Lab 6 Report: Path Planning

Team 18

Eric Delgado, Arianna Ilvonen, Jesus Diaz, Charles Ge, Dora Hu

---

1: Introduction [Dora]

Path planning and path following is a key component of robotics, wherein the subject takes in a start point and endpoint for the path planner to plan a trajectory to follow, and the path follower tells the car how to keep following the trajectory. Path planning and path following are key to navigating unfamiliar environments and optimal paths. The path planner requires knowledge of the car's odometry and its environment to determine which space does not have obstacles, as well as a start and end point to plot a trajectory for. Given these, the path planner can be implemented with search-based or sampling-based algorithms to find a trajectory from the start point to the end point. The trajectory the path planner finds is used by the path follower to publish drive commands at regular intervals to ensure the car continues following said trajectory. The path follower requires the car's odometry to localize, then search for a point ahead of the car on the trajectory to go to. The car is told to travel to this lookahead point by determining what angle it must turn to reach this point and publishing a drive command with this angle. By continually publishing drive commands telling the car to travel further along the trajectory, the car will eventually reach the trajectory's destination.

2: Technical Approach [Arianna, Dora, Jesus, Eric]

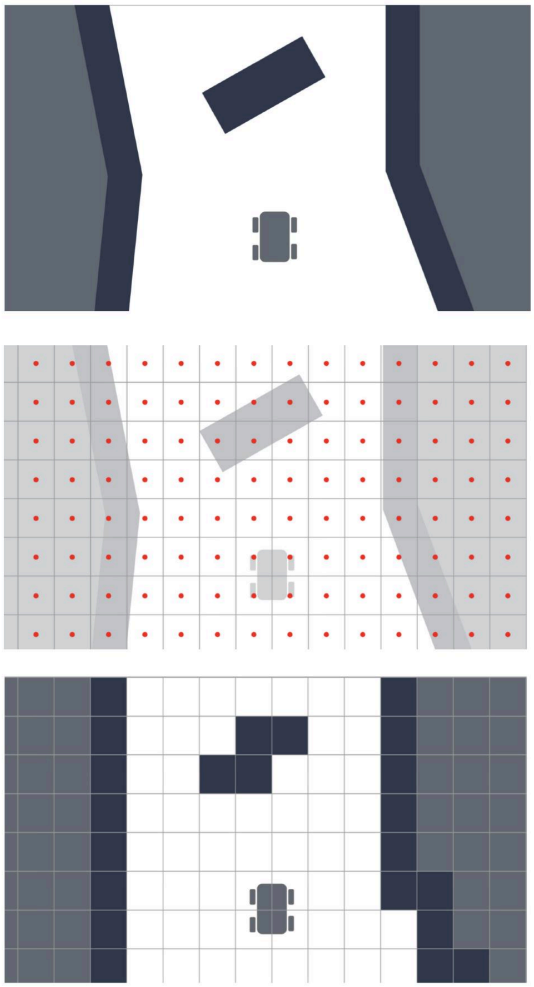
To implement path planning, we made two primary nodes: one for path planning and one for path following. We also reused our particle filter from Lab 5 to localize. Thus, collectively we relied on a map of the car's environment, the car's own location, LIDAR scans from the car's camera, and the start and end point provided by a user driving the car. The interaction between the path-planning and path-following nodes is characterized by the publishing and subscribing to a topic representing a trajectory, which can be generated with a variety of path finding algorithms like Breadth-First Search, A*, and Rapidly-Exploring Random Trees*.

2.1: Search-Based Path Planner: Map Discretization [Jesus]

Before implementing a path-planning algorithm, it was necessary for us to develop a method for discretizing the map data given to us into a traversable graph. In this graph, nodes would represent a particular area on the map and each node would be connected by unweighted edges to the four neighboring nodes—that is, the areas directly adjacent to that particular node in the four cardinal directions. We chose to keep our algorithm as simple as possible: discretize the map into a grid of 1x1 squares and assign all points that fall within the bounds of that square to that particular location node. To account for points that fell between these squares, we would also consider the bottom and left edges of grid squares as part of that particular location node. Nodes would be identified by the location of their center point, which we would also use to index into the map data and determine whether the area at a particular node was open space, an obstacle, or unknown.

One of the main advantages of representing the graph in this way is that it allowed us to identify nodes and neighbors from a given point with a simple floor operation that could be computed in fractions of a second. Because of this, it made more sense for us to build out our graph as we ran

our path-finding algorithms, and thus save ourselves the memory and performance overhead of pre-computing the graph for all nodes in the map.



$$\textbf{discretize}(x, y) = x : \lfloor x \rfloor + 0.5, y : \lfloor y \rfloor + 0.5$$

Figure 1: Visualization of the discretization of the 2D map space
with the formula used to assign (x,y) points to a particular node.

2.2: Search-Based Path Planner: Breadth-First Search Algorithm [Jesus]

Breadth-First Search (BFS) is a Search-Based algorithm that explores the nodes of a graph by visiting all the nodes at the present depth before moving on to the nodes at the next depth level. Here, “depth” refers to the distance of a node from the starting node in the graph. Because BFS

visits nodes in order of their distance from the source node, it is a useful algorithm for finding the shortest path between two nodes in an unweighted graph, as it guarantees that the shortest path will be discovered first.

BFS is typically implemented with a queue to keep track of which nodes need to be checked next, a *parent* map which we use to keep track of the previous node in the path as we run our search algorithm, and a *visited* set which tells us which nodes we have previously added to the queue. When we are ready to explore the next node, we remove a node from the front of the queue. This means that the node that has been in the queue for the longest time is removed first, which is necessary for the breadth-first nature of this algorithm. Once a node has been dequeued, we check its neighbors, add unvisited neighbors to both the back of our queue and our *visited* set, and add entries for these unvisited neighbors in the *parents* map, with each neighbor being mapped to the node that was dequeued. For the purposes of this lab, we also had to check if neighbors were open spaces before adding them to the queue. We repeat this process until we dequeue a node and find that it is our end node, at which point we can find a path by recursively tracing back through our *parents* map until we reach the start node.

2.3: Search-Based Path Planner: A* Algorithm [Eric]

A* Search is an informed search algorithm that efficiently finds the shortest path from a start node to a target node by using heuristics to estimate the best path through the remaining nodes. It is designed to guarantee an optimal solution; however, its computational demand can result in longer runtimes compared to other pathfinding methods. The heuristics chosen are implemented into a cost function (refer to Fig. 1) that assigns a value to every visited node, and decides to continue traveling along the path with the lowest cost. The heuristics chosen for this lab are common in path-planning applications and essentially represent the distance from the

start node and the distance from the goal node. This heuristic must be admissible, meaning it never overestimates the actual cost to reach the goal.

$$J = S * 10 + D * 14 + \text{Heuristic}$$

$$\text{Heuristic} = 10 * \sqrt{(x_{current} - x_{goal})^2 + (y_{current} - y_{goal})^2}$$

$$S = \text{Number of Straight Path Moves}$$

$$D = \text{Number of Diagonal Path Moves}$$

Figure 2: Cost Function associated with assigning values to traveled nodes in A*. Finding an appropriate, admissible heuristic is critical to the successful implementation of A*.

A* operates by first discretizing a grid of potential nodes and maintaining a priority queue of nodes to be explored, prioritizing them based on the cost function. Starting from the initial node, A* selects the node with the lowest value, expands it to its neighbors, updates their costs, and repeats this process (See Figure 2). The search persists until the goal node is chosen for expansion, ensuring the found path is optimal. A* is efficient because the heuristic helps to significantly reduce the number of nodes that are explored compared to brute-force algorithms such as BFS or DFS, effectively directing the search towards the goal.

