Lab Report: Path Planning

Team 21

May Huang
Mohammadou Gningue
Haley Sanchez
Nikhil Kakarla
Neil Chowdhury

Robotics: Science and Systems

July 3, 2024

1 Introduction

1.1 Lab Objective

Trajectory planning is critical for robots to properly and safely navigate various environments. Our team's objective for this lab was to implement planning and control algorithms and allow our robot to drive autonomously around the Stata basement, navigating at various speeds and around obstacles. Given a known map and a goal pose, our robot plans trajectories using a search-based algorithm, A*. The robot then localizes its origin position and follows the planned trajectory using pure pursuit control, which finds and uses a lookahead point to compute steering angle and velocity at each timestamp and actuates these changes in motion repeatedly.

1.2 Defining Success

We defined success in this lab with various quantitative and qualitative metrics. For path finding, we observed in simulation that the robot successfully determined a viable path from its position on the map to a user-specified end goal. We also optimized for speed of computing paths in making design decisions on our path planning algorithm. In simulation, we watched for the robot to accurately follow pre-planned paths using the pure pursuit controller. In both simulation and implementation, we wanted minimal oscillation in the robot's motion, so we worked to tune this by visualizing drive angle against time. This contributes to desired accuracy for the robot controller as well, since excessive oscillation can lead to positional error. We were able to understand how well our robot was navigating by visualizing drive angle and angle error against time, and we used this data to inform design decisions for the pure pursuit controller.

2 Technical Approach

Our approach to autonomously moving between two points includes two parts: **path planning** to generate trajectories and **pure pursuit** to follow them. We initially developed these components separately in simulation, and then combined them on the robot.

2.1 Developing Path Planning

2.1.1 Choosing an Algorithm

There are a variety of algorithms to choose from in Path Planning, including:

- **Search-based**: A*, Dijkstra, BFS/DFS
- **Sampling-based**: Rapidly-exploring Random Trees (RRT), Probabilistic Roadmaps

We chose to use the **A* algorithm** (Figure 1) for path planning because it is more sophisticated than (indeed, a generalization of) Dijkstra's algorithm, and because Search-based algorithms are guaranteed to be more complete than sampling-based algorithms. We broke the map into a grid and explored it using A*, with Euclidean distance to the goal point as our heuristic. In A*, the priority of exploring a new point is the current cost to reach it plus the heuristic, so a lot of inefficient paths are not explored, increasing runtime efficiency.

2.1.2 Generating an Obstacle Map

To generate the obstacle map, we applied a morphological dilation with a disk of radius 5 to the original map and then downsampled it by a factor of 5, as shown in Figure 2. We found that this was sufficient to prevent the path planning algorithm cut corners or go to close to the walls. Additionally, the downsampling step decreased the theoretical runtime by a factor of up to 25, leading to a great efficiency boost.

2.1.3 Other Improvements

In our original implementation of A*, we simply explored the four cells horizontally and vertically adjacent to the given cell. We found that this led to jagged and inefficient pathing; the algorithm sometimes optimized more using Manhattan distance than Euclidean distance. To resolve this, we allowed our search algorithm to jump to the any of the 8 cells surrounding a given search

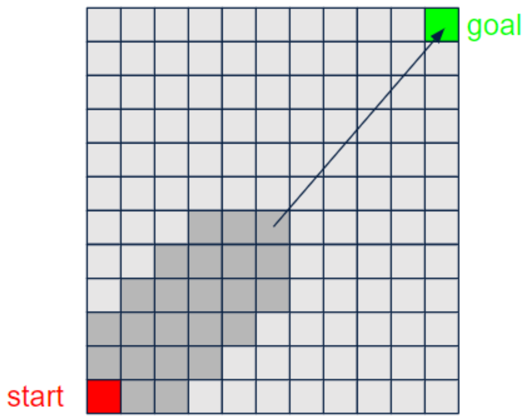

Figure 1: The A* algorithm: the gray squares show the growing search tree, while the line shows the heuristic used to guide the search.

