Lab 6 Report: Comparison of A* search and rapidly-exploring random tree sampling methods for path planning in an autonomous racecar following trajectories via pure pursuit

Team 6

Penny Brant
Yatin Chandar
Fritzgerald Duvigneaud
Nico van Wijk
Kristine Zheng
6.4200: RSS

April 27, 2023

1 Introduction (Penny, Yatin, Fritz, Nico, Kristine)

A **trajectory** can be defined a set of intermediate points that lie between a start and a goal position. These points can be used to divide a long path into smaller line segments that can then move around obstacles in an environment. Path planning and path following are the algorithmic processes of finding trajectories in a map environment when given a start and end point, and following the given trajectory.

With autonomous vehicle navigation, the importance of path planning and path following is clear when examining how people interact with these systems. Humans define navigation tasks as getting from place A to place B, and hence the instructions that we provide our autonomous vehicles are also in this format. Translating these start and end points into a trajectory that can be followed via driving algorithms is a clear problem that any robust system needs to be able to overcome.

In this report, we explore two different methods of path planning, **A*** and **RRT**, and compare both the speed and optimality of the paths that they generate.

Furthermore, we implemented **pure pursuit**, a path following algorithm that follows a line smoothly. We tested these solution in simulation, attempted to deploy in the physical racecar, and evaluated the performance of a trajectory following algorithm on different generated paths.

2 Technical Approach (Penny, Yatin, Fritz, Nico, Kristine)

2.1 Overview (Penny, Yatin, Fritz, Nico, Kristine)

Firstly, before running our path planning algorithms, we processed the map data (the map data we worked with for this lab is the map of the stata basement). The maps are provided as occupancy grid data which have probabilities of obstacles existing in discrete sections of space. From these, we generated a Boolean representation of obstacle presence in each location and dilated this to account for small errors in the map.

For creating trajectories, there are two broad classes of path planning methods: **search based methods** that discrete the map size and find optimal routes expanding out from the start point, and **sampling methods** that stochastically take points in the environment and build a web of connections until a route to the goal is found. For our two path planning methods, we intentionally chose one search based method (A*) and one sampling based, rapidly-expanding random trees (RRT).

Furthermore, to evaluate the practical usefulness of these generated paths, we tested a racecar's ability to follow our generated trajectories in simulation. For this, we utilised a pure pursuit algorithm which involves the car continually checking a certain lookahead distance from itself to find a point that intersects with the trajectory and then calculating steering drive commands to steer towards this point. Fig. 1 is a system overview for how the path planning and path following components of this lab integrates with the localization in the previous lab to build a successful program.

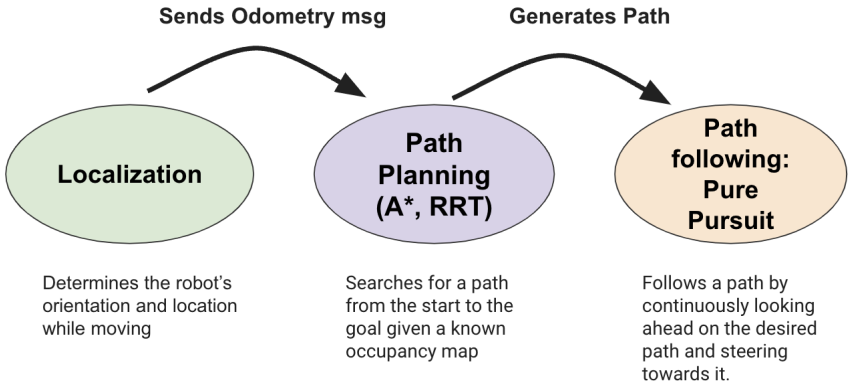

Fig. 1: Overview diagram of how localization, path finding, and path following interact and function. Localizatoin takes in LiDAR and odometry data and outputs estimates of the robot location, Path Planning calculates a trajectory given a start and end point, and path following allows the robot to follow the trajectory.

2.2 System Setup

We worked on a racecar with Hokuyo 2D LiDAR sensor for LiDAR data a NVIDIA Jetson TX2 Embedded Computer. For simulation, we tested it on RViz in ROS with 100 LiDar data points, the odometer data is obtained for wheel encoders.

2.3 Path Planning (Penny, Yatin, Fritz, Nico, Kristine)

The goal of path planning in this lab is to find optimal trajectories in an occupancy grid map, which contains information about collisions (walls and other obstacles), given the car's current position to a desired pose. We will discuss setup for the occupancy grid map, transformations between the world and pixel frame, and how A* and RRT use the given information to find paths.

2.3.1 Occupancy Map Setup (Penny, Yatin, Fritz, Nico, Kristine)

Occupancy grid data in provided in the form of an 1D Array listing the probability of an obstacle present for pixel in row-major order, starting with position (0,0) in the pixel frame. Map meta data includes the height and length of the map in pixel count. It also includes the map resolution which corresponds to the side length, in metres, of the real-world space box represented by each pixel.