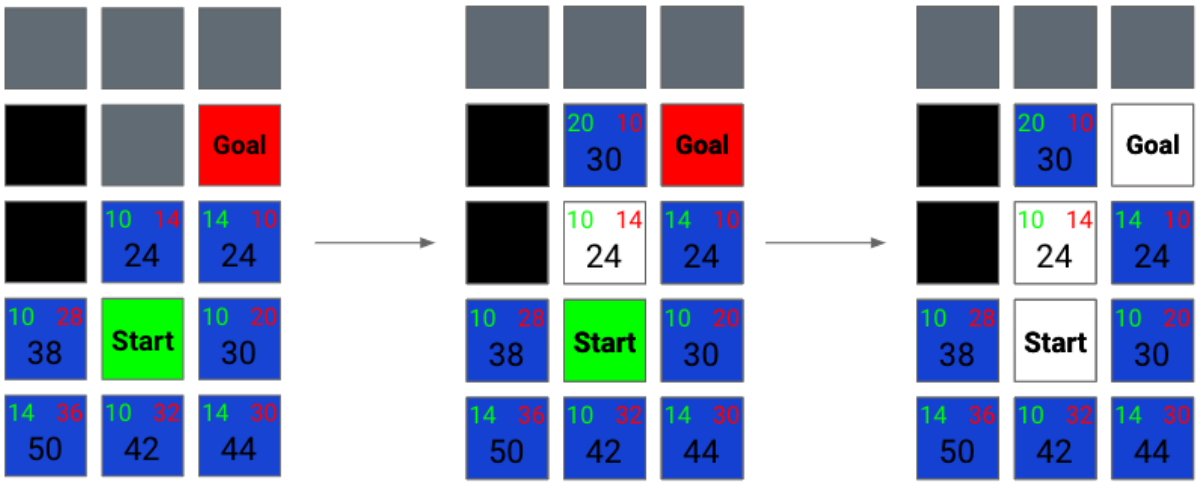


Figure 3: Visualization of A* Search on a simple graph traversal. In the leftmost state, the cost for all neighbors of the starting node have been assigned. In the middle state, the highest-priority node has been de-queued and the cost of its neighbors calculated and assigned. Since the goal node is in the neighbors of the current node, in the final state, an optimal path has been found from the start node to the goal node. Note: The number in green represents the distance to the starting node, the number in red represents the distance to the goal node, and the number in black represents the total cost of the node.

In addition to the algorithm's fundamental operations, A* also incorporates techniques to manage computational resources more effectively, such as closed sets and open sets. The open set contains nodes that have been visited but not expanded (i.e., their neighbors have not all been explored), and the closed set includes nodes that have been fully expanded. This management prevents the algorithm from revisiting and reprocessing nodes unnecessarily, thereby enhancing efficiency. Still, as shown in Figure 3, some of the more complex paths resulted in relatively high path-finding times.

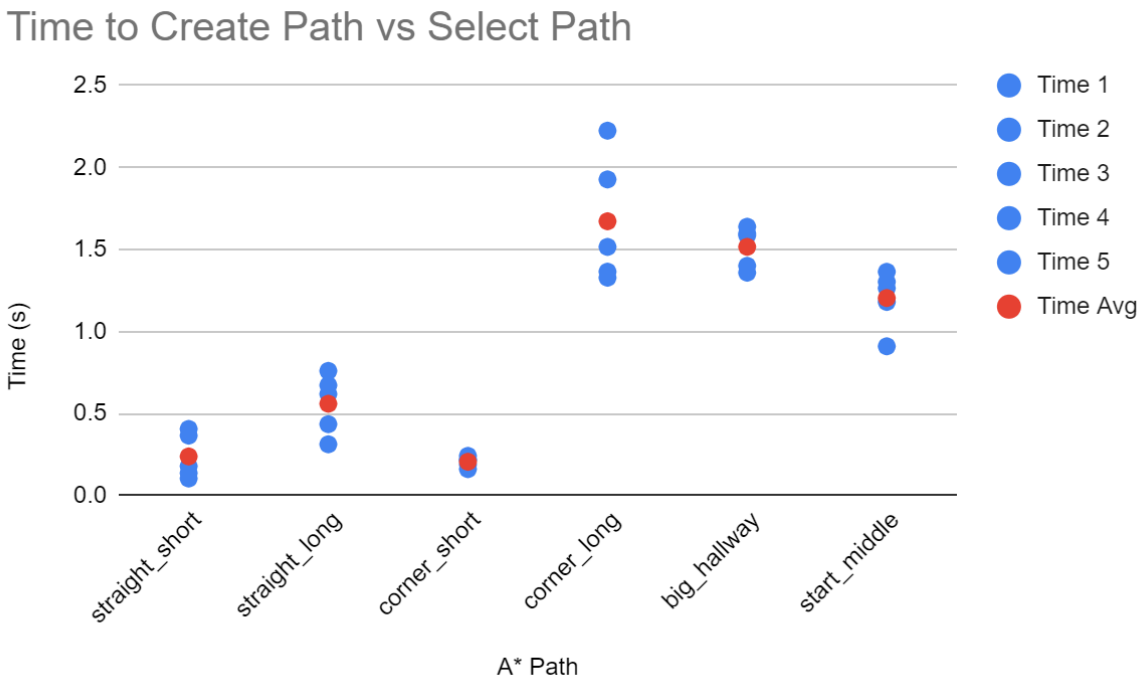


Figure 4: Scatter chart displaying the total path generation time for all 6 standardized test paths. This diagram displays the reduction in speed when navigating more complex paths, such as corner_long.

2.4: Sampling-Based Path Planner: RRT* Algorithm [Arianna]

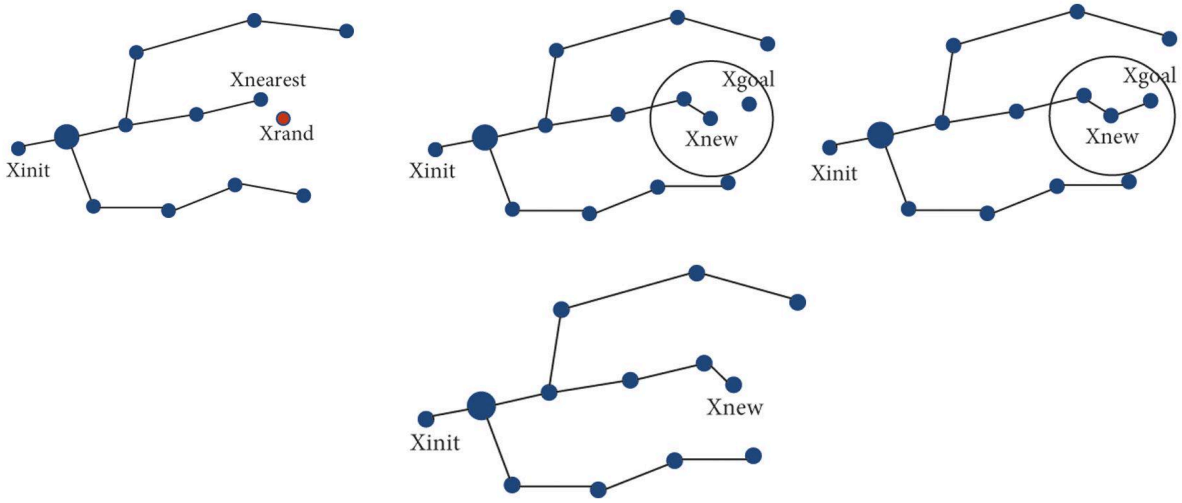


Figure 5: Figure shows a progression through the RRT algorithm, first a random point is generated, then it is connected to the closest other node available. This path is then checked for collisions, and, if there are none, added to the full tree as a node. Adapted from [1].

