RSS Lab 6 Report: Path Planning and Pure Pursuit

RSS Team 9: SDWCH
William Chen, Henry Hu, Christina Jung, Silvia Knappe, Diego Mendieta

April 25, 2020

Contents

1 Introduction **3**

2 Technical approach **3**

2.1 Assessment of motion planning algorithms 3

2.2 Path Planning: RRT* 4

2.3 Path Following: Pure Pursuit 8

2.3.1 Finding the Closest Path Segment 8

2.3.2 Determining the Target Point 9

2.3.3 Determining the Steering Angle 10

3 Experimental evaluation **11**

3.1 RRT* evaluation 12

3.1.1 Performance 12

3.1.2 Optimality 13

3.1.3 Path smoothness 14

3.2 Pure pursuit evaluation 16

4 Conclusion **17**

5 Lessons Learned **18**

1 Introduction

Author: Christina

Our team has been working on developing an autonomous racecar by implementing different algorithms which will build up to the final challenge. So far, we have implemented a wall following controller and a safety controller in Lab 3, developed a parking controller and a line following controller in Lab 4, and localized the racecar in Lab 5.

For Lab 6, which is our last lab before the final challenge, we were tasked with planning and control. Given a destination, the robot should be able to determine the path to the destination and follow it. Lab 6 is composed of multiple parts: path planning, pure pursuit, and integration. Specifically, we used a rapidly-exploring random tree algorithm (RRT*) for the robot to plan paths through an environment, and pure pursuit algorithm to proceed to follow the path. Finally, we integrated the two components to enable real-time path planning and execution. The work done in Lab 6 will be crucial to the final challenge, as the algorithms we develop are key components of autonomous operation.

2 Technical approach

Our simulated robot uses RRT* to plan paths through an environment, and pure pursuit in order to follow the planned path. In this section we will assess our choice of RRT* as our path planning algorithm, then go in depth into our implementation of RRT* and pure pursuit.

2.1 Assessment of motion planning algorithms

Author: Diego

To plan a path, there are two main types of algorithms that can be used: search and sampling algorithms. Search algorithms treat locations in a map as discrete nodes in a graph and use graph-based algorithms to find the shortest path. Sampling algorithms, on the other hand, use a probabilistic component to explore the graph for a valid path.

We did not consider any search algorithms because they generally perform slowly and because they are more difficult to implement for a continuous graph. The results of graph search algorithms are also very sensitive to the resolution of the rasterization.

Therefore, we considered three sampling algorithms for path planning: PRM, RRT and RRT*. To decide which specific algorithm to use, we evaluated the following properties for our three candidates.

- **Asymptotic optimality:** The ability of the result of the algorithm to approach the optimal path as the runtime grows
- **Query structure:** number of planning tasks we can execute
- **Necessity of search:** whether the graph will require a search algorithm after being constructed

A summary of our findings in shown in table 1.

Table 1: Summary of considerations for selection of sampling algorithm

Algorithm	Asymptotic optimality	Query structure	Necessity of search
PRM	No	Multi-query	Required
RRT	No	Single query	Not required
RRT*	Yes	Single query	Not required

We concluded that RRT* would be the most suitable candidate for our use given that it is asymptotically optimal and does not require an additional search algorithm which would result in a better runtime.

2.2 Path Planning: RRT*

Author: William

The optimal rapidly-exploring random tree (RRT*) algorithm is a randomized algorithm which finds a trajectory between desired start and end goal points in configuration space. As the name suggests, it works by randomly growing a tree of nodes, each representing potential robot configurations, while maintaining parent-child connections between them to represent possible trajectories to get the robot from one configuration to another. Each node has an associated cost – a measurement of difficulty for the robot to get from the start to the configuration represented by the considered node by traversing the tree graph. Critically, for each node created, nearby nodes are ”rewired” to optimize their costs. In doing this repeatedly, the algorithm generates paths from a given start and end point. As the number of nodes goes to infinity, the path generated converges to the optimal one. In practice, a large number of nodes will often result in a good approximation of the best path. What results is an algorithm with relatively low time complexity and good empirical performance.

It is convenient to define some notation for these nodes. In general, nodes will be denoted by $\mathbf{n}$. Their associated configuration will be denoted by $\mathbf{n}$.value, their parent by $\mathbf{n}$.parent, and their cost by $\mathbf{n}$.cost. Finally, the set of all nodes will be called $\mathbf{N}$.

We used RRT* to plan trajectories on a two dimensional map for our robot to follow. Thus, the configuration space that the algorithm searched over is that of the Cartesian plane – (x, y) points on a map. Moreover, the aforementioned nodes' connections can be interpreted as being the physical paths through the map that the robot will be following. Finally, the desired start and end goal configurations are the desired starting and ending locations of the robot. With these problem parameter definitions given, the RRT* algorithm can be defined as well.

