Lab 6 Report: Path Planning using A^* and Pure Pursuit Control

Team 8

Xavier Bell
Nicolaniello Buono
Mary Foxen
Elise Harvey
Eli Scharf

6.4200

April 27, 2024

Contents

1 Introduction

(Mary)

In this lab, we implement planning and control, two main components of autonomous systems, to allow our robotic car to navigate autonomously throughout the Stata basement from any start to end location. Planning is essential in creating a path from a vehicle's start point to goal destination, and a control algorithm is necessary to follow a desired trajectory. Designing more optimal planning and control methods is essential to advance autonomous systems. By integrating these core components of autonomy with our localization algorithm, we can deploy this system on our physical race car to successfully follow a desired trajectory that reaches our desired goal destination. Our autonomous race car used Light Detection and Ranging (LiDAR) data to localize its position on a map of Stata's basement. After knowing its position, the robot could plan a path from its current location to any desired location on the map. This can be seen in Fig. 1. Path planning is a difficult task since it requires a robust localizer. Path planning also requires an algorithm that efficiently plans optimal paths without collisions. Each small corner must be accounted for in the path planning algorithm. Successful path-planning algorithms offer a great deal of reward. Path planning has many applications in autonomous vehicles. Further, path planning has applications in most robotics areas. Any robot that has nondeterministic motion requires a path-planning algorithm.

Figure 1: Example of path planned by the race car. Using Pure Pursuit, the race car followed the given path to the destination.

For this lab, an occupancy map of the Stata basement and a LiDAR system provided information to help our vehicle navigate from a start to end point quickly and efficiently without any obstacle collisions. Finding a path on a map, let alone an optimal path, is difficult for computer systems. Different algorithms' may be considered to solve this path finding problem. Some algorithms have a longer runtime but always return an optimal path. Other algorithms prioritize a faster runtime at the expense of a non-optimal path. Some algorithms integrate over all possible paths, such as Breadth First Search (BFS) and A-star (A*). These algorithms are called search-based since they iterate and search

the map space deterministically. Other approaches to solving the path-finding problem are called sample-based algorithms. Sample-based algorithms search the map spaces randomly. These approaches build trees throughout the map of possible paths. To determine the most optimal planning method, we designed and tested both a search-based planning algorithm, A*, and a sampling-based planning algorithm, rapidly-exploring random trees (RRT). After testing our algorithms' runtime and average path distance generated, we found A* to be the most optimal.

In order to accurately follow our desired trajectories, we developed and tested a pure pursuit control algorithm. This algorithm allows us to follow a path given in the map frame. At a high level, this is done by selecting a reference point on the path a specific distance away from the car (the "lookahead" distance). Pure pursuit then calculates the needed changes the car needs to make to drive to that point.

Upon designing and testing our planning and control algorithms, we integrated these components with our localization algorithm to allow our autonomous vehicle to create and follow an optimal path in the real world, Stata basement, in real-time.

2 Technical Approach

(Eli)
Before building an algorithm, we preprocessed our graph. The map of Stata's basement was given to us as an OccupancyGrid. The grid was a 2-dimensional array with values ranging from -1 to 100. For a point, -1 signified uncertainty about whether an obstacle (mainly a wall) was present. The values 0 to 100 represented the percent certain that a specific pixel (or point) was occupied. We built a simplified version of the OccupancyGrid where any value between 0 and 10, inclusive, was marked drivable terrain. Any other value was marked as an obstacle. In the 2-dimensional array, our group built, 0 signaled drivable terrain while 1 meant an obstacle.

One challenge faced was ensuring the autonomous car does not drive too close to walls. Algorithms find paths on a pixel-by-pixel basis. This means an optimal path might only be a few centimeters away from a wall. This path would not leave enough room for the autonomous car to clear the wall. To prevent this issue, we dilated the walls. Dilating the walls created a buffer zone between the real wall and the wall the path-planning algorithm used. To dilate we applied a kernel that marked the current location as occupied if there was an occupied pixel with a certain distance. An example of the dilation kernel can be seen in Fig. 2. The buffer allowed the path-finding algorithm to get close to the wall in finding an optimal path. However, the real path stayed several inches away

from the real wall and allowed the race car to maneuver without hitting the wall.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 2: Example of a 5 by 5 convolution kernel. If there is an obstacle above, below, to the left or right of the center pixel, the center pixel will be marked as one. Marking the pixel as one denotes that location as occupied.