We took this information and used it to generate a 2D array Boolean array. We decided to use 0.5 as our threshold value for probability. If a pixel in the

occupancy grid had higher probability than 0.5 of being occupied or -1 (an unknown probability), we determined that it was occupied and set its value to 1. Otherwise, we reduced the value to 0. This gave us a 2D array that we could index into by providing any pixel frame coordinates and determine whether a pixel was occupied by an obstacle or not.

Next we used binary dilation, a technique that takes each occupied cell in an occupancy grid and "inflates" it so that the surrounding cells in a specified radius are also marked as occupied, as in Fig. 2. This has the effect of making walls thicker and passages narrower, giving an artificial buffer zone for the path planning algorithms, preventing them from generating paths that are unrealistically close to walls and obstacles. Binary dilation is a method that is included in the scipy library.

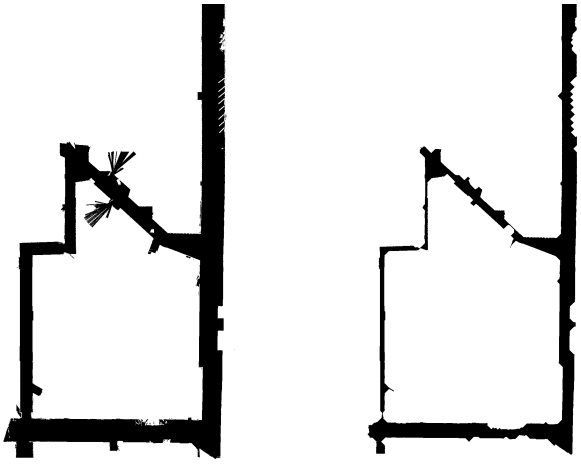

Fig. 2: Left: Stata basement occupancy map without dilation, Right: Stata basement occupancy map with binary dilation iterated 15 times

Finally, to also account for the fact the car is not a point mass and another method to add some padding around the car thus restricting moves from getting too close to obstacles and walls, we put a bounding box around the car. The idea behind this can be seen in Fig. 3 below. The car takes up space over multiple pixels so even though we are path planning between two pixel points in space, when collision checking we used a box with 0.35m dimensions, centered of the point we were evaluating for the path, and checked for any collisions within this bounding box. We only considered the point a valid addition to our path only if zero collisions were detected.

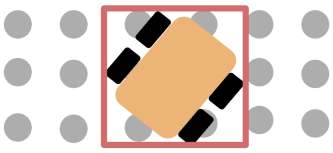

Fig. 3: The bounding box around the car position pixel estimates the real size of the car.

2.3.2 Transformations Between Pixel and World Frame (Penny, Yatin, Fritz, Nico, Kristine)

For efficiency, starting and goal points on the simulation map were transformed into the pixel frame before path planning was done. This prevented us having to make a transformation every time we wanted to check a viable point position for collisions.

Once we had calculated a trajectory of points in the pixel frame, we then converted back into the world frame to provide to the car. For any points in the pixel frame $(x_{pixel}, y_{pixel}, 1)$, the transformation from pixel frame to world frame after finding the path includes the following steps:

1. Multiply by map resolution:

$$\begin{bmatrix} x_{pixel} * (map\ resolution) \\ y_{pixel} * (map\ resolution) \\ 1 \end{bmatrix}$$

2. Apply the transformation (rotation and translation):

$$\text{point in world frame} = \begin{bmatrix} cos(-\theta) & sin(-\theta) & x_{map\ origin} \\ sin(-\theta) & cos(\theta) & y_{map\ origin} \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_{pixel} * (map\ resolution) \\ y_{pixel} * (map\ resolution) \\ 1 \end{bmatrix}$$

To transform from the world frame to the pixel frame after initializing the start and goal points, we apply the same procedure but with the inverse of the transformation matrix and by dividing the map resolution.

$$\text{world to pixel frame point} = \frac{1}{(map\ resolution)} * \begin{bmatrix} cos(-\theta) & sin(-\theta) & x_{map\ origin} \\ sin(-\theta) & cos(\theta) & y_{map\ origin} \\ 0 & 0 & 1 \end{bmatrix}^{-1} * \begin{bmatrix} x_{world} \\ y_{world} \\ 1 \end{bmatrix}$$

2.3.3 A* (Penny, Yatin, Fritz, Nico, Kristine)

A* is an admissible and complete search-based planning algorithm which is able to always choose the shortest path if it exists by favoring actions that will likely get closer to the goal. From the starting point in the pixel map, the next move

is selected from the 8 adjacent squares of step size n away and within the map bounds, shown in shown in Fig.4.

For neighboring pixels that bring the ROBOT closer to the goal (rather in the opposite direction), they are added to a priority queue of next moves which is ordered by lowest cost — the sum of the actual distance from the start to the potential point and the heuristic, aka estimated distance from the potential point to the goal. Updating the neighbors and the priority queue continues until the goal is reached or the queue is empty. Each node's parent node is stored so if the current node is within a step size n from the goal, the shortest path can be retraced to the starting point.

More specifically, A* selects for a path which minimizes the total cost function to the goal, $f(x = goal)$:

$$f(x) = cost(x) + heuristic(x)$$

where for state x, $cost(x)$ is the incurred cost or current path's distance from start state to graph node x and $heuristic(x)$ is the estimated cost or euclidean cost from node x to the goal (without regards to obstacles). This heuristic distance is guaranteed to be at most equal to the actual distance thus satisfying the admissibility requirement of the algorithm.

