LAB 6 REPORT: Path Planning Algorithms for AV Navigation on a Map

Tags

TEAM 6 - Rulan Gu, Bradyn Lenning, Aimee Liu, Sruthi Parthasarathi, Tyrin-Ian Todd

6.4200 Robotics: Science and Systems

4.24.2024

I. Introduction

Author: Rulan Gu

A key feature of autonomous vehicles is that they're able to drive from one point to another without any teleoperated input. For example, the Tesla's automated driving system is now able to autonomously drive itself to a location by simply inputting the desired destination. It accomplishes this by discerning its current location on the map with GPS, planning the best path to its destination given available roads, and then following that path accurately.

For these purposes, an autonomous vehicle must be able to consistently generate an optimal path to a destination very quickly and follow it with high accuracy in order to prevent crashes and efficiently move to the goal.

So far, our robot has accomplished step one of this process: it is able to find its current location in the Stata basement map by using the particle filter we developed in Lab 5: Localization. However, it hasn't used that information to do anything yet. In this lab, our goal is to implement path planning and following algorithms so that the robot can use its location information to autonomously drive from one location to another. We must first create an algorithm to generate a path from the position found through localization, then another algorithm to follow the

path before we implement this into the robot. This flowchart of dependencies is shown in *Fig. 1.*

Fig. 1. Dependency flowchart of localization, path planning, and path follower. *Localization determines the location of the robot on the map, path planning finds a path from this current location to a destination on the map, and path trajectory moves the robot along this path to reach its destination.*

Our desired optimization parameters for this Lab parallels those for real autonomous vehicles— efficiency of path planning and accuracy of path following. As we develop our algorithms, we must optimize for runtime, optimality of path, and error reduction. These metrics will be used to quantify as “efficiency” and “accuracy” in order to evaluate each experimented algorithm.

In order to accomplish this difficult task, we broke it down into smaller goals and experimented with multiple path planning and path following algorithms to find the best one:

- Path Planning
  - Search-Based Algorithms
    - Dijkstra’s Path Planner
    - A* Path Planner
  - Sampling-Based Algorithm
    - RRT* Path Planner
- Path following

    - Pure pursuit follower
    - Motion predictive control (MPC) follower
- Real-world implementation

Edited by Aimee Liu

II. Path Planning Algorithms

Author: Sruthi Parthasarathi

The first task is to plan a path that the robot should follow, given a start location and a goal destination. In order to optimize this search for a viable and efficient path, we experimented with three algorithms, each of which are described in further detail. Section II.A explains the Search-Based Algorithms with the Dijkstra's Path Planner and the A* Path Planner, and Section II.B describes a Sampling-Based Algorithm with RRT*'s Path Planner. These algorithms are then evaluated with a quantitative metric of "efficiency" dependent on runtime and optimality in Section II.C.

Edited by Aimee Liu

A. Search-Based Algorithm

Author: Sruthi Parthasarathi

The first algorithm we discuss here is a search-based one — such algorithms require that the search space first be converted to a graph, consisting of discrete nodes and edges. To achieve this, we first took the set of all unoccupied grid cells on the map, and randomly sample V nodes (without replacement) from them. Then, rather than connecting all pairs of nodes with an edge, we opted for a sparser graph to improve computational efficiency. This was accomplished by only adding edges of length at most r_{nn}, a parameter we set for the nearest neighbor radius threshold.

Lastly, we added the grid cells corresponding to our start and end positions as nodes themselves, so that we could run shortest path algorithms between them.

Searching for Paths with Dijkstra

The first shortest path algorithm we implemented was Dijkstra's. The algorithm operates as follows:

1. Initialize a dictionary with the shortest path distances to each node from the start node, which we will denote as d_i for a node i. At the start, for all nodes $i \neq s$, $d_i = \infty$, and $d_s = 0$.
2. Initialize a dictionary with the predecessor of each node (currently set to be empty for all of the nodes).
3. Then, we repeat the following steps until t is removed from the set of nodes still being explored:
    a. Take the node i with the smallest current d_i that is still in the set of nodes being explored, and remove it.
    b. For each of that node's neighbors j that are still being explored, check if the path to j through i is shorter than the current shortest path to j, and update d_j accordingly. That is, set d_j to be $\min(d_j, d_i + d(i,j))$, where $d(a,b)$ is the Euclidean distance between a and b.
        i. If the path through i is shorter, update the predecessor of j to be i.
4. Start at t and backtrack along the shortest path to t by repeatedly finding the predecessor of the node until you reach s. Reversing this list of nodes provides the path in forward order.

This implementation worked well and generates an optimal path that avoids obstacles as shown in *Fig. 2*, but was restricted in terms of computational efficiency by the fact that all directions in the graph are explored equally, forcing most of the computations to be done for paths in the graph that are irrelevant to the path to our destination. In order to combat this, we improved upon our existing algorithm with A*.

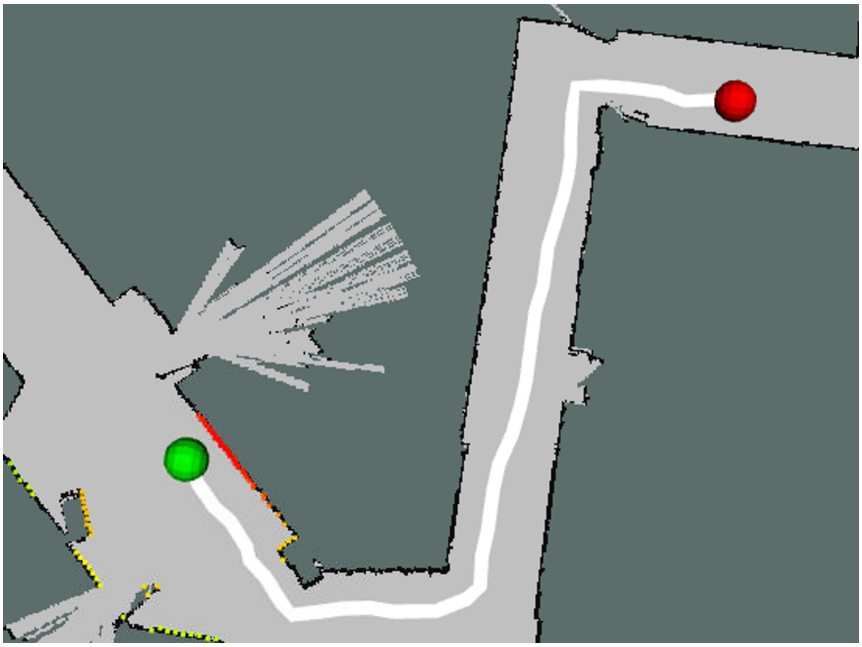

Fig. 2. Planned path generated by Dijkstra's algorithm with random sampling. *Path is generated from the green start point to the red goal position, as the path is generated between in white. The path avoids the obstacles and walls and reaches its intended goal.*

Improving Our Algorithm with A*

As mentioned above, the primary objective of A* is to additionally take into account some metric of whether a path is moving towards the desired goal or not, which helps streamline the process of finding the optimal path to a goal given a large search space of paths in the graph. We then remove the node with the smallest $d_i + h(i, t)$ at each step, where h is the heuristic that accounts for the estimated distance to the destination. When this value is smaller, we claim that the path is also more in the direction of the goal.