Another challenge our group faced was converting from pixel coordinates (the coordinates of the OccupancyGrid) to the map coordinates that the autonomous car could use to navigate. For this conversion, our group was given the transnational offset (t), the rotational offset of the pixel coordinate frame and map frame (θ), and the scaling factor (s). To convert from pixel space to map space we used the equation as seen in Fig. 3. To convert from map space to pixel space, we calculated the inverse of the matrices and multiplied them by the map coordinates. This solved for pixel coordinates. These equations allowed us to transition from a start coordinate in the map frame to pixel coordinates.

$$\begin{pmatrix} m_x \\ m_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

Figure 3: Matrix transformation used to convert from pixel x,y coordinates to map x,y coordinates.

2.1 Sampling-based Approach

(Eli)
The first approach our team implemented to solve the path-finding problem was a sampling-based approach. Sampling algorithms work incredibly fast and perform faster than search-based algorithms. Our group implemented the RRT algorithm. Our choice to implement RRT stemmed from RRT's high efficiency in higher dimension sample spaces. RRT is also probabilistic complete, meaning the algorithm guarantees a solution as the number of samples increases and approaches infinity. This offers comfort for an algorithm navigating Stata's complex basement. RRT's randomness also makes it robust to noisy and uncertain

data. Finally, RRT is simple to implement.

RRT works by building a tree of possible paths from the start location until it reaches the end location. As seen in Fig. 4, the tree grows outward from the start (red) towards the end (green) as more points are sampled and added to the graph.

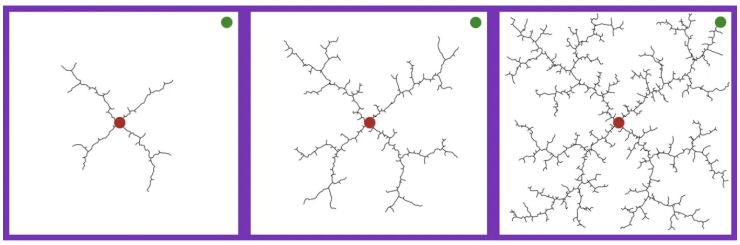

Figure 4: **Example of RRT algorithm expansion.** The tree grows larger after more points are sampled and added to the graph. Source: https://towardsdatascience.com/how-does-a-robot-plan-a-path-in-its-environment-b8e9519c738b

RRT's implementation is straightforward. The start point is set on the graph. A point is randomly sampled from anywhere in the map space. On the current RRT graph, the closest point to the randomly sampled point is calculated. From the closest point on the graph, another node is added to the graph in the direction of the randomly sampled point. The new node is added a step size distance away. We defined our step size as 20 pixels. We chose 20 because it allowed our algorithm to explore the map space quickly. Smaller step sizes took too long to explore the map and sometimes would not return an answer. For larger step sizes, the algorithm was unable to non-collision paths. Our group converged on 20 after comparing the runtime of the algorithm with the different paths found.

After adding a node to the graph a specific step size away, the algorithm checked for collisions. If the new node was in an occupied location, it was discarded. The algorithm also checked if a straight line could be drawn from the new node to the closest node in the current tree. Points were sampled on the line to check collisions. If any of these points landed on an occupied point, the algorithm discarded the new node. The number of points sampled scaled with the size of the line. The new point was added to the graph if the new node did not collide with an obstacle and a collision-free line existed. The closest point was set as the parent of the new node. If a new point was added, the algorithm determined if a non-collision line could be drawn from the new node to the goal point. If so, the algorithm returns the path. RRT backtracks at each node and uses the parent nodes to construct the full path. Using this implementation, RRT finds paths through complex spaces. However, these routes are not optimal. As seen

in Fig. 5, the route has unnecessary bends. Overall, the algorithm returns paths quickly even in complex spaces. The non-optimal path comes at the expense of RRT being an efficient algorithm.