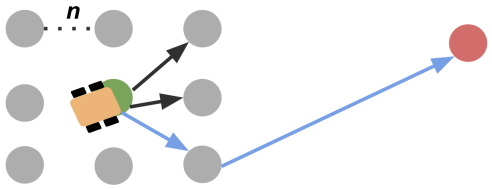

Fig. 4: Setup for A*: The algorithm searches through neighbors of the current state and keeps selecting moves with the smallest total cost until the shortest path is found if it exists.

2.3.4 RRT (Penny, Yatin, Fritz, Nico, Kristine)

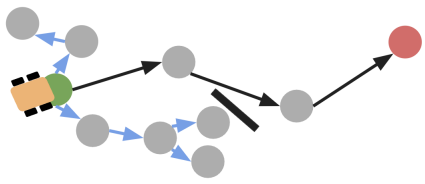

Fig. 5: Depiction of RRT algorithm over multiple time steps.

Rapidly exploring Random Trees (RRT) is an efficient algorithm used for searching through high dimensional discrete or continuous space, where search trees would otherwise be very computationally inefficient. In the case of a path planning problem, RRT randomly selects locations to explore and assesses whether the robot is able to navigate to the point. If so, the algorithm connects the point to the nearest validated point to create a branch. This branching continues over the entire space until the algorithm validates a point that is close to the goal. The branch that results in the lowest number of points from the start to the goal is returned as the path. RRT is not guaranteed to find the most optimal solution in the search space, but due to its immediate search over the entire space, will be faster than other queue- or tree- based planning methods.

RRT, while fast, often returns suboptimal trajectories, especially in open spaces. Fig. 6 shows how RRT can return paths that zigzag through open space, unnecessarily increasing their length and difficulty of traversal.

Fig. 6: Suboptimal, inefficient path

One solution to this problem is the more advanced RRT*, an algorithm that adds a distance heuristic to RRT. RRT* minimizes the distance of each point to the goal while checking for connectivity to return the most optimal path. It will

also prune the path after making each connection so that paths hug obstacles, like the paths generated by A*. However. RRT* is much more computationally expensive, taking multiple extra operations per test point than RRT. RRT has been shown to be approximately eight times faster than RRT*.

An intermediate solution that is very computationally inexpensive takes the path pruning of RRT* and runs it on the final path, once. The procedure is as follows:
1. Check if the line from the start to the first point in the path is traversable (this is always guaranteed, but it makes looping easier)
2. Repeat the first step for subsequent points until a point on the path is found that is not traversable directly from the start point
3. Remove the points in between the start point and the nontraversable point
4. set the nontraversable point as the start point and repeat.

This procedure greatly reduces the zig-zags that are common with RRT paths, and also minimizes the number of points needed to create the path. When coupled with occupancy grid dilation, RRT can return efficient paths quickly, as seen in Fig. 7.

Fig. 7: Optimized RRT path

2.4 Pure Pursuit (Penny, Yatin, Fritz, Nico, Kristine)

To help the robot nagivate and follow the paths computed using A* and RRT, we implemented Pure Pursuit. Pure pursuit is a path following algorithm that follows a given path through finding **goal points** along the path and adjusting the steering angle dynamically towards the goal points to create a smooth trajectory. The pure pursuit problem setup is shown in Fig. 8, the car located in world frame location (x, y, θ) aims to follow a trajectory, which is a line defined by a set of points joining the line segments.

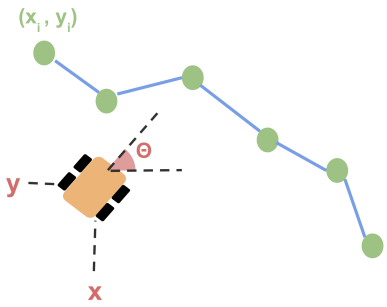

Fig. 8: Pure pursuit problem setup, the car is located in position (x, y) and facing direction θ in world frame. The trajectory the car wants to follow is represented by a line defined by set of points (x_i, y_i) shown in green.

In order to calculate a smooth trajectory along the path, the robot will continuously define goal points, defined as points on the trajectory that are a constant **look away distance(l)** away from the robot.

2.4.1 Goal Point Calculation

To calculate a goal point, we define a circle with radius l centered at the robot's location (x , y) and calculated the intersection between the circle and the trajectory.

To speed up the process of finding which line segment along the trajectory contained this goal point, we first iterated through the line segments to calculate the closest line segment from the car.

To do this, we find the closest point on each segment to the car. This is done by finding the projection of car location, point p, onto each line segment (v, w). As this point lies on the line segment it must therefore satisfy the following property for some constant t:

$$projection = v + t * (w - v)$$

Imagining that we extend the line segment indefinitely and find the orthogonal projection, we know that t should satisfy the following equation:

$$t = [(p - v).(w - v)]/|w - v|^2$$

However, our line segment is not indefinite, so once we solve for t, we bound it between 0 and 1 since if the orthogonal projection is outside the segment of line then the closest point will be one of its ends. Once we have all of the closest

points, we can determine the closest line segments based upon their closest point.