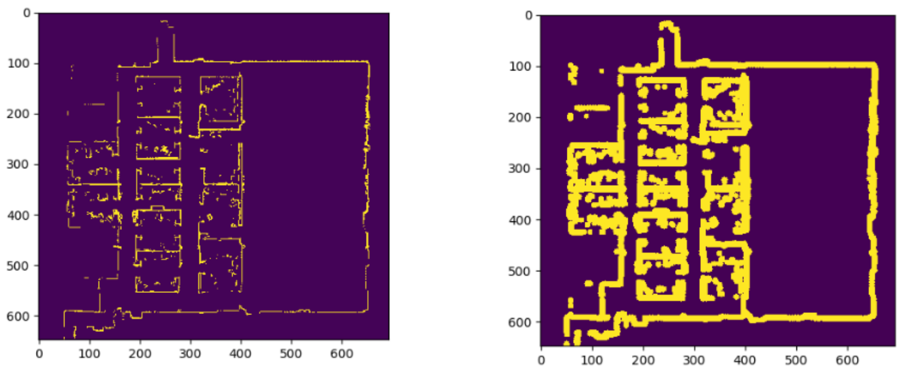

Figure 2: Morphological Dilation. Left: Original Map. Right: Dilated Map.

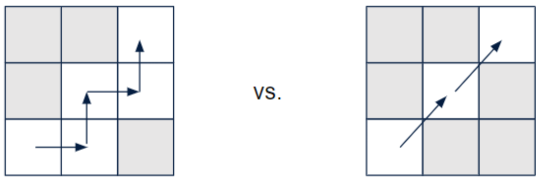

Figure 3: Improvements to pathing by exploring more neighbors. Left: original pathing, in which algorithm traverses only directly adjacent cells. Right: path found when algorithm traverses all surrounding cells.

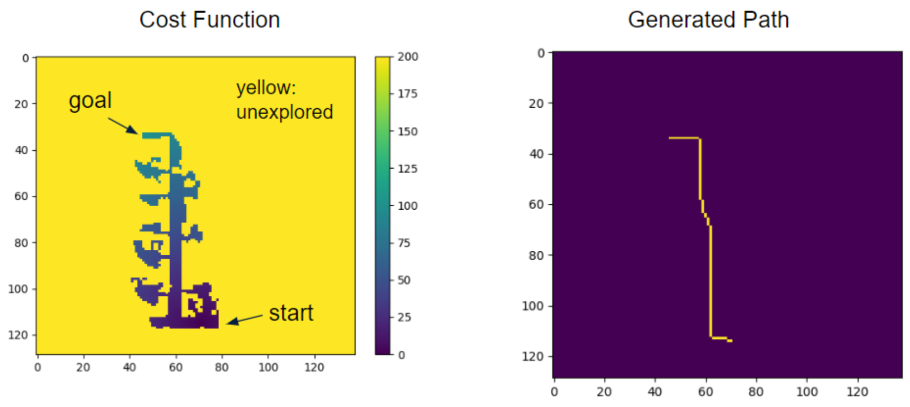

Figure 4: Example of a planned path. Left: Cost function from the start point. Right: Generated optimal path.

cell. Figure 3 shows that more linear paths can be generated with this optimization.

2.1.4 Evaluations

We found that random paths on the Stata basement were generated in 250 milliseconds or less in simulation, with shorter paths generated in around 70 milliseconds. While we were unable to test our pathing algorithm on the autograder, we found that the algorithm appeared to be finding near-optimal paths in all cases. Figure 12 shows an example of a generated path using our algorithm.

2.2 Developing Pure Pursuit

The next crucial part of our system was the Pure Pursuit algorithm. This method allows the robot to dynamically track a given path and adapt as conditions change. The implementation and design decisions we made are detailed below.

2.2.1 Algorithm

The basic pure pursuit algorithm is very similar to the previous labs. The robot picks a look ahead point on the given trajectory and uses trigonometry to calculate the desired drive angle and speed to efficiently reach that point. However, the complexity came in choosing the look ahead point. In order to accomplsih this, the robot first scans across all the line segments in the trajectory and calculates its distnace to each. Then, the robot takes the minimum of these values to determine which segment on the trajectory it is closest too. Then, the robot creates a circle of a specified radius around itself and finds where that circle intersects the line segments of the trajectories. This intersection represents a point on the trajectory that is a desired distance away from the robot and intersects the trajectory. The robot then uses this point to calculate a steering angle and navigate along the trajectory. The process repeats at every time step to allow the robot to navigate along the line dynamically.