Prior to starting the main loop of RRT*, some preprocessing is completed. Firstly, the algorithm requires both a starting and ending goal location, denoted $\mathbf{p}_{\text{start}} = (x_{\text{start}},\ y_{\text{start}})$ and $\mathbf{p}_{\text{end}} = (x_{\text{end}},\ y_{\text{end}})$ respectively. Moreover, as the algorithm will be growing a tree, it requires a root node. Said object is created before the full algorithm is run, with a configuration value of $\mathbf{p}_{\text{start}}$. Naturally, as this node is at the start location, the "cost" is 0. With this node initialized, the algorithm has both an end configuration and a starting node for the tree to branch out from, meaning the main RRT* loop can be executed.

The first step in the RRT* iterative procedure is to uniformly sample a location in free space. Our map information was given as a ROS OccupancyGrid message. This datatype discretizes the map into pixels, with values of 0 indicating free space, 100 indicating occupied space, and -1 indicating unobserved space (e.g. if a wall occludes a given region). These messages also provide the transformation information necessary to convert from integral pixel coordinates to Cartesian. Thus, random Cartesian free space coordinates can be generated via uniform random number generator. Any pixel with a value of zero counts as being in free space, so the random number generation function can be used to pick an arbitrary zero-valued pixel's coordinates. Then, it can be converted into Cartesian, thus giving a uniformly-sampled free space Cartesian point, denoted $\mathbf{p}_{\text{free}}$. This is the location that the tree will try to grow towards.

In order to get the tree to explore towards $\mathbf{p}_{\text{end}}$, this free space sampling procedure has an additional bias term. Said term, denoted goal_bias $\in (0,\ 1)$, represents the probability that, instead of sampling a random free space location, $\mathbf{p}_{\text{free}}$ is set equal to $\mathbf{p}_{\text{end}}$. That is, goal_bias is the probability that the tree will try to grow directly towards the goal point, if possible, thus allowing the tree to reach the end quickly. Once the tree does grow to reach $\mathbf{p}_{\text{end}}$, goal_bias is set to zero, so sampling once again becomes uniform, allowing the tree to spread and optimize further.

After finding a $\mathbf{p}_{\text{free}}$ value, the tree then expands towards the new free space point. To do so, several auxiliary functions are required:

cost(start, end) Calculates a numeric ”cost” to get from a starting configuration to an ending one. This cost function will be denoted $C(\text{start, end})$ For 2D path planning, we chose L_2 Euclidean distance.

test_free_space(path) Checks to see if a given path through the Cartesian plane goes through any occupied or unobserved grid cells. Returns true if the path is unobstructed, false otherwise.

nearest(config) Returns the closest node to a given configuration, using $C()$ as a metric for closeness. That is:

$$\mathbf{n}_{\text{nearest}} = \arg\min_{\mathbf{n}_i} C(\mathbf{n}_i.\text{value, config}) \tag{1}$$

where $\mathbf{n}_i$ denotes the set of all existing nodes (indexed by i).

steer(start, end) Generate a free space path from configuration start to another configuration that is closer to end than start is. Returns that final configuration.

Given these functions, define:

$$\mathbf{n}_{\text{nearest}} = \textbf{nearest}(\mathbf{p}_{\text{free}}) \tag{2}$$

This is the node closest to the randomly sampled free point. Then, to find the position of the new node:

$$\mathbf{p}_{\text{new}} = \textbf{steer}(\mathbf{n}_{\text{nearest}}.\text{value, } \mathbf{p}_{\text{free}}) \tag{3}$$

This gives a new configuration closer to $\mathbf{p}_{\text{free}}$ than $\mathbf{n}_{\text{nearest}}$.value, and will be the configuration of the new node. If, when using **steer**(), no path can be generated, then pick new $\mathbf{p}_{\text{free}}$ configurations until a $\mathbf{p}_{\text{new}}$ is found.

In order to ensure optimality, RRT* tries to find an optimal parent for a node at $\mathbf{p}_{\text{new}}$. Of the nodes spatially close to $\mathbf{p}_{\text{new}}$, the algorithm searches for whichever node can connect to the new node located at $\mathbf{p}_{\text{new}}$ and would give it the minimum cost. To do this, define the following helper function:

nearby(config) Returns a list such that all nodes in the list satisfy the following conditions. First, the cost connecting each node to the specified configuration must be less than some amount:

$$\text{cost}(\mathbf{n}.\text{value}, config) = C(\mathbf{n}.\text{value}, config) < \Delta \tag{4}$$

where Δ is some maximum cost. Secondly, all nodes in the list must be able to connect to the specified configuration. For each node, there must exist some path to config such that test_free_space(path) returns true.