After finding the corresponding index, i, of the closest line segment, we were able to speed up the process of finding the goal point through calculating the intersection between the line segments with the circle only using line segments that appear after the closest line segment. We define a circle with radius l centered at the robot's location (x, y), and calculate the intersection between this circle and each line segment defined by points (x_i, y_i) and (x_{i+1}, y_{i+1}). If the circle returns 2 intersections with the given line segment, we set the goal point as the intersection furthest along the path to avoid the robot going backwards along the path. A visual demonstration of this algorithm is shown in Fig. 9

Additionally, we set the look ahead distance, l to be proportional to the speed,

$$l = \mu \cdot speed \tag{1}$$

where μ is a experimentally determined constant, 0.833 in our case. The intuitive justification behind this choice is that when the robot is driving at a higher speeds, the goal points are adjusting constantly which is equivalent to the robot dealing with a trajectory with a higher curvature. Since we want to avoid the robot oscillating around the trajectory, setting a bigger look ahead allows the robot to maintain a similarly smooth path as when it's driving at a lower speed.

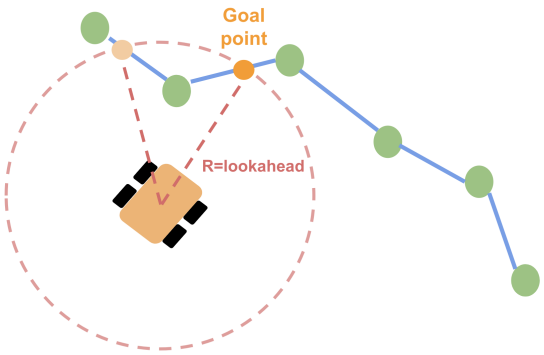

Fig. 9: Calculating the goal point along a trajectory. The vehicle is looking for a point lookahead distance away on the trajectoy, which leads to two points highlighted in yellow. We set the final goal point to be the intersection furthest along the trajectory

2.5 Converting Between World and Car Frame

After finding the goal point in world frame, we transform it back into the car frame since the car's drive commands are based in car frame. To do so, we utilized two spatial transformation matrices:

$$\begin{bmatrix} goalx(car) \\ goaly(car) \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} cos(-\theta) & -sin(-\theta) & 0 & 0 \\ sin(-\theta) & cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} goalx(world) \\ goaly(world) \\ 0 \\ 1 \end{bmatrix} \tag{2}$$

Starting with the car's world frame location on the right, the first matrix applied is a translation that translates the goal point relative to car frame origin, and the second matrix rotates the goal point to align with car frame, which returns the desired goal point relative to the car in car frame.

2.5.1 Drive Command

We calculated the steering command, where α refers to the steering angle of the car.

$$\alpha = tan^{-1}(\frac{2 \cdot L \cdot sin(\eta)}{l}) \tag{3}$$

Where L is the wheelbase length (distance between the front and the back of the wheels of the car, 0.32 in our case), η refers to the angle between the car's forward direction and direction to the goal point, and l refers to the look ahead distance (distance to the goal point).

3 Experimental Evaluation (Penny, Yatin, Fritz, Nico, Kristine)

3.1 Path Planning Algorithm Comparison

Using the Stata basement map, we were able to successfully use both A* and RRT to plan paths through the space to specified goal points. A successful path planning algorithm can be evaluated in two main ways - speed of planning, and efficiency of path. These are our evaluation metrics that we can use to compare RRT and A*'s capabilities. In addition, subjective evaluations and ease of integration can be used to compare the two algorithms.

3.1.1 Solve Time Comparison

To evaluate the solve performance of RRT and A*, we took the locations of the marked start and finish lines of the real-world Lab 6 race from the Stata basement hallway and found their coordinates in the simulator map. The start point was at (-10, 25) and the finish point was at (-41,0). These points were hard coded into both algorithms, which were then run to evaluate their solve

times and path lengths. Both A* and RRT were run a minimum of three times.

The solve time for RRT on a M1 Pro Macbook Pro ranged from a minimum of 0.089784145 seconds to a maximum of 3.304175854 seconds, with a mean of 1.254543543 seconds and a standard deviation of 1.08661582 seconds. The large standard deviation relative to the mean shows that there is a wide spread of solve times for the RRT algorithm. However, all the solve times are on the order of 1 second, which reveals the speed of RRT when given a large and challenging search space. One of the optimizations made to RRT was to remove all the occupied cells from the search space before randomly selecting a cell to test. This let the algorithm only search for points that were known to be empty.

Contrast these results with that of A*. Running on a 2019 Macbook Pro with an Intel i7 processor, A* took 77 seconds to find a path with a step size of 1, and 526 seconds to find a path with a step size of 5. These results are surprising, given that a larger step size took longer to solve, and that both step sizes took multiple minutes to solve given a fast processor. This indicates that many optimizations can be implemented in the future.

