Lab 6 Report: A*-Based Path Planning and Application of Pure Pursuit to Follow Paths

Team 16

Lasya Balachandran
Fiona Gillespie
Adrian Gutierrez
Katherine Lin

Robotics: Science and Systems

April 27, 2023

1 Introduction *(Lasya, Katherine)*

In this lab, we used path planning to find collision-free trajectories between a start point and a goal point and used pure pursuit for our car to follow these trajectories. We developed our solution by building off of our localization solution from the previous lab as well as further applying the Linux, Git, and the ROS skills we have learned throughout this semester.

Path planning and following are essential to building an autonomous car and have applications in many open areas of research, including autonomous driving and UAV navigation. Given a map where our car can localize itself, once our car can also efficiently build a collision-free path from its current position to a desired point and follow that path, the car can safely and autonomously travel throughout the map. This capability is useful in situations where the car must travel a path without any manual controls. Through this lab, we simulated our car operating its path planning and pure pursuit programs in RViz and attempted to translate the programs from simulation to hardware.

The path planning component of our solution involved creating a grid representation of the Stata basement map, which we dilated to better recognize and more easily avoid obstacles. After creating a new representation, we explored different path-finding algorithms and implemented the search-based A* algorithm to efficiently build a collision-free trajectory between start and goal points.

Once we built a trajectory, we used pure pursuit to follow the path. This component involved finding the closest portion of the trajectory to the car and extending this portion to find a point on the trajectory at a given lookahead distance. The car then drove to this point, which we referred to as the lookahead point. The integration of our path planning and pure pursuit components allowed our car to build and follow collision-free trajectories.

2 Technical Approach *(Lasya, Katherine, Adrian)*

In order to design a program that allowed the robot to find and follow a collison-free trajectory, we focused on implementing and integrating two components: using a path-finding algorithm to plan a collison-free path and using pure pursuit to follow a given path. Figure 1 shows a block diagram of this program, which we will discuss in the following subsections.

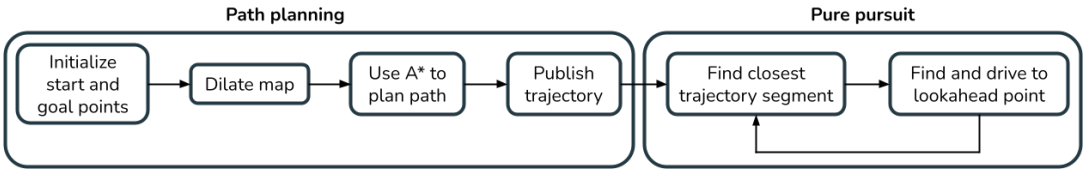


Figure 1: This block diagram shows our technical approach. The first four components show how we planned a trajectory using A* while the last two components show how the car followed the path using pure pursuit.

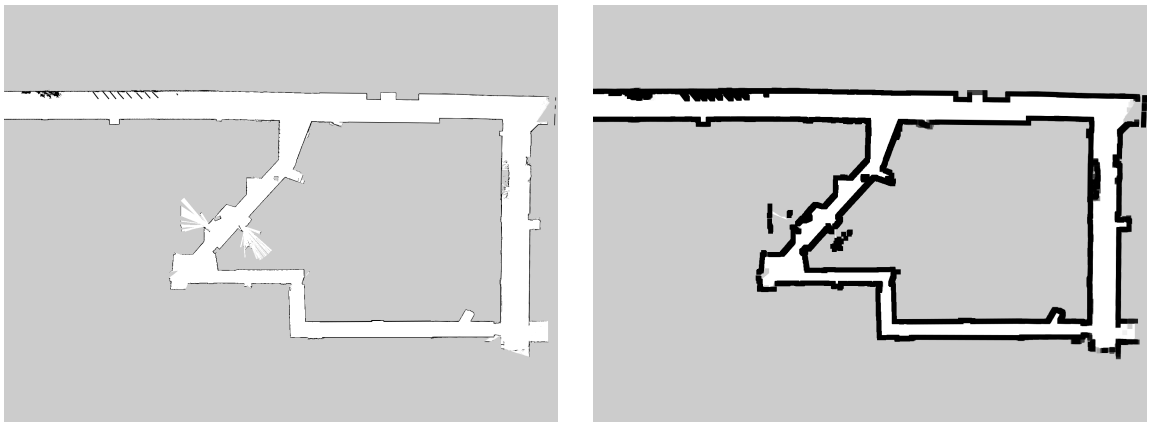
2.1 Path Planning *(Lasya, Katherine)*

We first designed an algorithm that could build a short trajectory from a start point to a goal point while avoiding obstacles such walls or other objects. This algorithm involved three main parts: creating a grid representation of the map and dilating the map, implementing a path-finding algorithm (specifically the search-based A* algorithm), and converting between real and pixel coordinates to switch between the map frame and the corresponding grid representation.

2.1.1 Creating the Search Grid and Dilating the Map *(Lasya)*

Our path planning program subscribed to the map topic, which published an OccupanyGrid message representing the map. This message included a list in row-major order corresponding to the probability of an obstacle appearing at each pixel. We resized this list based on the width and height of the map to create our grid representation. A value of -1 in this representation indicated that it was unknown if the pixel contained an obstacle. Otherwise, a value of 0 represented no obstacle, while a value of 100 represented an obstacle. For our path, we only considered pixels in which the value was 0 since any unknown pixels could contain an obstacle.

Given the grid representation and a start and goal point, our main goal then was to find a path that allowed the robot to travel between the points without crashing into any walls or other objects. Since search algorithms tend to cut corners to minimize distance or time, or possible paths in the grid representation of the map could be unfeasible in real life due to the car size corresponding to more than one pixel, we chose to dilate the map. We applied a greyscale dilation with a kernel of (15, 15), so neighbors of of nonzero-valued pixels also contained nonzero values. Figure 2 shows our original map of Stata basement compared to the dilated map.