The sampling based path planner, Rapidly Exploring Random Trees (RRT), functions by randomly generating points to build an expanding tree. It halts only once the tree has connected to the desired point, or the number of iterations has expired. The algorithm begins by generating a random point on the map, then checks to make sure that this point is not within an obstacle. Once a valid point has been generated, the algorithm attempts to connect it with the nearest other point available. Once the nearest point is found, the code generates a straight line between the two points. This set of steps is visualized in Fig. 4. Points are generated by the algorithm at a given step length along this path, and each point is checked to make sure that it is not within an obstacle. If any point is found to be within an obstacle, the point is removed. If all of the points are clear, this path is considered valid, and the node is added to the tree. Once the node is added, the algorithm checks if the point is within a given radius of the target point. If it is, a path is generated to the target and checked for collisions. If this path is collision free, the path is complete and exported. Otherwise, the program discards the path and continues.

Each Node is an object from a custom class containing its x, y location, a theta orientation in radians, and a pointer to its “parent” node. This structure allows us to “backtrack” to the origin from any given node, leaving us with a path from the origin to the node. The origin node is represented in the same way as the others but has no parent node.

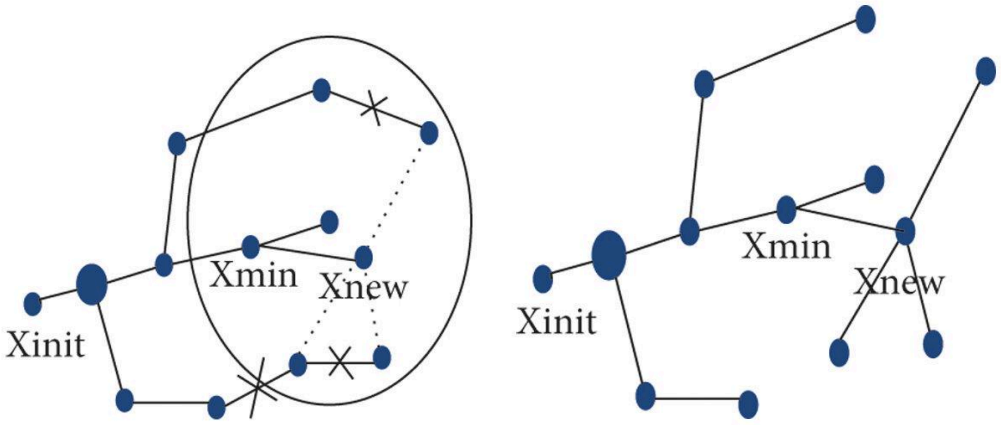


Figure 6: This diagram displays the optimizations made to become the RRT* algorithm. Instead of just finding the closest point, the algorithm now searches all of the points in a given radius, and connects to the node with the shortest total path to origin. It then checks all the other points in the radius to determine if connecting to the new point would give them a shorter total path to origin. This set of steps enables the tree to "rewire" itself and become more optimal over time. Adapted from [1].

With a fully developed RRT algorithm, the algorithm was optimized further by adding functionality from the RRT* algorithm. Instead of checking for only the nearest node once a point is generated, the new algorithm cycles through every node within a set distance from the new point and connects to the one that has the lowest total path length back to the origin. After the node is added, the program goes on to check if any of the nodes within the radius would have a shorter total path length if connected to the new node instead, as seen in Fig 5. These changes enable the tree to "rewire" itself over time; as new nodes are added, connections become more direct.

Once a path is found, it is converted to a trajectory and published to the path following algorithm. Then, the program continues to run iterations until a path with a shorter total path length is found. This allows the program to generate better paths over time.

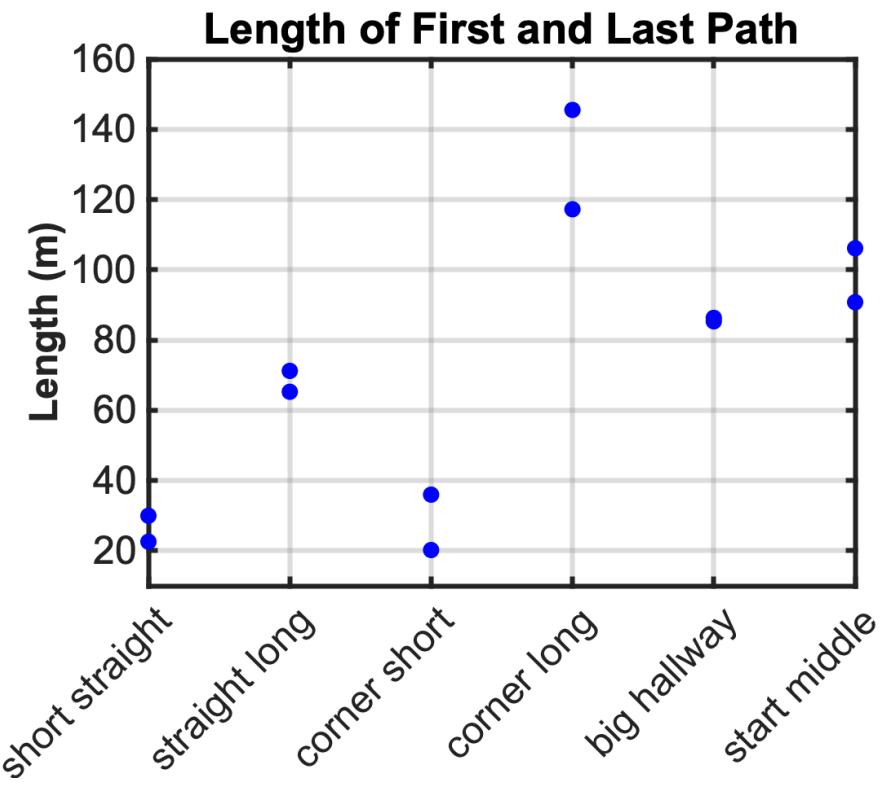


Figure 7: Scatter chart displaying the total path length of the first and last path generated by the RRT* algorithm in a two minute period for each route. This diagram displays the improvement in path optimality over time, especially in locations with complex environments like the corner long route.

In order to characterize the optimality of the paths generate and the performance of the code, I measured the length of time it took RRT* to generate a path for a variety of different routes and the length of paths generated by RRT* over a two minute period. All of the routes are shown in FIG. All of these trials are shown in Figure 6, with the average length of time to arrive at the first path plotted in red. From this graph, it's apparent that more complex environments, such as the corner long and start middle routes, take longer for paths to generate. This is likely because these areas of the map are narrower, with more obstacles. This means that the algorithm needs to generate and connect more points to achieve a valid path. Additionally, Figure 7 displays RRT*'s improvement from its first path to more optimal paths as time progresses. At the start of the trial, the route has multiple kinks and non optimalities, but as time progresses it

becomes more direct. Figure 8 shows the improvement in each route over the two minute trial period, displaying both the path length of the first generated path, and the path length of the last generated path. From here, it is apparent how large these improvements can be for the various pathways, especially the more complex ones. The corner long path, for example, has an improvement of over twenty meters in the two minutes the experiment ran for. Overall, this data demonstrates that RRT* takes at least a couple seconds to generate any valid path, and an even longer period of time to generate a semi-optimal path. Figure 9. Shows the computational cost of the RRT* algorithm by displaying one trial run of the corner long route. As time progresses, the path becomes much shorter, but the total number of points generated increases exponentially. While RRT* does not require discretization of the map, and is capable of responding to dynamic, real time changes in the map, the long compute time did not make it a good choice for our final path planner in this lab.

