Lab 6 Report: Path Planning

Team 17

Alicia Zang
Ari Grayzel
Robert Cato III
Sebastian Garcia
Shreya Gupta

RSS

April 16, 2022

1 Introduction

Section 1 by Shreya Gupta

Path planning is an aspect of robotics that has many applications. From navigation for mobile robots to pick and place for robots with joints, the ability to plan a path from one point to another is a critical part of the functionality of many of these robots. To look more specifically at an example relating to mobile robots, path planning used in warehouse robots that are required to move boxes from one point to another in a manner that is efficient in both computation time and path length.

The goal of path planning for mobile robots is relatively straightforward: traverse a path from point A to point B that strives to be optimal in regard to a certain criteria. The two criteria that we tried to optimize over were computation time and path length. We decided to prioritize path length over computation time as that tended to be the limiting factor in the performance of the algorithms. The motivation behind tackling autonomous path planning is that autonomous navigation is generally the base functionality of mobile robots. Any additional functionality is an overlay on top of the robot's ability to navigate within its environment.

In order to solve the problem of navigating from point A to point B, we implemented and compared the A* and RRT path planning algorithms and then used pure pursuit to follow the path. In order to implement path planning algorithms, we had to first modify the map in order to prevent the path planning algorithms from planning a trajectory that is extremely close to obstacles. We did this by eroding the map by about 1.5 times the largest dimension of the car so that if a car samples a point extremely close to the wall, it appears as "occupied" to the algorithm.

After the map has been eroded, we use a path planning algorithm to draw a path from one point to another. This can be done using sampling-based path planning algorithms such as RRT or search-based path planning algorithms such as A*. RRT specifically picks a random point on the map and draws an edge to the nearest point in the already existing tree. The edges and nodes must be in areas that are not occupied by obstacles. A* works by first discretizing the map and calculating a cost for every reachable state. It then moves in the direction of the lowest cost, assuming that the node and edge is in a region not obstructed by an obstacle. We found that RRT tends to be faster than A* but A* finds the shortest path. The paths generated by these algorithms are used by pure pursuit to navigate the robot.

2 Technical Approach

2.1 Occupancy Grid and Configuration Space

Section 2.1 by Sebastian Garcia

2.1.1 Problem

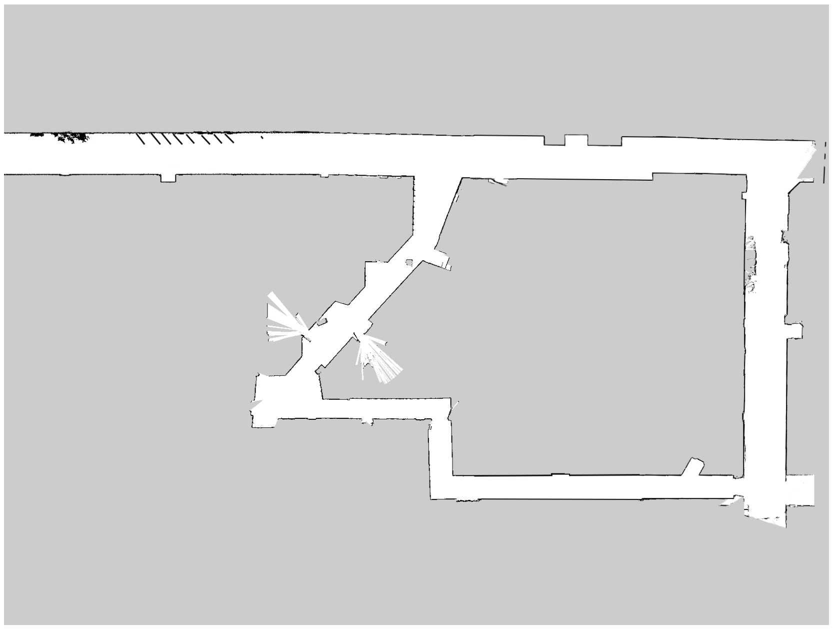
Before attempting to plan a path for the robot, we needed to define the space available to plan a path on. With any map, our path planning algorithms will generate paths that come very close to obstacles to minimize distance since we model our car as a point. In simulation this means that the car would be represented as a single pixel. Since the simulated car is larger than one pixel in both x and y directions, the car would come in contact with the obstacles when trying to follow the path. Modifying our algorithms to take the dimensions of the car into consideration would be computationally expensive, so we needed to find a different way to deal with this issue.