The graph in this case consists of all of the pixels in our map, and we allow edges between pixels that are adjacent or directly diagonal from each other. When only movement in these 8 directions are allowed, in order to weight diagonal and non-diagonal movements in the graph equally, the diagonal distance between two points is used for the heuristic. We can define this metric in (1):

$$h((x_1, y_1), (x_2, y_2)) = \max(|x_1 - x_2|, |y_1 - y_2|) \tag{1}$$

The heuristic function h takes in two points (x_1, y_1) and (x_2, y_2) and takes the maximum of the difference in x and difference in y.

To additionally improve the runtime, instead of random sampling, down sampling is used, where 1 point is taken for every 15 points in the x and y direction.

From running both algorithms in simulation, we discovered that A* was indeed faster than Dijkstra's in returning the optimal path generated in *Fig. 3.* The comparison between the two algorithms will be covered more in depth in Section II.C.

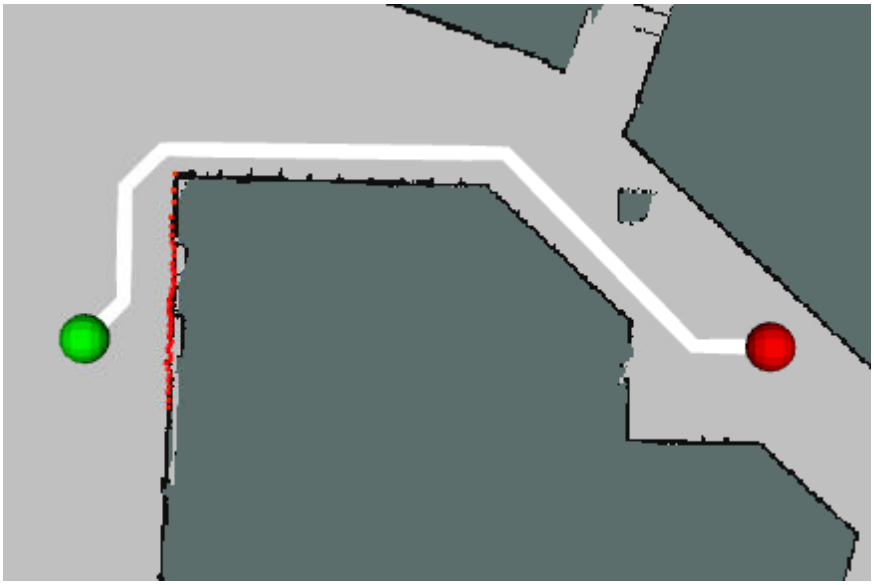

Fig. 3. Planned path generated by A* with down sampling. *Path is generated from the green start point to the red goal position, as the path is generated between in white. The path avoids the obstacles and walls and reaches its intended goal, similar to the original Dijkstra's algorithm. Down sampling causes this path to appear more geometric and less smooth. The runtime of this algorithm is near immediate and much faster in comparison to Dijkstra's.*

Edited by Aimee Liu

B. Sample-Based Algorithm

Author: Tyrin-Ian Todd

Sample based algorithms sample the continuous space of the map to build a path instead of breaking the map up into discrete parts and performing a search algorithm. This has the benefit of time-efficient pathfinding at the cost of optimality. Our implementation of Dijkstra's was actually a combination of a sampling and search based algorithm, as it sampled the continuous space to build a graph similar to a probabilistic roadmap. In this section, we will look at a pure sampling-based algorithm, the Rapidly Exploring Tree (RRT) algorithm.

Growing Paths using RRT

The RRT algorithm starts by initializing a tree at the start point. Next the RRT algorithm samples a random point on the map. It then "steers" the tree towards that point. When we steer, we first select the closest point to our sample, this will be the parent node. Then the algorithm adds a new point at a constant distance away from parent point that is in the direction of the sample. In our implementation we call this distance step. The algorithm will for this for n iterations or until the goal point is found. *Fig. 4* shows the pseudocode for this algorithm.

$$\textbf{RRT Algorithm}(x_{start}, x_{goal}, \text{step}, n)$$

RRT Algorithm (x_{start}, x_{goal}, step, n)

```
G.initialize(x_start)
for i = 1 to n do
    x_rand = Sample()
    x_near = near(x_rand, G)
    x_new = steer(x_rand, x_near, step_size)
    G.add_node(x_new)
    G.add_edge(x_new, x_near)
    if x_new = x_goal
        success()
```

Fig. 4. RRT algorithm pseudocode. *The algorithm takes x_{start} (start position), x_{goal} (goal position), step, and n (number of iterations) and generates a trajectory from x_{start} to x_{goal}.*

Improving with the RRT* Algorithm

RRT is not guaranteed to find an optimal path. To mitigate this we implemented the RRT* algorithm which creates more optimal paths than RRT, as shown in *Fig. 5*. RRT* extends the RRT algorithm with a rewire step. In the rewire, we look at all nodes within a search radius from the new node. Then for each node we note the cost from the new node to the root node through it. Lastly we set the parent of the new node to whichever node has the lowest cost. *Fig. 6*. visualizes this process of rewiring for RRT*. We repeat this process for n iterations to generate an optimal path as n goes to infinity.

Algorithm 2.
$T = (V, E) \leftarrow$ RRT*(z_{ini})

$T \leftarrow$ InitializeTree();
$T \leftarrow$ InsertNode($\emptyset$, z_{init}, T);
for i=0 to i=N ***do***
$z_{rand} \leftarrow$ Sample(i);
$z_{nearest} \leftarrow$ Nearest(T, z_{rand});
(z_{new}, U_{new}) $\leftarrow$ Steer ($z_{nearest}$, z_{rand});
if Obstaclefree(z_{new}) ***then***
    $z_{near} \leftarrow$ Near(T, z_{new}, $|V|$);
    $z_{min} \leftarrow$ Chooseparent (z_{near}, $z_{nearest}$, z_{new});
    $T \leftarrow$ InsertNode(z_{min}, z_{new}, T);
    $T \leftarrow$ Rewire (T, z_{near}, z_{min}, z_{new});
return T

Fig. 5. RRT* algorithm. *Extends the RRT algorithm with the rewire step to generate more optimal paths.*

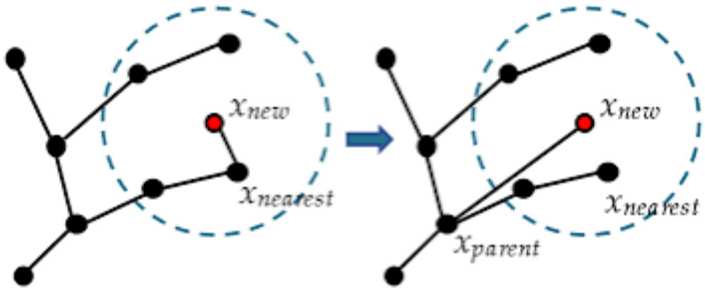

Fig. 6. Rewire step diagram. *As you can see in the photo all nodes within search radius are evaluated. The parent of x_{new} is set to the node with the lowest cost.*

The last optimization we added on top of the RRT* algorithm is a goal sampling rate. We choose a random number between 0 and 1 if this number is below our goal sampling rate we will choose the goal as our sample which will cause the tree to steer towards it. Additionally, instead of running for N iterations the algorithm concludes as soon as the goal is discovered. This helps the RRT* algorithm to converge quicker at the cost of optimality.