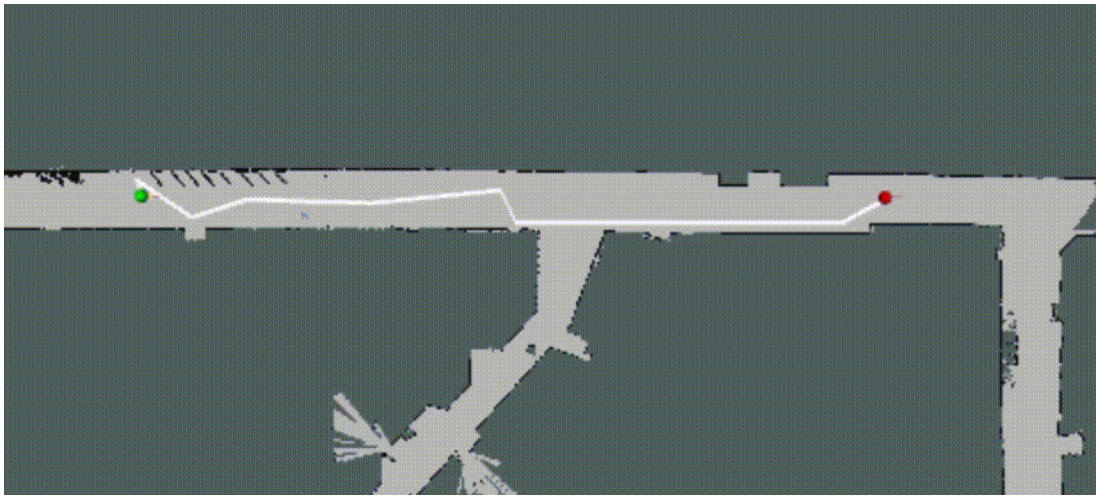


Figure 8: This figure shows the path generated for the straight long route over one trial run of two minutes. At the start, the path has more kinks and non optimalities, but as time progresses the path is replaced by more and more optimal versions. This shows how the RRT* algorithm progresses to a more optimal version over time.

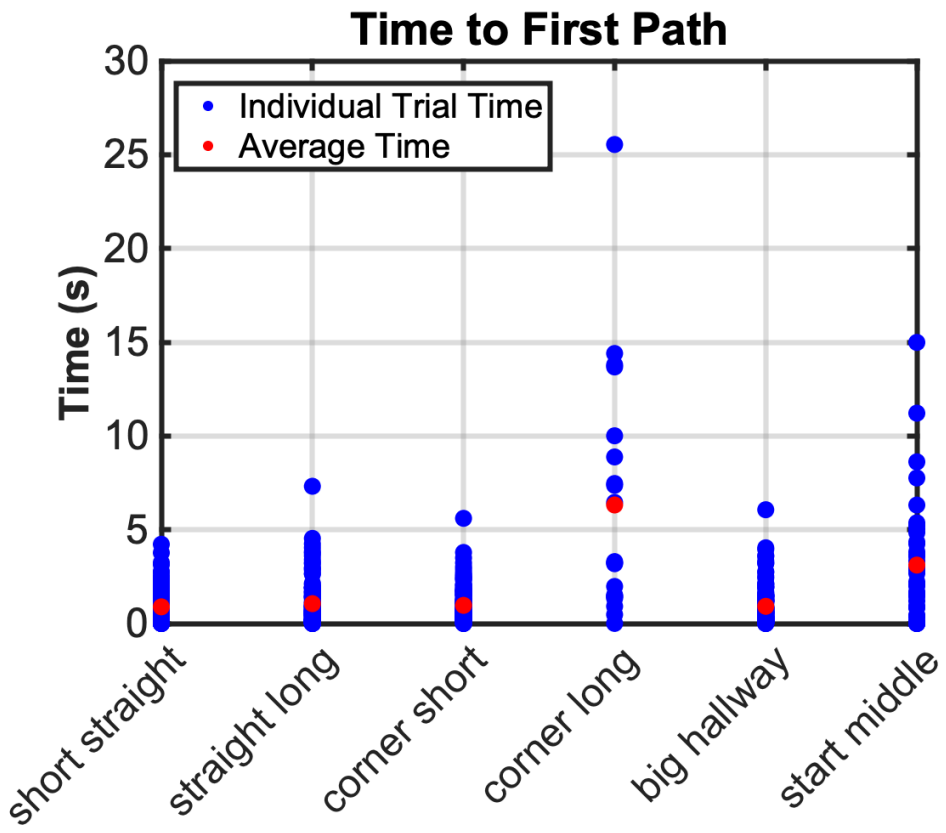


Figure 9: Scatter chart displaying the time to first path generation for multiple trials of each route (each trial represented by one blue dot), along with the average time to path generation plotted in red. This data suggests that the RRT* algorithm takes on average a few seconds to generate a path for any location, and sometimes takes much longer, especially in more complex areas of the map.

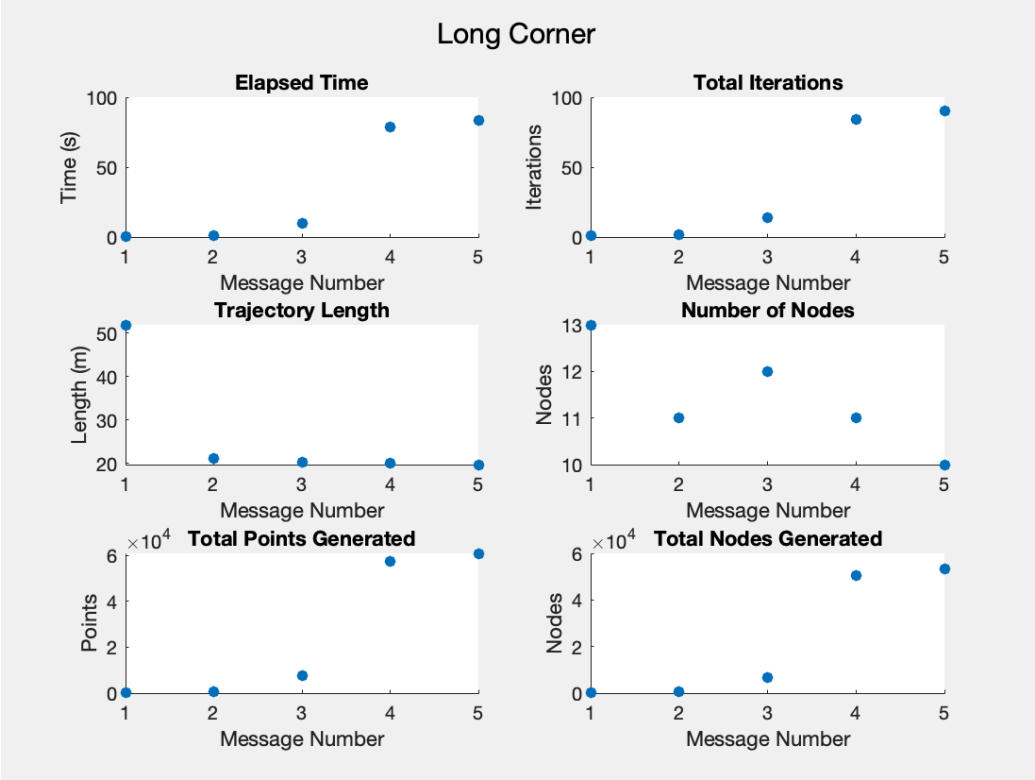


Figure 10: Graph shows the increase in total data generated over one two minute trial of the corner long route. As time progresses, the trajectory length gets much shorter, but the total points generated increases exponentially. This shows the computational cost of the RRT* algorithm.