(a) Original Stata Basement Map Before Dilation

(b) Map After Greyscale Dilation With (15,15) Kernel

Figure 2: The left image shows the map of Stata basement before dilation. The right image shows the map after greyscale dilation with a (15, 15) kernel.

After dilation, the obstacles in the map became more pronounced and occupied more pixels, so the path less likely traveled close to the walls or other obstacles.

2.1.2 Choice of Path-Finding Algorithm *(Katherine)*

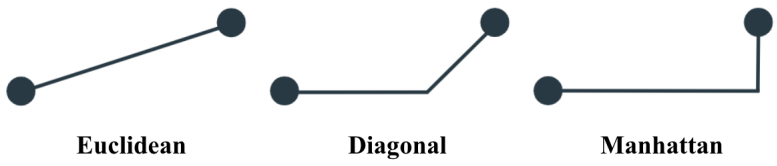
There were several clear frontrunners for algorithms that could find a path through a map: A*, which is a deterministic algorithm that was more efficient than traditional graph-searching algorithms such as DFS and Dijkstras, and RRT*, which was an optimized version of the rapidly exploring random tree algorithm. RRT* uses random sampling from the graph to reach the endpoint and explore unexplored areas of the graph. Due to its stochasticity, it is able to converge very quickly. On the other hand, A* is deterministic and methodical as it explores every node in the order the heuristic dictates. While A* is faster than something like DFS, it is much slower than RRT*. However, it is guaranteed to find the optimal path always, whereas RRT* may only find a near-optimal path.

Since the ability of the car to follow our generated path quickly is also crucial, we decided to use A*. In this way, we are guaranteed a better and less redundant path for our car to follow.

2.1.3 A* Implementation *(Katherine)*

The key insight behind A* has to do with the way the heuristic tries to estimate and minimize the cost of a path from start to end. To do this, at every node we calculate the heuristic (call it $f(n)$, where n is the node we are considering). The heuristic sums the cost of the path from the start node to the current node and the estimated cost of the path from the current node to the end. To reach our current node, we have already traversed from the start to the current node, and as a result, we know that the cost of the path from the start to the current node is the number of nodes on the path between them. However, since we have yet to reach the end, we can only estimate the cost from the current node to the end using a distance metric.

Our first design decision was what kind of distance metric we would use: Euclidean, Diagonal, or Manhattan.



(a) Distance Metrics

Figure 3: The diagrams show examples of how one might calculate the Euclidean, Diagonal, and Manhattan distances between two points, respectively.

In Figure 3, we show a visual example of how we would calculate the distance between two point using each metric. Euclidean distance is the traditional way of computing distance: $d = \sqrt{\Delta x^2 + \Delta y^2}$. Manhattan distance treats the world as if it has been divided into squares, and so finds the distance to be $d = \Delta x + \Delta y$. Diagonal distance is similar to Manhattan distance but considers the world to be made up of vertical, horizontal, and diagonal (45^o) movement. So it finds the distance to be $d = \max(\Delta x, \Delta y)$. To choose which metric to use, we looked at which would be most appropriate for how we looked at neighbors. Since for every node, we only consider its neighbors to be the nodes up, down, to the right, and to the left, we are working on a very rectangular grid. So the most fitting metric would be Manhattan distance.

With this decided, we defined our heuristic as $f(n) = g(n) + h(n)$, where $g(n)$ was the number of nodes on the path from the start to n, and $h(n)$ was the Manhattan distance between n and the endpoint.

From here, we built a heap of "frontier nodes" to keep track of nodes to explore, and a list of "closed nodes" that we had already explored. We started with just the starting node in the frontier. Then we would explore each node by looking through its neighbors (adjacent nodes) for viable nodes to add to the frontier. A node was viable if every pixel between the current node and the neighbor node was unoccupied. Given a viable node, we would calculate it's f-value, and check to see if it was in closed with a lower f-value (if so, this implies that we've already explored this node with a path that had less cost than the current, rendering re-exploring redundant), or in the frontier with a lower f-value (this would mean that we're already planning on exploring this node, and it already has a better cost). If both of those conditions are false, then we add this neighboring node to the frontier with its f-value and mark its parent as the current node. The frontier will then choose the next node to consider by finding the node with the smallest f-value, since we would like to explore the paths with the lowest cost first.

In order to boost the efficiency of our algorithm, we created the frontier to be a heap rather than a list. A heap always maintains the minimum node as invariant, making it easy and fast to find the next node to consider, as opposed to a list, which would have to iterate through every node to find the minimum. The frontier grows in size exponentially as well, making this increase in performance crucial.

In the end, when we reached the endpoint, we would stop exploring and instead backtrack to reconstruct the path created. We did this by tracing everyone's parent until we reached the starting node, at which point we had the full path. From here, we converted every pixel in the path to real coordinates and published them.