This algorithm is more likely to quickly generate an optimal path, however may not produce straight lines or give consistent solutions, as shown in *Fig. 7*. The RRT* may even fail to create paths at all. Further comparisons of this algorithm will be discussed in Section II.C.

Fig. 7. Planned path generated by RRT*. *Path is generated from the green start point to the red goal position, as the path is generated between in white. The path avoids the obstacles and walls and reaches its intended goal, however the path tends to be less straight and more inconsistent due to its branching nature. The runtime of this algorithm is quick for shorter paths but occasionally miss optimal solutions.*

Preventing Crashes with Map Dilation

The algorithm will occasionally generate a path that is too close to the obstacles. To improve safety, we are able to dilate the map as shown in *Fig. 8* to improve the safety of our paths. Dilating too little risks not seeing a wall properly, however dilating too much risks generating a wall from two close obstacles and removing an optimal path. A balance of dilating by a factor of 15 gave a suitable new, pre-processed map, which is used for all algorithms.

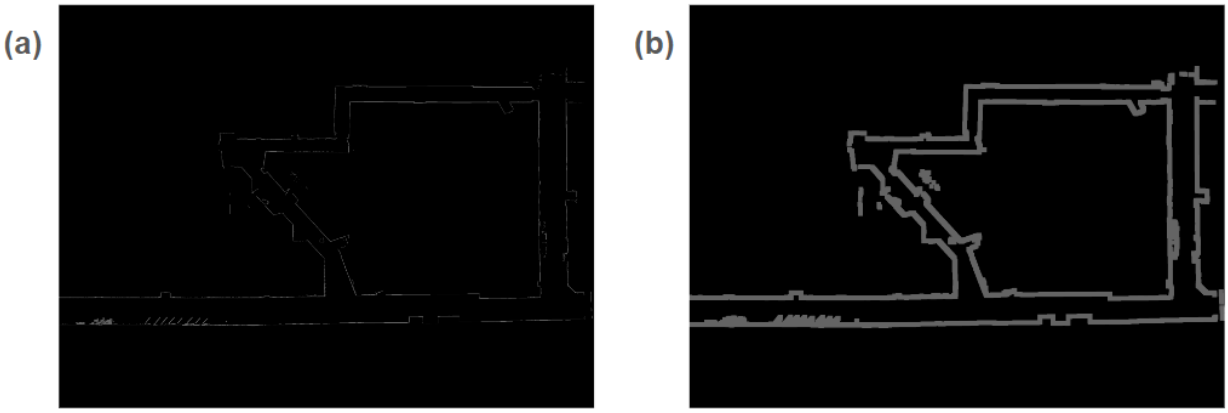

Fig. 8. (a) The map of obstacles for the MIT Stata Basement, and (b) the improved, preprocessed map of obstacles for the MIT Stata Basement. *Obstacle lines are more clear but areas with close obstacles become thinner and more difficult to pass through.*

Edited by Aimee Liu

C. Evaluating Our Path Planners

Author: Aimee Liu

In order to evaluate and compare how well each of our path planners work, we must use a quantitative metric to measure the efficiency of each algorithm. The best path planner should balance speed and optimality of its path, therefore the metric we use must decrease with path length and runtime. Path length is calculated by taking the trajectory points and summing the distances between

each point in the path on the map and runtime is recorded by logging the time the algorithm receives a goal point and begins and the time. However, path length varies with the location of the start and end points. A better metric to use instead can be the difference from the most optimal path length to the path length found. An extended Dijkstra's that uses all points from the occupancy grid would guarantee the most optimal path, therefore we will use this distance as our baseline. However, using all points takes an indefinitely long time as it generates edges between over 2 million points. We will instead sample 1 every 100 points and call this path closely optimal.

From this we can generate a simple formula, shown in (2).

$$\begin{aligned} Q_{EFF} &= \frac{1}{\text{opt. diff.} \times \text{runtime}} \\ &= \frac{1}{(\sum_{i=1}^{n-1} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2} - d) \cdot \Delta t} \end{aligned} \tag{2}$$

The "efficiency" quantitative metric, Q_{EFF}, examines the efficiency of the algorithm by computing the reciprocal of the distance between each point (x, y) on a n-length path, subtracted by the optimal distance, d, and multiplied by the difference in time from the start to the end of the algorithm, Δt.

For all algorithms, in order to maintain consistency, the same path will be followed in the Stata Basement map and the same number of points are used for down sampling or random sampling coordinates, one pixel for every 225 pixels. *Fig. 9* shows the start (green) and end (red) point being used for this evaluation, as well as an example path the path planner would generate between the two. The optimal distance of the path generated was 34.30 m. This path contains obstacles and walls that the path planners must avoid, which allows us to fully test the capabilities of our algorithms.

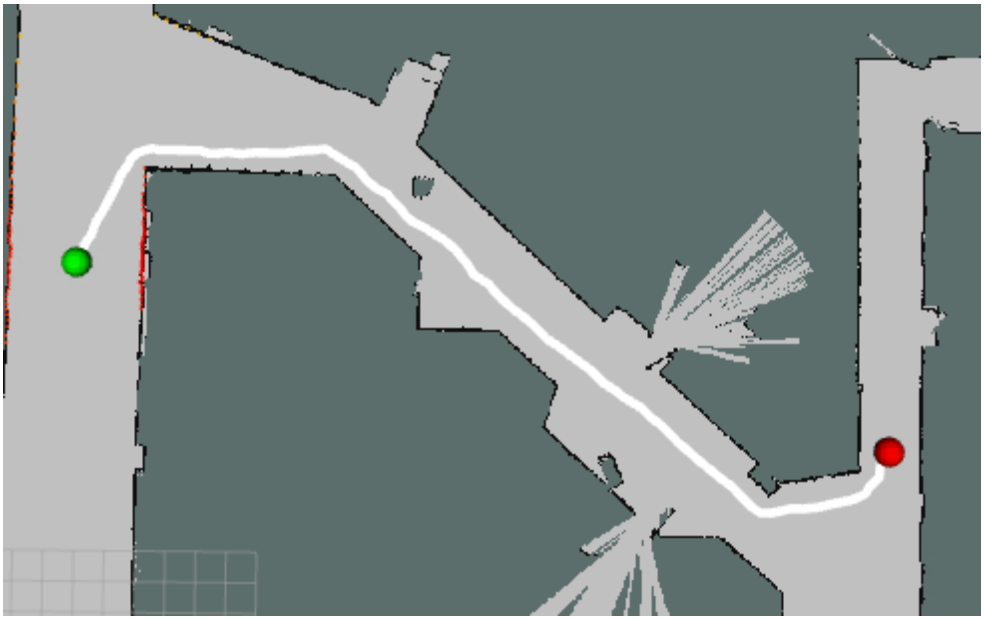

Fig. 9. Path used for evaluation of path planners. *The map uses the Stata Basement of MIT as a model. The green circle marker represents the start point and the red marker represents the end point of the path. The trajectory on the map is the path determined by the extended Dijkstra's algorithm.*