2.5: Comparison of Path Planning Algorithms [Eric]

To evaluate our path planning algorithms, we aimed to analyze both the path length and the time required to generate specific routes. We established six standardized starting and goal positions (illustrated in Figure 10) to assess our algorithms. Given that both BFS and A* are optimal pathfinding algorithms, we anticipated similar path lengths between them but expected A* to be quicker as it does not examine every node. We predicted that RRT* would both require more time and yield longer path lengths due to its method of randomly populating the space until a valid path is discovered, with optimality only theoretically achievable as time tends toward infinity.

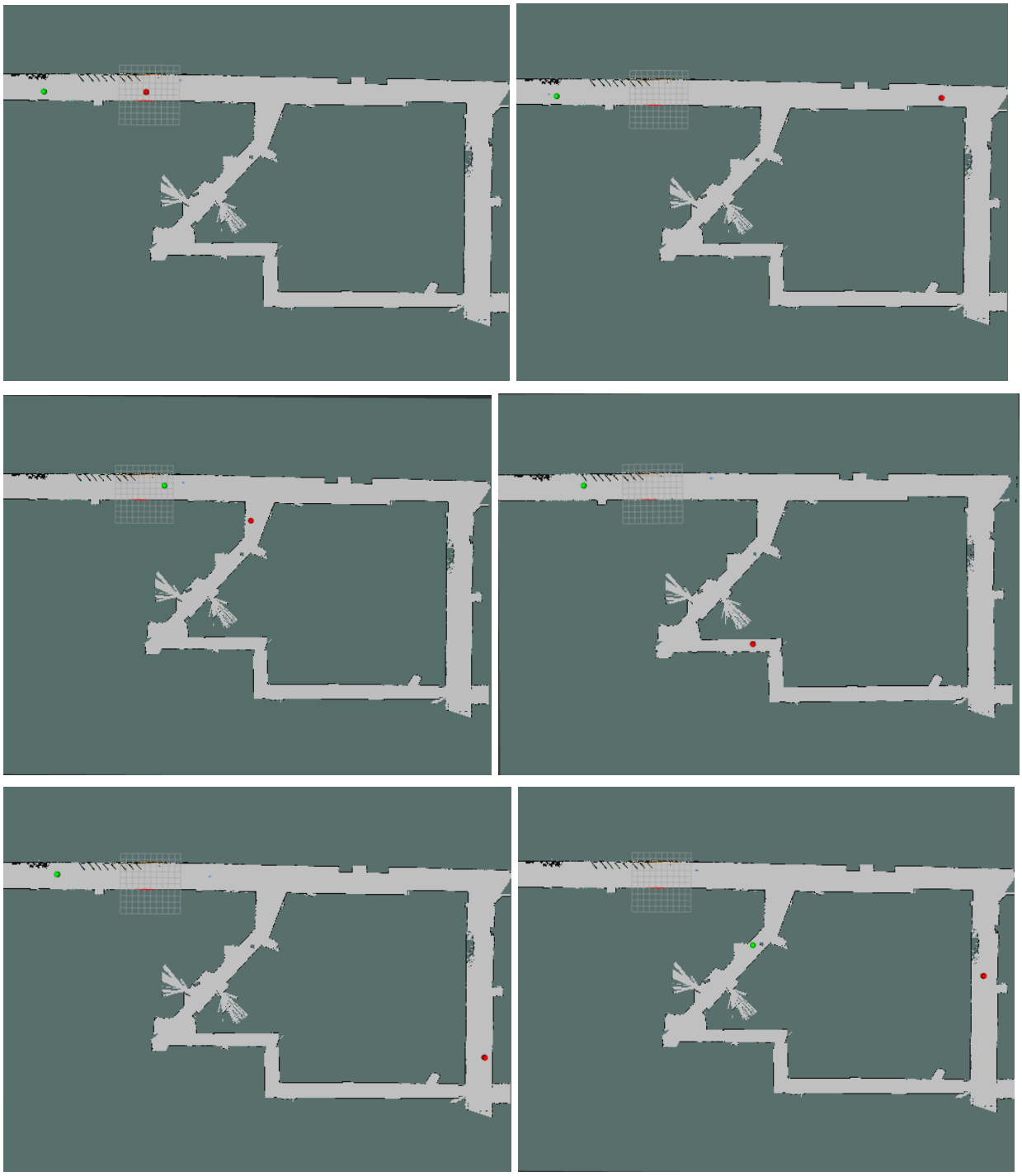


Figure 11: Testing Routes used to compare BFS, A*, RRT*. Start nodes are shown as the green dots while goal nodes are shown as red dots. Traversing in row major order, the paths are labeled as: straight_short, straight_long, corner_short, corner_long, big_hallway, and start_middle

The outcomes of our experiments are presented in Figures 11 and 12. As hypothesized, A* and BFS produced similar path lengths (optimal paths), whereas RRT* generated longer paths even after allowing each simulation a full minute to enhance path quality. Contrary to our expectations, however, BFS proved significantly faster than both A* and RRT*. Upon analysis, we determined that, given the small size of the discretized grid and the limited navigable space, the time saved by A* in not exploring every node was outweighed by the computational overhead of evaluating the cost of each node. This resulted in BFS, a brute-force approach, being more efficient under these specific conditions.

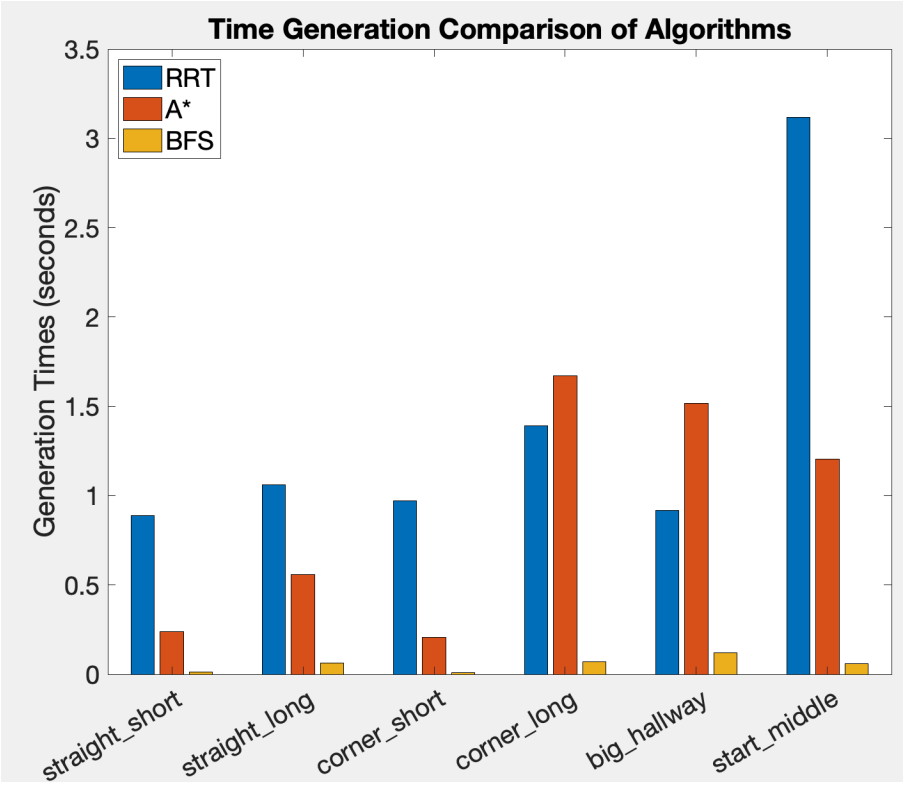


Figure 12: Data collected from simulation path planning of the 6 standardized routes. Comparison of the time needed to generate a trajectory by each algorithm. BFS is shown to be superior for this application.