3.1.2 Dilation vs Bounding Box for Padding

See Fig. 10 and Fig. 11 to see how A* always finds the most optimal path from the start to end point if it exists. When comparing bounding box and dilation, the dilation method generally found longer paths as shown in Table. 2; dilation acted as padding for the walls so the path was more uniformly and further distanced from the wall whereas the bounding box added as padding for the car so it sometimes oscillates in distance from the wall. As a result, the dilation method would be better to use in real world testing for paths that are more uniformly distanced from the wall.

Fig. 10: Left to right: A* using a bound box in stata basement with step sizes 1,3,5 with speeds of 309.336, 978.90, and 280.557 seconds, respectively. Generally, smaller step sizes result in smoother paths but may take much longer to find (Sometimes larger step sizes take longer, perhaps because there are more collisions or computer processing related challenges).

Fig. 11: Left to right: A* using dilation in stata basement with step sizes 1,3,5 with speeds of 77.035, 978.90, and 526 seconds, respectively. Generally, smaller step sizes result in smoother paths and are expected to take longer to find.

3.1.3 Path Optimality Comparison

While A* takes longer to solve for a path, it is guaranteed to return a shorter, more efficient path due to its use of a heuristic. This is its main advantage over RRT, which, even with the path post processing optimizations added, still generates longer paths than A*. As seen in Table. /reftab:tab1 and by comparing paths from A* in Fig. 10 and Fig. 11 vs paths from RRT in Fig. 7 and Fig. 6, the RRT graph is less precise when going around obstacles and does not follow walls as uniformly as either A* method - a result of the randomization in RRT.
.

Table 1: comparison of speed and distance between different A* methods (dilation, bounding box (BB)) and RRT

Methods + Conditions	Speed of Path Finding (sec)	Path Distance (m)
A* with dilation, step = 1	77.035	53.37
A* with dilation, step = 3	995.068	50.15
A* with dilation, step = 5	526.010	51.82
A* with BB, step = 1	309.336	51.922
A* with BB, step = 3	978.90	50.15
A* with BB, step = 5	280.557	52.082
RRT suboptimal	1.254543543	62.3
RRT optimized	1.254543543	57.7

We tested pure pursuit with both A* and RRT in simulation - A* successfully found a path that avoids obstacles along the path in the test path, and pure pursuit followed the path with high accuracy here.

A key challenge in building a successful path following algorithm was also dealing with localization drift. In this video, the red arrows represent the estimated localization location for the robot. While the estimated locations followed the planned path accurately, the actual car drifted. We found through our tuning experiments that having smaller look ahead distance led to more frequent adjustment for goal points and more frequent steering, which made drifting error worse in simulation. However, having a really large look-ahead distance led to overly smoothed trajectories that often ran into walls since corners, as shown in here. Therefore, during pure pursuit tuning, we aimed to find a balance between the two.

3.2 Pure Pursuit Tuning

The main component for pure pursuit tuning is the look ahead distance (l). We set l to be proportional to speed, where $l = \mu \cdot$ speed, and experimentally determined what μ should be. We tested this for various values of robot speeds and evaluated the results qualitatively, with the results being recorded and listed in Table.2 From these results, we decided to use $\mu = 0.8356$ where resulted in the most stable path following in simulation.

Table 2: Qualitative evaluation of effects of look ahead distance across four different robot speeds.

\ look ahead dist.	significantly increased drifting	over smoothing path	best path following
Speed = 1	$0.6, 0.4$	$1.0, 2.0$	0.84
Speed = 2	$1.2, 1.0$	$2.0, 3.0$	1.67
Speed = 3	$1.8, 2, 2.25$	$3, 2.75$	2.5
Speed = 4	$3.340, 2.4, 3.2$	$8, 4, 4.8$	3.3425

Various values were tested and human evaluated into 3 categories: significantly increased drifting of the robot, over smoothing robot path, and best path following. After evaluating the results, we determined the best proportionality constant between look ahead distance and speed as $\mu = 0.8356$

3.3 Cross-Track Error Analysis

We utilized cross-track error as a metric to evaluate the performance of our car's ability to track the generated trajectories. It provided valuable insight into shortcomings in both the localization and software portions of the software stack, and will be a great aid in motion planning efforts going forwards.

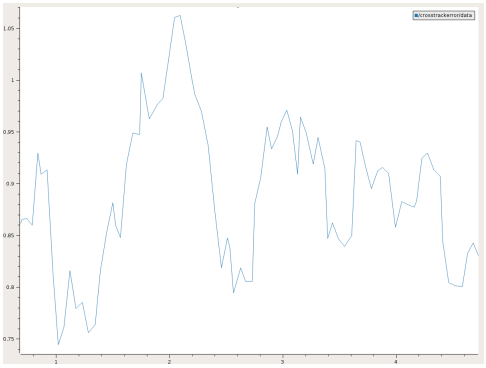

Fig. 12: A* Cross-Track Error Along Stata Basement Path At 1 m/s. The error stabled relatively stable and did not diverge.