Using this function, define the list of nearby, connectable nodes as follows:

$$\mathbf{N}_{\text{nearby}} = \textbf{nearby}(\mathbf{p}_{\text{free}}) \tag{5}$$

Thus, $\mathbf{N}_{\text{nearby}}$ is the list of all nodes that could potentially be a parent for the new node. As defined prior, the optimal parent would be one that gives the new node the lowest cost:

$$\mathbf{n}_{\text{parent}} = \underset{\mathbf{n}_i \in \mathbf{N}_{\text{nearby}}}{\arg\min}\ \mathbf{n}_i.\text{cost} + C(\mathbf{n}_i.\text{value},\ \mathbf{p}_{\text{new}}) \tag{6}$$

The new node can finally be created with the following parameters:
$\mathbf{n}_{\text{new}}.\text{value} = \mathbf{p}_{\text{new}}$
$\mathbf{n}_{\text{new}}.\text{cost} = \mathbf{n}_{\text{parent}}.\text{cost} + C(\mathbf{n}_{\text{parent}}.\text{value},\ \mathbf{p}_{\text{new}})$
$\mathbf{n}_{\text{new}}.\text{parent} = \mathbf{n}_{\text{parent}}$

Lastly, nodes near the new one are "rewired" to optimize their costs. If any of the nodes in $\mathbf{N}_{\text{nearby}}$ (excluding $\mathbf{n}_{\text{parent}}$) would have a lower cost by being a child node of $\mathbf{n}_{\text{new}}$, then said change in parent node occurs. That is, for each $\mathbf{n}_{\text{nearby}} \in \mathbf{N}_{\text{nearby}}$, if

$$\mathbf{n}_{\text{new}}.\text{cost} + \textbf{cost}(\mathbf{n}_{\text{new}}.\text{value},\ \mathbf{n}_{\text{nearby}}.\text{value}) < \mathbf{n}_{\text{nearby}}.\text{cost} \tag{7}$$

then the parent of the considered node should be switched to $\mathbf{n}_{\text{new}}$ and its cost updated. Likewise, all descendants of $\mathbf{n}_{\text{new}}$ must have their costs updated recursively, to reflect $\mathbf{n}_{\text{new}}$'s new parentage.

This sampling, steering, parent-selection, and rewiring process is the core loop of RRT*. A path is found once a node is sufficiently close to the end goal configuration. However, as the rewiring process allows for costs to be reduced retroactively, this means running this iterative procedure after successfully finding a path will improve the efficiency of the generated path. Once the loop is broken, the full path can be found by recursively considering a node's parents (starting at the ending node). This thus produces a sequence of configurations from the start to the end representing a path through free space, as desired for path planning.

2.3 Path Following: Pure Pursuit

Author: Silvia Knappe

Pure pursuit is a path tracking algorithm that controls the steering angle of the robot so that the robot follows a path. The algorithm is named pure pursuit since the robot picks a target point to pursue, so the robot is constantly chasing a target point on the path in front of it. Using pure pursuit, our task was to follow the path given to the robot with the RRT* algorithm described above. In order to find a target point on the path to pursue, there were 2 steps we needed to take: finding the closest path segment to the robot, and obtaining the target point given the segment. After a target point is acquired, we were then able to calculate the appropriate steering angle for the robot so that it could follow the path.

2.3.1 Finding the Closest Path Segment

From the RRT* algorithm, we obtained a list of points that formed the path that the robot should follow. As described above, the robot needs to find a target point on this path, but to do so, the robot must first know which segment of the path it is closest to. To find the closest segment, we project the location of the robot onto each path segment, and find the distance between the robot and the projected point. The segment that corresponds with the shortest distance is the closest segment to the robot.[1].

Let us consider the case with one line segment from point v to w, and robot at some point p. If we consider the line between v and w, which can be parameterized as $v + t(w - v)$. The projection of p onto the line vw occurs at the value t that is shown in equation 8, where $p - v$ and $w - v$ are vectors:

$$t = \frac{(p - v) \cdot (w - v)}{|w - v|^2} \tag{8}$$

Since we only want to consider the point t that falls on the line segment vw and not the line vw, we only consider values of $t \in [0, 1]$.

We then obtain the actual value of the projected point on the segment p_{proj} by substituting the value of t into $v + t(w - v)$, and take the distance between p and p_{proj} to get the distance between p and vw.

This process needs to be done for all segments on the path, and the closest path segment to the robot is the one with the minimum distance between the

[1] The math used for section 2.2.1 comes from a stackoverflow post linked from the lab handout: `https://stackoverflow.com/questions/849211/shortest-distance-between-a-point-and-a-line-segment/1501725#1501725`

robot location and the projected point. Since the given path could have many points, the process of finding the closest line segment was heavily optimized using numpy. Once the robot knows which segment is closest, it can start finding the target point to pursue.