Efficiency of Dijkstra's, A*, and RRT*

The sampled Dijkstra's path planner had a total path distance of 35.21 m, which is 0.91 m more than the optimal acquired from the extended Dijkstra's algorithm. It's runtime was about 15.49 seconds, which gives it an efficiency of 0.0709. From these results, Dijkstra's can be determined as the path planner that creates the most optimal path but risks runtime.

The A* path planner had an average total path distance of 36.79 m, which is 2.49 m more than the optimal. It's runtime took an average of 0.05743 seconds, which gives it an efficiency of 6.9929. This makes A* the fastest path planner, though it sacrifices some optimality.

The RRT* path planner had an average total path distance of 49.66 m, which is 15.36 more than the optimal. It's runtime took an average of 71.70 seconds, which gives it an efficiency of 0.00091. In addition to this large run time and long path,

RRT* often was unable to find the path at all due to its random probabilistic nature as it often got stuck and stopped in corners. This caused it to fail 1 out of the 5 trials. This makes RRT* the most variable and inconsistent path planner.

Final Evaluation of Path Planners

Based on this evaluation and the quantitative metric used, A* is the most efficient algorithm for our purposes. *Fig. 10* plots the optimality difference vs. the runtime of each algorithm over 5 different trials each. Based on this graph, A* has the fastest run-time of the three algorithms while still maintaining a high efficiency. This gives A* the highest average efficiency of 6.9929, which is 2 magnitudes better than Dijkstra's efficiency and 4 magnitudes better than RRT*'s efficiency.

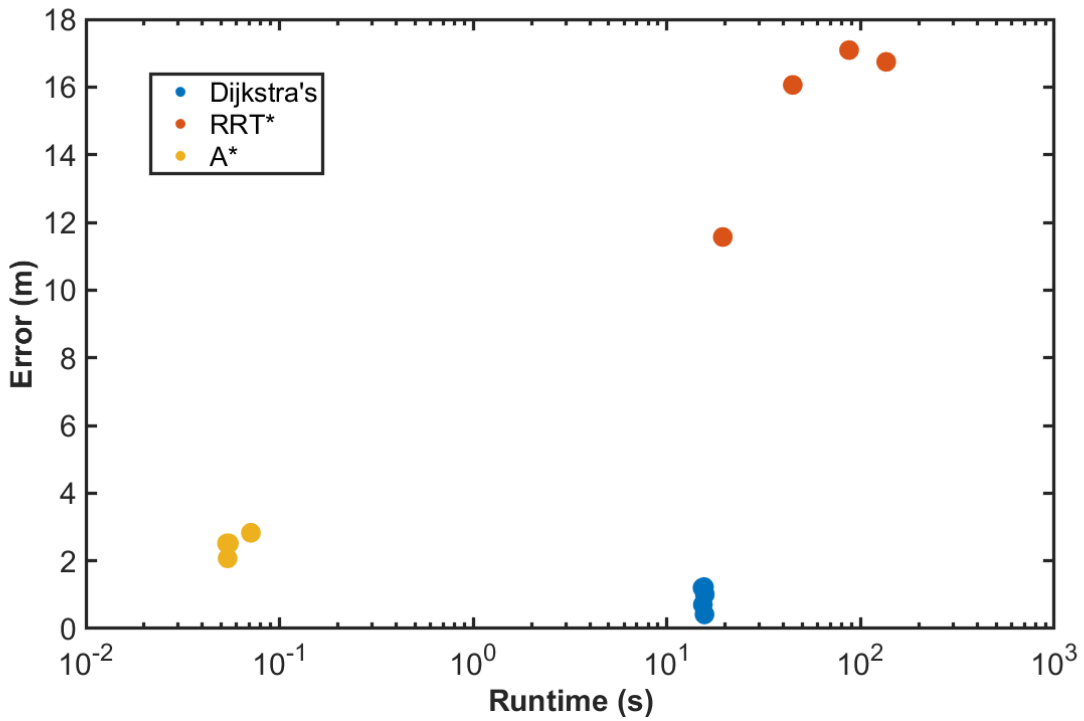

Fig. 10. Graph depicting the error from the optimal path (m) vs. runtime (s) for the 5 trials of each of the three algorithms. *Due to the large difference in runtime, a log scale was used for the x-axis. A*, shown in yellow, has the lowest runtime and a moderately low difference in optimality. Dijkstra's, in blue, has a higher runtime with the lowest optimality difference. RRT*, in orange, has the highest runtime and optimality difference of the three algorithms. An efficient algorithm*

should be closer to the bottom left of the graph, and the closest set of points to this area is A.*

Each algorithm has its strengths and weaknesses: Dijkstra's is guaranteed to provide the most optimal path if you are willing to wait the long runtime, A* is faster but sacrifices some additional distance, and RRT* becomes more efficient for larger maps but is not as optimal or speedy in smaller maps. Depending on the applications, any of the following algorithms may prove useful, however for our needs, A* is the best path planner for our robot and will be utilized as we move on to the next step and develop our path following algorithms in Section III.

III. Path Following Algorithms

Author: Rulan Gu

After planning a path, the robot must also be able to follow it. In order to find the best algorithm for the robot to follow a path, we experimented with two different ideas: pure pursuit and model predictive control. Then, we evaluated each follower by calculating the perpendicular distance between the robot's current position and the path to find which one was more accurate.

A. Pure Pursuit

Author: Rulan Gu

One of the path following algorithms was pure pursuit. The pure pursuit algorithm involves drawing a circle of radius *lookahead distance* around the robot, centered at its rear axle, and setting the target point to the intersection of that circle and the path. Then, the target point is used to calculate the appropriate steering angle for the robot to turn to. As the robot moves along the path, the target point and resulting steering angle are constantly updated based on the current location of the robot. The robot moves at a constant speed the whole time; pure pursuit only changes the steering angle. *Fig. 11* visualizes this algorithm as it looks ahead its received path.

