Lab #6 Report: Path Planning

Team #11

Matt Jens
Aileen Ma
Dylan Goff
Anika Cheerla
Quincy Johnson
6.141/16.405

April 20, 2022

1 Introduction (Dylan, Aileen)

Whether it is exploring the surface of Mars or maneuvering through city streets, being able to plan and follow a desired path is an essential component of robotic navigation. Having this capability enables us to use known maps to generate an efficient path from point A to point B that a robot can take, while avoiding obstacles along the way.

For this lab, our objectives were to implement a path-planning algorithm that can generate a trajectory for the robot to follow and to design a pure pursuit controller that makes the robot follow this computed trajectory. We considered the trade-offs between a sample-based algorithm and search-based algorithm. Additionally, we considered how to best represent the search domain, avoid close collisions on the map, accurately converting locations between the pixel and the world frame, and implement a useful heuristic. With pure pursuit, we also focused on tuning gains, changing look-ahead distances based on various paths, and integrating with the localization lab.

There are two key challenges to account for: design trade-offs between path planning and path following and accounting for edge cases with pure pursuit. Overcoming these challenges is important in order to allow our robot to work robustly in a wide range of environments.

2 Technical Approach

In this section we will outline the goals of a state space search algorithm and the two types of algorithms that accomplish this. In particular, we will elaborate on why we decided upon an A* algorithm for path planning and the specifics of this implementation. Next we will discuss methods of fine tuning our path planning algorithm to perform better, including morphological dilations and choosing a heuristic function. Finally, we will discuss our implementation of pure pursuit in order to achieve our goal of path following.

2.1 Path Planning

Path planning algorithms seek to find an trajectory from a start location to an end location, ideally one which minimizes distance and avoids obstacles. In order to do so, they typically represent potential positions along the trajectory as nodes and connect the nodes via edges, representing the movement of the robot along the trajectory. In this section, we will discuss two kinds of path planning algorithms: deterministic search-based planning algorithms, and nondeterministic sampling-based planning algorithms.

2.1.1 Search-Based Planning (Aileen)

Search-based planning algorithms systematically iterate over potential candidates for the next step of the trajectory until the goal is found. These algorithms are guaranteed to find a path if one exists. Search-based algorithms include breadth first search (BFS), depth first search (DFS), Dijkstra's algorithm, and A* search. Of these algorithms, all but DFS will find the optimal path. BFS, DFS, and Dijkstra's algorithm are all uninformed algorithms, which means that they do not consider information about the current state of the node relative to the goal. On the other hand, A* is an informed algorithm, and will use the distance to the goal node as a heuristic to decide which nodes to prioritize in the queue. This can often help A* find an optimal path faster than the other alternatives because the heuristic function will roughly correlate with the true cost of the optimal trajectory.

We decided to use A* as the planning algorithm to be implemented on the car itself. Given a start node, end node, and map, A* will add neighboring nodes (nodes in the cardinal, inter-cardinal, and secondary inter-cardinal directions) as potential candidates as shown in Figure 1. For each neighbor, A* will calculate the "cost" of each node, which will take into account the distance traveled so far to reach this node in addition to a heuristic function that will estimate the distance left to reach the goal node from this node. It is due to this heuristic function that A* is categorized as an informed algorithm, as this information can allow A* to prioritize nodes with a lower expected total cost. The effect of prioritization is ideally minimizing the number of nodes needed to search through, ultimately resulting in a faster algorithm.

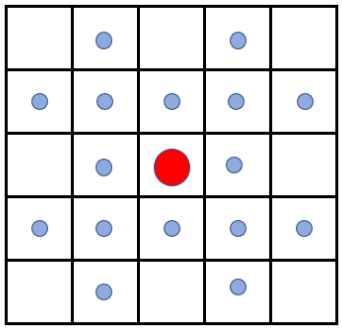

Figure 1: A* will branch out from the main node (shown in red) to discover neighbor nodes (shown in blue). This represents the cardinal, inter-cardinal, and secondary inter-cardinal directions.

The choice of heuristic function is important because there a few key qualities that the heuristic function must have. First, the heuristic function must always underestimate the true cost to the goal node (admissible). This is because if heuristic cost functions could return overestimated values, then the optimal path may potentially not be the first path returned if the heuristic function mistakenly pushed it to a lower priority. Second, the heuristic function must provide an estimate less than the distance from a neighbor to the goal in addition to the cost to reach a given neighbor (consistent). This property guarantees that the estimated cost at any given point is greater than or equal to the cost at the prior time step. For this problem, we used the Euclidean distance heuristic, in which we took the straight line distance from a node to the goal. This is guaranteed to be admissible because it is impossible for a calculated path to be shorter than the straight line path. Moreover, this is also consistent because the straight line distance is necessarily as short as any other path from a node to a goal node. Euclidean distance heuristic also has the added advantage of being robust in calculations for various directions, and is easy to implement. Our regular cost function determines the cost by adding the current cost to reach a node to the cost to reach a neighbor node. There are many potential cost functions to consider, such as Manhattan distance (horizontal/vertical) or diagonal distance, but these are not as robust as Euclidean distance because we are looking in 16 potential directions.