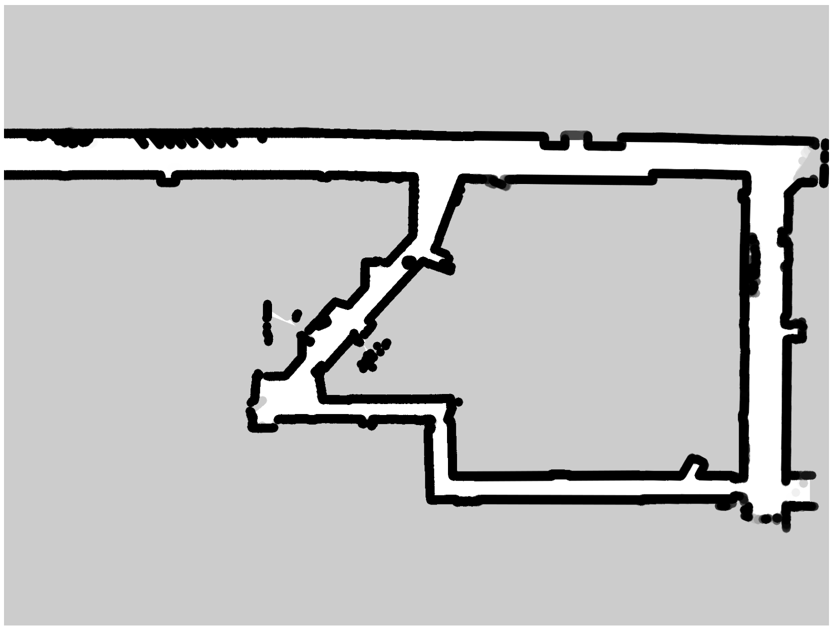
2.1.2 Setup

Since our path planning algorithm takes in the map we use as an Occupancy Grid, we must create that Occupancy Grid with black pixels denoting obstacles. Once we have that, a way to reduce the complexity of the problem while preventing collisions is by creating buffer zones in which free spaces within a certain range of obstacles are marked as occupied. We preprocessed the image and eroded it offline to save time when running the algorithm.

2.1.3 Approach and Implementation

To inflate or expand obstacles on the map being considered, the scikit-image and scipy python libraries were used to erode the image in spots that were occupied. More specifically, we used the erosion method in which we pass in the original greyscale map, with black pixels indicating occupation, to get a new version of the map, with wider black pixelated areas. The eroded map will be loaded in during the map callback as an Occupancy Grid in our path planning algorithm, and the original map will be what is contained in the '/map' topic so it can still be used for localization purposes. This way, a path will be planned that leaves enough space between the physical car and any obstacles around it. The original map and eroded map are shown below, respectively.





2.2 Image and Map Space Conversions

Section 2.2 by Robert Cato III

2.2.1 Problem

Motion planning on the robot requires two different frames: the map frame shown in RViz and the image frame used by the occupancy grid. The general flow for planning a path is then to:

- take a start pose and goal pose in the map frame
- convert the poses to the image frame and pixel space
- run the planning algorithm
  - check collisions by pixels in the occupancy grid
- convert points in a found path back into the map frame
- publish the map frame trajectory

2.2.2 Approach and Implementation

The conversion between map space and image space is mostly handled in the map callback for the planning algorithm. We extract the image's position, rotation, and resolution from the message received by the callback.

A pose matrix T transforming from pixel space to the world space is created using the map's position vector p and rotation matrix R

$$T = \begin{bmatrix} & R & & p \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then, using this matrix and the image's resolution, the conversion is:

$$\begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = T \begin{bmatrix} v \cdot \text{resolution} \\ u \cdot \text{resolution} \\ 0 \\ 1 \end{bmatrix}$$

The point (x, y) in the world space corresponds to the indices (v, u) in the occupancy grid where u is the row index and v is the column index
The inverse conversion is simply:

$$\begin{bmatrix} v \cdot \text{resolution} \\ u \cdot \text{resolution} \\ 0 \\ 1 \end{bmatrix} = T^{-1} \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix}$$

2.3 Sample Based Search: RRT

Section 2.3 by Alicia Zang

2.3.1 Problem

In order for the robot to travel from one point to another, it will need a path in order to do so. One way of finding a path is using Rapidly Exploring Random Trees, or RRT.

2.3.2 Approach and Implementation

For the algorithm, we are given the current position of the robot and the end goal. To start, we randomly choose a point in the map that is unoccupied (with 5% probability of choosing the end node) and then found the nearest existing node, shown below

```
for node in found nodes do
    if distance between random node and node is less than minimum distance then
        minimum distance = distance between random node and node;
        minimum distance node = node;
    end
    return minimum distance node;
end
```

Algorithm 1: Finding nearest node to the random node

In the initial case, this is the start node.

We then find a straight line from the two nodes. In order to check if the edge passes through an obstacle, done so by moving a newly created node along the edge and see if the occupancy grid of the created node is occupied. If so, we will sample another point in the map. Otherwise, we set the nearest found node as the random node's parent and add it to the list of found nodes.

```
node 1;
node 2;
intermediate node = node 1; maximum distance of an edge; step size = set number;
distance = minimum (maximum distance of an edge, distance between node 1 and node 2);
for step in range of distance divided by step size do
    if pixel at coordinate of intermediate node is occupied then
        return False;
    end
end
if remaining distance between node 1 and node 2 greater than step size then
    set node 2 = intermediate node;
end
set node 2's parent to node 1;
return True;
```

Algorithm 2: Checking validity of an edge

By limiting the length of an edge, one can explore the space more by utilizing far away random nodes, and, in the case where the random node is the end node but no node in the found nodes list is close, it will create a new node closer to the goal.

We then repeat the process of finding nodes until a node is within a set distance from the end node and an edge can be created.