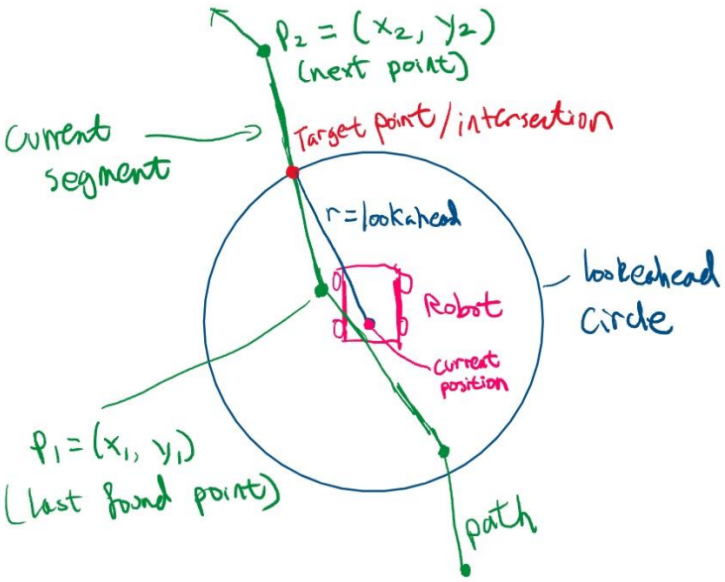

Fig. 11. Pure pursuit diagram. *The diagram displays the key components of the pure pursuit algorithm: the lookahead circle around the robot's current position, the path, the current path segment, and the target point or intersection.*

The first step to designing our pure pursuit algorithm is to create a logical flowchart that outlines the algorithm's edge-cases, such as what to do when no intersections are found. *Fig. 12* shows the flowchart that details the general logic of our pure pursuit algorithm.

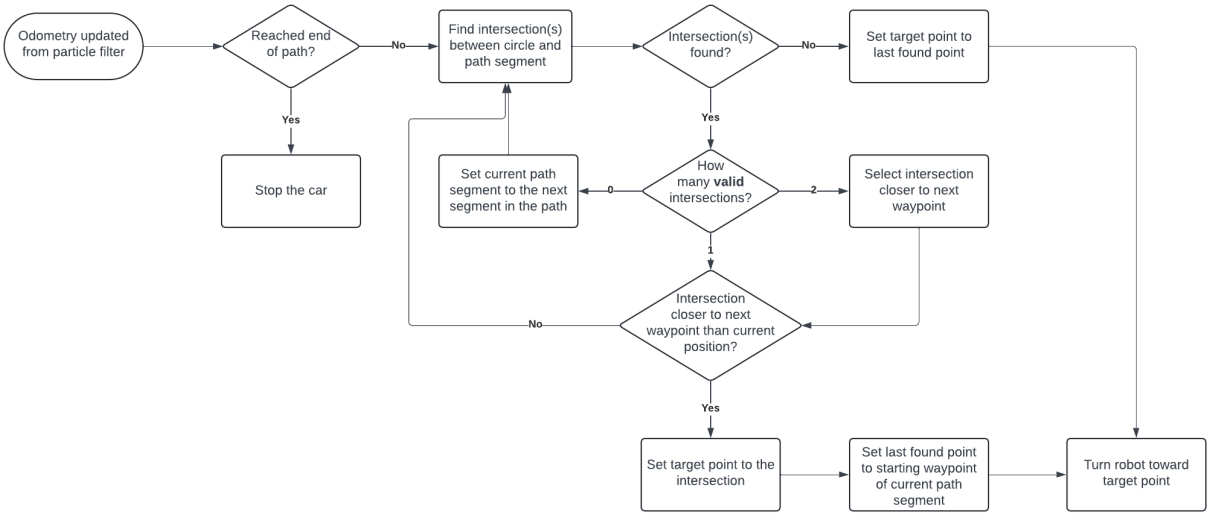

Fig. 12. Flowchart describing the logical flow of the pure pursuit algorithm. *Pure pursuit steers itself towards the target after all conditionals are met, where the end is not reached, an intersection is found between the circle and path closer to the next waypoint, and the target point is set to this intersection.*

A valid solution described in *Fig. 13* can be defined as an intersection that is within the rectangle defined by the endpoints of the current path segment, as only these solutions are actually on the segment.

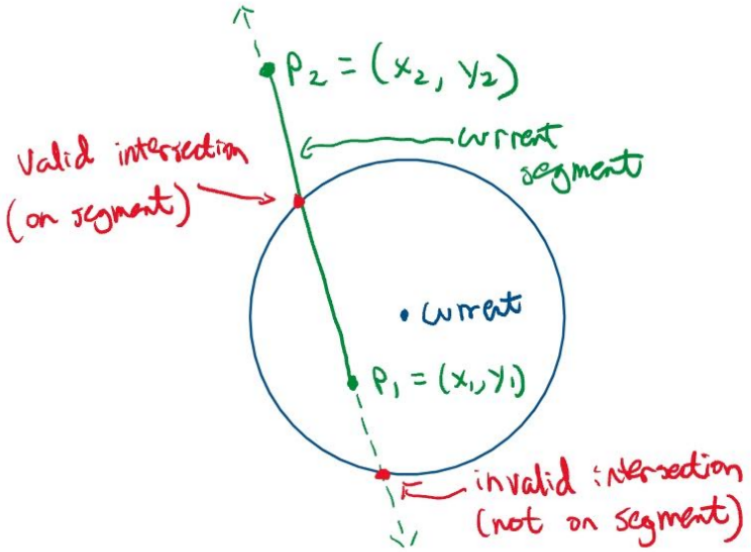

Fig. 13. Diagram depicting valid vs invalid solutions. *The valid solution is bounded by the rectangle with corners on p1 and p2 and thus on the line segment. The invalid solution is not on the segment and therefore is not an intersection between the circle and the current path segment.*

In order to find the target point (or intersection), the system of equations composed of (3), the circle equation, and (4), the line equation, is solved for x and y:

$$(x - current.x)^2 + (y - current.y)^2 = \text{lookahead}^2 \qquad (3)$$

$$y - y_1 = (\frac{y_2 - y_1}{x_2 - x_1}) * (x - x_1) \tag{4}$$

In (3) and (4), $current.x$ and $current.y$ are the current x and y positions of the robot. Lookahead is the radius of the circle that the algorithm looks for intersections on. (x_1, y_1) and (x_2, y_2) are the endpoints of the path segment that the algorithm is currently looking at.

Next, the robot must turn towards the target point. As shown *Fig. 14*, the robot has already found the target point but must calculate the steering angle. The goal is for the robot to turn its steering wheel to angle δ, so that it'll follow the path in the figure to eventually reach the target point (TP).