2.1.4 Converting Between Real and Pixel Coordinates *(Lasya)*

In order to apply A* on the grid representation of the map to create the trajectory, we converted between real coordinates (x, y) and pixel coordinates (u, v). To convert from pixels to real coordinates, we first scaled the real coordinates by the map resolution as specified by the OccupanyGrid message. We then applied a rotation and translation as specified by the OccupancyGrid message's orientation and position, respectively. Equation 1 shows how we converted from pixels to real coordinates.

$$p_{pixels} = \begin{bmatrix} u \\ v \end{bmatrix} = (R_{map}^{grid})(\begin{bmatrix} x \\ y \end{bmatrix} \cdot \text{map resolution}) + translation_{map}^{grid} \tag{1}$$

In order to convert from real coordinates to pixels, we reversed these operations, as shown in 2.

$$p_{real} = \begin{bmatrix} x \\ y \end{bmatrix} = \frac{(R_{map}^{grid})^{-1} \cdot (\begin{bmatrix} u \\ v \end{bmatrix} - translation_{map}^{grid})}{\text{map resolution}} \tag{2}$$

In both 1 and 2, we specified the rotation matrix R_{map}^{grid} using 3

$$R_{map}^{grid} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{3}$$

where θ indicated the yaw of the grid representation of the map, which we found using the Euler to Quaternion conversion of the info.orientation.z field of the OccupanyGrid message.

2.2 Pure Pursuit *(Adrian)*

To follow our trajectory we decided to use pure pursuit. Our pure pursuit model is given a trajectory in the form of a PoseArray, which is an array of x and y coordinates, and Odometry data, of the car's current x and y position, to decide how to follow the trajectory.

2.2.1 Finding Closest Trajectory Segment *(Adrian)*

For our pure pursuit to work, we must look for the closest trajectory segment to our car. We start by interpolating between all adjacent points on our trajectory and these will be our set of trajectory segments.

With these, we then calculate the closest point on each trajectory segment to our car. By using the distance formula we are able to determine which one of these points is the closest to our car. Figure 4 gives an illustration of this process.

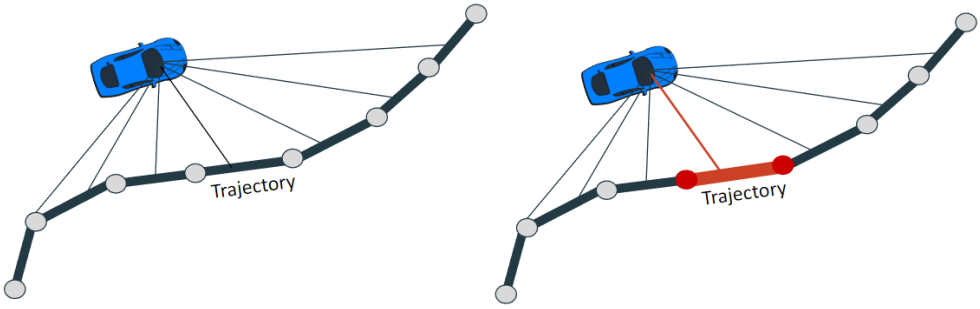


Figure 4: The left image shows the interpolation of the trajectory points. The right shows the closest trajectory segment that is chosen.

2.2.2 Finding Lookahead Point *(Adrian)*

After we retrieve the closest trajectory segment to our race car we extend it in both directions. In our program, we are given a lookahead distance, which is how far from the car we want to look for a point on the trajectory to drive to, and we must find the intersection between the circle with a radius given by the lookahead distance and the extended trajectory segment. Figure 5 shows this method.

As shown in the figure we find two different intersections. For the purposes of this project, we assumed that we will always drive along the trajectory in approximately the same direction the car is in. In other words, we choose the intersection point that is most parallel with the current direction we are driving in. To do this we center our coordinate frame to the car's position (the racecar is at x=0, y=0) by shifting the car's coordinates and the two intersection points by the car's current position in the world frame. Then we create a vector to both points and a vector of 1 in the direction the car is facing. By rearranging the equation of a dot product, $v1 \cdot v2 = |v1||v2|cos(\theta_{v1,v2})$, we are able to decide which point to drive to.

2.2.3 Driving to the Lookahead Point *(Adrian)*

To drive to our determined lookahead point we calculate the angle in the cars frame to the point. From there we set up an AckermannDriveStamped object and set the angle to what we found and the speed to a preset desired speed.

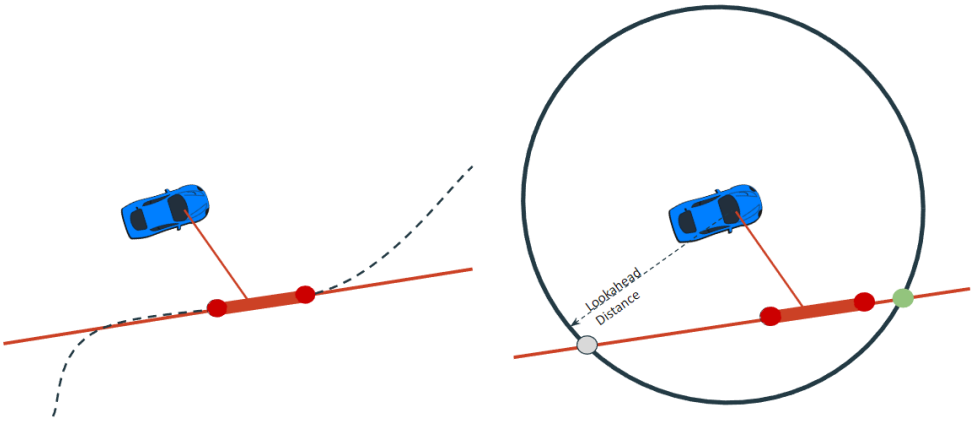


Figure 5: The left image shows the extension of the closest trajectory segment. The image on the right shows how the lookahead point that is chosen based on the lookahead distance.

