-nighter before race day. Thus, I also learned that I can work on robotics for 27 hours straight without sleeping, which is not really good, but I am surprised I could do it. Stick a fork in me, I'm done.

6.3 Liane Xu

This challenge taught me about thinking creatively and debugging. Our plan for how to approach each part of the challenge evolved a lot throughout these couple of weeks through debugging, doing reality checks, and accidentally discovering things that work. Even though the final challenge was supposed to be a culmination of things we'd already implemented or learned, we struggled to get everything working together as one system, and tuning our components to solve the new problems. I'm grateful for my teammates.

6.4 Michael Zeng

Most of all, this challenge demonstrated to me the seemingly exponential relationship between the number of constraints and considerations in a robotic system and the complexity of the software required. Although every module in the final challenge was fairly simple in itself (like stopping in front of a stop sign), integrating these into a system intended to drive the entire course reliably was incredibly difficult and time-consuming. The final challenge also, once again, exemplified the importance of accurate localization. Our most persistent problem in Part B of the challenge was "losing" localization, which not only is a problem in itself but made it harder to debug other problems with our path follower or U-turn procedure. Prioritizing this in Labs 5 and 6 could have made the final challenge easier.