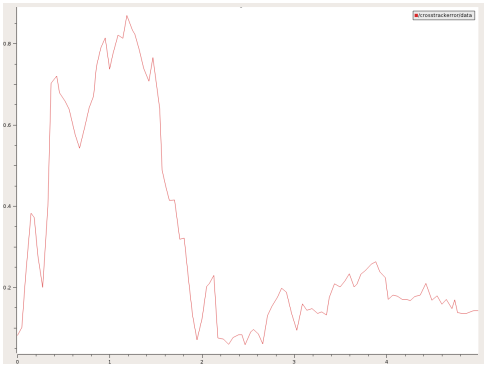

Fig. 13: A* Cross-Track Error Along Stata Basement Path At 2 m/s. The cross check error was initially large but converge to smaller values.

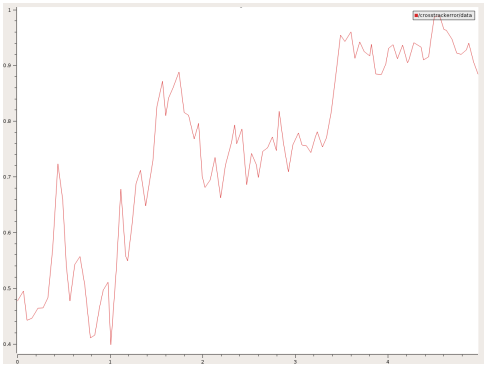

Fig. 14: A* Cross-Track Error Along Stata Basement Path At 3 m/s. The error was at comparable scale to lower speeds but diverged.

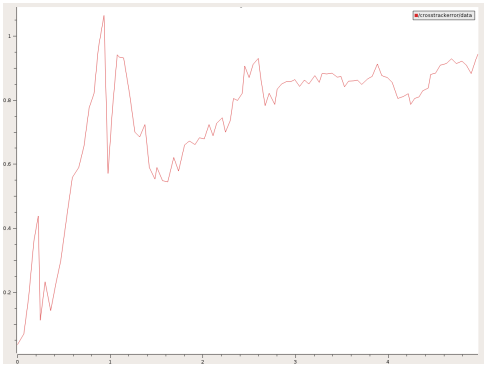

Fig. 15: A* Cross-Track Error Along Stata Basement Path At 4 m/s. The error stayed relatively smaller compared to slower speeds at the beginning, but diverged.

Experimental evaluation of the effect on speed vs. cross-track error (CTE) revealed that increasing speed tended to decrease the CTE, as shown in Fig. 12, Fig. 13, Fig. 14, Fig. 15 Not shown in the data plots is that the divergence between the CTE, which is between the path and the estimated odometry, and the true odometry also decreased significantly with increasing speed. This is inline with the results of our previous evaluation of our localization algorithm in lab 5 which showed the same trend. For further testing we utilized high speeds to achieve the best results.

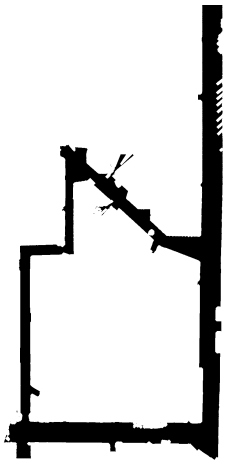

Fig. 16: Dilated Map of Stata Basement At Level 5. The unoccupied path the robot is able to move through (shown in black) are much smaller and further from the wall.

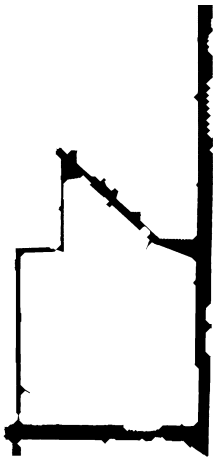

Fig. 17: Dilated Map of Stata Basement At Level 10. The unoccupied path the robot is able to move through (shown in black) are much smaller compared to smaller levels and further from the wall.

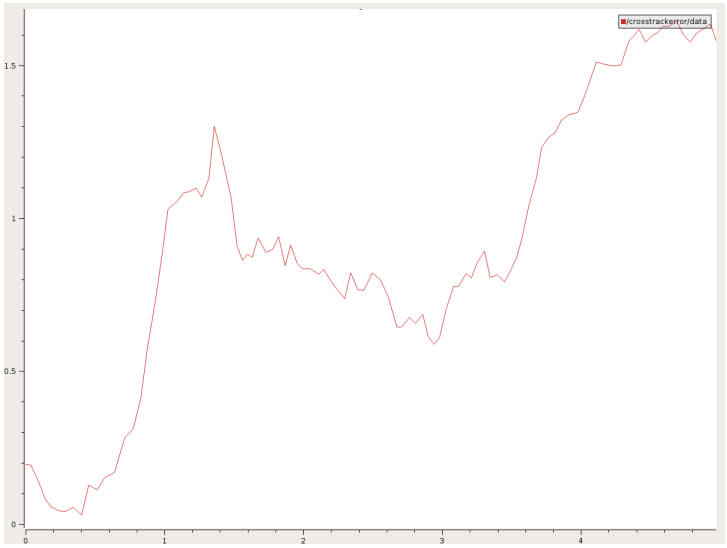

Fig. 18: A* Cross-Track Error Along Stata Basement Path At 3 m/s, Dilation Level 5. The CTE's magnitude was large compared to other tests.