2.1.2 Sample-Based Planning (Matt)

There is another way to find a path between two points, and that is using sample-based planning. This method for finding a path takes a Monte-Carlo approach compared to the more deterministic search-based planning described in the previous section. Our team implemented RRT to test search-based planning. As displayed in Figure 2, this algorithm builds a tree containing nodes and branches that fill up a continuous space randomly with a bias to Voronoi regions (empty space), and then it finds a path between the start and end nodes.

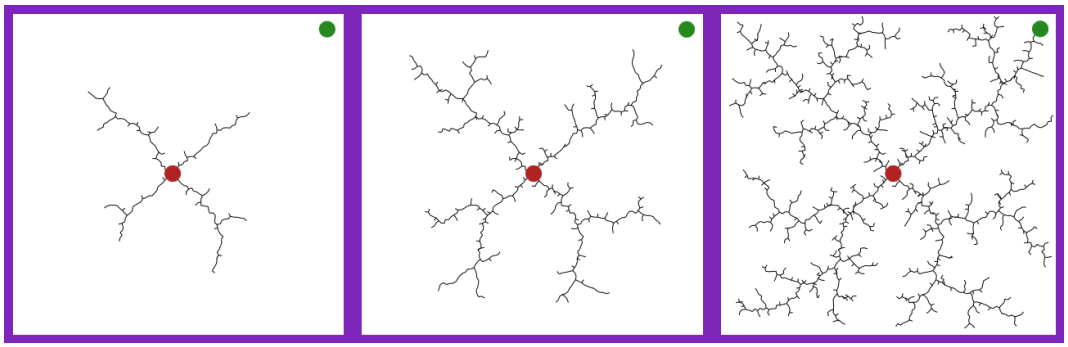

Figure 2: The RRT algorithm in action. This sampling-based planning algorithm finds a path between the start (red) and end (green) points by building a tree with nodes and branches.

2.1.3 Choosing a Planning Algorithm (Matt)

Like any problem with multiple approaches, there are trade-offs between the two types of algorithms, and the selection depends on what factors we would like to weigh more heavily. Sampling-based planning finds a path quickly, but this is a non-deterministic process and sometimes this path is

inefficient and non-optimal. Search-based algorithms are able to find an optimal path deterministically, which is important when there are many obstacles and pathways in a space and consistent testing is useful. The downside to search-based algorithms is that the space must be discretized and complexity may become exponentially large with each node, which is in contrast to sampling-based algorithms, which is able to find a path in continuous space while complexity does not increase exponentially. Some sampling-based algorithms, such as probabilistic roadmap (PRM) are capable of planning multiple paths after a single roadmap construction, while other sampling-based algorithms such as RRT must rerun the algorithm every time the start and goal nodes are updated. With these trade offs in mind, we ended up favoring search-based algorithms, and in particular, we chose the A* algorithm. The map is not too large and this means that the exponential complexity of A* in both space discretization and search will not have such a significant effect. A* will generally be faster than other uninformed search based algorithms due to the use of a heuristic. Although it will be faster to generate a path from a sampling-based algorithm, having an optimal path is a high priority because we will be racing other teams, and we would like to have the shortest route possible.

2.1.4 Morphological Dilations and Erosions (Matt)

Once we got path planning to work, we noticed that the shortest path found often hugged walls and obstacles tightly. Because the car is not a singular point, the wheels would run into the side of the wall or obstacle along its path. To avoid crashes, one solution is to add some buffer around the obstacles in the simulated map. The way we added this tolerance was by applying erosion and/or dilation filters. After we applied an erosion filter to the Stata basement map, the map's walls and obstacles (the black-colored areas) became larger, which enabled the robot to stay towards the middle of hallways and avoid collisions. Figure 3 shows the effects on the provided Stata basement map after applying an erosion filter.

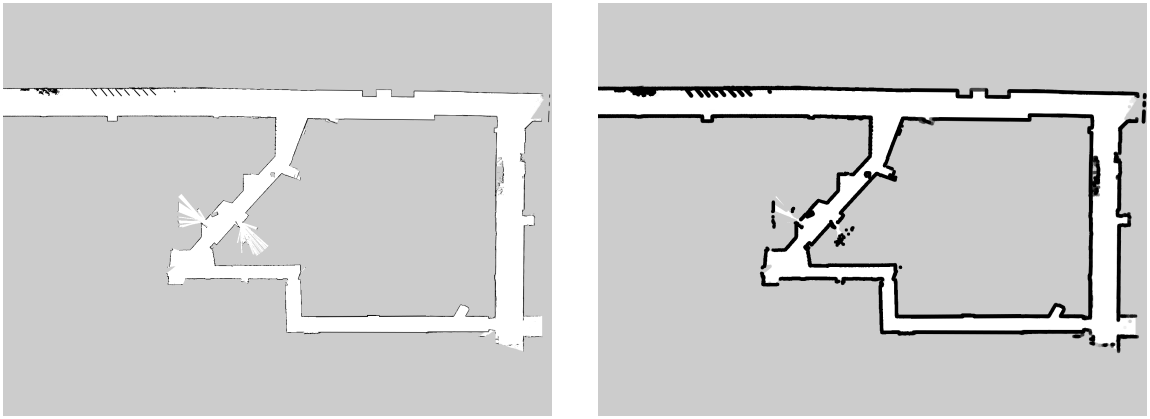

Figure 3: The Stata basement map before (left) and after (right) applying an erosion filter. This version of the map helps to prevent the robot from hitting obstacles as it follows a path.