2.2.2 Decision Choices

While implementing our pure pursuit algorithm, our team made a variety of key design decisions. The first was to use primarily numPy operations for our calculations. These optimized operations sped up calculation and allowed the robot to fully process the trajectory at each time step. Additionally, our team implemented a secondary safety controller. In addition to the standard safety controller on the car, our pure pursuit algorithm immediately stopped the robot if it veered too far from the desired trajectory. This functioned to stop the robot from potentially crashing if it veered from the trajectory. Also, our team decided to use a fixed look ahead distance. Although a dynamic look ahead distance that varied depending on the curvature of the path would have been beneficial, our team decided to use a fixed look ahead distance for simplicity and ease of tuning. Finally, our team implemented a steering control multiplier which dampened the oscillations of the robot. Both the fixed look ahead distance and steering control multiplier were tuned on the live robot. In this way, our team made a variety of core design decisions that impact the success of our Pure Pursuit algorithm.

2.2.3 Results

Our pure pursuit controller was able to properly follow the given staff path. Testing at speeds ranging from 1 m/s to 10 m/s, our robot in simulation was able to fully follow the staff path without crashing. This proved the effectiveness of the pure pursuit algorithm and was corroborated by our score of 98 percent on the gradescope auto grader.

2.3 Implementing in Simulation

2.3.1 Algorithm

Now that we had 2 functioning algorithms for path planning and pure pursuit, we needed to combine them to get the robot to follow the planned path in simulation. We needed to change what some of the subscribers and publishers from different nodes were using to ensure that they were able to share information between the two of the scripts. Additionally, we needed to deal with multiple ros files running at once. This required us to integrate the rostopic listeners and ensure that all topics made it to the destination nodes. Moreover, we integrated a backup safety controller that would stop the car if the car veered too far from the planned path. This worked in addition to our normal safety controller to ensure that the robot did not crash. However, after solving these problems, we were able to integrate the path planning and the pure pursuit controllers. This means that in simulation, we were able to click on a goal pose, the robot would plan its path to that pose, and then use its pure pursuit controller to reach the pose.

2.3.2 Experimentation

Once we got the 2 algorithms to work together we needed to visualize them working in simulation. Below in figure 5 is an example of a path being made with our path planning algorithm once we clicked on an end goal point. In this example we can see that the path does not run into any obstacles and that there is a buffer between the path and the other obstacles, including walls. This shows that our path is viable and would not lead to a collision.

Now that we have a path planned, we needed to use our pure pursuit algorithm to follow the path. In the figures 6, 7 and 8, we can see that the car does follow the white planned path accurately. Additionally, we can see 2 red arrows which represent the lookahead points that are used in order to correctly update the odometry data to follow the path.

In figure 9 the angle and drive angle errors are shown for the pure pursuit controller following the staff planned path. We did this to verify that the pure pursuit was working well. In the figure we can see that when angle error has a large spike, we see a corresponding dip in the absolute value of the drive angle. This shows that when there is a corner present, the angle error grows and the drive angle is changed in order to correct for this. This shows that our pure pursuit works effectively and is able to follow the path.

2.3.3 Autograder Results

We ran into many errors when checking and validating our results with the Gradescope auto-grader. One of the first things we realized was that our code was reading the values backwards for the path planning so we reformulated our