After reaching the end node, a path is found by going through the parents of the nodes, starting with the end node. Since the start node has no parent, the algorithm will then return the list of points for the graph.

```
path = [given node];
current node = given node's parent;
while current node is not none do
    add current node's parent to the path;
    current node = current node's parent;
end
return path;
```

Algorithm 3: Finding path to start node from a given node

The full pseudocode of the RRT algorithm is shown below:

```
found nodes = [start node];
while number of steps not reached do
    Get random node;
    Get nearest node in the found nodes list;
    if edge can be created then
        Add random node to found nodes list;
        if random node is within a certain distance from end node then
            if edge can be created between random node and end node
             then
                return found path from end node;
            end
        end
    end
    return empty list if no path is found in the number of steps;
end
```

Algorithm 4: RRT Pseudocode

2.4 State Space Search: A*

Section 2.4 by Robert Cato III

2.4.1 Problem

Our core problem for motion planning is finding a path from our robot's current position to some goal position we set. Diverging from RRT above, we not only want to find a collision–free path, but also the shortest path. We implemented A* to find the shortest path between two positions in our map.

2.4.2 Approach and Implementation

To approach the A* algorithm and state-space search in general, we discretized the continuous world space into a grid. Each location in the grid is a box of five by five pixels centered on the robot's position when planning was started. Movement from any given state is then restricted to the eight adjacent squares in the grid space. The final piece for implementing A* search is an admissible and consistent heuristic. We chose to use the Euclidean distance from a given state to the goal state (in pixels) for our heuristic. This is a common method in motion planning and guarantees that any path found will be the shortest With a discrete space and defined transition between states, the A* algorithm was implemented as follows:

```
while agenda do
    expand-state = agenda.min-cost-state;
    cost-so-far = agenda.min-cost-so-far;
    path-so-far = agenda.min-path-so-far;
    expanded.add(expand-state);
    if expand-state close to goal-state then
        return path-so-far;
    else
        for children not in collision and not already expanded do
            move-dist = linalg.norm(expand-state - child-state);
            dist-to-goal = linalg.norm(goal-state - child-state);
            new-state = child-state;
            new-cost-estimate = cost-so-far + move-dist + dist-to-goal;
            add child state and information to agenda;
        end
        sort agenda by cost-estimate of each state;
    end
end
if path-so-far then
    convert each pixel state to map space and add to traj;
end
```

Algorithm 5: A* Pseudocode

For simplicity, all components in the algorithm are in the discrete pixel space. States were not converted to the world space until the very end when publishing the trajectory. We handled state-space collision detection by checking the occupancy values for all 25 pixels (the 5x5 square) in a given state.

2.5 Pure Pursuit Controller

Section by Ari Grayzel

2.5.1 Problem

Planning paths is only useful if our robot is able to follow them. The dynamics of a car are such that it is non-trivial to follow an arbitrary path, especially at high speeds. The car can effectively only control its forward velocity and turning angle, meaning that lateral movements are difficult to control and physical parameters such as the axle length constrain the turning radius of the car. Given that it may be impossible to perfectly track a trajectory we seek to develop an algorithm to control our steering angle and forward speed to minimize total transit time of a trajectory while minimizing average deviation from the path and keeping maximum deviation within acceptable bounds. For the purposes of this lab we will define the acceptable maximum deviation as deviation such that we are guaranteed no collision with known obstacles.

2.5.2 Approach

To achieve these goals we implemented a pure pursuit controller. The pure pursuit controller works by tracking a point at some distance r ahead of the vehicle on the path. The controller searches for all points at distance r from the robot that intersect with the planned trajectory and from them selects the point which is furthest along the path. The controller then calculates the constant steering angle which would be required to intersect the point by traveling along an arc. This process is repeated every time that the vehicles estimated pose is updated. We implemented the algorithm as described in Coulter 1992, "Implementing a Pure Pursuit Path Tracking Algorithm".

2.5.3 Implementation

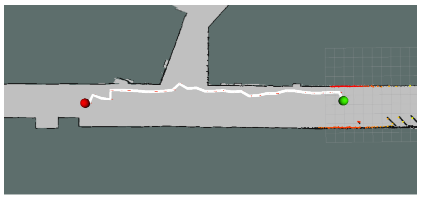
We chose to run our algorithm with a static speed and a static lookahead distance which would be calculated as a function of vehicle speed. This simple approach yielded highly effective performance. Details about our tuning process and performance figugures can be found in the Experimental Evaluation section. Future work may be to try to improve performance further through the use of mechanisms such as variable speed and lookaheads based on local characteristics of the path such as curvature.

3 Experimental Evaluation

3.1 RRT and A*

Section 3.1 by Robert Cato III and Shreya Gupta
In simulation, both path planning algorithms (RRT and A*) returned collision free paths between given start and goal positions using the eroded occupancy grid. Our A* implementation also returned collision free paths when brought to the physical platform.