2.2 Path Following (Anika)

In order to follow the path derived from our planning methods, we decided to implement pure pursuit, which is illustrated in Figure 4.

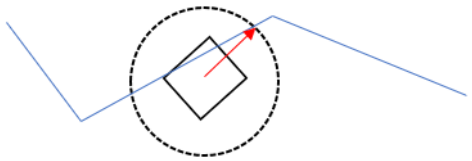

Figure 4: Simple representation of the pure pursuit algorithm.

Pure pursuit moves the robot from its current position to a look-ahead point in front of the robot. It does so by drawing a circle with a radius of the look-ahead distance, calculating where that circle intersects with the path, and then outputting a steering command. The steering command δ is specified by the below equation, where L is the length of the car, η is the angle between the car's heading and the line between the car and the look-ahead point on the desired path, and L_1 is the look-ahead distance.

$$\delta = \tan^{-1}\left(\frac{2L\sin\eta}{L_1}\right)$$

The result of our path planning algorithm is a piece-wise linear trajectory. For that reason, we perform pure pursuit on segments rather than the entire line. We do so keep a running track of what segment the car is currently on and perform pure pursuit only on that segment. While designing our pure pursuit algorithm, we realized that the look-ahead distance had to depend on both the speed of the car and the curvature of the path the car was following. Specifically, the look-ahead distance should increase when the car is traveling quickly or when the car is traveling along a long, straight path. It should decrease when the car is traveling slowly or when the car is traveling along a path with high curvature. Since the construction of the path is a piece-wise linear function specification, the curves are divided into many short segments. Therefore, we could use the length of the next segment as an indicator of how curvy the following path would be.

$$L_1 = d/d_0 \times k \times v$$

We used the equation above to dynamically determine look-ahead distance L_1. d is the length of the next segment, d_0 is a baseline look-ahead distance (we defined to be 2.0m), k is a multiplier we tuned and v is the velocity of the car.

2.3 Integrating Path Planning and Path Following (Quincy)

The integration of path planning and following works as follows: a start pose is initialized though localization data, an end goal is initialized, points on the path are created, the full path is published as a trajectory, and then pure pursuit follows this trajectory. The trajectory is constructed by the path planning algorithm once a start and end position is specified. This trajectory is then fed to

the pure pursuit algorithm as a series of poses in the world/map. These poses are parsed as points making up the path of the trajectory. In order for the pure pursuit algorithm to more accurately follow this path, we interpolated points in the given trajectory so that any resulting line segments being followed is more continuous. Lastly, the pure pursuit runs as specified in previous section on this modified trajectory.

3 Experimental Evaluation

In order to verify our code for trajectory planning and path following, we devised several simulation test cases to validate our implementations both individually and when integrated together. However, as demonstrating working code in simulation is insufficient, we also tested the integrated trajectory planning integrated with path following on the racecar in order to demonstrate that our solution works on hardware.

3.1 Path Planning (Aileen)

The A* algorithm will find the optimal path, but we need to add several conditions to constrain the optimal path such as dilating wall obstacles to prevent collisions. We also needed to add more neighbors than originally anticipated to find a more efficient path. This proved to be quite fruitful and we were able to follow the solution trajectory with high accuracy. This is shown in Figure 5 when we compare our solution with the instructor solution.

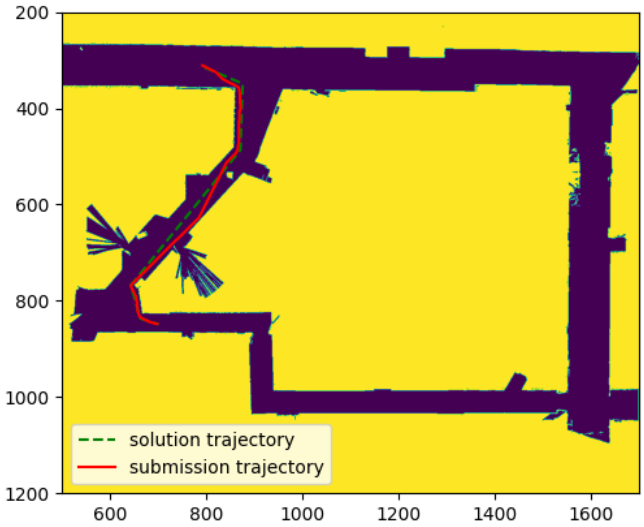

Figure 5: Our path planning algorithm compared with the solution trajectory, with a score of 2.999/3.0

3.2 Path Follower (Dylan)

Our implementation of path following involved the design of a pure pursuit controller. Before implementing this controller with the trajectory planning code, though, we first tested the pure pursuit controller on a pre-generated trajectory provided by staff. Evaluation of our controller involved measuring the distance between the car and the desired trajectory.