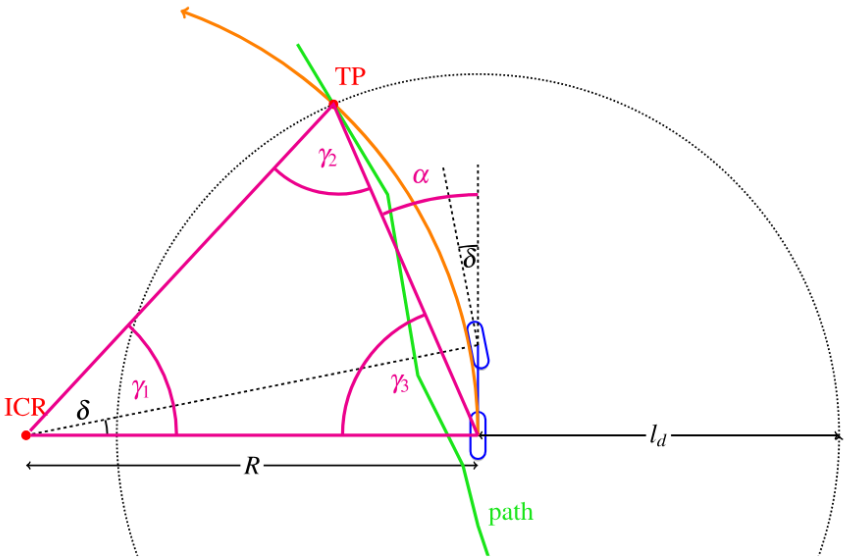

Fig. 14. Diagram showing the steering wheel angle. *This diagram shows the idea behind turning toward the target point: the robot is expected to turn in a curve (in red) with steering wheel angle δ to eventually reach the target point (TP).*

The formula for the angle error, α (shown in *Fig. 14*), is given by (5):

$$\alpha = tan^{-1}(\frac{TP.y - current.y}{TP.x - current.x}) - \theta \tag{5}$$

The formula for the steering angle, δ, is given by (6):

$$\delta = tan^{-1}(\frac{2 * \text{wheelbase} * sin(\alpha)}{\text{lookahead}}) \quad (6)$$

Note that for the new variables, TP is the target point, theta is the current absolute heading of the robot, and wheelbase is the distance between the front and rear axles of the car.

The robot turns its steering wheel to the angle δ at each pose update from the particle filter and consistently follows the path it receives. *Fig. 15* shows the pure pursuit working in simulation:

https://drive.google.com/file/d/1NTzjiXjUMrhXeGGAQ6PqTjypn48IyGru/view?usp=sharing

Fig. 15. Pure pursuit in simulation. *The lookahead radius and intersection point is displayed by the blue ring around the robot and the green dot ahead of it. The robot is capable of following the white path line effectively as it can navigate straight edges and wide curves with ease.*

Edited by Aimee Liu

B. Model Predictive Control (MPC)

Author: Bradyn Lenning

In addition to Pure Pursuit, we developed model predictive control for this lab. Model predictive control optimizes commanded controls for the next N time steps into the future. A model is created that can predict the state of the robot over the next N timesteps give an arbitrary input control. To implement this, the bicycle model for robot steering was used. Equation (7) is used for finding the next state given the current state and the applied turn angle and acceleration.

$$
\begin{aligned}
\text{Given:} \quad & x, y, \theta, v, \text{ acceleration, turn angle} \\
\text{Compute:} & \\
\text{Next } x: \quad & \text{next\_}x = x + v \cdot \cos(\theta) \cdot \text{time\_step} \\
\text{Next } y: \quad & \text{next\_}y = y + v \cdot \sin(\theta) \cdot \text{time\_step} \\
\text{Next } \theta: \quad & \text{next\_}\theta = \theta + \frac{v \cdot \tan(\text{turn\_angle})}{L_{\text{car\_wheelbase}}} \cdot \text{time\_step} \\
\text{Next } v: \quad & \text{next\_}v = v + acceleration \cdot \text{time\_step} \\
& \text{next\_}v = \min(\max(\text{next\_}v, \text{min\_velocity}), \text{max\_velocity})
\end{aligned}
\tag{7}
$$

The next state is calculated recursively in order to find the states for the next N timesteps for a given array of accelerations and turn angles. This array of accelerations and turn angles are then optimized to minimize a cost function. In order to not get stuck in local minima, combinations of accelerations ranging from negative to positive and turn angles left to right are sampled. The most promising combination is chosen by using the cost function, then this most promising combination is optimized over using the SciPy Minimize function.

The key to making model predictive control successful is carefully developing a cost function. For the parking controller from the Lab 4: Visual Servoing lab, this cost function was a function of the distance away and angle at which the cone was at. In addition, cost was added for regularizing turn angles and accelerations. In order to adapt this controller to this lab, the cone position was replaced with a point along the path. This point was chosen to be the point closest to the goal along the path that was within a given radius of the robot. *Fig. 16* demonstrates MPC working in simulation as it follows a given path.

https://drive.google.com/file/d/10Fsay2f2ZI2YQg0jf0R5W792qAn87o1F/view?usp=drive_link

https://drive.google.com/file/d/10Fsay2f2ZI2YQg0jf0R5W792qAn87o1F/view

Fig. 16. Model Predictive Control in simulation. *The robot uses MPC's cost functions to direct itself to the next point. Notice how the target position for the*

robot shown by the MPC path generated moves forward along the path as the robot moves forward.

In addition to finding the goal position differently, the cost function was updated to assign an additional cost to having the projected path be too close to a wall. This helps to prevent collisions. The distance from each point in the projected path to each point in the lidar data was calculated and the cost associated was proportional to the minimum distance found, shown in *Fig. 17*.