3 Experimental Evaluation *(Lasya, Fiona, Adrian)*

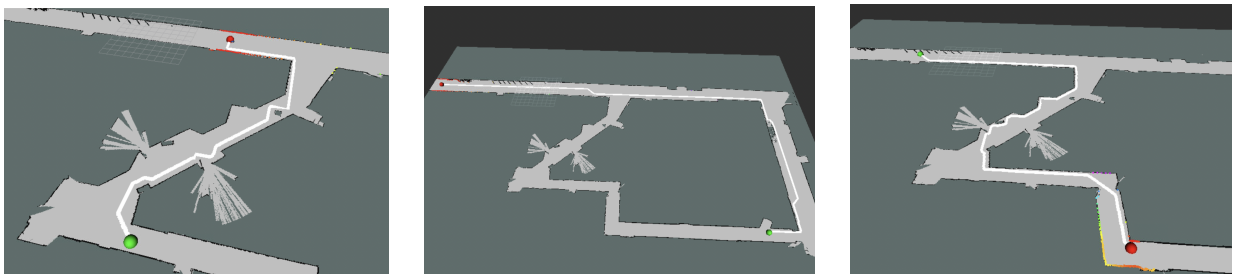
We evaluated and validated our path planning and pure pursuit in simulation, testing important metrics such as a qualitative analysis of the planned path, efficiency of our algorithm, and error between the planned path and our car's actual path. We also tried to integrate our design with the hardware but were unable to fully debug issues with loading pre-built trajectories for our car to follow. We will discuss our evaluation process and possible limitations for both simulation and hardware in the following subsections.

3.1 Simulation Evaluation *(Lasya, Fiona)*

In simulation, we evaluated our individual path planning and pure pursuit algorithms as well as the integration of the two components.

3.1.1 Path Planning *(Lasya)*

Once we had a working path planning implementation, we tested it on several different paths in RViz. Three of the paths that we tested in simulation are shown in Figure 6.

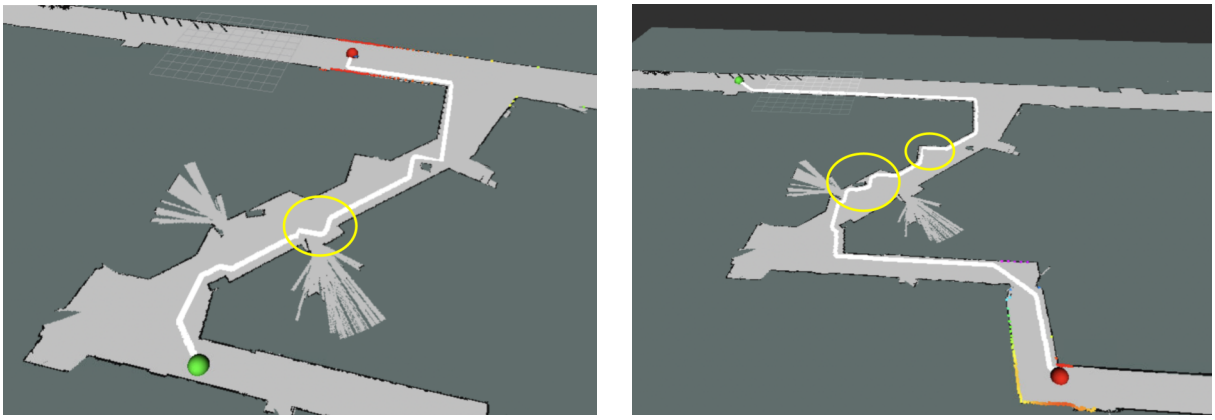


(a) Path 1 (b) Path 2 (c) Path 3

Figure 6: The figures show three planned paths in RViz where the red dot represented the start, the green dot represented the end, and the white line represented the planned path between the two points.

In each of the three paths, we see that the algorithm was able to successfully find a short, collision-free path between the two points while maintaining a slight distance from the walls and other obstacles. How-

ever, while our algorithm planned a collision-free path between the two points, based on Figures 6a and 6c, we also see that these paths were not the most direct ones. In locations where the wall jutted out a bit, the paths traveled inside the area before returning back outside, as shown in the circled areas in Figure 7.



(a) Limitations of Path 1

(b) Limitations of Path 3

Figure 7: The figures show two short, collision-free planned paths in RViz but not the most direct collision-free paths between the start and goal points. The circled areas represent the areas where the paths traveled more than necessary.

This limitation could be due to our algorithm prioritizing efficiency over accuracy by only visiting every pixel once. In the paths in Figure 7, we see that by traveling into the circled areas, the paths reached pixels that were closer to their respective goal points. Since the algorithm visited neighboring pixels in order of increasing distance from the goal point and did not revisit the pixels to check if there were shorter paths, the algorithm created paths through neighboring pixels in the circled areas.

In addition, since we valued the efficiency of our path planning algorithm, we used the time from setting the goal point to publishing the trajectory as an evaluation metric. The algorithm planned the three paths in Figure 6 in 5.705 seconds, 6.788 seconds, and 5.561 seconds over distances of 47.91 m, 119.6 m, and 87.82 m, respectively. Figure 8 displays the amount of time needed to plan these paths as well as several other paths in Stata basement.

From the graph, we can see that the algorithm consistently found paths up to about 120 m in the range of about 4 to 8 seconds. Shorter paths took closer to 4 seconds, while longer ones took closer to 8 seconds to plan the path. Based on these times, we considered our path planning algorithm to successfully and efficiently find a collision-free path between the start and goal points.

3.1.2 Pure Pursuit *(Fiona)*

After implementing our pure pursuit algorithm, we tested it through simulation to validate the logic and design. To individually test pure pursuit and also allow for parallelization, pure pursuit was first tested on pre-computed trajectories.

We tested how well our pure pursuit algorithm could follow a loop through Stata basement. Figure 9 shows the goal trajectory on the left in white, and the followed trajectory on the right in red. On the straight segments of the path, the racecar followed the goal trajectory reliably. However on the corners, there were some small deviations, which are circled in yellow.