The pure pursuit controller interpolates the trajectory. As a result, we can generate several equations for lines connecting adjacent points by taking linear regressions. We can then easily find the distance from the car to each of these lines using the known equation for the distance from a point to a line as shown in Equation 1:

$$Distance = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}} \tag{1}$$

where the equation for the line connecting two interpolated points is defined as

$$ax + by + c = 0 \tag{2}$$

and the racecar is located at the point (x_0, y_0). By calculating the distance between the line segments closest to the racecar and taking the minimum, we can determine the distance of the robot from the trajectory.

A video demonstrating controller performance with the car traveling at 4[m/s] can be found in the appendix, while a graph displaying car error (deviation of the car from the trajectory) as a function of time can be found in Figure 6. The trajectory used for this evaluation is shown in Figure 7.

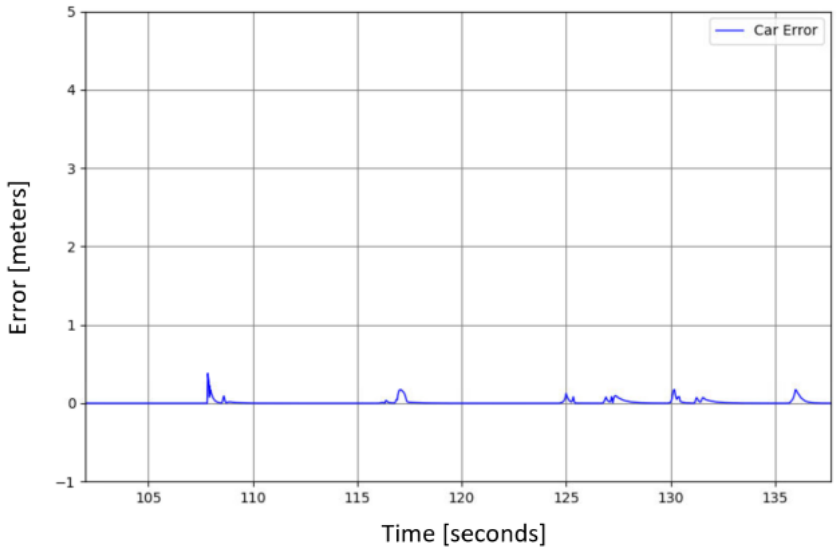

Figure 6: Deviation of car position from trajectory at a speed of four meters per second. Note that error remains at zero for the most part, with small spikes appearing at turns. This shows that the car mostly follows the trajectory, but has slight errors when taking corners. A video showing the simulation that provided data for this plot can be found in the appendix.

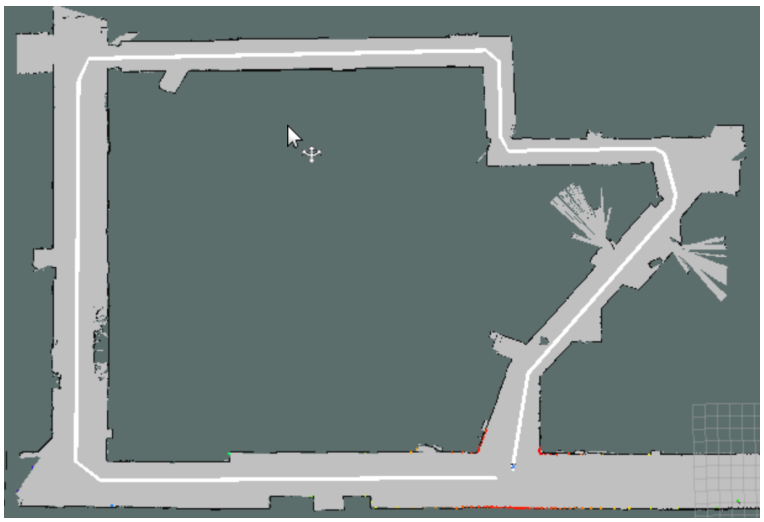

Figure 7: Map corresponding to trajectory used for the test in Figure 6

These results show that the car is capable of following pre-planned trajectories with great accuracy, as the error (as shown in Figure 6) remains close to zero throughout the course of the simulation.

In evaluating our controller, it was also important to determine the optimal lookahead distance. While our code dynamically sets lookahead distance in order to be able to navigate both sharp turns (small lookahead) and straight lines (long lookahead), we had to choose a default lookahead distance that the code uses on straight lines. This lookahead distance is then scaled down when traveling along turns. If the default lookahead distance is too small, the racecar will oscillate along the trajectory and may even become unstable. If the default lookahead is to large, then the car will be slow to recover from turns. A visualization of this behavior is shown in Figure 8.