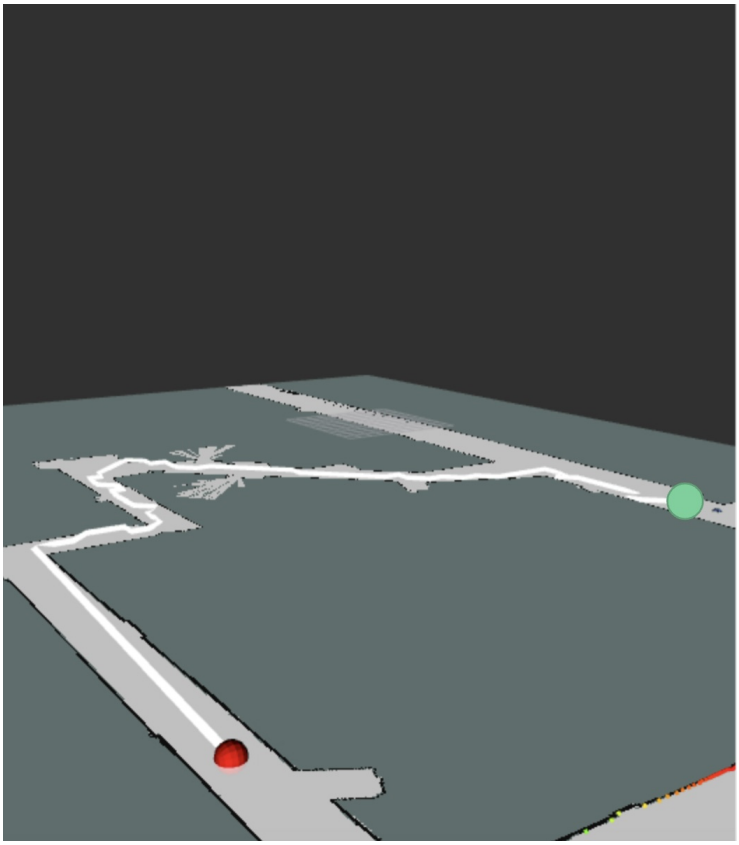

Figure 5: **RRT algorithm successfully finds a path between the selected start and end points in the simulation.** There are kinks in the path found. This results from the fact that RRT is not an optimal path-finding algorithm.

2.2 Search-based Approach

(Xavier)
Search-based algorithms are useful for their ability to return optimal–or very close to optimal–routes at the expense of runtime. Search-based paths should also be smooth and suitable for the robot to follow. We examined two different search algorithms–one a natural extension of the other.

Our group's first implementation of a search-based algorithm was Breadth First Search (BFS). BFS returns an optimal route and its implementation is relatively

simple but heavily suffers on runtime on larger grid sizes. BFS is initialized with a starting point and a queue of the nodes to visit. Then, while the queue is not empty, BFS will dequeue a node and mark it as visited. This node is examined to determine whether it corresponds to the designated goal point. If not, all unvisited neighboring nodes are added to the queue. This can be thought of as exploring outwards, as seen in Fig. 6. An empty queue means all reachable nodes have been visited. If the queue empties before the goal node is found, no path exists. Upon termination, if an optimal path exists, BFS has found it. BFS explores the entire grid thus it knows the shortest path to the goal node. A dictionary stores each node and its parent node. BFS simply backtracks using this parent dictionary to return the optimal path. BFS returns an optimal path, but it has struggles with runtime on large grids, compared to RRT, as seen in Table 1.

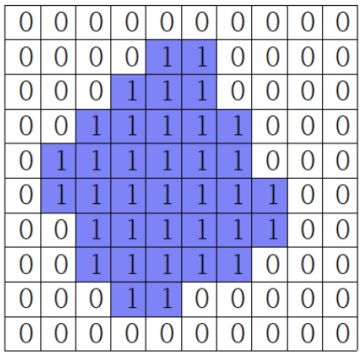

Figure 6: **BFS visualization.** In this example, BFS starts at the center and explores neighboring nodes and moves outwards. BFS marks visited nodes with a 1. This approach allows BFS to ensure optimal paths are found. Source: https://stackoverflow.com/questions/55416057

Our group improved on BFS by implementing the A-star (A*) algorithm. A* cuts down on runtime by avoiding searching the entire grid. When humans are asked to find a path from point A to point B, we unknowingly use many heuristics in our search. Similarly, computers can utilize heuristics to find a path efficiently. A* uses BFS as the groundwork, but will remove cells from the queue based on the estimated cost to the goal. The actual distance to the goal cannot be truly known during the path planning because any number of obstacles could occlude the path. The distance to the goal is approximated using a heuristic to provide information about the most optimal next step. A* depends on a strong heuristic to guarantee an optimal path.

The heuristic used depends on the method of traversal through the grid. If only the left, right, up, and down cells are accessible then the heuristic used

to estimate distance from the goal is the Manhattan distance. The Euclidean distance is used for motion in any direction. In our case, we can traverse to all eight adjacent cells and must use the Diagonal distance. Diagonal distance is found as the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively.

A* struggles with large map sizes. Any search algorithm must search at least the immediately adjacent square in order to progress. The complexity of the search is at worst $O(N^2)$. Higher resolution maps must be down-sampled for reasonable running time. Fig. 7 shows the performance enhancement from running A* on a down-sampled map.