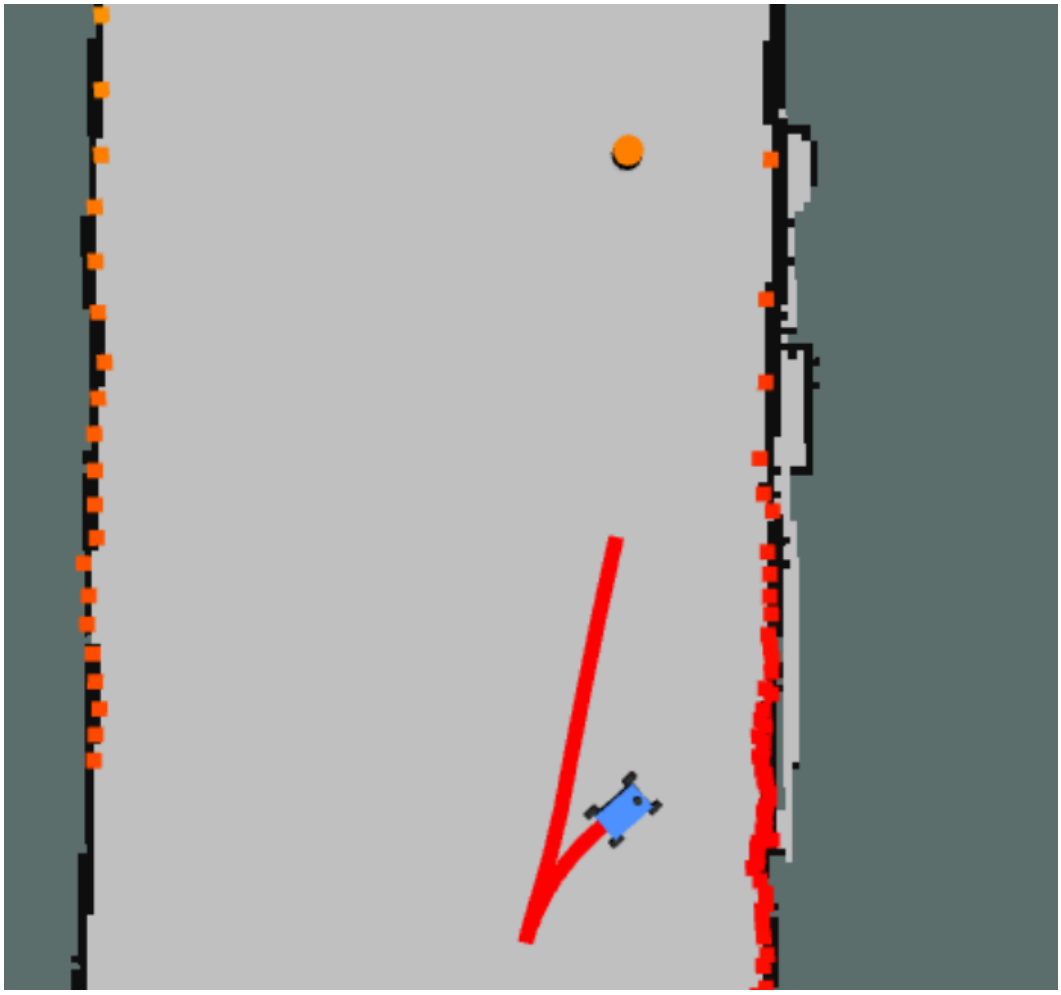

Fig. 17. Model Predictive Control avoiding crashing into the wall. *The planned path is shown in red. MPC chooses to first back up the drive forwards to the orange goal point due to the cost associated with driving too close to the wall.*

Edited by Aimee Liu

C. Evaluating Our Path Followers

Author: Aimee Liu

To evaluate our path followers, we must quantify its accuracy to following the desired trajectory given by the path planner. We can do so by tracking the distance of the robot from the line it is aiming to follow. The best way to do this consistently is finding the perpendicular distance from the robot to the piecewise line of the trajectory that the robot is between. This can be calculated using the formula displayed in (8) [2].

$$Error = \frac{|as + bt + c|}{\sqrt{a^2 + b^2}} \tag{8}$$

The equation describes the error of the robot at a certain timestep where the robot is at point (s, t) following a piecewise line $ax + by + c = 0$. The closer the error is to 0, the more "accurate" the algorithm is at making the robot stay on path. We can display this by graphing this error as the robot runs through a path and estimating the average error of its run.

Accuracy of Pure Pursuit and Model Predictive Control

Pure pursuit maintains a very low error close to 0 throughout its run. There are slight increases in error as the robot turns a corner, however it overall has an average error that maintains close to zero. *Fig. 18*. shows a graph of the error using Pure Pursuit in the middle of the robot's run.

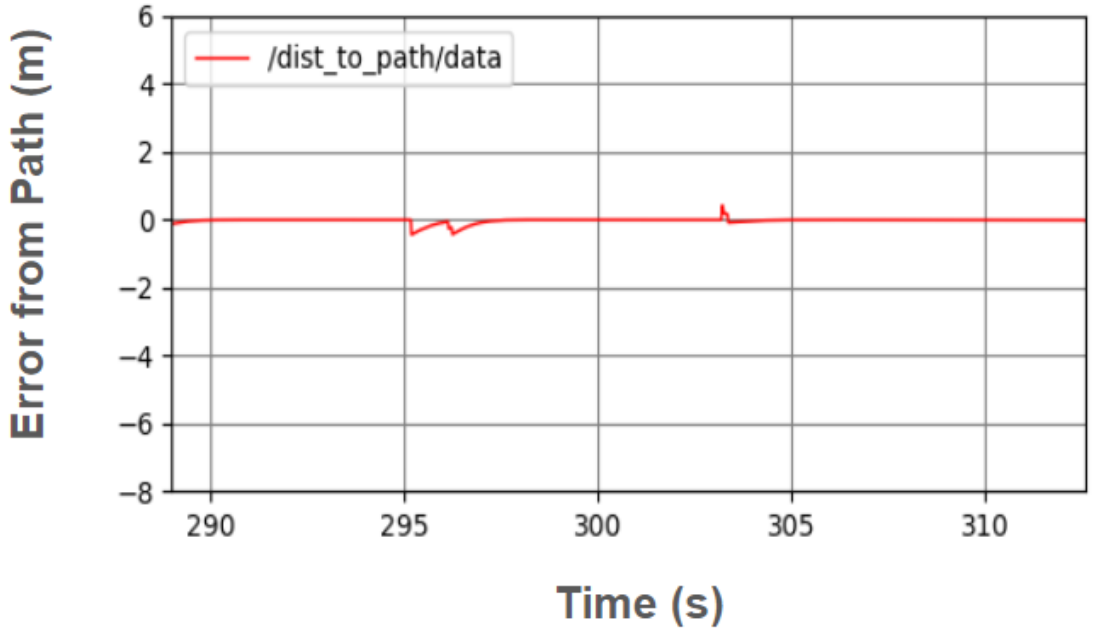

Fig. 19. Graph of the error distance (m) over time (s) for Pure Pursuit error. *The graph measures error from the path in meters over the y-axis and simulation time in seconds over the x-axis as the robot follows a path with Pure Pursuit. The path consistently stays at an average very near 0, except for when it spikes during turns or slight curves in the path. The spikes are very low (< 0.2 m) and are tolerable for our purposes, making Pure Pursuit a very optimal path follower.*

MPC also maintains a relatively low error, averaging at around 0.2 m, however it has much greater spikes when the robot attempts to correct itself when reaching a difficult corner or overshoots, resulting in much oscillations in error. This is likely due to additional costs to prevent the robot from crashing into walls that make the robot create new, more costly routes in exchange for safety. *Fig. 19* shows an example of this in a graph of the error using MPC in the middle of the robot's run.