To quantify our performance, we also tracked the error between the position of the lookahead point on the trajectory and the racecar's position. Figure 10 displays the error seen over the course of the Stata basement

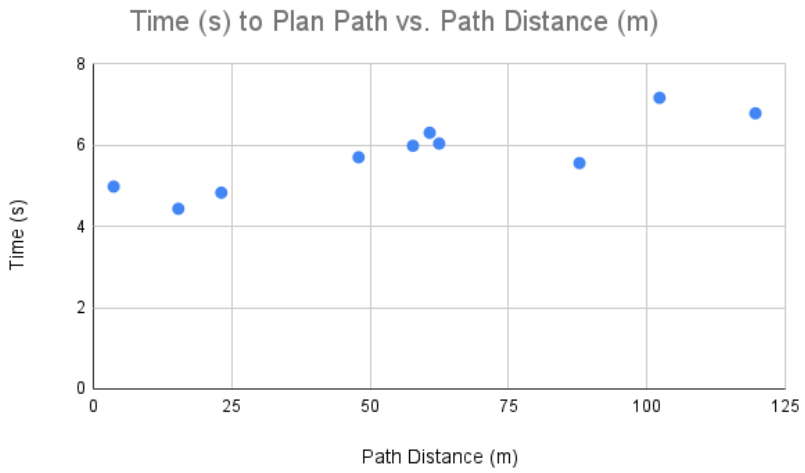


Figure 8: This graph shows the amount of time needed to plan paths of various distances on the Stata basement map. The times shown are from when the goal point was set to when the trajectories were published.

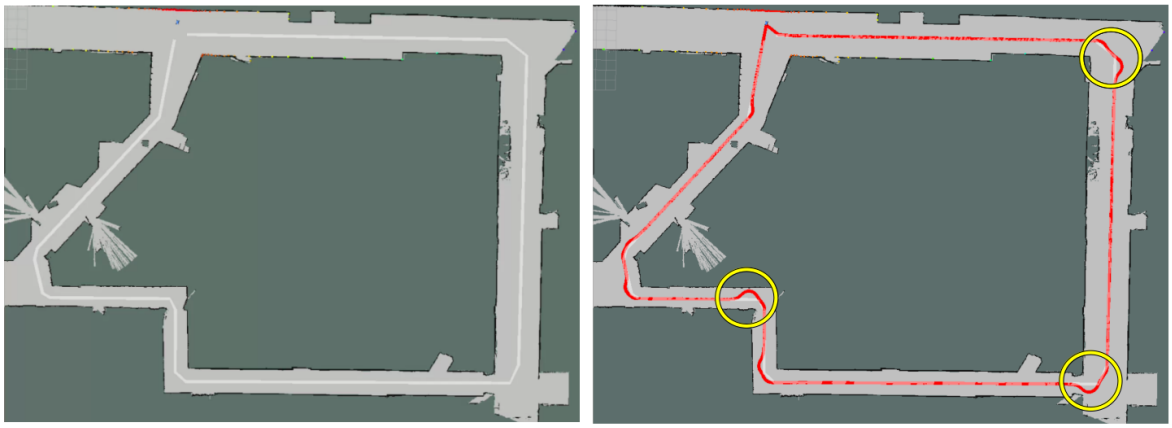


Figure 9: We tested our pure pursuit algorithm in the Stata basement. On the left, the white line shows the trajectory to follow. On the right, the red line shows the actually trajectory the racecar drove through. Note the small deviations from the goal trajectory on the corners, as highlighted in the yellow circles.

loop.

Since our lookahead point is 1m in front of the car, it makes sense that most of the data centers around 0.92m, as that is very close to our lookahead distance. When the racecar is turning on the corners, the error has some spikes which range up to 1.1m, which is 0.18m above where the car typically was. An extra error of 0.18m is reasonable, as the hallway is relatively wide and the racecar was able to drive through the trajectory from start to end without getting stuck or hitting obstacles.

We also tested how the performance could handle driving at higher speeds. Figure 11 shows the error from running through the Stata basement loop at 1 m/s, 2 m/s, and 3 m/s from left to right.

When the car ran at 1 m/s, the lookahead point was typically around 0.92m from the car, but the distance

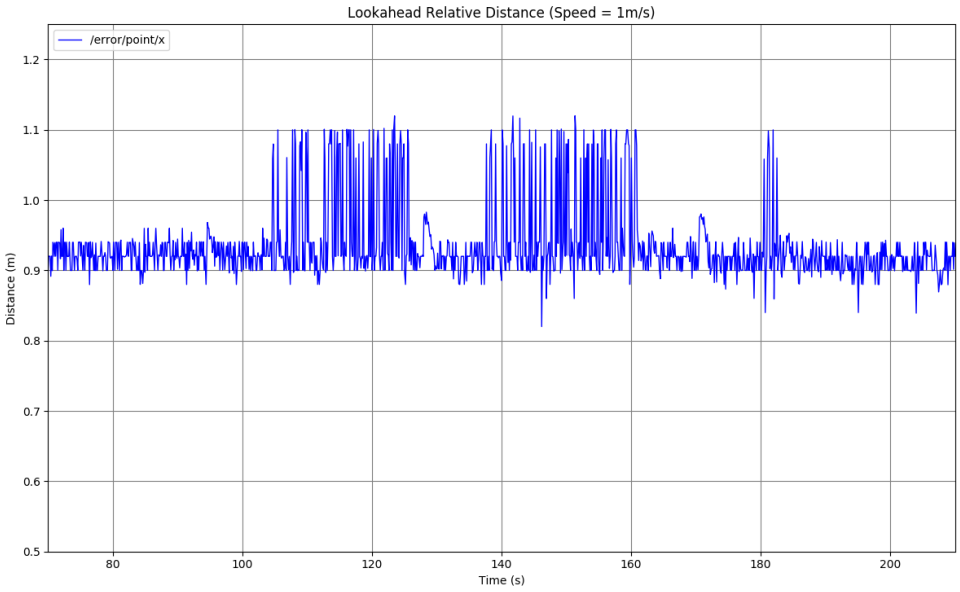


Figure 10: When driving at 1m/s in simulation, the racecar was consistently about 0.92m away from the lookahead point. However on corners, this distance rose to about 1.1m, which is a 0.18m increase.