Our experimental evaluation will focus on the differences between RRT and A*, ultimately leading to our decision to bring A* to the physical robot. Below, you can see visualizations of the paths generated by the algorithms with RRT on the left and A* on the right. Since the RRT algorithm is non-deterministic as it randomly samples points, the figures on the left are just an example run. If the algorithm were to be run again, the path would look different even with the same beginning and end points. Each of the following three sets of figures show the visual comparison between RRT and A* for different configurations


(a) Example RRT Run

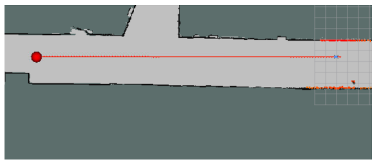
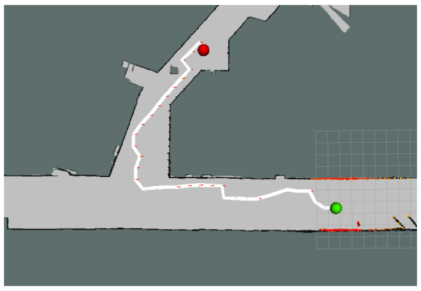

(b) A* Run

Figure 1: Path planning on a straight hallway


(a) Example RRT Run

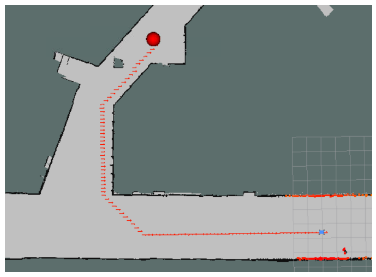
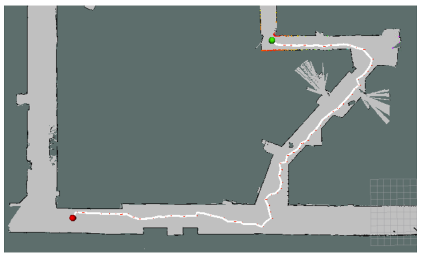

(b) A* Run

Figure 2: Path planning around a corner


(a) Example RRT Run

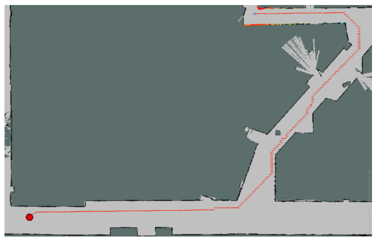

(b) A* Run

Figure 3: Path planning across the map

Algo		Straight Hall	Around Corner	Across Stata
	Start Position	(-3.2, -0.599)	(-3.2, 1.588)	(-20.06, 26.13)
	Goal Position	(-30.58, -0.599)	(-14.53, 11.94)	(-50.20, -0.434)
A*	Path Length (m)	27.65	28.85	72.99
	Avg Plan Time (s)	0.0983	0.6500	1.8714
RRT	Avg Path Length (m)	30.89	34.24	79.42
	Avg Plan Time (s)	0.143	0.1429	0.8649

Table 1: Path Length and Plan Time for A* and RRT for three different scenarios. Start and goal positions are recorded as (x, y) in meters.

We compare two core evaluations for A* and RRT in Table 1: Path length and the time it took to plan a path. Regardless of the test scenario, RRT found a path between the start and goal points in significantly less time than A*. However, A* always found the shortest path between the two points (considering the eroded occupancy grid). These shortest paths were several meters shorter than those found by RRT. In extreme cases, the path returned by RRT is *much* longer than the one returned by A* because it wraps around the map before finding the goal. Figure 14 in the Appendix shows an example of this.

	over 50 runs	Straight Hall	Around Corner	Across Stata
Path Length (m)	mean	31.478	38.919	79.93
	standard dev	1.204	26.5704	2.2933
Plan Time (s)	mean	0.0945	1.4573	0.3638
	standard dev	0.0914	2.5973	0.2235

Table 2: Path Length and Plan Time means and standard deviations for RRT over 50 runs. Start and goal positions are the same as that stated as Table 1.

In Table 2, we are comparing the standard deviations in path length and path time over 50 runs for RRT. We're only recording the means and standard deviations for RRT because it is a non-deterministic algorithm. A*, on the other hand is deterministic so it mean and standard deviation have no useful meaning. The standard deviation for path length around the corner is much higher than the standard deviation for straight hall or across Stata because there were a few trials that returned a path that wrapped around the map, as mentioned before (Figure 14 in the appendix). The possibility such an inefficient path to be planned using RRT makes it a poor choice for our physical robot, since for any given run, there is a chance that the robot traverses this extremely poor path.

Keeping in mind that our car might need to plan a path and race against others, we wanted to limit the total amount of time between start and goal initialization to arriving at the goal point. For a physical platform traveling at finite and low speeds, the time limiting factor is the distance along a path the car has to follow. For that reason, and A*'s path length optimality, we chose to use A* on the physical car rather than RRT.

3.2 Pure Pursuit Controller

subsection by Ari Grayzel
The effectiveness of a pure pursuit controller is largely determined by one key parameter, the lookahead distance. To tune the lookahead distance we established a step testing protocol. We constructed a trajectory with a long, straight section, a one meter horizontal step and then another long straight section. This allowed us to run objective, repeatable tests to asses the effects of different lookahead distances on controller performance.

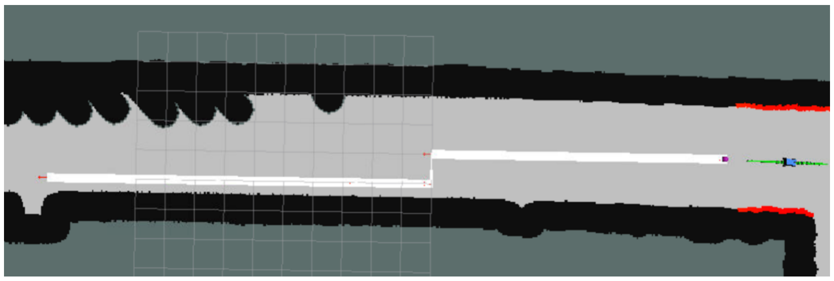


Figure 4: Step test trajectory

We tuned our controller by varying speeds and lookaheads. For a set speed of 2m/s we recorded the following step responses for lookaheads of 0.5m, 1m and 2m respectively.