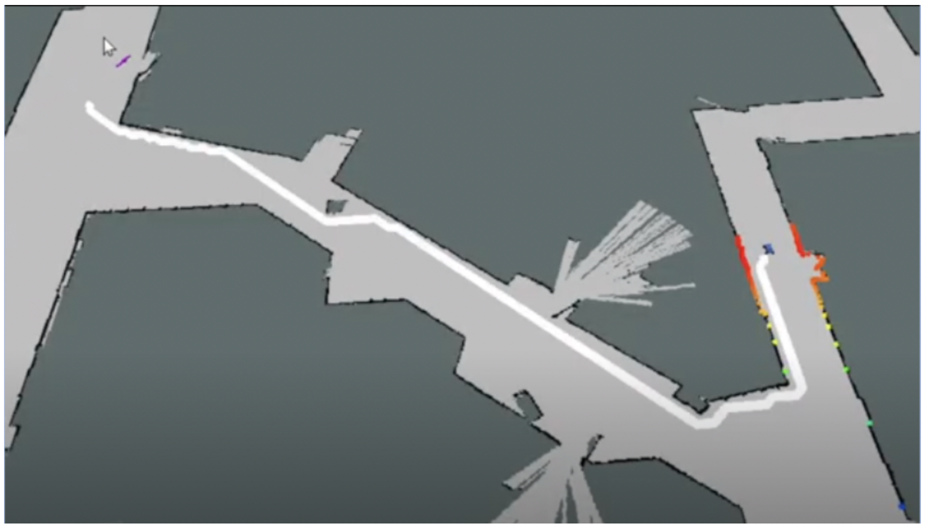

Figure 5: Example of a planned path when using the interactive goal point in Rviz

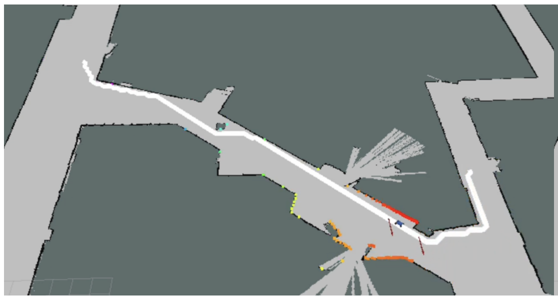

Figure 6: Following Path: Example 1

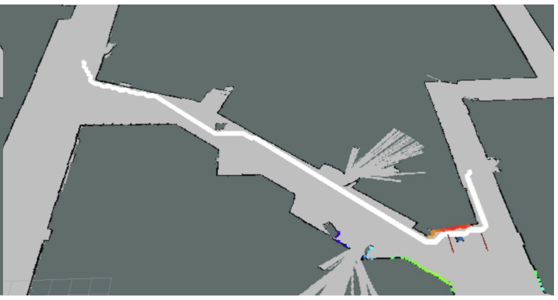

Figure 7: Following Path: Example 2

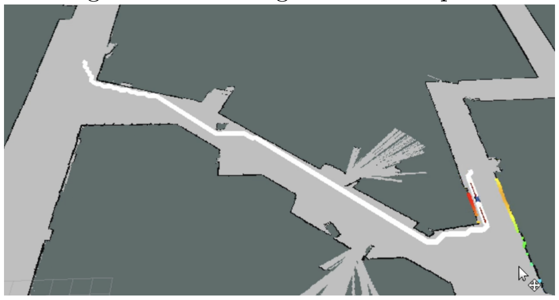

Figure 8: Following Path: Example 3

path planning in order to fix that and then tried to use the auto-grader again.

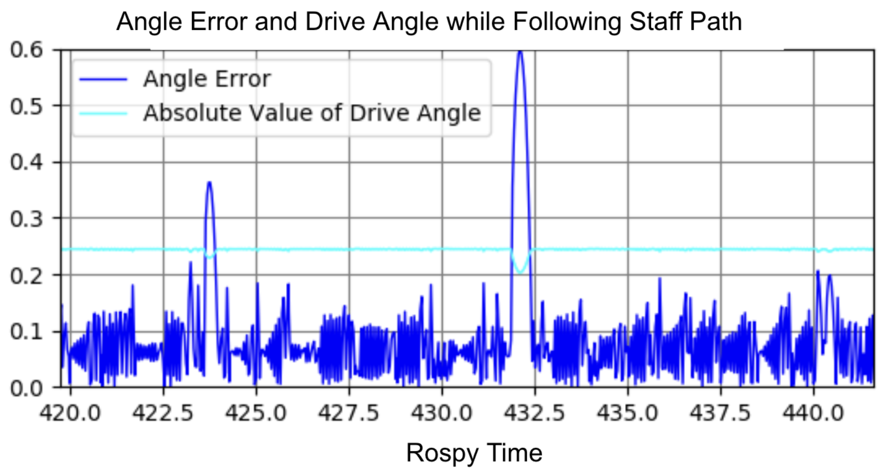

Figure 9: Absolute errors of our algorithm in simulation

We then ran into other errors that would not run our program at all and was not informative about why these errors was occurring. We then spoke with the TAs and found that many other teams were facing this same issue so we should focus on making our program run well on our physical robot. They also said that they will release information regarding manual grading. Therefore, we moved on to the physical robot after we verified that in simulation our algorithms were working well together.