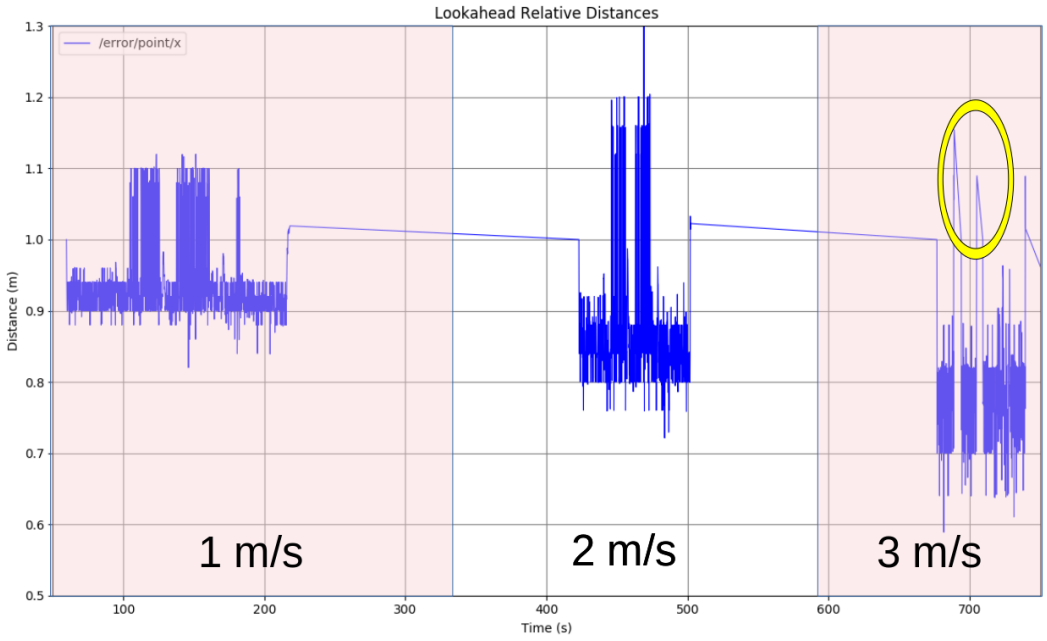


Figure 11: The lookahead point's relative distance to the car is recorded from the simulated car driving through the Stata basement loop at 1 m/s, 2 m/s, and 3 m/s from left to right. As the car's speed increased, the error on corners also increased. At 3 m/s, the car needed manually intervention to return to the path, circled in yellow.

increased to 1.1m on corners. For 2m/s, the lookahead point was typically a bit closer at 0.85m from the car, but on corners the distance increased up to 1.15m. Whereas the corner error increased by 0.18m for the 1m/s test, the corner error increased by 0.3m for 2 m/s. When the car ran at 3 m/s, the car was typically

about 0.75m from the lookahead point, but on some corners got stuck and needed manual intervention twice. Figure 12 shows the followed trajectory at 3 m/s, with yellow circles highlighting where the car got stuck. We suspect the static lookahead distance is the limitation for poorer performance at high speeds and corners.

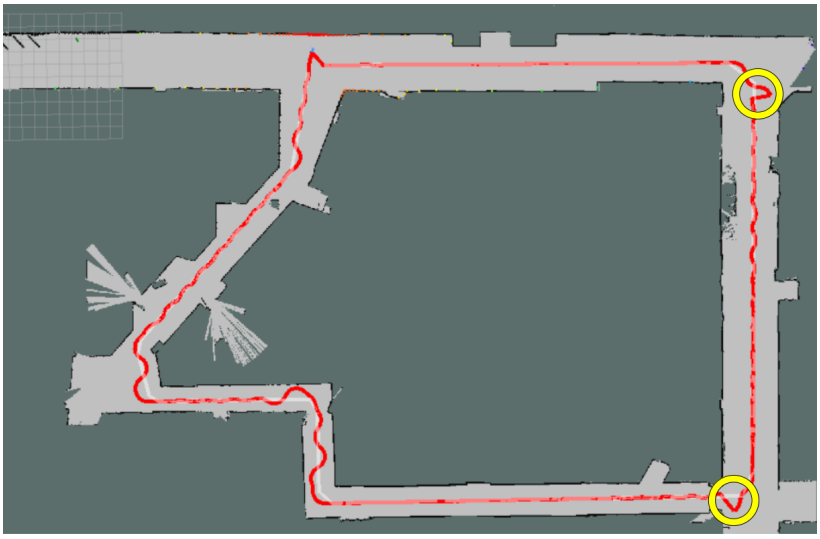
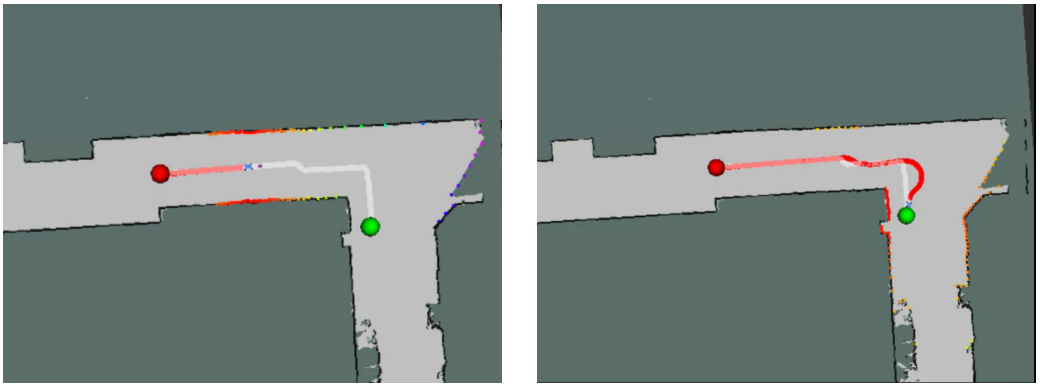


