Team 15 + Irene Terpstra, Miles Silva, Matthew Leonard, Lili Sun

6.141

April 15, 2022

LAB 6 REPORT: Real-time Path Planning Using A* Search and Pure Pursuit

1 Introduction

In this lab, we implemented path planning and following algorithms to allow our robot to efficiently travel from any starting point to any given end point, despite the presence of arbitrary obstacles. In an open, two-dimensional space, path planning would be a trivial problem, as the robot could easily orient itself directly towards the goal and move in a straight line. However, the obstacles can cause a complex tree of branching possible paths to form as the robot must make a choice at each possible fork, and so a thorough and sophisticated path-planning algorithm is necessary. Possible options include discrete, search-based methods, such as the A* algorithm, and stochastic, continuous sample-based methods such as RRT*. Each method uses a different approach to reduce the space of all of the possible paths that the robot could explore to a discrete, computationally feasible subset. In addition, because these optimal paths can be so complex, we must implement a sophisticated controller that allows the robot to follow an arbitrary, pre-planned path as closely as possible. As in the previous lab, the robot must use a Monte Carlo filter for localization, which means that there is some inherent error that the controller must handle. We were asked to implement pure pursuit, a commonly-used control algorithm wherein the robot finds the closest point on a path and follows it using a PD-style controller, similar to labs 2 and 3. In order to pass the tests, the robot must find a path that is close to the optimal path in simulation, with a maximum permitted deviation. The algorithm is constrained to a maximum computation time, so it must also be efficient. We chose to implement a search-based path planning algorithm, which represents the world as a set of discrete pixels, similar to a chessboard but with over a million squares instead of 64. We used A*, an informed search algorithm which employs an heuristic to rank possible paths by how optimal they appear to be, and chooses to explore the most

promising paths. The A* algorithm significantly improves computational efficiency over uninformed search algorithms. We implemented a standard pure-pursuit controller, with some modifications to help it choose the correct point on our path as quickly as possible. We also replaced our localization solution with the one provided by the staff. We worked on an RRT* algorithm, but decided that the A* algorithm was sufficient for our purposes. By implementing an efficient and effective path-planning algorithm and Pure Pursuit controller, we have made significant progress in developing a robust autonomous vehicle. Autonomous robots are routinely required to travel from a starting point to an ending point, and so we can use this lab as a building block to develop a high-level strategy that chooses these starting and ending points. Planning and following a path between these points as quickly as possible is important for a race scenario.

2 Path Planning (Irene)

In order for the robot to successfully travel through its environment from one place to another, the robot needs to plan a path that avoids obstacles. To design a path, we used the search based algorithm, A* to find a path. We developed and tested the algorithm in simulation and the algorithm successfully plans a trajectory in a known occupancy grid map from the car's current position to a goal pose.

2.1 A* Algorithm Overview (Irene)

The A* algorithm works by generating paths by moving through the occupancy grid. A path is a list of locations on the occupancy grid. In each iteration, a path is pulled from the queue and multiple new nodes for the potential path are generated, moving up, down, left, and right (and diagonally) from the last node of the path. The new nodes are then stored in a heap.

In order to make the search more efficient the algorithm uses a cost function and heuristic to make the list of possible paths a priority queue. The cost function is based on how long the path is and the heuristic is based on the distance of the end of the path to the goal point. These two values allow the algorithm to follow the shortest paths that are closest to reaching the goal.

2.2 A* Algorithm Implementation (Lili)

After deciding on an algorithm, we needed to define the specific search space, cost, and heuristic for it to work correctly for our setup. As stated above, the A* algorithm is run on discrete graphs, while our robot operates in a continuous space (i.e., the real world). The algorithm also requires a specific cost function in order to actually optimize, as well as a well-defined heuristic in order to determine the proper search direction towards the goal.

2.2.1 Search Space (Irene)

One of the most challenging aspects of implementing the algorithm for our team was locating our robot in the occupancy grid map. In order to search the map for a path that reaches the end goal without hitting the wall, the robot needs to be able to know where the wall is. The layout of the map the robot is in is defined by the occupancy map. The robot is able to successfully locate itself in the real world coordinates using the localization algorithm that was implemented in lab 5. However, the coordinates in the real world do not match the coordinates of the occupancy grid. Offset of the occupancy grid from the real world coordinates is given in the definition of the occupancy grid. In order to convert the robot location into the coordinates of the occupancy grid we need to shift the coordinates, divide by the resolution of the occupancy map then rotate the coordinates by the yaw of the occupancy map. By property converting the coordinates, we can check whether the path of the robot is going through an open space.