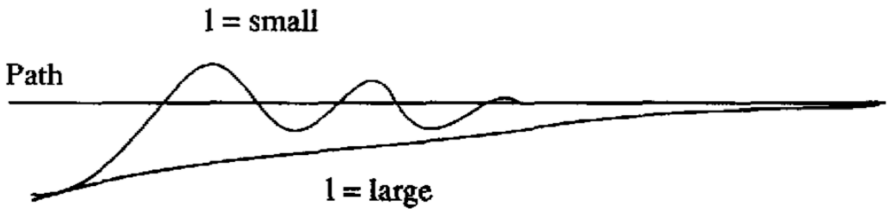

Figure 8: Plot showing how the car will oscillate with too small of a lookahead distance (l), but if l is too large, it will recover slowly.

In order to determine the optimal default lookahead distance, we ran multiple trials on the trajectory provided by staff and summed the error (distance from car to trajectory) at each time step. The results of these trials are shown in Table 1.

Table 1: Lookahead Distance Optimization

Max Lookahead [m]	Error Sum
0.5	Unstable
0.6	102.86
0.75	18.86
1	19.03
2	43.71

From Table 1, it appears that the optimal default lookahead distance is somewhere between 0.75 and 1. We can see that the error sum skyrockets when the default lookahead distance is lowered to 0.6; this is because the car begins to become unstable, oscillating on straight lines (see Figure 9). When lowered to 0.5, the oscillations become so severe that the car is unstable and cannot complete the trajectory.

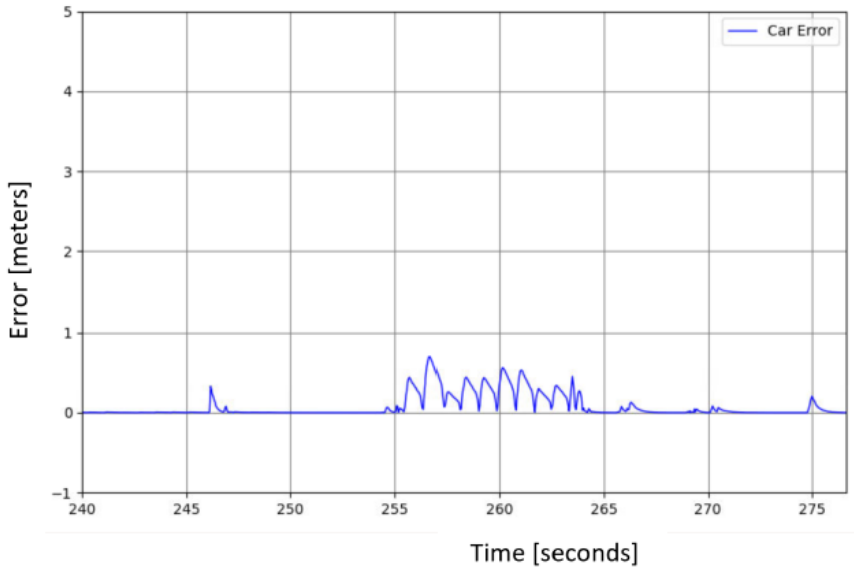

Figure 9: Car error as a function of time when default lookahead is set to 0.6 meters. Note the oscillitory behavior of the error, indicating that the lookahead distance is too small, thus causing the car to oscillate along straight lines. Note how the error at curves is similar in magnitude to Figure 8, but that oscillations can occur along straight trajectories.

When increasing the lookahead distance past 0.75, from Table 1 we can see that the total error also increases. This is because the car is not able to recover to the trajectory as quickly after taking a turn (see Figure 10). However, while 0.75 represents a slight increase in performance compared to a default lookahead distance of 1 meter, we choose to use 1 meter due to the fact that this increase in performance is small and because 0.75 lies rather close to the boundary for unstable behavior.