Figure 12: The red line shows the racecar's trajectory at 3 m/s. The yellow circles show the two places where the car needed manually intervention to return to the path

3.1.3 Integration of Path Planning and Pure Pursuit *(Fiona)*

After validating our path planning and pure pursuit algorithms individually, we integrated both algorithms together to successfully create custom trajectories. Figure 13 shows a visual of this integration.



(a) Planned Trajectory (b) Followed Trajectory

Figure 13: We tested our integrated path planning and pure pursuit algorithms together in simulation. On the left, the white line shows the trajectory to follow. On the right, the red line shows the actually trajectory the racecar drove through. Note the small deviations from the goal trajectory on the corners, as highlighted in the yellow circles.

The red marker is the start point, the green marker is the red point, the white line is the planned trajectory, and the red line is the followed trajectory. As noted in the pure pursuit evaluation, the performance on

straight segments is quite strong but the performance on the corners is more prone to error.

Figure 14 shows the error found for this path. The lookahead distance stays around 0.92m as it did in the loaded trajectory when the car ran at 1 m/s. On the corners, the error increased to 1.0m before the car could correct itself back to the trajectory. This error range is small enough to safely follow the path. This integration was exciting for us to see how our two main parts of this lab could successfully work together to create and follow any custom trajectory!

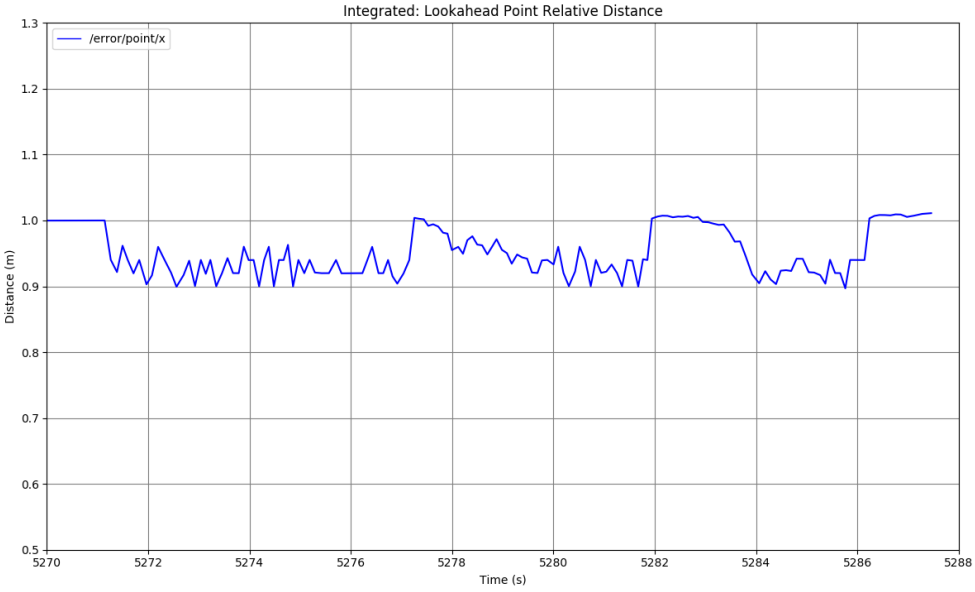


Figure 14: We found the error of our integrated path planning and pure pursuit algorithms.

3.2 Hardware Evaluation *(Adrian)*

Our future plans would be to apply our program to hardware. We did prioritize the capabilities of our program in simulation to have data to evaluate. When we attempted to run the program on our car we continuously ran into errors with using the controller and even with pre-built trajectories. As shown in figure 15 the pre-built trajectory given to us would not load correctly when run on the racecar and attempts to fix this during office hours did not fix this problem.

4 Conclusion *(Adrian)*

In this lab, we were able to create a program for our robot to create and follow paths to goal points. To create our paths we used the A* algorithm and to follow the resultant paths we used pure pursuit. Both algorithms did great in simulation.

In the future, we do hope to tune our A* algorithm so that we can create more optimal paths. Though we were able to find collision-free paths we believe we could optimize that algorithm to find more direct paths, or we could also attempt different algorithms to see which one might work better for us. For our pure pursuit algorithm, we would like to create an adaptive lookahead point. This would allow us to keep more on track with the trajectory. And we would also like to further apply our integration of path planning and pure pursuit on hardware. Though we attempted this we would like to move more quickly next time so that we have more time to do the integration on the racecar.