Once the robot was successfully able to identify obstacles in the occupancy grid the A* search algorithm was able to calculate routes that started at the start node and ended at the goal location. However, these paths often followed the wall very closely. While the robot can move directly next to the wall in simulation, in the real world the robot needs to be a certain distance from an obstacle so that it does not hit it. Therefore, rather than finding obstacles in the original occupancy grid we created a new version of the occupancy grid where the walls were dialed. Dilation of an image makes it so that each pixel takes on the maximum value of the pixels around it. When applied to the occupancy grid, dilating the image makes it so that all the obstacle sizes are increased or the thickness of the walls are enhanced. To dilate the map, we used the skimage library’s morphology.dilation function.

(Lili)

The occupancy grid given as input into the path planner was based on the Stata basement map, which itself was discretized as pixels. We decided to use the occupancy grid map for the discretization of our search space, as it was simple, easy to manipulate with our libraries, and worked well. After testing our algorithm and implementing various optimizations which will be described later, we decided that it was unnecessary to downsample the grid we were searching on as its performance was fast enough.

2.2.2 Cost and Metrics (Lili)

For the cost function between the parent point and the new point, we decided to use the Manhattan metric. For the heuristic function to determine the cost estimate to the goal, we used the square of the Euclidean metric.

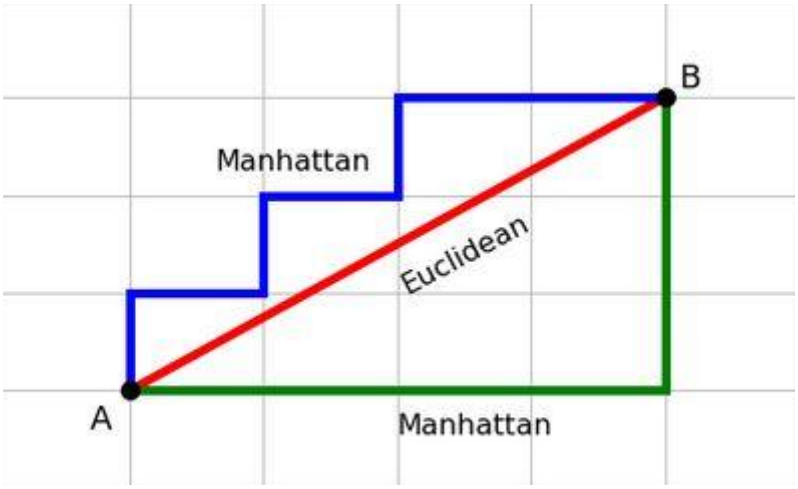

Figure 2.1: Visual description of the Euclidean and Manhattan metrics.

If we have two points: A at coordinates (x_A, y_A) and B at coordinates (x_B, y_B), then the Manhattan distance is the number of grid lengths to get from point A to B while the Euclidean distance is the length of the shortest line. For the Manhattan distance we simply add the two displacements in the x and y coordinates (equation 2.1), and for the Euclidean distance we use the Pythagorean theorem (equation 2.2).

$$\text{Manhattan Distance} = |x_A - x_B| + |y_A - y_B|$$

Equation 2.1: Manhattan distance formula.

$$\text{Euclidean Distance} = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$

Equation 2.2: Euclidean distance formula.

We decided to use the Manhattan metric as the cost function because our discretization was granular enough to accurately represent the actual cost of motion. We opted for the Euclidean distance as the heuristic as it was a direct measure of the distance from a certain point to the goal, which is the direction we wanted to search in.

2.2.3 Optimization and Performance (Lili)

After our initial implementation of the A* algorithm, we found that it was too slow for the autograder, and took longer than 2 minutes to find the desired path. The algorithm's lack of speed was largely due to our lack of using specialized data structures. Part of the A* algorithm includes keeping track of all the points that have been visited so far in order to ensure that the program properly terminates. Every time we want to add a potential new node into the list, we need to check it against the visited list and see if it's already there. The way we were originally checking visited points was by keeping an array and looping through it to check the new node against each existing one. Looping through the array takes linear (O(n)) time. We replaced the visited list with a set data structure instead, which has an average constant lookup time, $O(1_{avg})$. The second large bottleneck was finding the node in the queue to expand, i.e. the one that had the highest priority. Again, we used a rudimentary array to implement the queue originally, which as stated above, took linear time to find the minimum on every iteration. We switched the array out with a priority queue. In a priority queue, finding the minimum element takes O(log n) time, which is much faster. Both these optimizations allowed our algorithm to run quickly enough when searching for paths.