2.3.2 Determining the Target Point

When executing pure pursuit, the robot looks a certain distance ahead of it to find a target point to follow. This distance is called the lookahead distance, or r. The lookahead distance forms a circle of radius r around the robot at which it is checking for a point to follow. We can find the lookahead point by checking where the circle of radius r intersects with the closest path segment we found above. Let us consider the robot at some location q with the closest line segment going from p_1 to p_2 as shown in Fig. 1 below.[2]

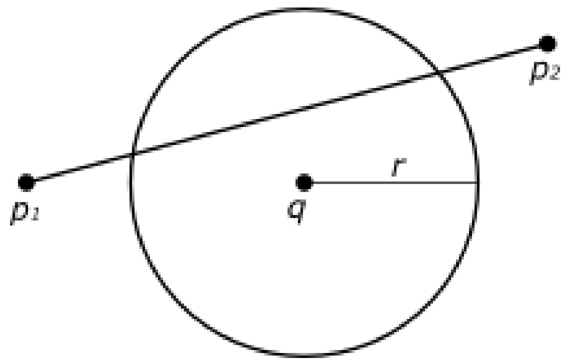

Figure 1: Robot with lookahead distance intersecting path segment

We can parameterize the path segment to be

$$p_1 + t(p_2 - p_1) \tag{9}$$

for some $t \in [0, 1]$ We want to find some point x that lies on the circle and intersect with the path segment. x lies on the circle if

$$|x - q| = r \tag{10}$$

So, the segment from p_1 to p_2 intersect the circle when

$$|p_1 + tv - q|^2 = r^2 \tag{11}$$

[2]Image for Fig. 1 and math corresponding to section 2.2.2 is also from a link from the lab: `https://codereview.stackexchange.com/questions/86421/line-segment-to-circle-collision-algorithm/86428#86428`

We can then use properties of dot products and expand the equation to get a quadratic equation where we can solve for t:

$$t^2(v \cdot v) + t * 2(v \cdot (p_1 - q)) + (p_1 \cdot p_1 + q \cdot q - 2p_1 \cdot q - r^2) = 0 \tag{12}$$

This quadratic, in the form $at^2 + bt + c = 0$ has coefficients $a = v \cdot v$, $b = 2(v \cdot (p_1 - q))$, and $c = (p_1 \cdot p_1 + q \cdot q - 2p_1 \cdot q - r^2)$.

The discriminant of this quadratic equation is then

$$d = b^2 - 4ac \tag{13}$$

Based on the value of the discriminant, we can tell if the segment intersects with the circle or not, and how. There are 4 different cases:

a. $d <= 0$: In this case, the circle and segment do not intersect. In this case, we want to increase the lookahead distance r until $d > 0$.

b. If $d > 0$, then we can obtain a solution $t = \frac{-b+\sqrt{b^2-4ac}}{2a}$. We only want to consider the '+' in the $\pm$, since we want to only consider points in front of the robot. If $t < 0$, then that means that if we extended the segment far enough backward, it would intersect with the circle. So, in this case, we increase the lookahead distance r until $t > 0$

c. In this case, if a solution $t > 1$, then that means if we extended the line segment far enough forward, it would intersect with the circle. In this case, it means that we should be looking at the next line segment in the path to determine a target point. So, we set $p_1 = p_2$, and $p_2 = p_{new}$, where p_{new} is the next point in the list obtained from RRT*. From there we recalculate the values from equations 9-13 to see if the new path segment is able to give us a target point.

d. In the last case, $0 \leq t \leq 1$, so we have found a solution that is on the line segment. We can then determine that the target point the robot should follow is $p_t = p_1 + t(p_2 - p_1)$

2.3.3 Determining the Steering Angle

Now that the target point has been determined, the steering angle θ of the robot can be found. If we want the robot to turn towards the target point, it will be turning around a circle with radius R. We call this radius R the turning radius. We also use the wheelbase length L of the robot to determine θ. These quantities are all visualized in Fig. 2

To determine the steering radius, we transform the target coordinate to the frame of the robot, and use the y-component of the coordinate y_t as shown in

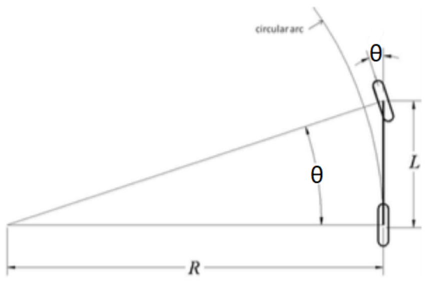

Figure 2: Relation of steering angle to turning radius.

equation 14[3], where r is the lookahead distance.

$$R = \frac{r^2}{2y_t} \tag{14}$$

With the turning radius, the steering angle is then

$$\theta = \tan^{-1}(\frac{L}{R}) \tag{15}$$

We repeat this process for each odometry message the robot receives, and publish the steering angle so that the robot successfully follows a given path. Some improvements that we made were to lower the speed and lookahead distance of the robot if the given segment is very short, so that the robot can better go around curves.