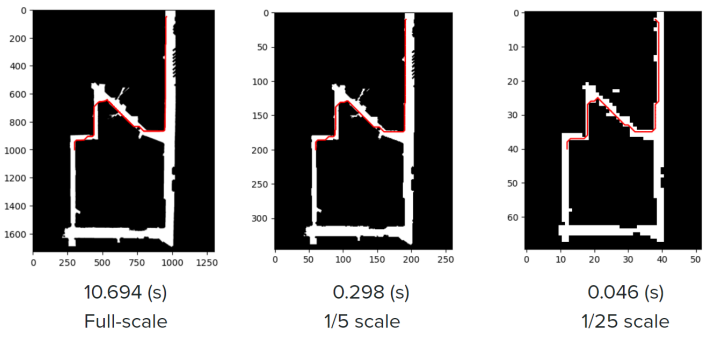

Figure 7: **A* greatly benefits from a smaller map size.** Down-sampling can provide this smaller map size, but care must be taken to not remove obstacles or obstruct tight passageways

Importantly, the map must not be down-sampled to a degree that removes obstacles or obstructs tight passageways. On the final implementation, our map was down-sampled to half size as runtime was still fast enough to not be cumbersome. Furthermore, path creation is a one-time event and accuracy was emphasized over speed.

2.3 Pure Pursuit Control

(Elise and Nico)
Considering the prevalence of nontrivial trajectories as a result of a complex environment, pure pursuit is naturally suited to to follow a path–particularly one that is represented as piece wise line segments–by utilizing look-ahead. While other controllers may be robust as well, an implementation such as PID control may require interpolation in conjunction with potentially extensive edge case handling for traversing between line segments. Pure pursuit naturally interpolates between line segments and has few parameters to tune, making pure pursuit a simple and robust choice for this use-case. An illustrated overview of pure pursuit is shown in Fig. 8.

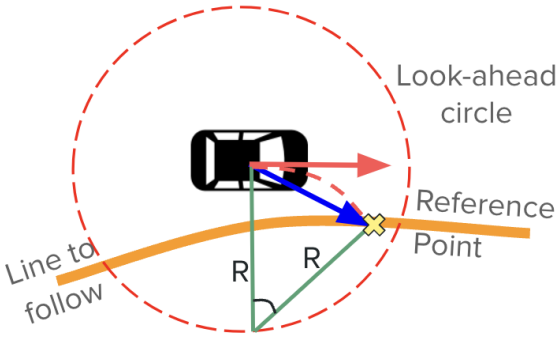

Figure 8: **High-level implementation of pure pursuit**. Pure Pursuit follows a point a set distance away from the car. The reference point is updated as the car moves forward and allows the car to follow a path.

To follow a given trajectory, we created lines connecting the points published by the path planner. This creates a path that pure pursuit can follow. Using some vector calculus, we calculate the desired point to target along the path.

In considering a triangle where the hypotenuse is the look-ahead distance, the known leg of the triangle is the vector from the car to the closest point on the current line segment being followed, and the leg of unknown length is co linear with the line segment being following. We solved for the unknown leg of the triangle to get a vector $\vec{T}$ in the world frame as shown in Fig. 9.

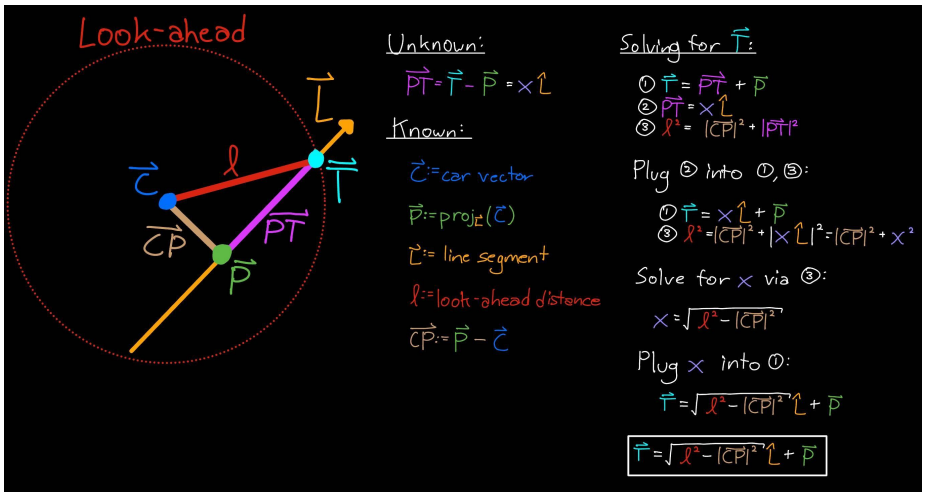

Figure 9: **Math to find target point $\vec{T}$ at intersection of look-ahead and a line-segment.** By projecting the car onto the line, we can create a triangle with two known sides and whose unconstrained vertex is the target point $\vec{T}$. By solving for the length of the unknown side corresponding to $\vec{PT}$, we find $\vec{T}$ in the world frame.

After we find $\vec{T}$ in the world frame, we used a transform to get $\vec{T}$ from the world frame to the car frame to pass to our pure pursuit controller.