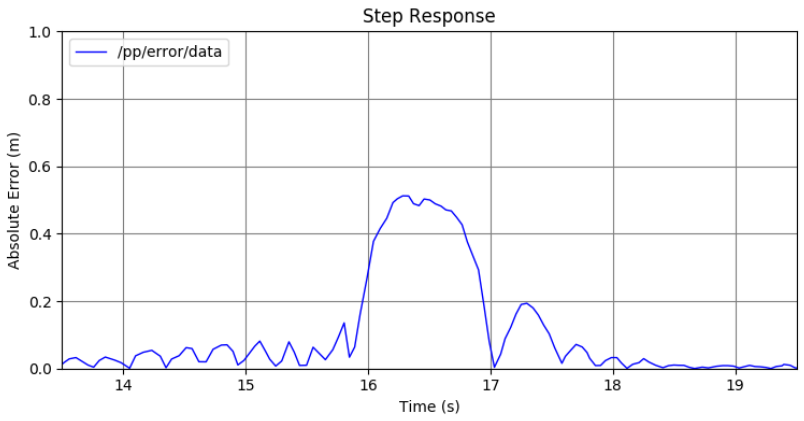


Figure 5: Step test at 2 m/s speed with 0.5m lookahead

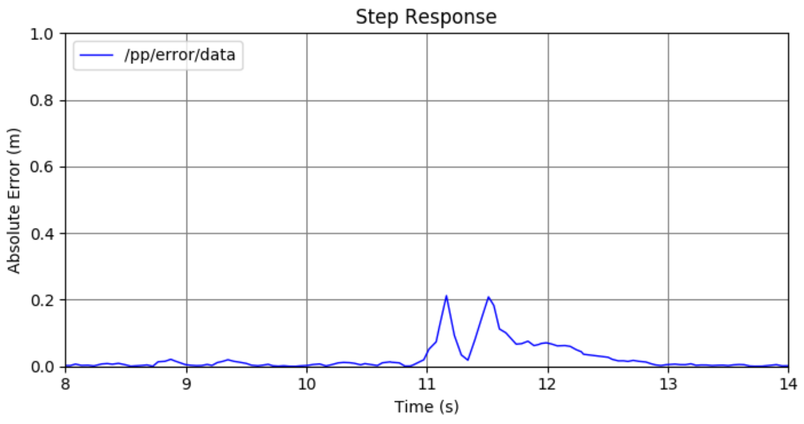


Figure 6: Step test at 2 m/s speed with 1m lookahead

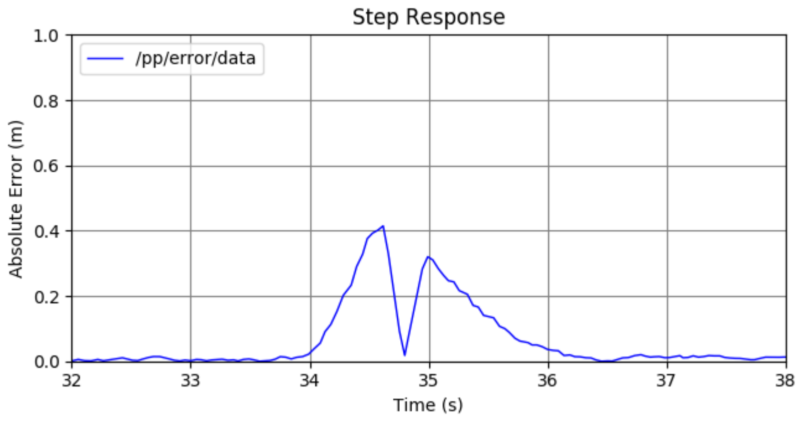


Figure 7: Step test at 2 m/s speed with 2m lookahead

From the step plots, there is a clear instability in the 0.5m lookahead distance, as seen by the wiggles in the signal both before and after the step. Comparing the responses of the controller with lookahead 1m to lookahead 2m, we see a similar shape to the response, but with the 2m lookahead we see a wider response and a higher peak–indicating that the robot was both off by more and for longer than with 1m lookahead. Repeating this experiment with a vehicle speed of 4m/s and with lookahead distances of 1m, 2m and 4m yielded these results:

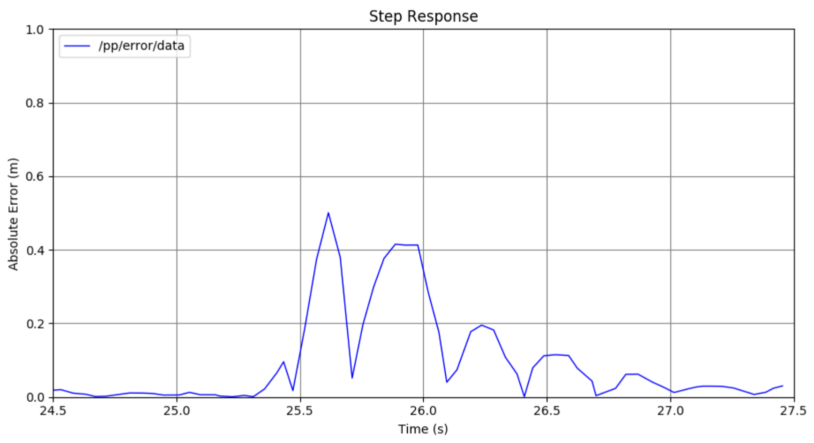


Figure 8: Step test at 4 m/s speed with 1m lookahead

[H]