3 Experimental evaluation

Author: Henry

We evaluated our path planning algorithm and path following controller's simulated performance using code instrumentation and `rqt_plot`. We evaluated our implementation of RRT* on performance, optimality, and path smoothness. We evaluated our pure pursuit controller on its absolute error. We collected all evaluation data in simulation, in the Stata basement.

[3]Math acquired from paper mentioned in Piazza: `https://www.researchgate.net/publication/319714221_Pathfinder_-_Development_of_Automated_Guided_Vehicle_for_Hospital_Logistics/download`

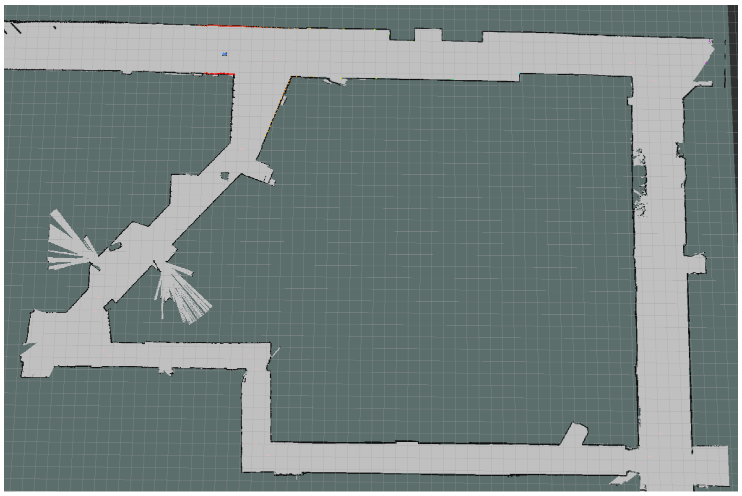

Figure 3: Evaluation environment, the basement of Stata, overlayed with a 1m grid

3.1 RRT* evaluation

3.1.1 Performance

The faster the RRT* algorithm is able to add new nodes to the search tree, the better the overall result. Evaluated on a Racecar virtual machine, our optimized algorithm is able to place 4680 points in two minutes (40 pts/second).

We were able to identify and optimize two main performance bottlenecks in our algorithm:

- Deep recursion in the event of updating child nodes during rewiring
- Slowness in checking for obstacles along the line segment between new nodes and their parent on the RRT* search tree

In the first case, by stopping the recursion after one call, we were able to increase the number of points placed by about 15%, from 700 to 800. We chose to only update the direct child nodes during rewiring, instead of recursively updating the cost and depth of all descendants, because these values are only heuristic and do not need to be perfect.

In the second case, by using `numpy` to vectorize our code, we were able to increase performance of the specific subroutine `generate_line_path` from an average runtime of 0.02 seconds to 0.0002 seconds, a 100-fold speed-up. This led to the number of points being placed growing from approximately 800 to 4600, about a 6-fold increase.

3.1.2 Optimality

As explained in section 2.1, the RRT* algorithm is asymptotically optimal, meaning that given an infinite number of iterations, the result is optimal. However, in our implementation, we focus on obtaining a smooth usable path, by reinforcing the first path found. Due to its probabilistic nature, the algorithm sometimes fails to find the most optimal path. Specifically, in the case where a shorter path is narrower, while a longer path is very wide with few obstacles, our RRT* algorithm has a non-trivial chance of returning the longer, simpler path.