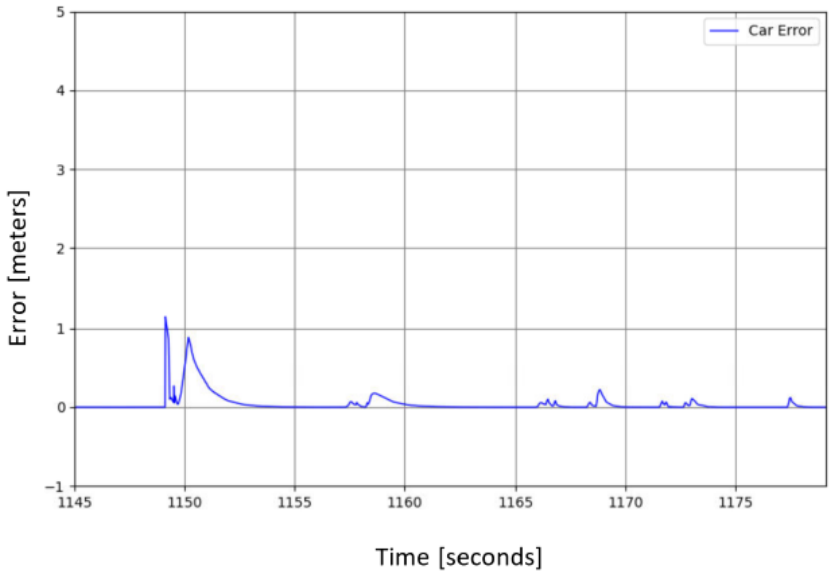

Figure 10: Car error as a function of time when lookahead is set to 2 meters. Note the decreased performance compared to Figure 8 on the curves indicated by the higher magnitude spikes in error. Also note how performance is similar to Figure 8 on straight lines, as expected.

3.3 Path Follower + Path Planning (Dylan)

With the pure pursuit controller now optimized and verified to make the race car follow the path with little error, we can now integrate and evaluate the path follower and path planning algorithms together. Performance evaluation follows the same logic as in the previous path follower section and it can be seen that the error remains close to zero for a custom trajectory, as shown in Figure 11, thus demonstrating that our integrated code works well together. The trajectory itself is shown in Figure 12. A video showing the car following this trajectory can be found in the appendix.

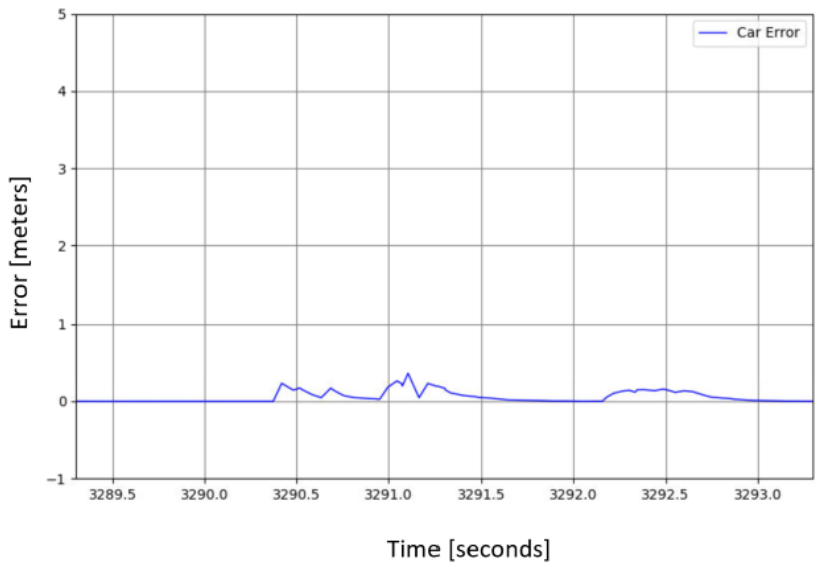

Figure 11: Plot showing error as a function of time along a custom path generated by the path planning algorithm.

Figure 12: Custom path generated and followed by the integrated path planner + path follower. This trajectory corresponds to the one used to obtain data for Figure 11

A more rigorous evaluation of our integrated code was obtained from the Gradescope test provided, the results of which are shown in Figure 13. It can be clearly seen that the generated path is followed almost perfectly, and indeed, the grade received from the autograder is around 99.8%.

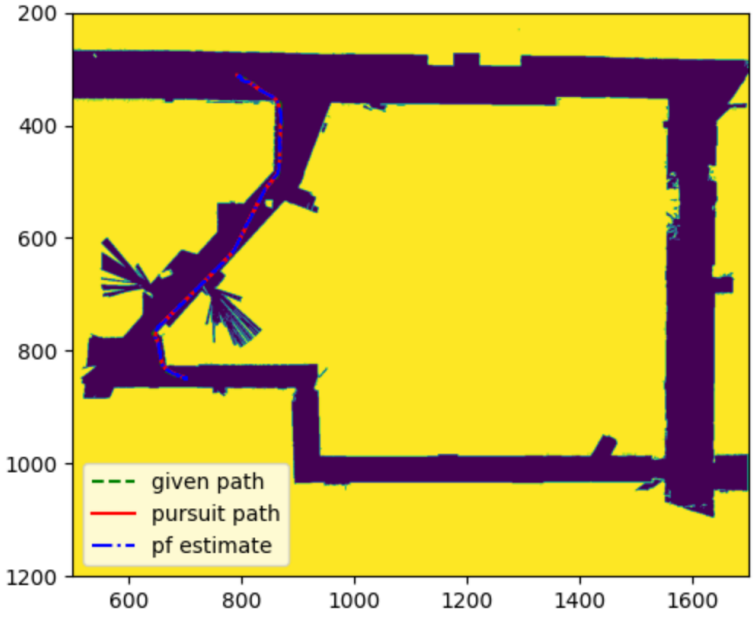

Figure 13: Results from Gradescope test. Note how the car follows the generated trajectory almost perfectly.