3 Pure Pursuit (Miles)

Once we have planned a trajectory, we need an algorithm for our car to follow it. To solve the problem of path planning, we implemented pure pursuit. Pure pursuit works by continually choosing a point on the trajectory a given distance in front of the car, and having the car follow that point. The high level structure behind the algorithm is described in Figure 1.

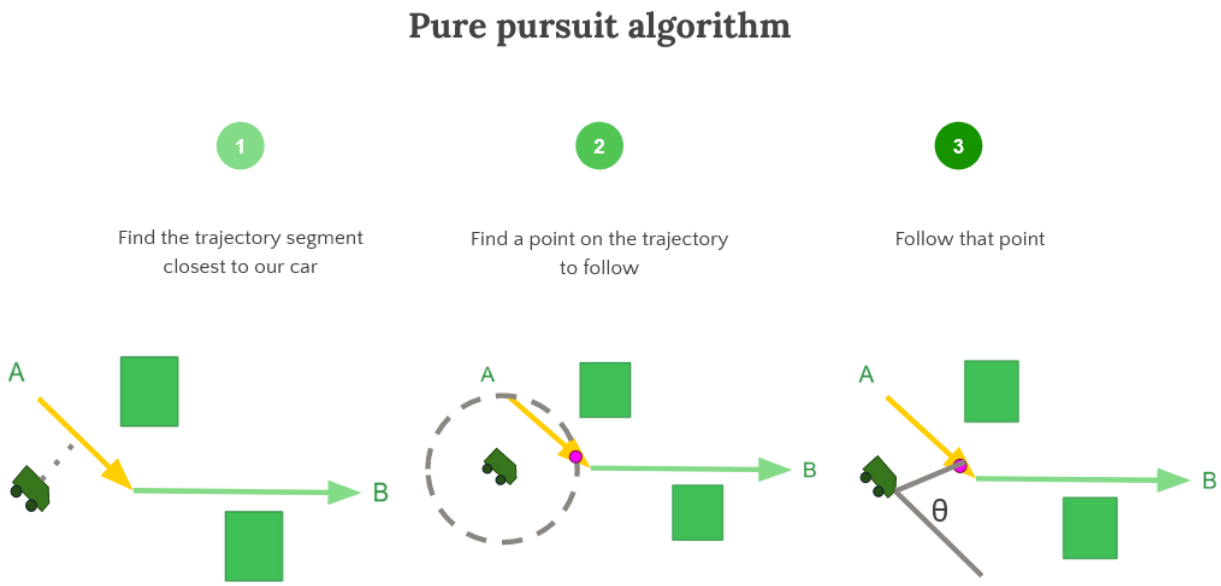

Figure 1: Pure pursuit. Our algorithm works by first finding the closest trajectory segment to our car, then finding a point on that segment or after that segment that is a given distance away from our car, then allowing our car to follow that point. The algorithm allows us to follow any arbitrary trajectory that we need to.

The first part of our algorithm is to find the trajectory segment that is closest to our car. To find the closest segment, we find the distance between our car and each trajectory segment, then find which value is smallest. The distance is calculated as the orthogonal distance between the car and the line segment itself (not the infinite line along the line segment), or simply the distance between the car and the nearest point on the line segment if an orthogonal line would not connect the two.

Next, we find the point on the trajectory to follow. We simulate a circle around our car of radius *lookahead_distance*, then loop through the trajectory segments to find intersections of the circle with the trajectory. To save time, we begin the loop from the segment that is closest to the car, and continue along the path until we have found two intersections. Then, we choose the intersection that is farthest along the path as our goal point to follow. If we cannot find an intersection because we are too far away from the trajectory, we choose the closest point on

the trajectory from the car as our goal point, allowing us to follow a trajectory even if we start far from it, or some outside influence occurs that pushes us away from the trajectory.