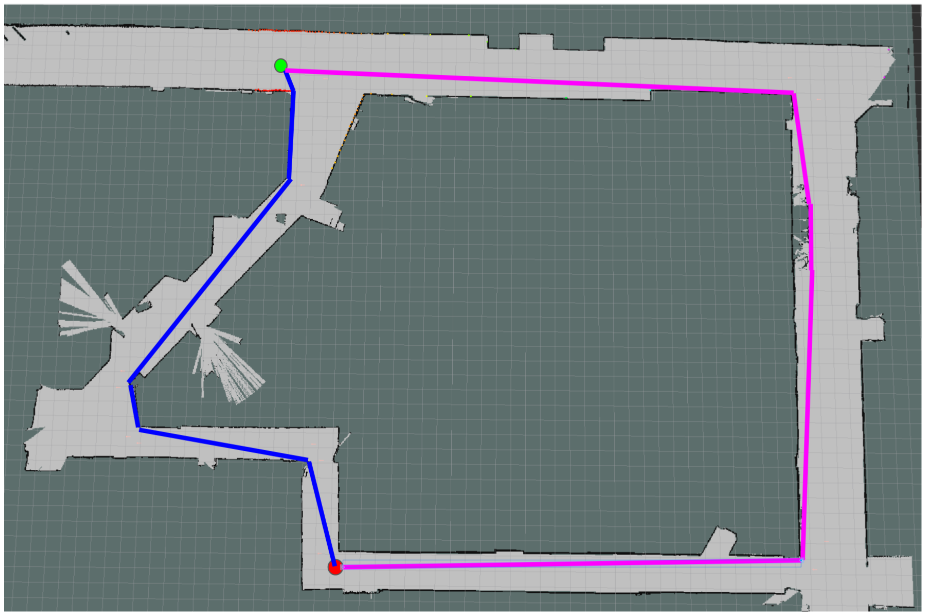

Figure 4: Visualization of Optimality evaluation environment.
Red node: End node.   Green node: Start node.
Blue path: Shortest corner-to-corner path.   Pink path: Alternative path.

We evaluated the performance of the path planning algorithm on the start and end points shown in figure 4.

By using the "Measure" tool on RViz, we were able to determine that the corner-to-corner (optimal) length for the shorter, blue path was 47.7 meters, while the length of the longer, pink path was 98.8 meters.

Because the left path (blue) is narrower, contains an obstacle, and contains more turns, it is more difficult for our RRT* algorithm to find. We ran the path planning algorithm twenty times, and the algorithm successfully discovered the shorter path only 55% of the time, as shown in figure 5.

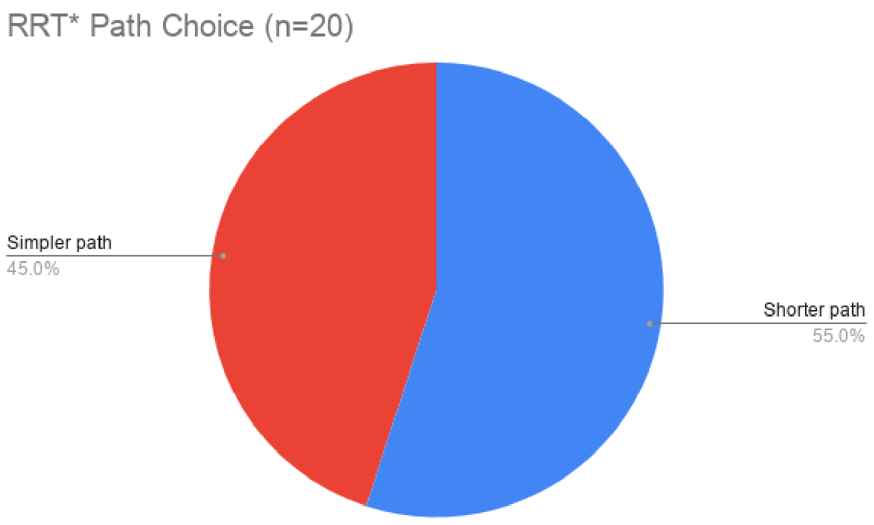

Figure 5: Path Choices made by RRT* (n = 20)

Note that the optimality metric was not a design priority for this lab, as our primary motive was to obtain a feasible trajectory for the robot to follow. In more narrow, obstacle-heavy spaces such as the blue path, our robot would likely perform more poorly, and not be able to travel as fast anyway.

Figure 6 shows an intermediate result for the RRT* search during this evaluation. Notice that there are RRT* tree nodes on both the left and right paths, showing a race-like dynamic between the searches on either front.

3.1.3 Path smoothness

One additional metric for the success of our path planning algorithm is the smoothness of the path. Given the optimal path, we looked to obtain the smoothest path while avoiding obstacles and walls.

The most natural metric for us to use for smoothness is percent inefficiency compared to a corner-to-corner measurement of the path from start to end point. To find the corner-to-corner measurement of a path, we take the corners of walls along the turns of the path and connect them, resulting in the path with