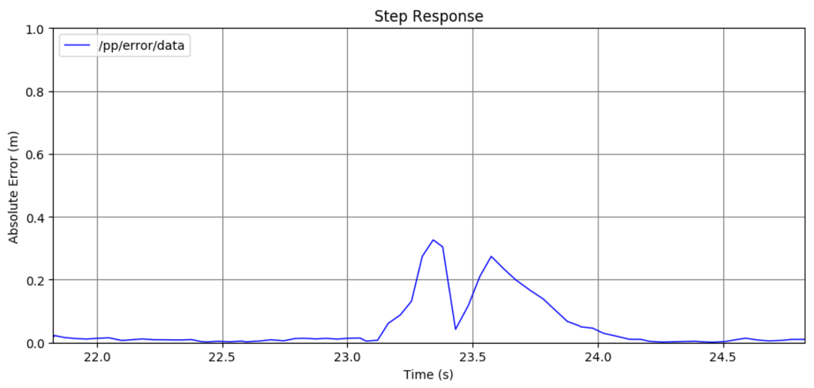


Figure 9: Step test at 4 m/s speed with 2m lookahead

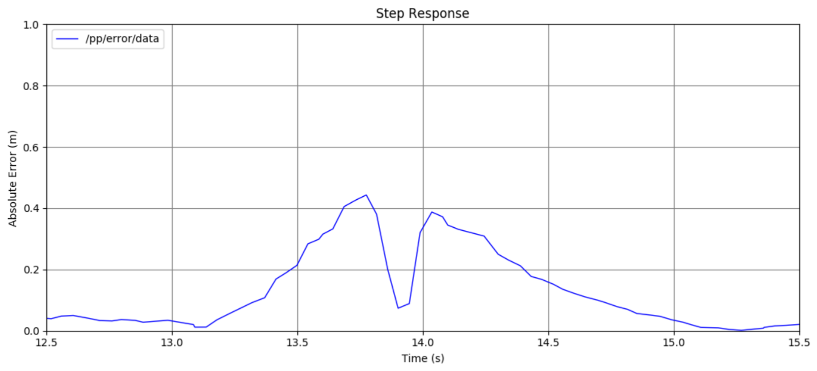
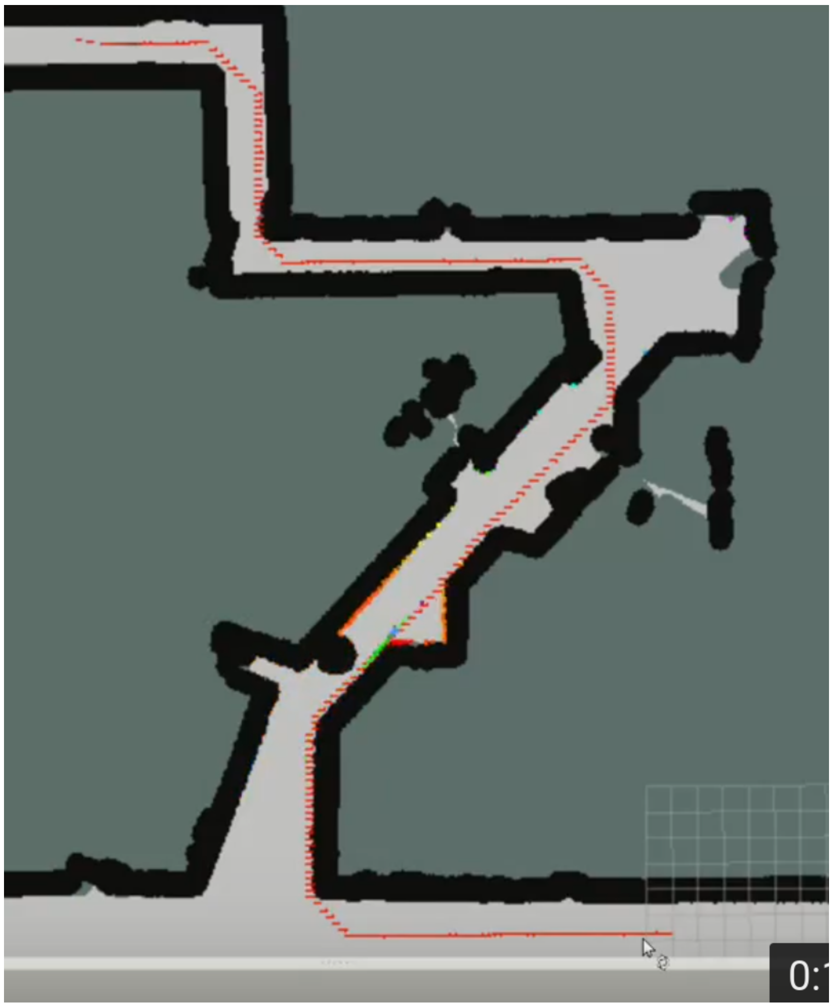