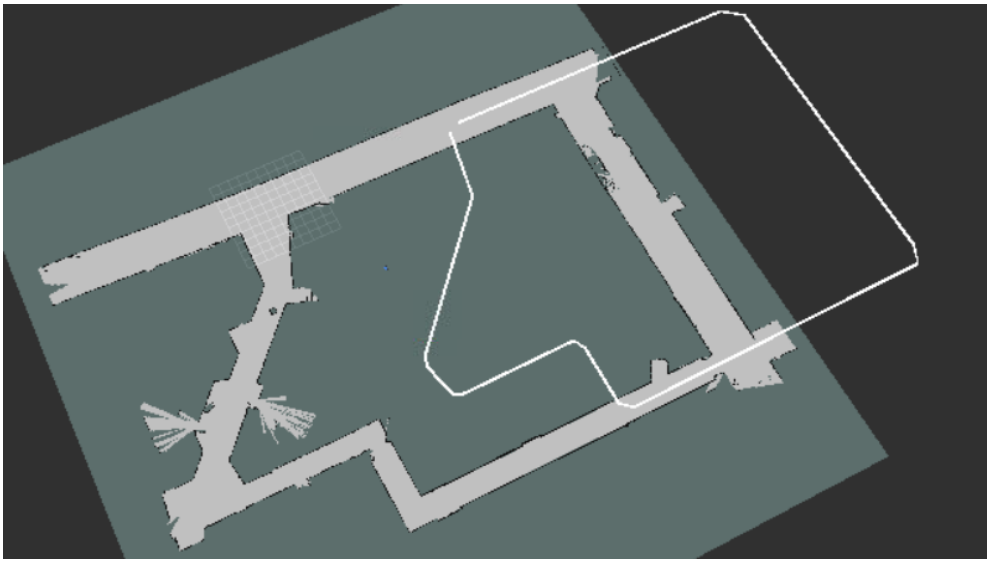


Figure 15: The pre-built trajectory given to us would not load in correctly leading to issues with testing our pure pursuit algorithm on the racecar.

5 Lessons Learned

5.1 Lasya

For the technical part, I found it a bit challenging since in terms of the path planning, I have only ever implemented simpler search-based algorithms such as BFS and DFS. I had briefly heard about A* prior to this lab but never implemented it before, so I had to learn how it worked before trying to help my teammates debug the algorithm. I had also never heard of the listed sampled-based algorithms, including RRT and RRT*, and I found limited information about those algorithms online, so it was particularly challenging trying to implement those algorithms. However, since I did not come into this lab with much prior knowledge in the area, I was able to learn a lot that I can apply in the future.

I feel that our communication has improved in terms of trying to understand what different members of the team have done on different components of the lab. Since we split into two groups of two to work on path planning and pure pursuit, we found it helpful for each of us to have a higher level understanding of the part we did not work on. Specifically, when it came to the briefing, we found that having a higher level understanding of those other parts allowed us to provide each other better feedback for the briefing.

In addition, I have found our collaboration to be an essential part of working on the labs. For the last two labs, since I have an M1 Mac, I have had issues submitting on Gradescope, and for the most part, I have had to rely on my teammates to submit whenever I wanted to test the autograder score after I made a change to the code. Also, similar to the past few labs, dividing up the work definitely helped us be more efficient.

5.2 Fiona

The technical aspects of this lab were quite difficult. The path planning involved complex algorithms, which made it hard to tell if there were bugs in the implementation or if our specific implementation was too slow. The pure pursuit geometry was also difficult and took me a few tries to actually understand what was going on. However, I do think we were able to really divide up the work efficiently for this lab which speaks to our growth as a team.

On the communication side, even though we weren't able to achieve everything we wanted, I think we were able to express the work we did do well through the briefing and report. One of the big issues from the last lab was deciding when to stop debugging and start focusing on the briefing/report. For this lab, we timeboxed ourselves more on the implementation, which I found helpful for adequately preparing for our briefing. Finding that balance can be difficult for class projects or even in the real world, and I think these labs are very useful to prepare for that.

5.3 Adrian

The geometry of finding a lookahead point was difficult. The online posts I followed explained what variables were being used so that I could replicate it in my pure pursuit algorithm but what each equation was exactly doing wasn't clear. Thankfully after running through the steps myself a couple of times, I was able to make sense of it.

It was unfortunate that we were not able to get the program running on the racecar. Though we attempted it a few times we decided it would be best for us to not spend too much time on this portion as we would run into the problem of eating away time that could be used for the final challenge. I believe that this choice has allowed us to better set ourselves up for the final challenge as compared to how we did for this lab.

Our communication between pods has still been good. Not many problems have arisen and we've been doing handoffs well. Also, we've been able to communicate within our group great as well. I found that making sure we work in person (even if our work could be done remotely) is helpful as we keep each other accountable and it speeds up work since we are able to bounce ideas off each other better.

5.4 Katherine

This lab, we experienced a lot of issues with the technical side of things. We got off to a late start due to general busyness, but once we started we met often, worked hard, and made good progress every day. Since I was working on path planning, I wrote the A* algorithm and made most of the design choices with respect to that algorithm. However, since we didn't start out working with a heap, our algorithm was incredibly slow at first. After a suggestion from Lasya, we changed our frontier representation and it immediately sped up. However, the Gradescope submission continued to tell us that our path planning was timing out despite our path planning in simulation finishing almost instantaneously. We spent a very long time trying to fix this but were unsuccessful in the end. However, I am quite proud that our path planning works very efficiently and accurately in simulation and feel accomplished considering where our algorithm started.

On the teamwork side of things, we split two-and-two this week to work on the two parts of this lab, so I primarily worked with Lasya. I think our teamwork was amazing: we spent two straight days together, twelve hours each day, doing nothing but debugging together (there were, of course, many other days we spent working together, but none quite as grueling as these). While a situation like that might inspire some friction, especially given our frustration over the Gradescope not working, Lasya and I both stayed sociable and worked through it. The things we accomplished with respect to the algorithm would not have been possible if we were working separately.

Our pod has been the same as ever; we are all having some issues with the robots but the teams in the pod are all very helpful in letting us know if they had an issue with a certain thing, so we can be cautious about it.