Finally, we follow the point. We have a constant pre-defined speed, and choose our steering angle to minimize the angle error between our car and the point (theta in Figure 1). To minimize oscillations, we implement a PD-style controller to determine our final steering angle. The equation is shown in equation 1.

$$\alpha = \arctan \frac{y}{x} + \frac{d\theta}{dt} \quad \textbf{(1)}$$

Where α is our steering angle, θ is the angle error, and (x,y) are the coordinates of the goal point in the reference frame of the car.

Adaptive Lookahead

To improve the accuracy and performance of our pure pursuit algorithm, we decided to implement an adaptive lookahead model. Our lookahead distance scales at each time-step to best accommodate for the situation the robot is in at each moment. We found that the scaling procedure allows us to reap the benefits of long lookahead distances (namely less oscillations and smoother driving), while also maintaining the accuracy of short lookahead distances when turning sharp corners.

We scale our lookahead distances based on two factors: length of the segment the car is closest to, and the distance the car is along the segment. We also cap the lookahead distance at a minimum of 1.0 meter, because we found that below this threshold, we observed extreme oscillations and other unwanted bugs. We found that these two factors performed very well in our tests, and obviate the need to include other factors such as speed in our lookahead distance equation.

We increase the lookahead distance when we are on long, straight segments of our trajectory. Because a long, straight segment means we will not have to turn, we can safely use a long distance to reduce oscillations in our driving. On shorter trajectory segments, we will likely be

turning corners or following more curvilinear paths, so we must use a shorter lookahead distance to navigate these paths safely.

We also decrease our lookahead distance as we approach the end of a trajectory segment. We decrease the distance because as we reach the end of a segment, we cannot be sure whether on the next segment we will be continuing along a relatively straight path, or turning a sharp corner. We dramatically decrease the lookahead distance at the end of trajectory segments to compensate for this uncertainty, so that if we do indeed make a sharp turn, we can do so as accurately as possible.

Putting the above together, we arrive at the following lookahead distance equation (2).

$$ld = \log_{10}(L) \cdot [0.7 \cdot (1 - a) + 0.3] \quad \textbf{(2)}$$

Where L is the length of the segment closest to the car, and a is the fraction that the car is along the trajectory.

4 Experimental Evaluation (Lili)

After implementing the path planner and the pure pursuit algorithm, we needed to ensure that they worked reliably and in different scenarios. The two parts of this lab needed to first be evaluated alone, each with their own set of requirements. After both performed well alone, we needed to test how well they worked together.

4.1 Test Procedures (Lili, Matthew)

For testing the path planner alone, we wanted to observe our algorithm's solutions in two main ways: how quickly it came up with a path, and how optimal it was. To test for optimality we mostly relied on the Gradescope autograder, as we did not have a set optimal solution for every path we tried. However, we could see when testing turns and straightaways if there was behavior that was clearly suboptimal. So part of our testing was qualitative. The path planner did not need to run in real time, so it was okay for it to perform slowly. However there was still a desired lower limit for the speed to be reasonable. When testing the path planner we tested many different scenarios and lengths of paths. With our heuristic, sometimes the path planner

would have trouble searching for the optimal path that went far past a very sharp turn. Therefore we made sure to include sharp turns as one of the examples of scenarios we made sure to test. The path planning autograder had specific conditions that would result in a score of zero points. First, the autograder gave only two minutes to find a certain optimal path that covered part of the basement floor, so we used that as a metric on whether or not our planner was quick enough. Second, it would abort any path that entered an occluded area on the map (i.e. an obstacle). Third, it would fail any solution that had a cumulative deviation from the path that exceeded `delta_path_max`*`path_length`.

We performed initial tests of Pure Pursuit by inputting various specific predetermined paths into the simulator, and visually inspecting the robot's ability to follow these paths. We also ensured that we tested a variety of scenarios and path lengths for Pure Pursuit, and included sharp turns and corners. We also used the autograder for more quantitative testing. The autograder assigned a score based on the percentage of the path that the robot could follow before encountering a failure condition. There were similar failure conditions to the path planning. However, we had 500 seconds to follow our path instead of 120, and the penalty for deviation from the path was *instant*, not *cumulative*.

4.2 Test Results

For the path planner, we tested multiple different start and end points. From our visual observations, the path planner never went into occluded regions of the map, meaning that the dilation and occupancy grid transformations were working as expected. Prior to adding dilation, the path would sometimes slip through the pixels of the walls and cut through in areas that it was not allowed to be in. We can see in Figure 4.1 the path it generated for the start and end points given in the autograder. In addition, we saw that our robot could find a path between almost any pair of points on our map within two minutes on our machines. This was also encouraging, although we were concerned that we would lose time if the autograder simulation was less efficient than our machines. Finally, we saw that the robot sometimes made possibly significant deviations from the optimal path that we observed visually. This is most visible in Figure 4.1 near (900, 400).