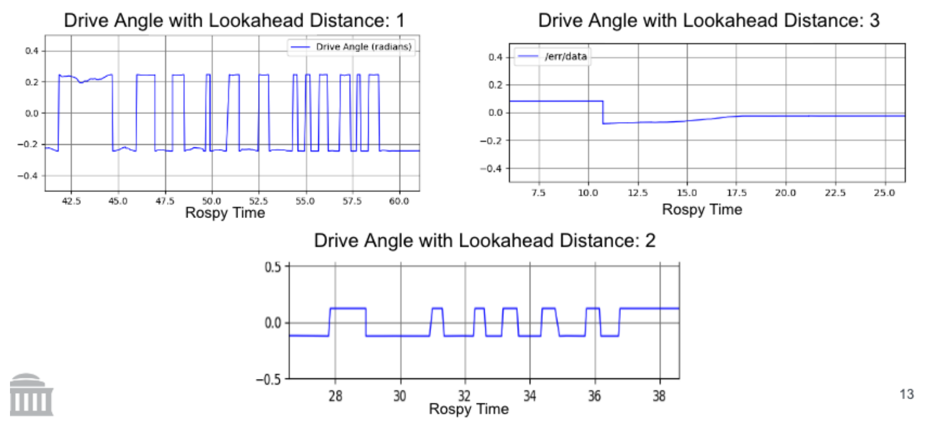

Figure 10: Testing robot oscillations with different look ahead distances

2.4 Path Planning on Physical Robot

After getting both modules and the integration to work on the simulator, it was time to transfer the code onto the physical robot to tune and test. At first, we ran into many issues. Primarily, one of the libraries that worked on simulator and was suggested in the reading did not exist on the robot. However, after refactoring our code to use a different library, we were able to get the path planning to work fully on the robot. Once we visually confirmed that the robot was able to follow trajectories, we then needed to tune the constants in our model.

2.4.1 Tuning

One of the major concerns with our path finding model was the oscillations. These oscillations occurred due to the nature of the pure pursuit algorithm. Generally, one way of dampening the oscillations is to increase the look ahead distance. This would enlogate the arcs that the robot was following and, especially in the straight sections, dampen the oscillations. Therefore, we did a variety of trials to determine the optimal look ahead distance for our robot. The results are shown above in Figure 10. A look-ahead distance of 1 meter caused extreme Ackermann angles and oscillation when trying to follow the path. A look-ahead distance of 3 meters caused a failure to follow the path around sharp angles like a corner. A look-ahead distance of 2 meters allowed for the smoothest path-following.

These graphs represent the drive angle across a variety of trials with different look ahead distances. It is evident that the look ahead distance of 1 resulted in large oscillations of our robot. Additionally, the look ahead distance of 3 created no oscillations, but the look ahead distance was so great that the robot was unable to follow the trajectory and instead drifted away from the path.

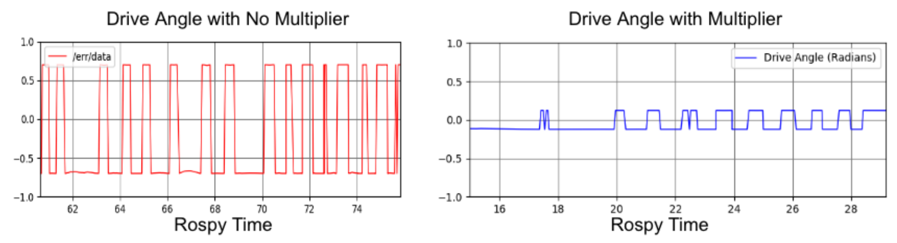

Figure 11: Final Result of Path Planning on Robot

However, the look ahead distance of 2 proved to be the optimal choice. The robot was still able to follow the path, but the oscillations were much smaller than the those with the look ahead distance of 1. Therefore, we chose the look ahead distance of 2 to be our optimal choice.

In addition to tuning the look ahead distance, another key idea we needed to test was the use of a multiplicative constant on the drive angle. This constant served to decrease the drive angle at each time step and therefore dampen oscillations. In order to validate the use of this constant, we compared the robot drive angle with the multiplier against a baseline without. The results are shown in Figure 11.