3.4 Hardware Implementation (Dylan)

While verifying in simulation is helpful, the most important step in demonstrating that our controller works is to demonstrate it working on hardware. Following a similar approach as before, after implementing our code on the race car we generated a custom trajectory and had the race car follow it. The distance of the car from the path (car error) as a function of time is shown in Figure 14, while the path itself is shown in Figure 15.

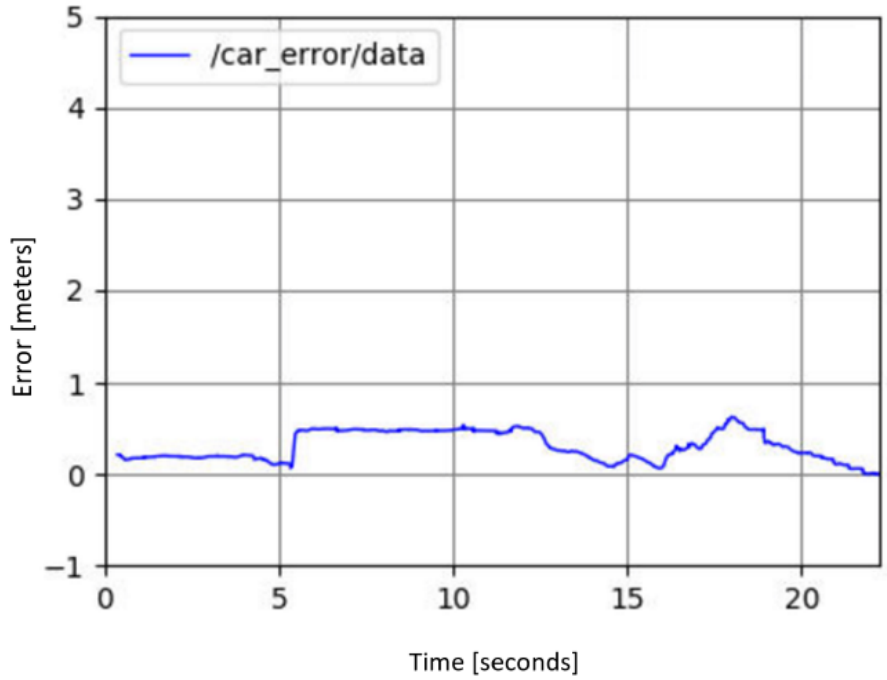

Figure 14: Car error as a function of time. Obtained from running code on actual robot.

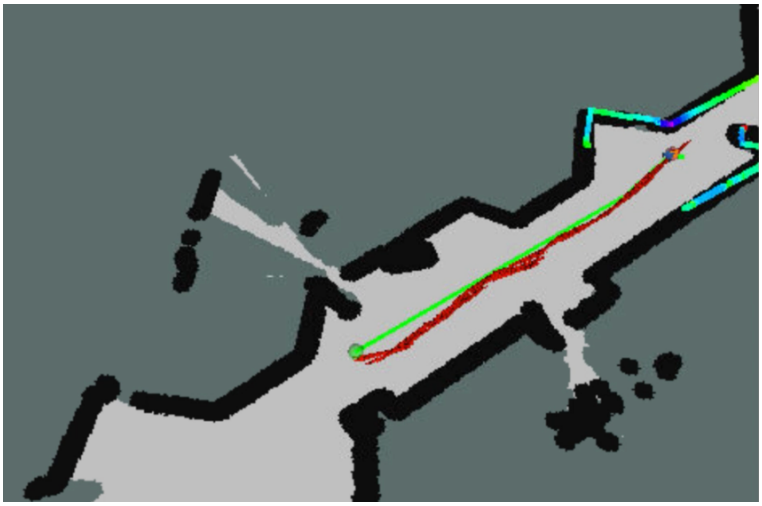

Figure 15: Path generated by code to obtain data for Figure 14. Generated trajectory is shown in bright green (before LIDAR data), path taken by car is shown in red.

From Figure 14, we can see that the error stays relatively close to zero over the course of the path. Figure 15 gives a visualization of the calculated trajectory and the path the robot takes, while a video showing this behavior in real life can be found in the appendix. Overall, both quantitatively and qualitatively, the code demonstrates solid performance on hardware for path planning and following.

4 Conclusion (Aileen)

We achieved the primary objective of this lab by implementing a path planning and path following algorithm such that our robot can successfully navigate a loop in the Stata basement.

First, we implemented an A* path planning algorithm. We decided upon a search-based algorithm because it was deterministic and guaranteed to produce an optimal path, which made testing more consistent. We favored discretizing the map into a grid space with piecewise points comprising the trajectory, as it was easiest to implement this with A* at the cost of necessitating a more robust pure pursuit implementation.