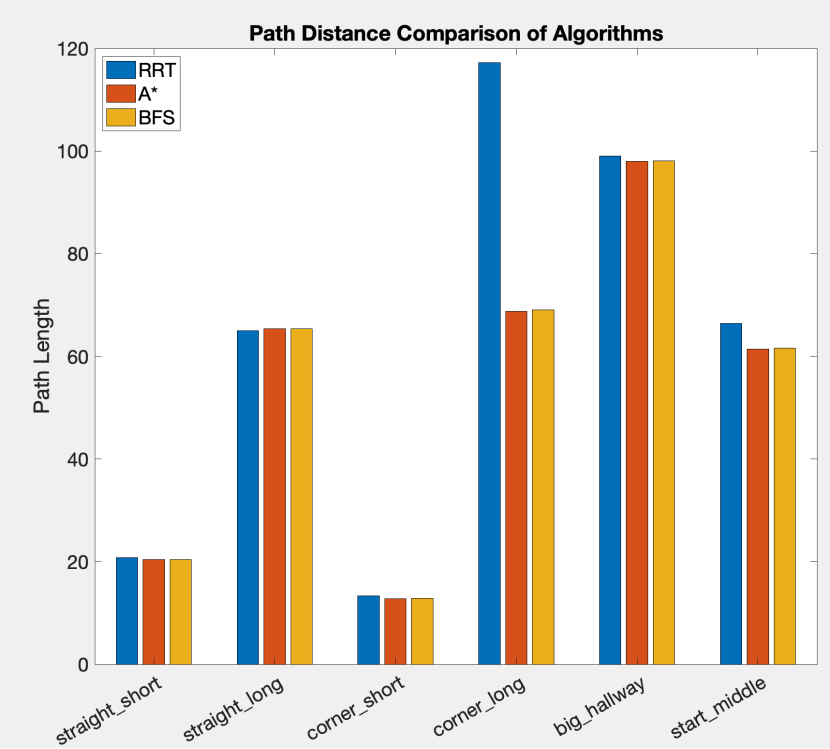


Figure 13: Data collected from simulation path planning of the 6 standardized routes. Comparison of the path distance created by each algorithm. BFS and A* shown to be optimal path-planners.

2.6: Path Following Algorithm: Pure Pursuit [Charles]

We use a pure pursuit controller to steer the robot along a given trajectory computed by the path planning algorithms described above. The pure pursuit controller subscribes to a live localized position topic (`/pf/pose/odom`) and a topic publishing the received trajectory consisting of line segments (`/trajectory/current`). The controller then outputs drive commands to the robot (`/vesc/input/navigation`). Figure 13 below depicts this algorithm logic.

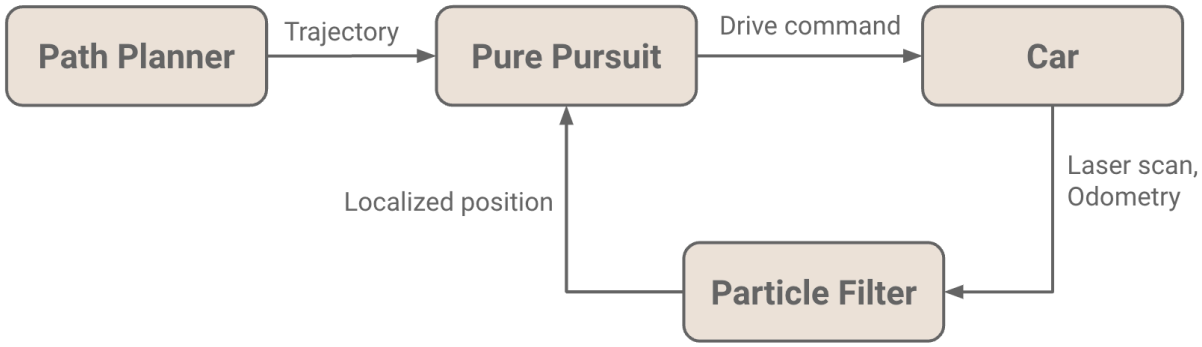


Figure 14: Diagram of the implementation logic for the pure pursuit controller.

The pure pursuit controller operates in three main stages. First, it computes the distance between the robot location and every line segment of the trajectory to determine the closest line segment. For each line segment, the perpendicular distance between the robot point and the infinite line passing through the segment endpoints is determined using the classic point-to-line distance formulation. Then, taking the robot position as the vector $\vec{p}$ and the endpoints of the considered line segment as the vectors $\vec{a}$ and $\vec{b}$, the parallel component of the distance between the robot position and the line segment is given using equations (1) and (2) below:

$$\vec{v} = \frac{\vec{b}-\vec{a}}{\left\|\vec{b}-\vec{a}\right\|} \tag{1}$$

$$d_{||} = max\left[\left(\vec{a}-\vec{p}\right)\cdot\vec{v},\ \left(\vec{p}-\vec{b}\right)\cdot\vec{v},\ 0\right] \tag{2}$$

where we compute $\vec{v}$ as the normalized tangent vector along the line segment in (1) and use it in (2) to find the parallel components of the signed distances from $\vec{p}$ to $\vec{a}$ and from $\vec{b}$ to $\vec{p}$. [2] Taking the maximum of these signed distances with 0 yields no parallel distance component when the projection of $\vec{p}$ onto the infinite line lies between the two segment endpoints and the positive distance to the closer endpoint when the projection of $\vec{p}$ does not lie between the endpoints, as desired. Finally, we combine the parallel and perpendicular components of the distance to find the total distance to each line segment and determine the closest line segment of the trajectory. This code is vectorized with numpy arrays for efficiency.

The second stage of the algorithm computes a navigational target point by finding the intersection of a circle of radius equal to the lookahead distance centered at the robot location with the trajectory path. Each line segment starting with the closest segment computed in the first stage is checked for intersection points with the circle. It can be shown with some vector algebra that the intersection points are given by $\vec{a} + \left(\vec{b}-\vec{a}\right)t$, where t are the solutions to the quadratic $c_2t^2 + c_1t + c_0 = 0$, where the constants c_2, c_1, and c_0 are given by the equations below

$$c_2 = \left(\vec{b}-\vec{a}\right)\cdot\left(\vec{b}-\vec{a}\right) \tag{3}$$

$$c_1 = \left(\vec{b}-\vec{a}\right)\cdot\left(\vec{a}-\vec{p}\right) \tag{4}$$

$$c_0 = \vec{a}\cdot\vec{a} + \vec{p}\cdot\vec{p} - 2\vec{a}\cdot\vec{p} - r^2 \tag{5}$$

and r is the lookahead radius. [3] If solutions for t exist, the algorithm selects the target point as the one further along the trajectory (the greater root of the quadratic). If no target point is found

after searching the remaining trajectory, the robot has deviated too far from the trajectory and a stop command is written to the robot.