The graph on the left represents the baseline level of oscillation without the multiplicative constant. The oscillations have enormous magnitude and resulted the robot veering back and forth during the straight sections. However, in the presence of the multiplicative constant, the oscillations are quickly dampened. Moreover, the multiplicative constant did not affect the robots ability to follow the trajectory. Therefore, we were confident that the multiplicative constant on our drive angle was effective and worthwhile.

2.4.2 Final Result

After tuning our look ahead distance and validating our multiplicative constant, we conducted final tests to ensure that our robot was functioning correctly. Overall, the robot was able to plan its own trajectory, localize in the Stata basement, and pursue the trajectory to the finish line. The images in Figure 12 show the results. In the images, one can notice the Lidar data matching the walls of the Stata map. This demonstrates effective localization. The white line represents the robots planned trajectory. Additionally, the red arrow represents the look ahead point of the robot. The pose tracked the planned trajectory and provided a smooth look ahead point for the robot. Therefore, our robot was able to complete the tasks laid out in lab 6.

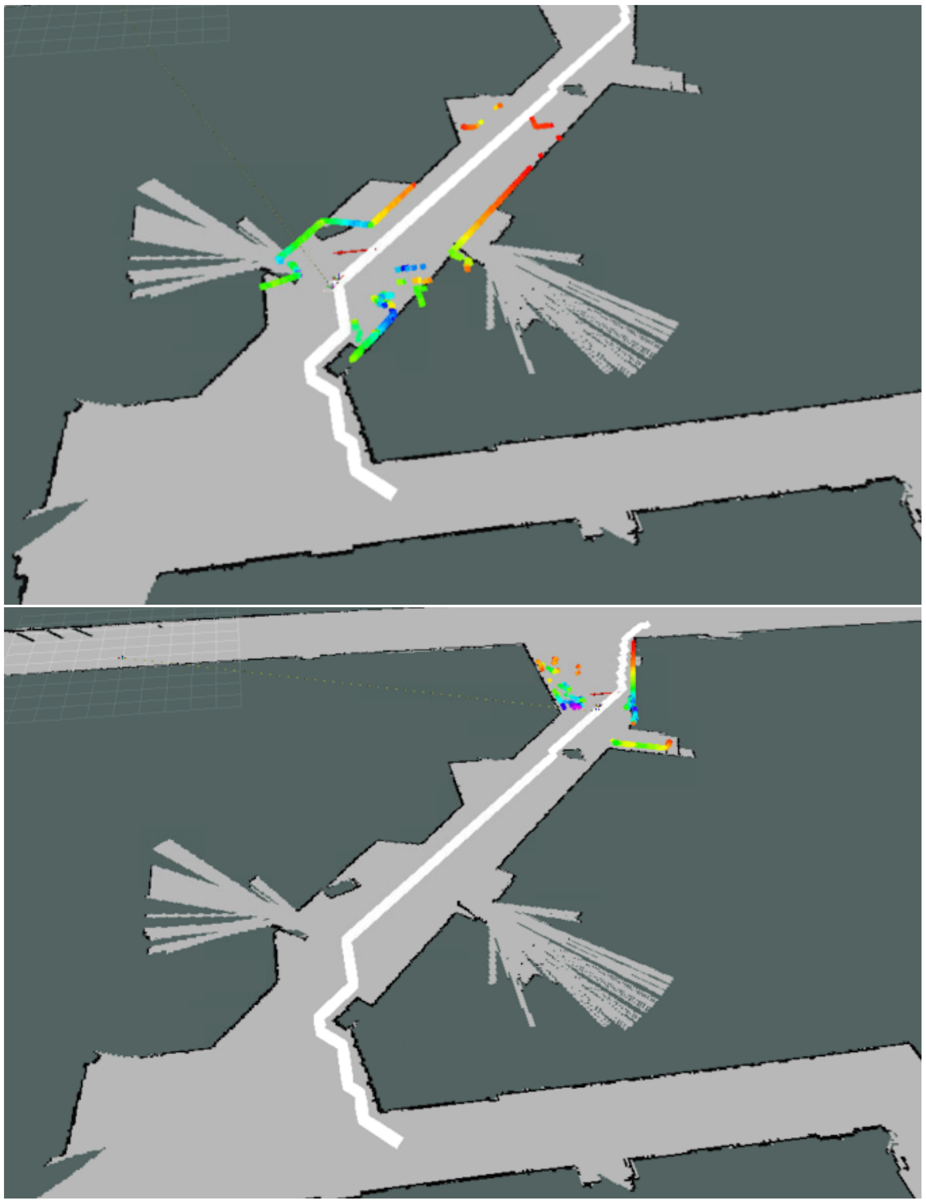

Figure 12: Validating the use of a drive angle multiplier

3 Conclusion

We acheived our goal of implementing a successful, effective path planning algorithm. The robot was able to navigate the Stata basement without hitting obstacles, and the paths met the design criteria, which we were able to verify through autograder results. Observing how the robot behaved in simulation, we verified that it was finding efficient, safe paths reliably. Moving our algorithm onto the robot in the real world, we spent time carefully tuning the lookahead point and steering control multiplier to dampen and minimize oscillations, running many trials in the Stata basement. Our pure pursuit controller was able to follow the given staff paths as well as a planned path from our path planning algorithm.

In the coming weeks we plan to further tune our existing code on the robot while preparing for the final challenge. Our path planning will be necessary for the city driving module of the challenge, and it will be critical to integrate the functionalities discussed in this report with collision avoidance and object detection. Modifying our algorithm to take into account intermediate points in path planning as well as improving robustness through extensive testing on the physical robot will be necessary for successful integration of all modules in the final challenge.

4 Lessons Learned

4.1 Haley

This lab helped me learn more about the importance of prioritization, especially when on a tight timeline. Normally, we try to fine tune everything as much as we can but this time around we had to focus on making sure we could get the program working first. It helped me see how valuable an MVP can be over pouring many more hours into this lab. Especially since we needed to start shifting gears over to the final project in order to complete all of the class tasks within the semester.

4.2 May