Figure 10: Step test at 4 m/s speed with 4m lookahead

This clearly matches the same pattern seen at 2m/s, with instability at too low lookahead(1m) and larger error at higher lookaheads. With this pattern we constructed a general control policy for our simulated robot:

$$\text{Lookahead} = \frac{v}{2}$$

This policy generalized well from the step response test into trajectories generated from our A* path planning algorithm. We ran the controller on this path:



The controller tracked the path very well, even at max speed of 4 m/s. The error plot can be seen below:

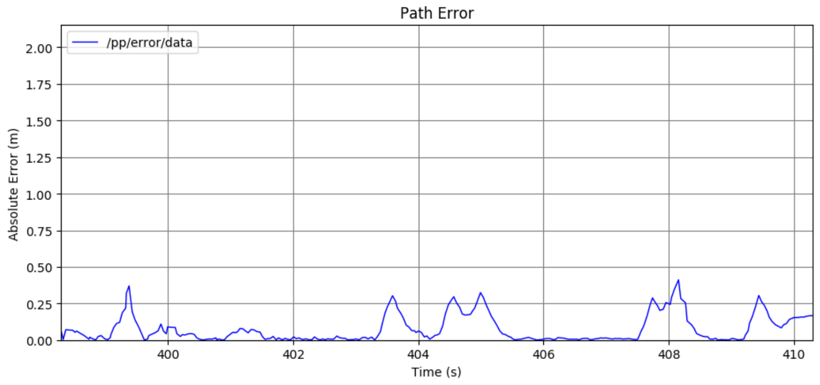


Figure 11: Error vs. Time on A* generated trajectory

As can be seen, there are spikes in the error, however the magnitude of these errors never exceeds 40cm. This error is well within the distance that we eroded the map by, meaning that with this much error we will never crash into a known obstacle.
Translating our controller onto physical hardware came with some challenges. Primarily, rigorous testing was much more difficult due to the imprecise nature of the real world and our ability to repeatably place the robot for beginning test runs. That being said, we were able to successfully port the algorithm and see good experimental results.
The largest change we had to make was to re-tune the lookahead distance for the real robot. When running with the lookahead distance rule we settled on in simulation, there was noticable instability when running the robot at 1m/s.

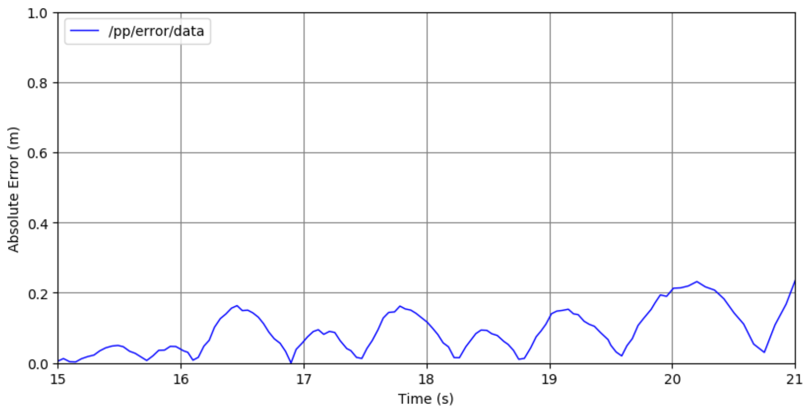


Figure 12: Error vs. Time for hardware platform at 1 m/s with 0.5m lookahead

After some time spent tuning we came to the conclusion that an increased lookahead distance was necessary, and developed a lookahead policy for the robot platform of:

$$\text{Lookahead} = v$$

While not perfect, this lookahead resulted in significantly improved stability performance (spikes are when the robot turns a corner):

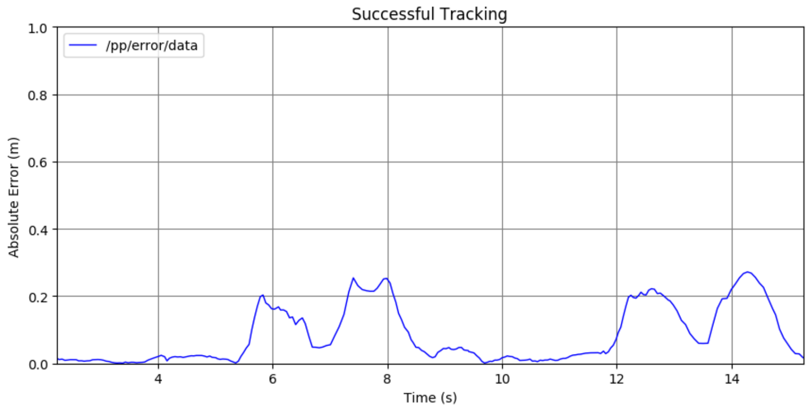


Figure 13: Error vs. Time for hardware platform at 1 m/s with 1m lookahead

3.2.1 Limitations

Our pure pursuit controller has been shown to be quite effective in a range of path following tasks. Perhaps the biggest downside of the implementation is that the relatively large lookahead distances we use mean that the robot is early to react to changes in the path such as corners. This can have the undesirable effect of the robot 'cutting corners' and deviating from the path in sections of high curvature. At the moment this addressed through our map pre-processing, the dilation ensures that the planned trajectories never get too close to a corner to where this ”corner cutting” effect could cause a collision. Possible future solutions to this problem could include adding a function to vary the robot speed or lookahead distance based on the local curvature of the trajectory.