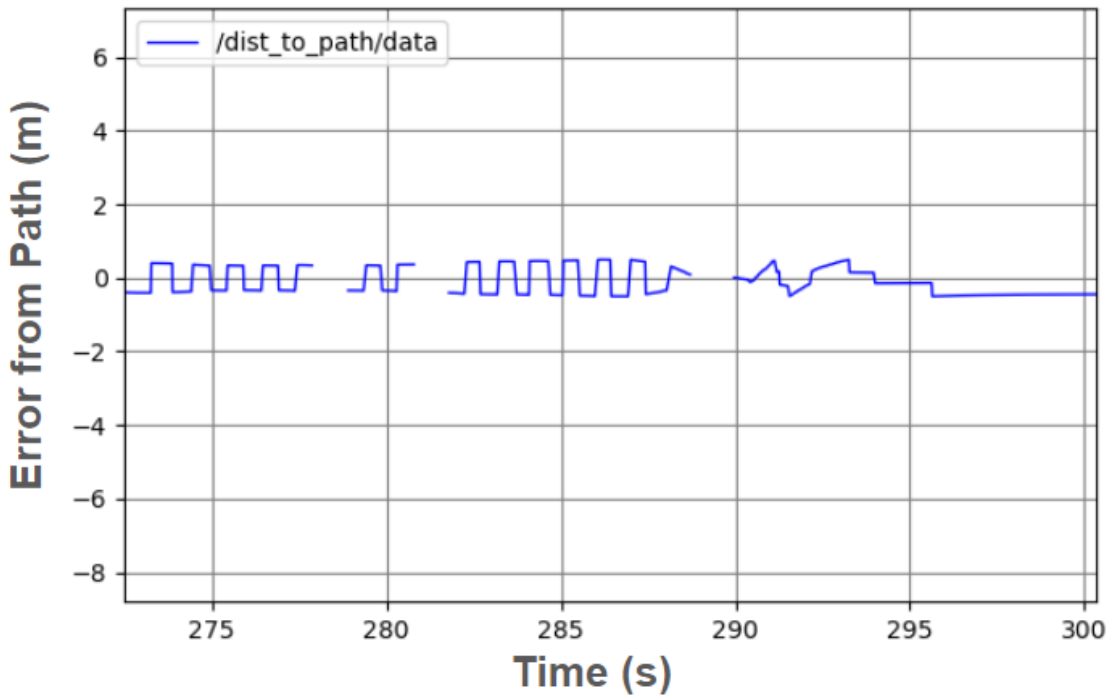

Fig. 19. Graph of the error distance (m) over time (s) for Model Predictive Control (MPC) error. *The graph measures error from the path in meters over the y-axis and simulation time in seconds over the x-axis as the robot follows a path with MPC. The path often oscillates or stays constant distance away from intended path, but has an average error of roughly 0.2 m from the path. These errors are likely due to the MPC balancing the other cost functions along with following the path.*

Final Evaluation of Path Followers

Pure pursuit and MPC both have their own pros and cons to implementation. MPC is much more inaccurate compared to pure pursuit, however it allows us to use its additional cost functions to improve the safety of the robot and better handle obstacles if there are any errors with the planned path, which allows it to correct itself and continue on its own instead of stopping completely. For our current application, pure pursuit's speed and accuracy are more ideal, however MPC can still be used as a viable and safer option if we decide to pursue a more careful model.

Edited by Bradyn Lenning and Rulan Gu

IV. Synthesis and Robot Integration

Author: Bradyn Lenning

Before implementing the system onto the robot, a lot of work was put into getting all of the systems working together in simulation. To start, we used the path followers to follow paths generated by the path planners instead of the sample paths. This was tested extensively using the ground truth data in simulation.

In the real world, there is no way to get the ground truth pose of the robot. This means that it was necessary to get our system working with the localization particle filter from the previous lab. Accomplishing this was straightforward as it just required using a different launch file that had path follower use estimated odometry from localization instead of the ground truth data.

Implementing this system onto the physical robot was in theory very similar to running it in simulation, but integration proved to be exceedingly difficult. After debugging issues with launch files using incorrect parameters on the physical robot, issues with the particle filter localization and the path follower started to pop up. After fixing most of the known problems, the robot was able to localize and move along a path to a goal location as shown in
Fig. 20.

https://drive.google.com/file/d/193Z1W32eqsQCIWtnSI7n-ZIyXv0inswX/view?usp=drive_link

https://drive.google.com/file/d/193Z1W32eqsQCIWtnSI7n-ZIyXv0inswX/view?usp=drive_link

Fig. 20. Model Predictive Control working in the real world. Though the robot moves slowly and has wavering localization, we can still see the robot moving

along the path while localizing successfully.

Edited by Aimee Liu

IV. Conclusion

Author: Rulan Gu

This report presents our implementation for path planning and following on our autonomous robot. Our robot is now able to drive from a starting location to a goal location in the Stata basement. To do this, we first implemented three different path planning algorithms, both search-based and sampling-based, and chose the best one. Then, we implemented two path following algorithms, pure pursuit and MPC, so that the robot could follow its planned path.

Further work still needs to be done on optimizing the pure pursuit follower. Although it works very well in simulation, the robot seems to get “confused” in the real world and often drives off the path because it is looking at the wrong target point. Further debugging and investigation is needed for the pure pursuit follower to work smoothly in the real world.

Overall, our path planning algorithm has been successfully implemented, but our following algorithm still needs some work. Once completed, these features will allow our robot to quickly and autonomously drive from one location to the next.

Lessons Learned

Each member of the team contributed heavily to the success of the project. As a result, we each learned various lessons in both the technical and communicative aspects of the lab.

Author: Rulan Gu

This lab was very interesting and taught me about a lot about path planning and following algorithms. It showed that even with this complicated and daunting technically challenge, it could be tackled quite quickly by assigning individuals components to teammates and then integrating it together.

Author: Bradyn Lenning

This lab taught me a lot about the importance of testing individual components rigorously before trying to integrate. Integration on the robot proved to be extremely challenging with unexpected bugs showing up in several of the individual components. Debugging these would have been much easier before integration, though it is difficult as many of these bugs do not show up until integration. Maybe a solution would be trying to test individual components on the robot.

Author: Aimee Liu

This lab was very extensive and interesting as it taught much about how an AV navigates its environment from a map and expands from the localization lab previously. I found path planning particularly interesting and creating metrics to analyze all three experimented methods very informative. I also learned the importance of working with a team and assisting each other whenever difficulties arise or to meet deadlines.

Author: Sruthi Parthasarathi

This lab was very neat in the way it broke a daunting task into several independent components. As a result, I learned a lot about how to think about larger systems in such a modular way, and this lent itself well to delegating tasks based on the strengths of each individual team member.

Author: Tyrin Todd

During this lab I learned about how to plan a path with the RRT algorithm and compare that algorithm to other path planning algorithms. I thought the tradeoffs (time, how optimal the path was) between each algorithm was very interesting.

References

[1] “Lab 6: Path Planning” Robotics: Science and Systems (MIT Course), Apr. 23, 2024. Accessed: Apr. 26, 2024. [Online]. Available: https://github.com/mit-rss/path_planning

[2] “Distance of a Point From a Line - Definition, Derivation, Examples,” Cuemath. Accessed: Apr. 26, 2024. [Online]. Available: https://www.cuemath.com/geometry/distance-of-a-point-from-a-line/