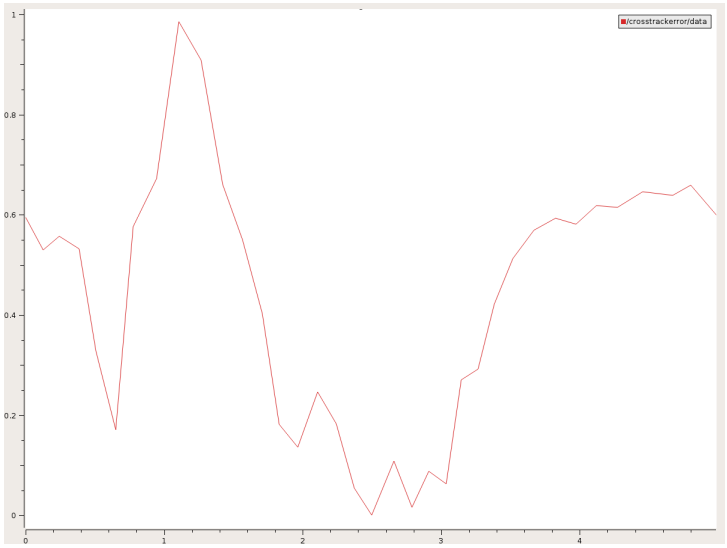

Fig. 19: A* Cross-Track Error Along Stata Basement Path At 3 m/s, Dilation Level 10. The magnitude of the error is smaller than all other tests.

We also tested the CTE across different levels of dilation. The dilated maps at levels 5 and 10 are shown in Fig. 16, Fig. 17 We found that the CTE decreased with increasing dilation, as shown in Fig. 18 and Fig. 19 This shows

that utilizing dilation may be a viable strategy in later testing to achieve more accurate path following. However, this increased accuracy came at the cost of longer path-planning time. To resolve this trade-off, we may employ larger map resolutions in the future, which should decrease computation times while providing similar benefits.

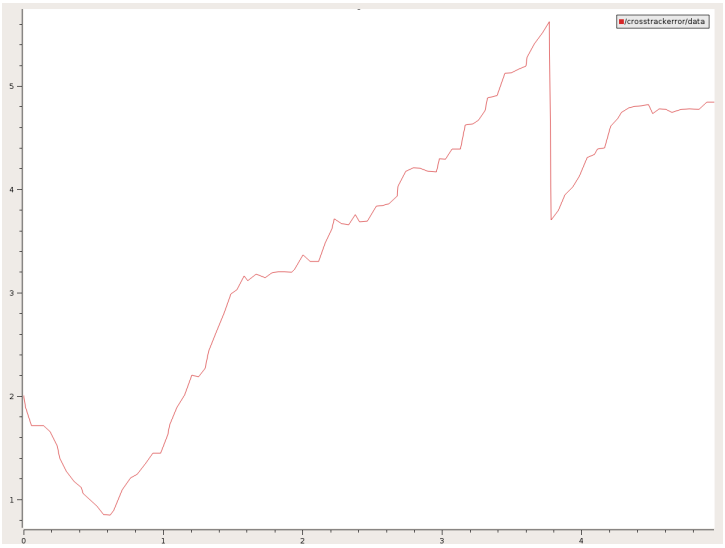

Fig. 20: RRT Cross-Track Error Along Stata Basement Path At 3 m/s. The CTE diverged quickly and at a much faster rate compared to A*.

As aforementioned, our pure-pursuit controller was unable to successfully follow the trajectories generated by RRT. Accordingly, the cross-track error diverges quickly, as shown in Fig. 20

3.4 Real World Testing

We were able to run the path planning (using A*) on the car's onboard computer to verify that it could localise itself and generate routes as seen in Fig. 21. We were also able to verify that the pure pursuit controller functioned as expected, although during our testing window, our car's computer battery died before we could test dilate the real car's map and move the generated path far enough away from the wall to avoid trigger the car's safety controller on sharp turns such as the right angle early in the path shown in Fig. 21

Implementing our solution on the physical race car came with its own challenges. Firstly, an important aspect of path planning is knowing the car's location throughout both planning and following a trajectory. To this end, we implemented a Monte Carlo Localization (MCL) algorithm that we built and

reported on for a previous lab.

In addition to this localization solution, we still needed to initialise the location of the car and also publish a goal for it to plot a path to. Within the simulation, we were able to to do this using UI features of rviz to interact directly with the map. On the car itself, we had to use a provided initial pose publishing node, and a separate goal publishing node that we created ourselves to establish the coordinates of these two path ends directly in the world frame.

We also faced hardware challenged with the race car itself. Notably, when we were first ready to trial our path following solution, our car would not drive. Through iterative testing of the components, we isolated the problem to the motor battery. Following this, we were dependent on a brief time period of borrowing another design team's battery whilst it was available to conduct all of our real world deployment and thus were not able to fully robust test this solution on the race car. It should be noted that we were able to get support from course staff to fix the battery the following day - it turned out to be a issue about the teleop USB and EM interference, but this could've been a different problem. This debugging process took around 20 hours straight between our team and it significantly slowed down our progress).
In the end, we realized again hardware debugging comes with it's own set of challenges and it was a valuable learning experience regardless.