3.3 Physical Implementation

In addition to re-tweaking our lookahead distances, we had to make several other changes to get the path planning code working on the robot. One such change was updating our ROS topics so that we were publishing to the right topics on the hardware platform. The biggest change we probably had to make was using our pre-processed, dilated occupancy map for the path planning algorithm while still using the original, undilated map for localization. This is because while we want the dilated map for a buffer during path planning, our localization algorithm would clearly fail if we attempted to use the same map, as it would not match the physical reality of the space the robot was in. Ultimately this was a relatively simple software implementation and we were able to get the platform working well on the actual car.

4 Conclusion

In this lab, we successfully implemented, tested, and compared two different path planning algorithms in simulation along with a pure pursuit controller for the car to follow the planned paths.

We ultimately chose to use the A*(A-Star) search algorithm instead of RRT for path planning on the physical car due to its optimality in being able to find the shortest path from one point to another. We defined this as optimal, since in a real life situation where we would want to travel as fast as possible, a shorter distance would take less time, making up for the slower process of finding a path.

For our pure pursuit controller, we found that in simulation the optimal look ahead distance to prevent instability when following a path is about 0.5 times the speed of the car. However, because the world is not perfect with a lot of noise and room for error, the optimal look ahead distance on the physical car was about the same as the speed that it traveled.

We had found that our pure pursuit controller would cause our car to cut corners occasionally and that a potential fix was to change the look ahead distance. Another way to fix this issue in the future would be to add a larger buffer zone in our path planning algorithms by changing the Occupancy Grid so that turns would not be as tight in corners.

Overall, we found that our implementations are effective in the stata basement environment, and more rigorous testing along with optimizing A* would improve the speed and safety of the path planning and path following process.

5 Lessons Learned

5.1 Alicia Zang

One communication lesson I learned was planning on a timeline and setting a time when to have all the parts ready by. This helped streamline the integration portion on the robot. A technical lesson I learned was that splitting a problem into smaller and manageable pieces would improve the rate in which I would make progress, as I was able to just focus on the smaller part, then piece them back together.

5.2 Ari Grayzel

My biggest technical lesson from this lab was the power of unit tests. I have always known generally that they were a good and useful thing but being able to run repeatable tests for the tuning was just so incredibly helpful here. I will definitely try to make as standard and repeatable tests as I can for all future labs and similar work I do.
A communication lesson I learned from this lab was the power of parallelization and how asking for help when you get stuck is so necessary. Our team has always tried to effectively work in parallel but it wasn't really until this lab that I feel like we really pulled it off in a big way. Additionally, there were many points at which I feel like I had gotten stuck but my teammates helped me figure out what was wrong with my work which was super helpful.

5.3 Robert Cato III

A major technical lesson I learned in this lab was how to anticipate errors and use case checking to make sure that our code could work on any platform (docker, the autograder, and the robot). This was important for the imports we were using, including loading the eroded map image into an occupancy grid. ROSPack() was critical in being able to access the desired image on any platform. On the communication side, I learned how effective parallel tasking is. We were all able to work more independently because of that; the entire lab was much smoother.

5.4 Sebastian Garcia

On the communication side of things, one thing that was reinforced during this lab was the importance of smart parallelization, as in formulating tasks such that they could be done completely independent of one another, but later integrated to have a fully functioning system. A technical lesson I learned was how to better create test cases for the different algorithms, using code to keep a controlled environment when testing for comparison.

5.5 Shreya Gupta

During this lab, I learned about how to write clean code that is structured in a way that makes it easy to change aspects of the code because it exists in modular sections. I also learned how to understand and modify other people's code. On the communication side, I learned the value of paralleling work and also how to communicate technical concepts both verbally and in a written fashion. Because we were working asynchronously most of the time, I had to explain/understand code over text conversations.

6 Appendix

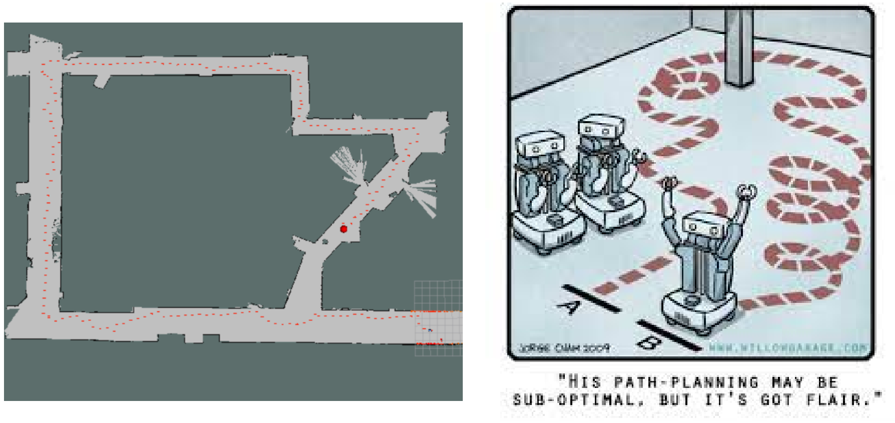


Figure 14: RRT path wraps around the entire map