The fact that we passed all of the gradescope tests for path planning proves that we did not violate any of the three conditions for path planning failure. First, our official time on the

autograder simulation did not exceed two minutes. This confirms that our informal benchmarking on our own machines was accurate, and that our algorithm was fast enough. Second, we have a precise, digital confirmation that we did not enter any occluded areas in our map. This backs up our visual inspection of our solution. Finally, it proves that we did not deviate a significant cumulative numerical amount from the optimal path. This assuages the concerns that we had when we saw our solution on the map. It appears that, although there was a significant deviation from the optimal path, the cumulative deviation was not enough to cause a test failure. We received an overall score of 90.49%. We did not find an in-depth explanation of the exact metric used to compute this score.

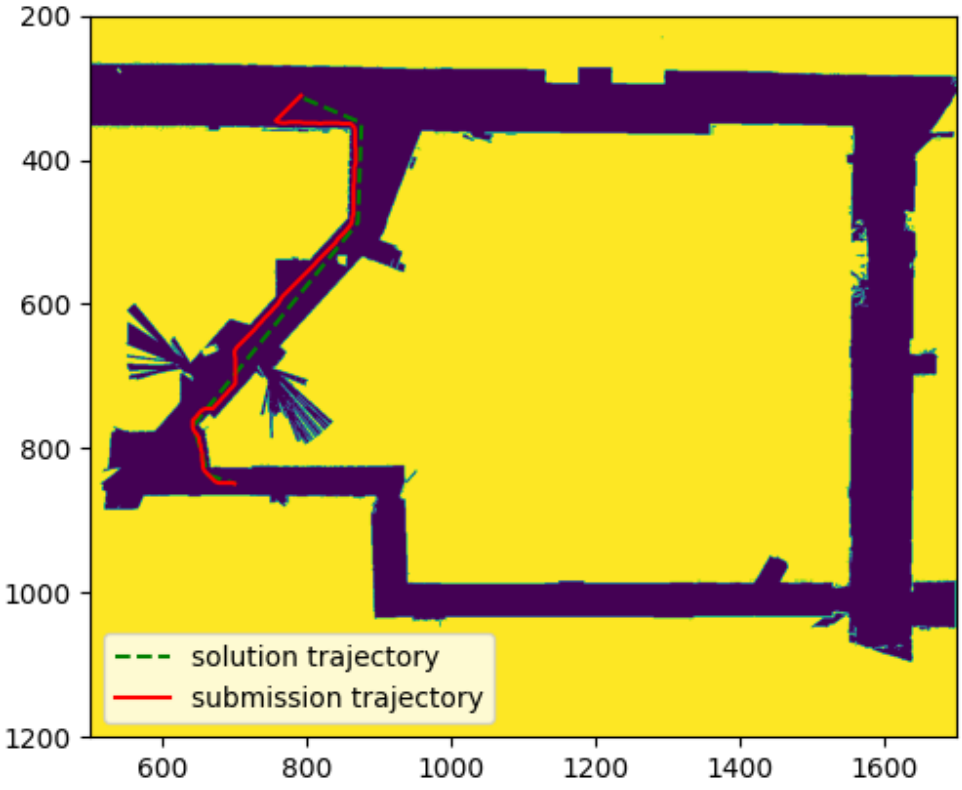

Figure 4.1: Path Planner in Simulation

The pure pursuit also proved to work very well. It followed 95.09% of the path in the autograder before it deviated more than 1.0 meters from the optimal path, which ended the test. The result is shown in Figure 4.2. This means that neither speed nor object collisions were a problem for our robot.