Second, we implemented pure pursuit in order to achieve our goal of path following. For any given trajectory, we seek to follow the closest point on the trajectory segment to the car itself, subject to the look-ahead distance. We also needed to modify pure pursuit to adapt to different look-ahead distances depending on whether the trajectory ahead followed a line or a curve. We implemented PID control to reduce oscillations in following the desired trajectory.

Finally, we integrated the two modules and combined them with the localization module from Lab 5. In doing so, our robot is able to estimate its current location via localization, plan a path with A* to a given goal location, and follow this path using pure pursuit with real time location updates from the localization module.

While we originally expected that the division of labor would be quite clear given the modularity of this lab, everyone actually collaborated quite extensively on all parts of the lab. Especially because design decisions from one team would affect those of another team, it was extremely important to communicate and help one another resolve issues. As a result, we all gained experience with both path planning and path following, along with resolving the difficulties involved with integration of the two modules.

Moving forward, there remain a couple of areas of improvement upon which we would like to build upon in the final project. First, we would like to fully develop RRT* and compare it with A*, weighing the tradeoffs of a more optimal path with the time spent on planning. Second, we would like to develop a better heuristic function, specifically testing with dubins curves. Third, we would like to explore the tradeoffs that favor path following over path planning, such as trajectory representation via spline curves rather than piecewise points. Fourth, we would like to spend more time tuning the parameters of pure pursuit to have smaller oscillations at higher speeds and follow corners more closely, along with eliminating any random occurrences where the robot spins in circles.

5 Lessons Learned

5.1 Aileen

I was really looking forward to this lab because path planning is my favorite topic in robotics. It soon became clear how challenging this lab was, especially in the sense of discovering how to collaborate as a team. The team split into two groups to work on the path planning module and path following module. Some design decisions favored one team over the other; for example, representing trajectories as piecewise coordinates was easier for the path planning team to implement, but more difficult for the path following team to execute. Learning how to balance these decisions could be challenging, and we resorted to creative solutions as a means of compromise. Moreover, this lab was highly collaborative, and we all relied on one another from debugging to calendar reminders. With the final competition coming up soon, I am proud to see the work our team has done so far and combine them for the last challenge.

5.2 Matt

This lab brought many challenges, and required the team and myself to keep fine-tuning our teamwork while also pushing us technically as we approach the final race. From the technical side, this lab did not give us much direction to tackle the challenge. We essentially had to construct the structure of the code on our own while also implementing complex algorithms like A* and RRT. Choosing the direction for our lab was exciting, yet presented challenges like coordinating how to approach each subset of the problem and modularizing tasks where we could. By communicating even more with each other in our group and and in person, we were able to work the problem. From the teamwork aspect, I found that we have been developing a nice rhythm for dividing up the work, recognizing what are strengths and weaknesses are, and accommodating each others' schedules. And I believe the biggest lesson we've taken to heart is being flexible with each other and keeping spirits high. We've really come together as a team as we face tougher and tougher tasks, and I'm very happy about that.

5.3 Anika

This lab was fairly challenging to work on and debug as the guidelines for it were broad. We learnt a lot about integrating difficult topics in robotics. Combining path planning and path following in an efficient and accurate way required a lot of planning both technically and in the communication aspect. We had to focus on making good design decisions that took into consideration both how much time we had to complete the lab and how well we wanted our path planning to work. As with every lab, we dealt with quite a few technical difficulties in both hardware and software that required a lot of time to solve. We found that switching off on solving problems and allowing for a new set of eyes for issues we've dealt with for a long time seemed to make our workflow much better. Finally, our team has really improved in balancing tasks and figuring out how to best work together, which manifested itself in a successful completion of this lab!

5.4 Quincy

Although this lab was more modular and structured than the last, the technical challenges presented were abundant. This lab really focused on design decisions more than anything else as there were

several trade-offs that had to be considered. For example, one such trade-off was speed versus accuracy when it came to path planning algorithms. There were several other trade-offs to be explored, however due to technical difficulties and time we didn't get to fully implement some possible design decisions (RTT and dubins curves) that could provide better insight into these trade-offs. Overall, I have gained a greater understanding of how different components of autonomous systems (localization, path planning, and path following) work together, and I look forward to the final challenge which only add to the amount of coinciding components.

5.5 Dylan

This lab was definitely one of the most challenging yet, and really required us to work together as a team. This lab required us to think outside of the box and consider many edge cases which can occur and create errors in our code. I believe we all learned more about problem solving while putting our heads together to solve challenges and come up with those critical edge cases. We faced many challenges, the most daunting of which was hardware. This has led to a lot of late nights for our team as a whole, and I believed that has made it so that we have had to learn to work together well in order to power through tough hours and tough problems. Working with this team has been very rewarding thus far, and I look forward to working with all of them on the final challenge.

6 Appendix

YouTube links demonstrating performance can be found at the following links:

Figure 6: https://youtu.be/n9rSkupNDkY

Figure 11: https://youtu.be/_r_qx5EmSNc

Video of car in real life: https://youtube.com/shorts/umixedlU-sU

Rviz of car in real life: https://youtu.be/IS3XY4rTcZM