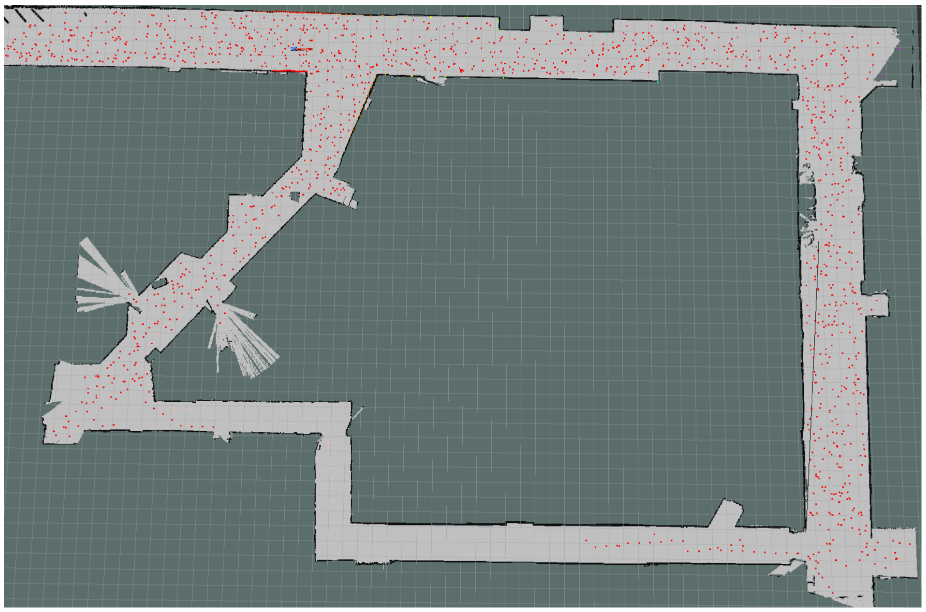

Figure 6: Nodes (red) in a typical RRT* search tree, before a path is found

the least distance. A very smooth path would follow this ideal path closely, while an uneven path would inefficiently zig-zag around the ideal path.

Note that due to any actual path remaining a safe distance away from corners, we wouldn't expect *any* path finding algorithm to obtain 100% efficiency.

Table 2: Percent inefficiency of RRT* and RRT paths (n = 20)

Path	Corner-to-corner distance	Average RRT* Trajectory length (n=20)	Percent inefficiency	Average RRT Trajectory length (n=20)	Percent inefficiency
Simpler path (Right)	98.778	103.495	4.7%	111.293	12.7%
Shorter path (Left)	47.734	48.625	1.9%	59.800	25.3%

As shown in table 2, our implementation obtains a very favorable inefficiency percentage of only 1.9% for a short path and 4.7% for a longer path. The reason why the algorithm performs better on shorter paths is that we reinforce the search tree by picking points along the path. Given that a constant number of points is chosen, this reinforcement is more effective for shorter paths. For reference, Table 2 also includes the smoothness for RRT search, where no rewiring is done. Paths found using RRT indeed have a more cubic, uneven quality and are less efficient.

Figure 7 shows the qualitative smoothness of one path found by our algorithm.

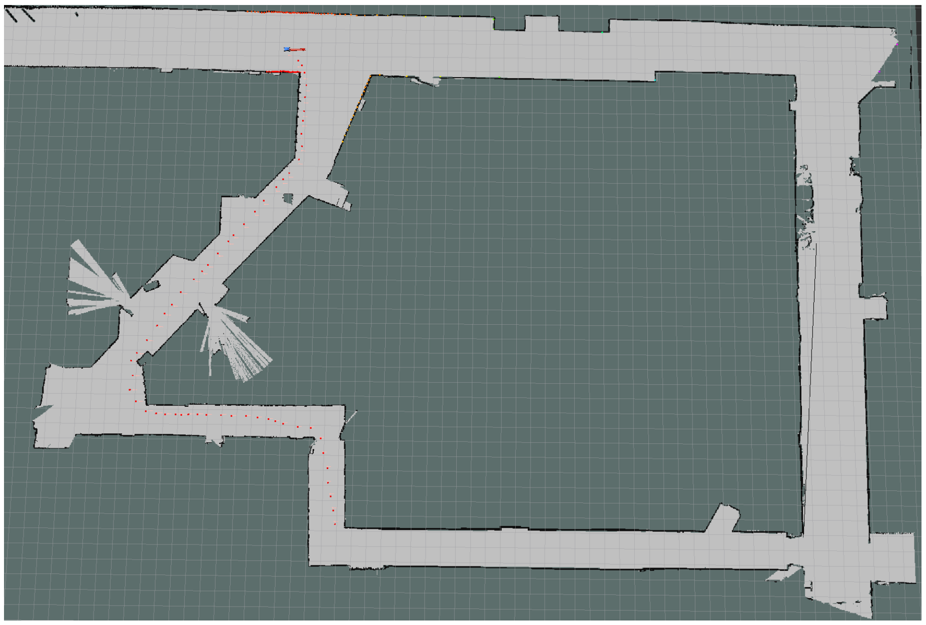

Figure 7: Nodes along the path found by RRT*. Note the smoothness of the path.

3.2 Pure pursuit evaluation

We were able to determine the absolute error for our pure pursuit controller by comparing the true pose of the robot to the planned trajectory in simulation. We evaluated the absolute error on the most difficult stretch of the loop trajectory in the Stata basement.

The calculation that we performed to determine absolute error was the Euclidean distance between the true position of the robot, given by the simulation, and the nearest point on the trajectory.

In simulation, the time-averaged error along the trajectory was only 0.046 m. At its peaks, the error stayed within 0.6 meters, and would usually peak between 0.2 and 0.4 meters. These peaks generally corresponded to whenever the robot turned sharp corners. The `rqt_plot` for the error over time is shown in Figure 8.