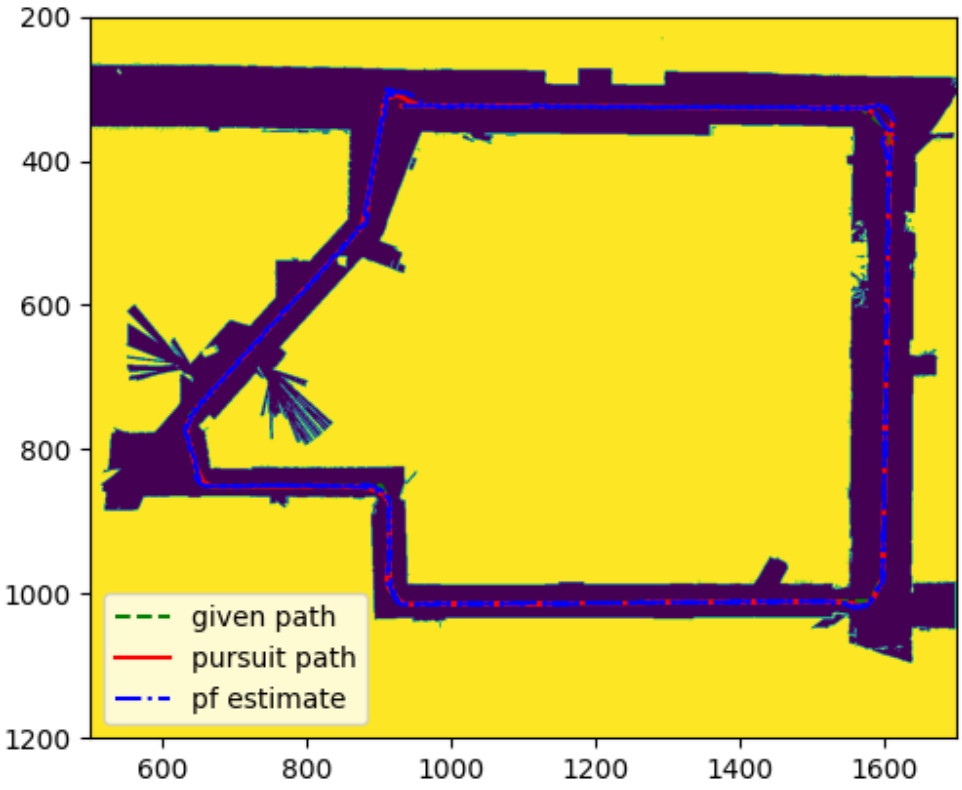

Figure 4.2: Pure Pursuit in Simulation

Finally, we received similarly high scores when we integrated the two aspects of the lab for the final test. We scored a 97.36% for planning and following the path in Figure 4.3.