In addition to the math we used to calculate our reference point, we needed to handle several edge cases. First, we handle the case when the start point of a trajectory was not where the car's position is. If the start point was in front of the car, we created a dummy segment that connected the car's current location to the start point. This extends the path we created such that the car can still follow. On the other hand, if the start point was behind the car, we prioritized the next closest point on the trajectory that was in front of the car. This allowed us to avoid the car spinning to try and find the start of the path.

Next, we had to decide how to handle multiple intersections of the path with the look-ahead circle. Similar to the above, we prioritized points earlier in the path. We made this choice to ensure we did not skip parts of the planned path in the event that the path avoids an obstacle.

The last major edge case we handled was when the car's look-ahead circle does not intersect with the current line segment. In this case we make our target point a distance of 1 meter along the line segment from the projection of the car onto the line segment as shown in Fig. 10.

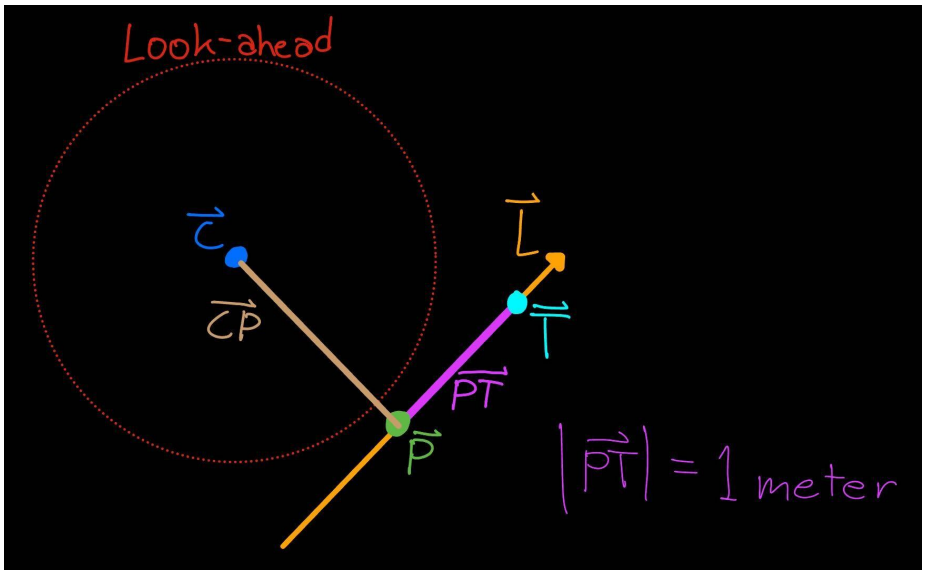

Figure 10: Handling of edge case when look-ahead doesn't intersect the line segment. When there are no line segments inside of the look-ahead distance of the car, we make out target 1 meter along the direction of the line segment from the point P, the projection of the car C onto the line segment L.

While this solution is likely not rigorously robust to challenging situations, it was a robust solution for our use case, given that our car was speed-limited and that the environment wasn't exceedingly challenging for cornering or localization.

2.4 Integration

(Elise)
The goal of this lab was to plan a path between two points in the Stata basement and then have the car drive along it. In Fig. 11, we see successful planning and following of a trajectory done in simulation.

Figure 11: Car path-finding with A* and following in simulation. The algorithm returns an optimal path, given the heuristics. Compared to RRT, the algorithm has few unnecessary turns.

For testing in the real world, we needed to use our localization algorithm from lab 5 in addition to the tools described above. Our localization algorithm estimates the physical location of the car using LiDAR, probability, and odometry updates. Thus, our three main parts to integrate are (1) the path planning algorithm, (2) our trajectory follower (pure pursuit), and (3) our localization algorithm. First, we plan our desired path using the path planning algorithm. Next, we need to run our trajectory follower. We used our localization algorithm to estimate the car's location. This location is then passed into our pure pursuit implementation so the car will adjust its movements to follow the path. After implementing these changes, the car worked in the real-world, performing with the desired behavior seen in simulation with the exception of more noise in the sensor data and localization as a result. Fig. 12 shows a side-by-side of the car autonomously navigating and the path it is following.