Figure 9 shows the same error graph, but overlayed on the route taken, to more clearly show the correlation of the absolute error to turned corners. The amplitude of the error graph is magnified about 4 times relative to the map background, in order to more clearly show the trends. The typical width of the hallway is about 4 meters across.

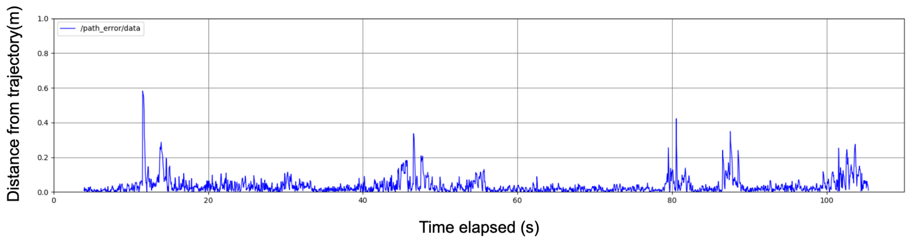

Figure 8: Absolute error over time

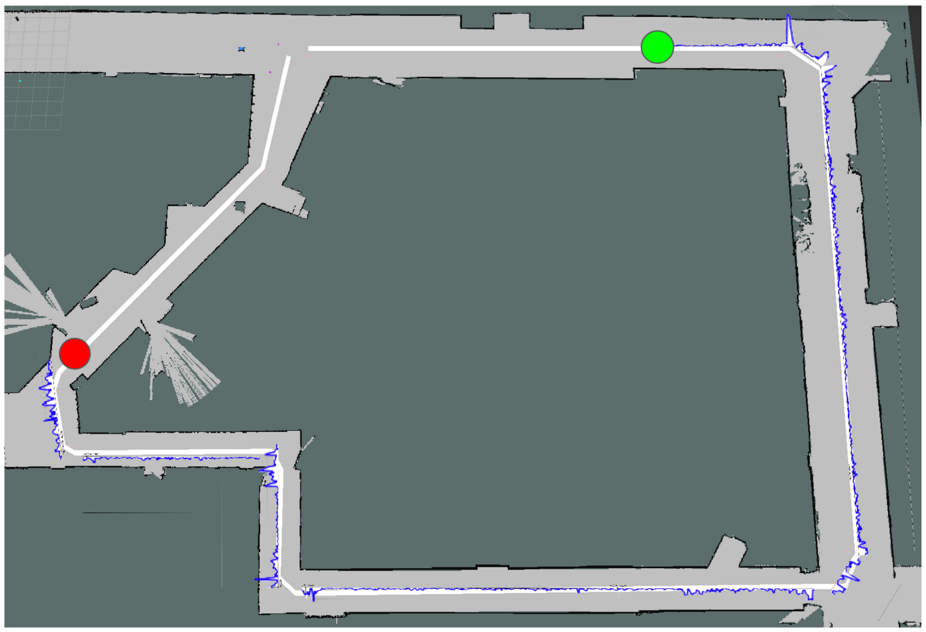

Figure 9: Absolute error (magnified 4x) along the loop trajectory.
Green node: start point. Red node: end point.

4 Conclusion

Author: Christina

In conclusion, we have demonstrated that our robot is able to determine the path to its destination and follow it. Through assessment of different motion planning algorithms, we determined that RRT* is the optimal path planning algorithm. Furthermore, we attained path following through pure pursuit. Finally, we integrated the path planning and path following algorithms. Upon evaluation of the algorithms through numerical evidence, we were able to conclude that we have successfully implemented the two core parts of autonomous operation: planning and control.

5 Lessons Learned

William:

- Learned to implement RRT* for a real robotics problem.
- Learned to alter existing algorithms to better suit a given problem, altering it to improve performance and time efficiency.
- Henry taught me about the absurd efficiency of vectorization and absurd inefficiency of recursion. Recursion be like recursion be like recursion be like recursion be like recursion be like recursion be like recursion be like recursion be like recursion be like...

Henry:

- I enjoyed learning about the RRT* algorithm, the first sampling search algorithm that I have ever implemented.
- Learned how to work and communicate remotely with the team.
- Learned about self motivation to do work (and work inertia)

Christina:

- Learned more about different types of algorithms, especially RRT* which was completely new to me.
- Learned (and still learning) how to work remotely with the time difference

Silvia:

- Learned about many different types of planning algorithms.
- I liked seeing the different parts (eg. localization, pure pursuit, path planning) all come together to make the robot move around the map.
- Learned about remote team collaboration and working together better remotely.

Diego:

- Learned how to implement a sensor model and reduce runtime by pre-computiong certain variables
- Learned to work remotely with teammates