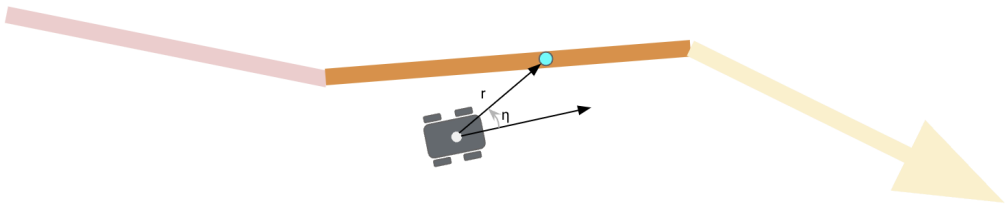


Figure 15: Example diagram illustrating how the pure pursuit algorithm works. The orange segment is the closest line segment on the trajectory. The cyan dot is the computed target point which is a distance r, the lookahead distance, from the robot location.

Lastly, the pure pursuit algorithm issues a steering command using the current robot pose and the target point on the trajectory that is a lookahead distance from the robot location. The steering angle command δ is derived from the pure pursuit formula in equation (6) below:

$$\delta = \tan^{-1}\left(\frac{2L\sin(\eta)}{r}\right) \tag{6}$$

where L is the wheelbase length of the car and η is the angle between the vector from the car's position to the target point and the car's current orientation vector (also depicted in Figure 2). [4]

3: Experimental Evaluation [Charles, Dora, Arianna]

We tested the pure pursuit controller qualitatively in simulation and quantitatively in real world situations by combining it with the path planning algorithms. In evaluating just the controller in simulation, we found that greater driving velocities necessitated greater lookahead distances to manage performance. The pure pursuit controller tracked straight portions of the trajectory better with greater lookahead distances and tracked curved portions of the trajectory better with lesser lookahead distances. Further work on the pure pursuit algorithm could incorporate dynamic lookahead distances based on the trajectory portion and steering wheel angle. Ultimately, we found that a lookahead distance equal to 0.8 times the velocity plus 0.2

meters produced the best handling, stability, and path following across multiple choices of velocities, segment lengths, and segment curvatures.

When testing our path planner and path follower both in simulation and in-person, we attempted 0.6 m/s, 1.0 m/s, 1.5 m/s, 2 m/s, and 2.5 m/s. For each speed in each situation (simulated or real-world), we ran our car along paths calculated from the same start and end point—from the stairs to the entrance of the lab room—to control our results. Our trajectory is pictured in Figure A.

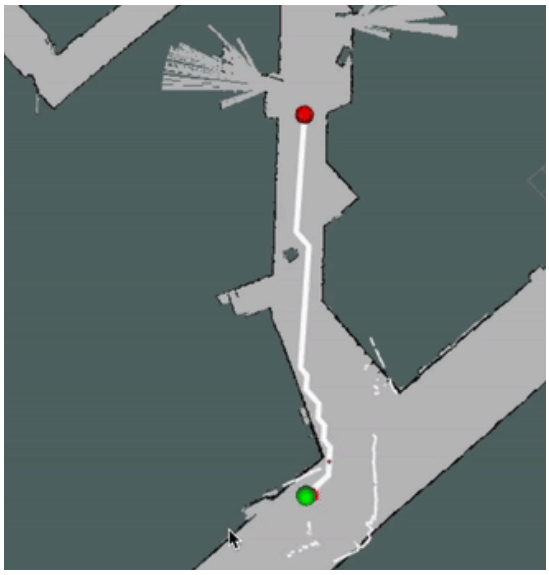


Figure 16: The trajectory from the common start and end point for all our test runs.

During these runs at differing speeds, we recorded the distance of the car from the trajectory at regular intervals. Then we calculated the average distance of the car from the planned trajectory and plotted the average errors of simulated runs against their real-world counterparts with the same speed, as shown in Figure B. We found that on average, the real-world runs had more error than the simulated runs—however, both simulated and real-world

runs had very small amounts of error, indicating on average, very little deviation from the intended trajectories calculated by the path planner.

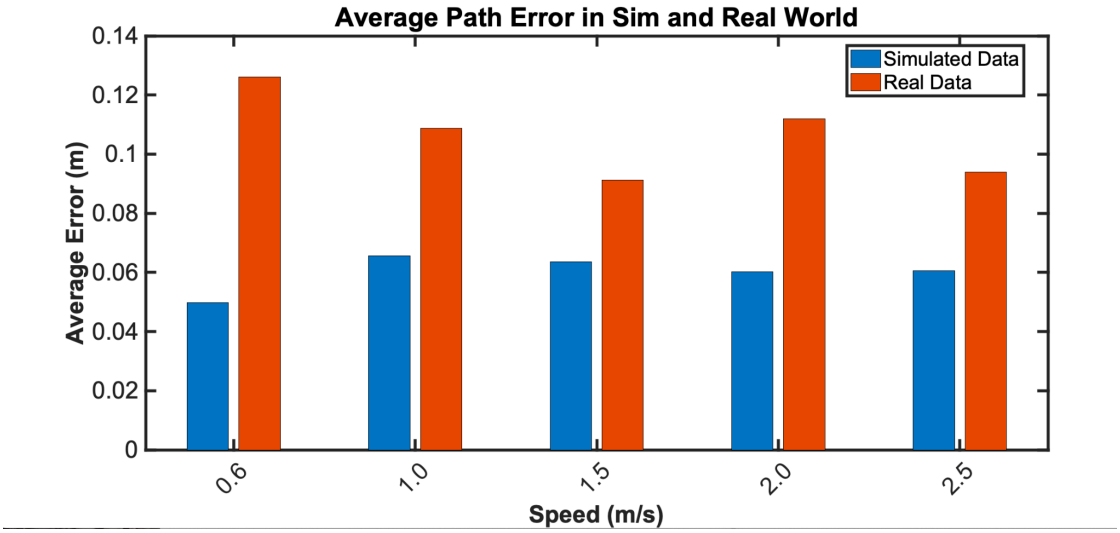
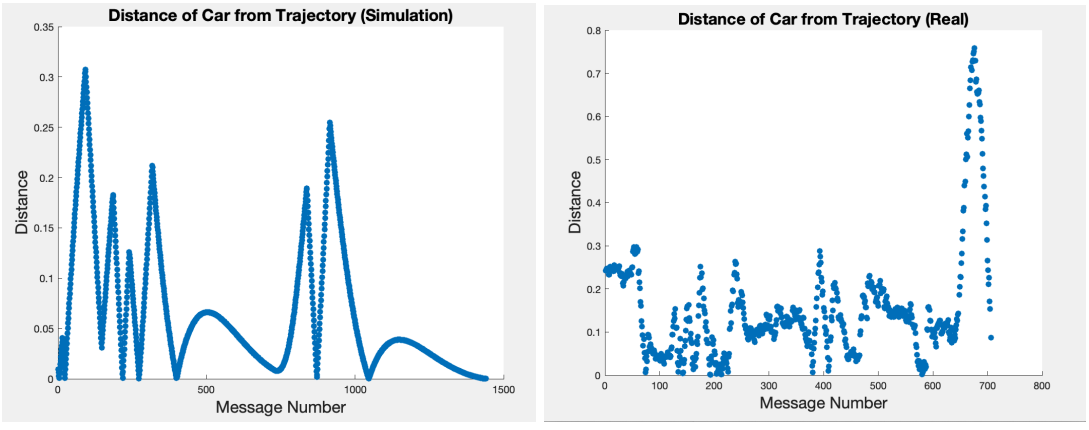
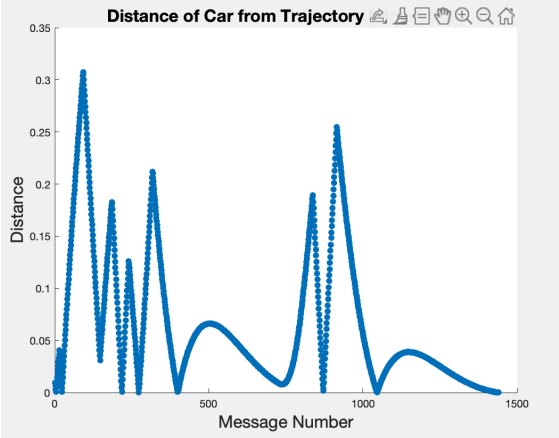