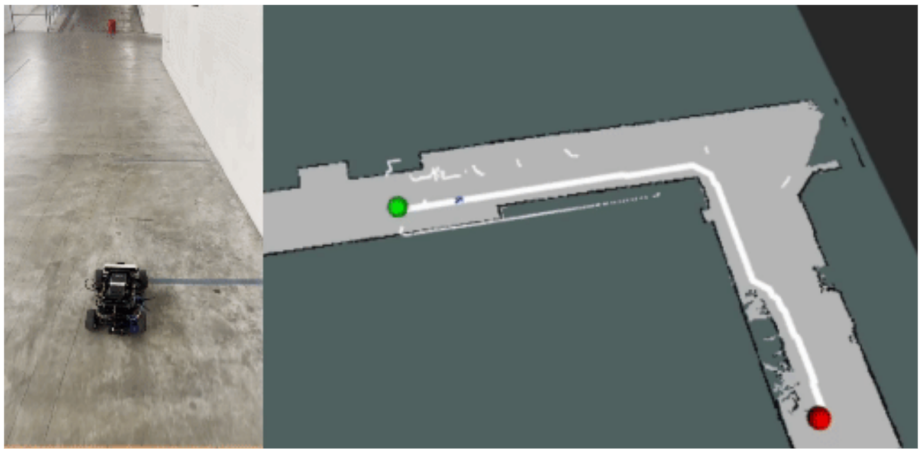

Figure 12: Car path-finding with A* and following in the real-world Stata basement. The race car is able to localize itself in the basement and follow a path from one point to another without issue.

3 Experimental Evaluation

RRT is a probabilistic algorithm. Measuring its error is difficult. In section 3.1 we explore RRT's error by comparing the output of RRT to the respective optimal paths. A* and BFS are both algorithms which return the optimal path. In this way, error is approximately zero for both and we could not perform the same analysis that we did for RRT. To compare the algorithms we tested each algorithm in real time and reported the time it took to calculate a path. This is further discussed in Path Finding Analysis.

3.1 RRT Error

(Eli)
RRT is a randomized algorithm. Quantitatively measuring its error is thus difficult. RRT returns different paths every time. To understand RRT, we use two paths calculated by RRT and their respective optimal paths. In Fig. 13, the path found by RRT is approximately 3 meters longer than the optimal path. This equals a 9.5 percent increase in path distance. For longer paths, such as the one in Fig. 14, the path RRT returns is approximately 14 meters longer than the optimal path. This correlates with a 16.4 percent increase in path length. Due to the nonoptimal nature of the paths returned by RRT, our group implemented A* and used A* to plan a path for the autonomous race car.

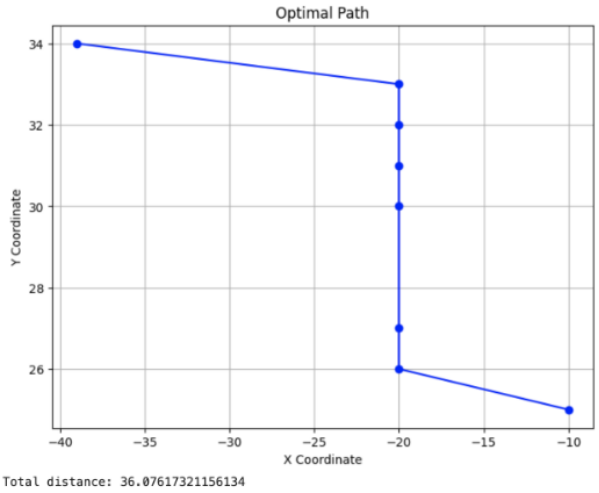

Figure 13: **RRT performance vs. optimal path over mid-distance.** Optimal path is on top. RRT returned path below. The difference in distance between the optimal and RRT path is approximately 3 meters.