This lab taught me a lot about how to approach more complex tasks on the robot, as well as the value of completing work on the physical robot even when some parts of the algorithm were imperfect. We delivered our briefing after the original deadline, and I'm glad we were able to request an extension ahead of time as it gave us some space to ensure we understand our path planning algorithm well before moving into the final project. We started planning for the final project while still wrapping up work on path planning, and I think that this exercise of thinking about how our code would be applied and integrated on other tasks will be valuable in moving forward.

4.3 Mo

This lab helped me realize the importance of good documentation and communication and passing over work between team members. Different team members worked on different parts and when integrating the code on the robot, I had trouble understanding some parts of path planning and pure pursuit. It made working with the robot while other members weren't there more difficult. I think we still have to master understanding the workflow of our code once it gets on the robot. I also realized the importance of developing that MVP even when pure pursuit and path planning wasn't perfect, and focusing on the details of tuning later. There was a lot more to handle than initially expected when trying to get on-robot results.

4.4 Neil

As I worked on most of Part A (path planning), I learned the importance of visualization in the debugging process. For example, to check if the cost function in A* was working properly, I used Matplotlib to display it as a 2D image until I got it working properly. Visualizing intermediate parts of the algorithm was key to making it work in the end.

Integration between path planning and pure pursuit was one of the more difficult parts of this lab. In hindsight, we should have changed our method a bit – get a barely functional version of path planning working quickly, and then get it to the people working on pure pursuit so the eventual finished version could get integrated quickly. This was a lesson I took away from the lab: it's better to get all the components barely working quickly than strive for perfection before developing and integrating future components.

4.5 Nikhil

Working on the robot, I learned a lot. The first thing is the important of troubleshooting and being adaptable. This was in large part due to the fact that one of the core libraries used in our path planning was not present on the robot. Therefore, I needed to adapt and rewrite some of the path planning code to account for this. Additionally, I learned the difficulty of running many programs on the robot. The parallel deployment of our localization and path planning modules made debugging more difficult because it was often difficult to decipher which module was malfunctioning in the case of an error. I overcame this by using rostopic echo and rosnode list. I will continue to apply this knowledge for the final project.

5 Credits Page

5.1 Nikhil

- Path Planning on Physical Robot
- Tuning on Physical Robot

5.2 Neil

- Path Planning

5.3 Mo

- Pure Pursuit

5.4 May

- Introduction
- Conclusion

5.5 Haley

- Implementing in Simulation