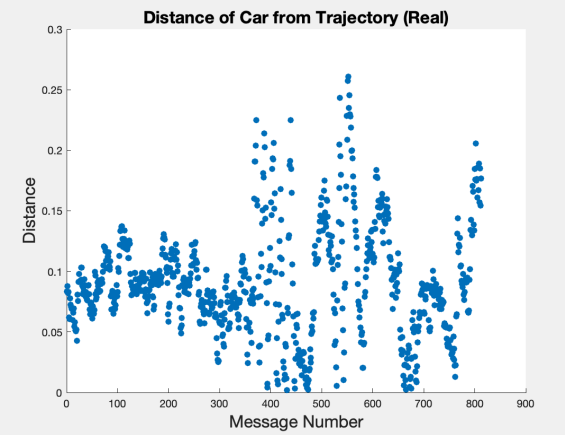
Figure 17: Plotting the average error of simulated runs and real-world runs.

Furthermore, we observed that the errors of the car from the actual trajectory followed a similar pattern for many trajectories, which correlates with a relatively accurate path follower that may deviate from certain trajectories on a path-by-path basis. Our levels of error are plotted in Figures C-J.

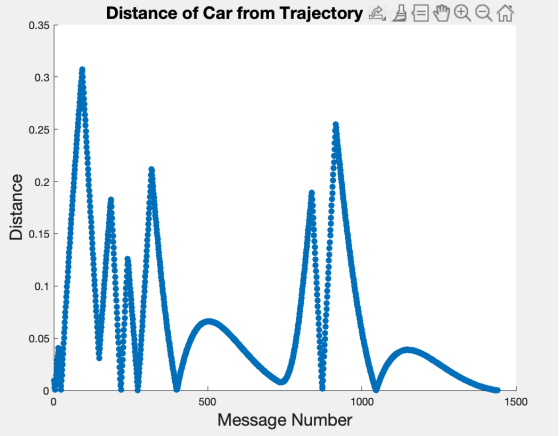


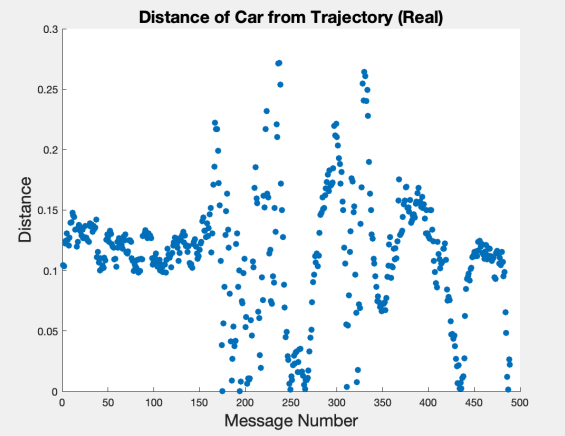
Figures 18 & 19: Errors of car driving at 1.0 m/s in simulation and in the real world.



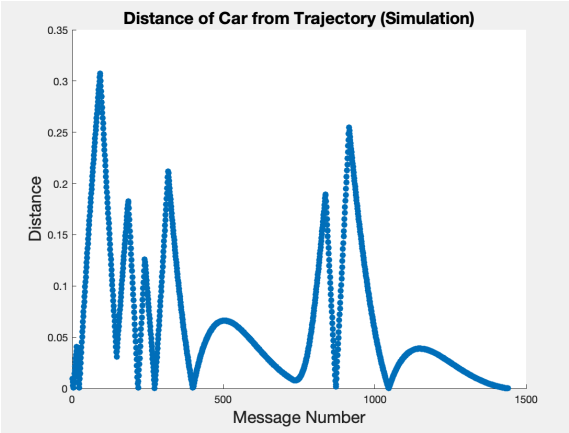


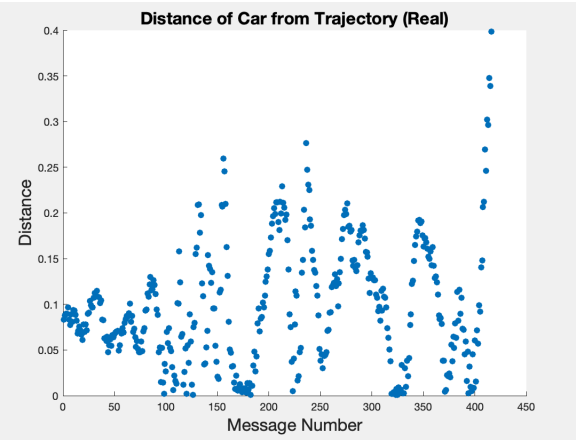
Figures 20 & 21: Errors of car driving at 1.5 m/s in simulation and in the real world.





Figures 22 & 23: Errors of car driving at 2.0 m/s in simulation and in the real world.





Figures 24 & 25: Errors of car driving at 2.5 m/s in simulation and in the real world.

Simulated runs tended to get much similar results, due to the absence of sound from the outside environment. Meanwhile, the points where the real run saw different patterns in spiking of error between different speeds, implying that some factors were not consistent across real-world runs. This could include noise from the outside environment and timing, as we were working near several other teams in the audience and the amount of time we spent on each real-world run varied; the spikes on the right side of the graphs could be representative of there being no lookahead to look for after reaching to end.

4: Conclusion and Lessons Learned [Dora, Eric]

With respect to the technical aspects of this lab, we learned to anticipate unexpected failures, especially during the integration stage. After seeing many teams run their robots at high velocities and sometimes crash, we took extra precautions in our approach. We edited our code to dilate the map so our path planner would not generate corner-clipping paths, and we ran our robot at very slow speeds to start. We also ran our safety controller for the slower speeds until we were confident with our path following abilities. Additionally, although our path planner and path follower functioned effectively in simulations, extensive debugging was required to achieve similar success on the actual vehicle. This was primarily due to mis-published topics and loading the incorrect launch files on the vehicle. In the future, it would be wise to be more meticulous regarding our testing methods to ensure our implementation performs as anticipated.

This lab was challenging in that our attempts to try different path planners resulted in most team members doing much individual work and said work took longer than expected to complete, setting back our general timeline for completing different modules of this lab. In the final stage, integration, our whole team was present to troubleshoot and evaluate the performance of our car. This experience enabled us to keep working for longer stretches of time by taking

turns contributing ideas and streamlining our testing process. In the future, we hope to replicate this experience while starting the final challenge even earlier to give ourselves more time to fine-tune our car's performance.

5: Citations

[1] J. Dai et al., "Autonomous Navigation of Robots Based on the Improved Informed-RRT∗ Algorithm and DWA," *Journal of Robotics*, vol. 2022, Feb. 2022, doi:10.1155/2022/3477265.

[2] "Find the shortest distance between a point and line segments (not line)," Stack Overflow, https://stackoverflow.com/a/58781995 (accessed Apr. 26, 2024).

[3] G. Rees. "Line segment to circle collision algorithm." Stack Overflow. https://codereview.stackexchange.com/a/86428 (accessed Apr. 26, 2024).

[4] L. Carlone. (2024). Lecture 5-6: Embedded Control Systems [PDF document].