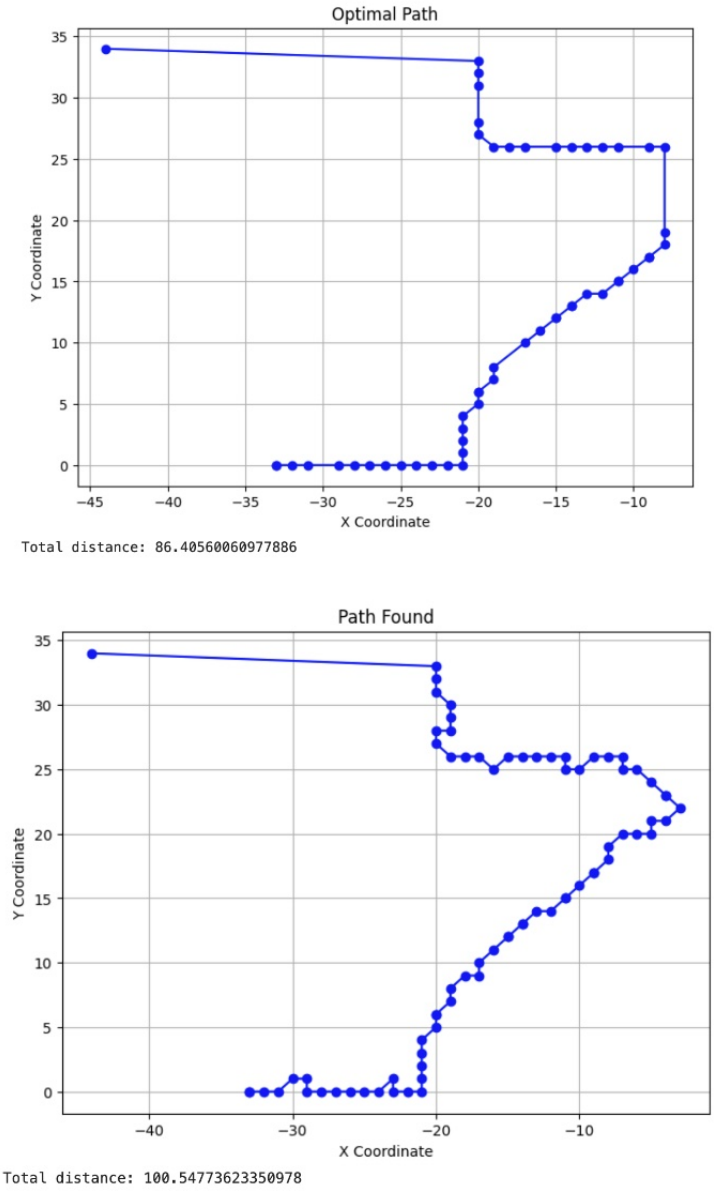

Figure 14: **RRT performance vs. optimal path over longer distance.** Optimal path is on top. RRT returned path below. The difference in distance between the optimal and RRT path is approximately 14 meters.

3.2 A* Error

(Nico)
A* can guarantee optimal paths if the heuristic used never overestimates the cost. For our use-case, the diagonal-distance heuristic with the parameters we chose can not possibly overestimate the cost. Therefore, we can guarantee that our A* implementation finds an optimal path in the maps we pass to it.

3.3 Pure Pursuit Error

(Nico)
Due to the nature of selecting a point with look-ahead, the car will effectively create a radius between the intersections of the piece-wise line segments. Since our car does not have full control authority over three degrees of freedom, the car is guaranteed to deviate from the piece-wise trajectory lines regardless of control approach for trajectories whose points are not co-linear. Additionally, any errors in localization will exacerbate error from these phenomena. We decided that it was not feasible to generate accurate error of pure pursuit on the car and resorted to qualitative metrics instead. These included seeing how many points the car visited in the trajectory and noting when the car strays beyond the look-ahead distance of the trajectory.

The pure pursuit implementation was robust and nearly always visits all points along a trajectory in addition to rarely deviating from the path beyond the look-ahead distance.

3.4 Path Finding Analysis

(Xavier)
Initially, our group implemented three path-finding algorithms: BFS, A* and RRT. As seen in Table 1 each algorithm has its strengths and weaknesses in respect to worst-case runtime and optimality. BFS has a worst-case runtime of N^2 and this generally cannot be improved. A* also has a worst-case runtime of N^2. However, the heuristics in A* make it a faster algorithm in most scenarios. RRT has the best runtime of $Nlog(N)$.

Algorithm	**Runtime**	**Optimality**
BFS	$O(N^2)$	Finds optimal path
A*	$O(N^2)$	Not always optimal, but gets close
RRT	O(NlogN)	Not Optimal

Table 1: Comparison of Algorithm Performance. RRT has the best runtime, but this comes at the expense of non-optimal paths returned. Our group used A* as our final implementation because heuristics decrease the runtime and A* returns mostly optimal paths.

Our team opted to use A* as our final implementation since it generally found an optimal path and had a comparable runtime to RRT. Fig. 15 showcases that the two algorithms' actual runtimes only differed by a matter of seconds in testing. Since our grids were not too large, A* returned quickly.

	10 x 10 Grid	100 x 100 Grid	500 x 500 Grid
A*	0.0010 (s)	0.101 (s)	3.557 (s)
RRT	0.0102 (s)	0.0373 (s)	0.0885 (s)

Figure 15: RRT performance vs. A* performance. Time (seconds) to return path based on given grid size. The performance of both algorithms only differed by seconds for our grid sizes.

4 Conclusion

(Eli)
The path-planning lab allowed our autonomous race car to navigate Stata's basement. The only information our race car had was a static map of Stata's basement. With that information, our autonomous race car could navigate between any two points in Stata's basement without human input. The team performed well. Our path-finding algorithm found optimal paths anywhere in the basement. The algorithm returned a path in a matter of seconds.