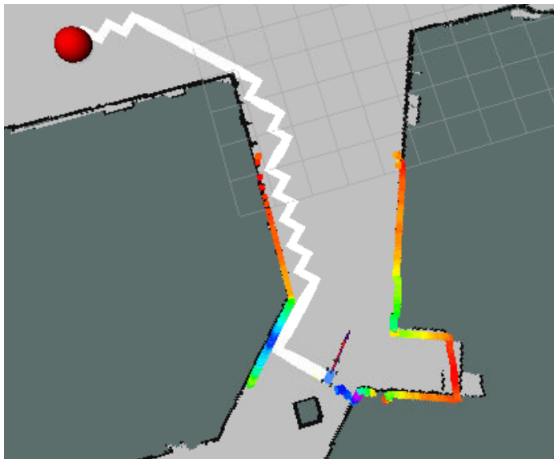

Fig. 21: Path Planning on the physical race car as visualised in RViz. Purple arrow is estimated odometry from localization, red arrow is car's internal odometry , rainbow dots are LiDAR scan data and the generated path is in white.

4 Conclusion

Ultimately, we achieved our primary goal in this lab of collecting enough data to viably compare a search based path planning method (A*) with a sampling based method (RRT). To achieve this, we developed and explored methods of improving collision detection robustness, such as map dilation, and we also wrote a pure pursuit algorithm to allow an autonomous vehicle to navigate along these paths to examine their feasibility. All of these methods were tested and refined for functionality within simulation.

Furthermore, we deployed our code upon a physical race car and were able to simultaneously localise and path plan. To do this, we used A* because of its reliability, despite being slightly slower, but have considered ways to improve the smoothness and speed of generation of the created path. Although not all of these improvements were implemented for a full end to end test due to delays from unexpected hardware issues, we confirmed that pure pursuit enabled the car to follow a generated path and set ourselves up in a position where our solution is complete enough to translate and improve upon for our final challenge.

4.1 Future Work

To make out path planning algorithms more effective, we plan to further optimize our A* code to decrease the long solve times, as well as integrate the path optimization code from RRT to reduce the number of points in the resultant path. This, combined with the already optimal distance paths that A* generates, should create traversable paths that are efficient and easy for the pure pursuit algorithm to follow.

5 Lessons Learned

5.1 Penny

In this lab I learned how localization, path planning, and path following, along with other concepts in the class can come together to build incredible things. I also again learned that hardware debugging is incredibly difficult. There were many cases where the simulation results were different between our team just because of the differences in our computer, and this only became a larger issue when we tried to deploy it to the car. For example, our car stopped responding to teleop completely at one point, which we eventually learned that it was because of EM interference, but the process of debugging was very difficult and definitely slowed down our progress. I'm looking forward to doing more path planning things and hopefully with less hardware debugging in the future!

5.2 Fritz

This lab was gave me a lot of insight into the complexity and power of combining separate software systems (trajectory planning and localization), but also the significant challenges inherent to doing so. We would not have been able to accomplish the progress we made in the lab without efficiently working together. I also feel as though the few portions of the lab we didn't complete to our satisfaction were not achieved due to circumstantial and hardware problems. I am quite proud of the work we've done and esecially our impressive performance in simulation. It was also gratifying that our localization solution from the previous lab held up quite well.

5.3 Nico

I feel as though a continual lesson in this class is to be more prepared for the unexpected. This lab, we had budgeted the time a full day before things were due to get the real world implementation up and running but a broken motor battery limited our testing window and left us reliant on other teams for support until we were able to get TA support in fixing it the following day. Hence on the final day, time pressure led to incomplete testing. In an ideal world, we'd have been further ahead to give ourselves more a buffer but considering we were still catching up from delays in earlier labs due to health and personal situations, we'd budgeted as much of a cushion as we could and yet it was unfortunately barely not enough.

That said, despite my disappointment in not being able to end to end record the car on the TA's challenge route, I was quite impressed in my team's ability to simultaneously explore RRT and A*. I feel as thought I learn a lot practically about both search and sampling path planning methods, and my ROS debugging skills continue to rapidly improve the deeper I have to dig into system files every lab. Seeing the paths appear in RViz was also oddly satisfying.

5.4 Yatin

I feel like I gained a full perspective of the autonomy stack during this lab. Linking perception, localization, and path planning was challenging, but also very informative. It made me realize how much I had learned about ROS. I still have a hard time with transformations, but I am happy that I was able to implement RRT on a real map of the Stata basement.

5.5 Kristine

This lab was incredibly helpful in understanding the full ecosystem of autonomous driving. Like a few other team members had mentioned, combining our previous methods from localization for path planning was challenging but rewarding. We had a lot of challenges while debugging, including an error in our

transformation matrix from the world to pixel frame that resulted in path finding working correctly sometimes. These set backs took a lot of time and effort to resolve on everyone's part and we are grateful for all the staff's support. Overall, I am satisfied that we implemented path finding (in simulation)successfully in a real map, especially since A* had been an algorithm as well as other methods were only explored theoretically in previous courses.