While completing the lab, our team faced the challenge of finding a path algorithm that returned an optimal path efficiently. Our group used A* as the path-planning algorithm. A* does not work as efficiently as RRT. Our group will further investigate efficient algorithms that find optimal paths. Our group will research RRT Star (RRT*). RRT* uses the same principles of RRT for path-finding. However, RRT* updates the shortest path after adding each new node. Updating path distances enables RRT* to return closer to optimal paths.

Our group will also further investigate our localization particle filter at higher speeds. At higher speeds, the localization algorithm needs to work faster. The algorithm has less time at each location to predict its location. Our group will further research possible heuristics to be added to the localization algorithm. The group will also investigate fine-tuning the localization algorithm's parameters for higher speeds. These improvements will help our team succeed in the final challenge.

5 Lessons Learned

(Xavier)
In this lab, I continued to refine my CI skills in planning, communicating and presenting. We set early deadlines knowing that we would need more time. While we were very close to not finishing on time, we finished all required components of the lab. I also learned to better plan my time to contribute to the project. I also continued to make graphics for the briefing but only contributed my relative portion of this report. In the future, I wish to be more active in the early stages of report/briefing creation.

With regards to the technical aspect, I lead the development of the A* algorithm. I also debugged the simulation to robot pipeline and updated our particle filter. I had a good time learning A* because I have heard a lot about it but never got the chance to learn before this lab. I will continue to learn as much as I can about the car so that debugging takes less time and I can be more helpful to my teammates.

(Nico)
For the CI aspect, I learned to more effectively distribute tasks at early dates to eliminate blockers and allow additional time for debugging. With this approach I think our CI components are also more robust.

With regard to the technical component, I worked on A* and pure pursuit control. I also learned to start a bit earlier than I had in previous labs. For A*, I helped debug the implementation and then created a wrapper for the implementation to use in our ROS2 package in addition to adjusting parameters (like map buffering/dilation) to get planning working better for our use case. For pure pursuit control I helped implement the pure pursuit controller itself in addition to target point selection and its various edge cases. When integrating all of these parts on the car, I helped with debugging, but am grateful I learned to better communicate my progress and blockers so we can work more efficiently as a team. Going forward I want to be more intentional with starting earlier to identify limitations of implementations and how they function in the real-world. I'm eager to refine the work we've done here and add more agency and robustness to our car's decision making.

(Mary)
In this lab, I improved my CI skills in learning to plan early and communicate more effectively with my teammates. Directly after completing our lab 5 briefing, we took the time to review the lab 6 requirements and divide work based on our personal preferences. After a week and a half, we completed our first set of tasks. We used the remaining part of our second week to debug and optimize our code. As a result, we left just enough time to integrate path planning and line following with our particle filter on the physical car. In the future, I hope to leave my schedule more open in the days before our final briefing in order to

be more available to help out with last minute debugging on the car.

After questions were raised about optimality and runtime during our briefing, I learned that I am also interested in testing and refining BFS further in simulation and on the car. Because we down-sampled the map of the Stata basement, our search algorithms were able to find an optimal path much more quickly. In the time frame of this lab, we chose to modify the search-based method A* instead of BFS. However, BFS may also provide an optimal solution and a faster runtime if investigated further. For the final challenge, we will have to work on quickly generating an optimal path, and using BFS may allow us to decrease this portion of our algorithm's runtime. I look forward to improving our algorithms in the final challenge as well as integrating new machine learning algorithms to create a faster and more successful autonomous vehicle.

(Elise)
For the CI component, I learned that we should set ambitious deadlines and have a set buffer time. For example, we set an early deadline of having parts A and B of the lab done by Wednesday evening. We did not finish them until the following Monday. Our larger buffer time allowed us to debug and optimize the code we had written.

For the technical component, I specifically worked on the pure pursuit control. My work included helping find out how to connect and follow points on the trajectory and handling edge cases. I also helped to debug with others! I think debugging is a skill that I will use a lot moving forward. I also have improved on understanding the bigger picture in terms of goals and integration of the moving parts.

(Eli)
For the technical component, I worked on implementing RRT and BFS. From the last lab, I learned that I needed to start early. After building the algorithms, I tested our them and found that they worked well on smaller and medium-sized inputs. However, our group faced unforeseen bugs when implementing the algorithm in simulation. To debug these errors our group did a great of testing. To make testing easier, I make an interactive Python notebook. The notebook allowed the team to test and change our code without changing any code on the race car. The interactive notebook also made debugging easier as print statements could be easily logged.

For the CI component, I learned the importance of completing slides early. Due to my observance of Passover, I needed to prerecord my slide presentation. This meant my slides needed to be completed earlier that the team's slides. However, I found that making slides soon after I implemented the algorithm worked well for me. It was easy to make the slides with the work fresh in my mind.