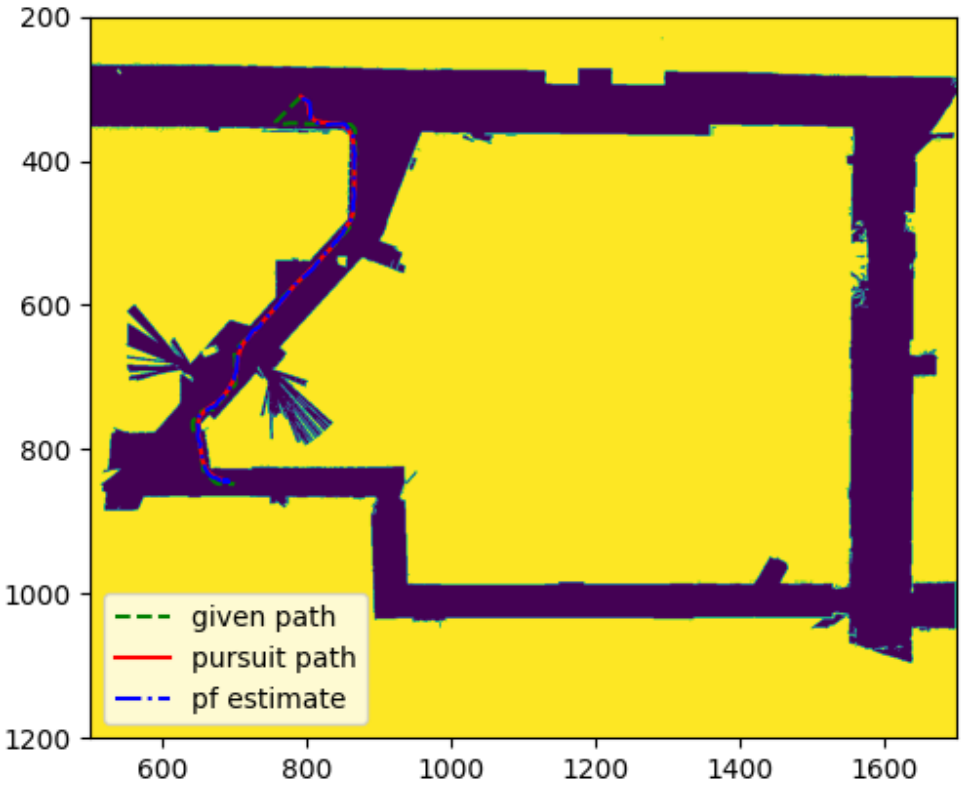

Figure 4.3: Integrated Path Planner and Pure Pursuit in Simulation

4.2.1 Test Limitations and Observations

A* will only work as well as the cost and heuristic are defined. Since our cost heuristic is only based off of the position of the robot, it is likely that we could achieve a higher score for optimality by using a more realistic heuristic such as a Dubins curve. The Dubins curve will ensure that the robot only considers paths that are physically realizable, which should improve our path planning. Because the robot sometimes seems to classify path optimality based on Manhattan distance instead of Euclidean distance, there is also possibly a bug in our heuristic code. However, as we have already achieved scores in the 90s, this test does not provide a significant incentive for fixing the strange deviation we noted in figure 4.1.

Conclusion

We see that our path-planning and following algorithms performed very well. Our path-planning algorithm ran within the two-minute time limit, even when it was required to find an optimal path through the Stata center basement. The basement represented a sizable proportion of our occupancy map, which had over a million squares and an exponential number of paths that might need to be checked in the worst case. Because our path paneer successfully found a path to follow to the goal point, we can conclude that our heuristic served as a good predictor for path optimality in simulation, and that our algorithm effectively leveraged the information it had to prioritize better paths. Although we cannot be certain about what situations our robot could be deployed in, we believe that it is likely any occupancy grid map of a realistic location will also contain only obstacles that follow broadly geometric patterns, as well as regions of many adjacent squares of empty space. We believe that the heuristic we developed will work for most realistic maps.

Our Pure Pursuit controller also functioned quite well, and our robot followed each path it found in a wide variety of situations including straight paths, corners, and curves, with minimal error and smooth behavior. It could handle essentially any situation that would naturally occur in the real world. We are therefore confident that, in a race situation where the robot is required to plan and follow a path quickly, it will perform well.

However, there remains some work that could be undertaken to improve our system. We also noticed various forms of odd behavior from our robot. For example, it chose a path that was longer than the optimal path because both paths had the same ‘taxicab distance.’ It also occasionally lost complete control in some situations when it tried to follow paths with Pure Pursuit. Because our algorithm was a complicated system with many interacting components, the root causes of these behaviors were difficult to identify and address. However, many of these behaviors did not significantly impact the robot’s overall performance, and we addressed the ones that did by adjusting relevant parameters. In the future, we could scrutinize these deviations more carefully and find any underlying problems in our implementation that might be causing them.

Although our final challenge does not directly require path planning, or following a specific pre-planned path, it is likely that we will have an opportunity to use at least some of what we have done here to improve our final challenge score. Although we will not know the layout of the track in advance, it is possible that we will be able to develop some kind of advanced knowledge of the path we will follow, possibly through vision. We could then use our pure pursuit algorithm to follow that trajectory closely.

Lessons learned

Miles: I learned how to implement pure pursuit, and why it is harder than it first appears! I also learned how to optimize our algorithm to improve accuracy, and run more robust tests of our algorithm's performance. On the CI side, I have become a much more confident and capable presenter, and I learned to create more engaging and informative visuals (such as our pure-pursuit visuals). Our meeting with the CI instructor was helpful, and one of my biggest takeaways was creating more interesting and useful title and end slides for a presentation.

Irene: I learned a lot about implementing search algorithms in different frameworks and that the most challenging part of getting the algorithm to work was being able to understand the search space the problem was giving us. To get the algorithm to run in simulation, involves understanding the way the occupancy map is exactly defined and how to discover that information when the documentation is unclear. On the CI side I learned a lot about how to make a more effective presentation. Our team's talk with our CI instructor was very helpful and the feedback on the lab report helped me understand how to write a better report.

Lili: I learned more about the importance of using good data structures when implementing certain algorithms. I definitely underestimated the importance of them, and never actually had a good sense of what O(log n) looked like against O(n) in real life. Part of our debugging included visualizing the point the algorithm was currently expanding, and after changing data structures, it moved visibly so so so much faster than before. On the CI side I learned a lot about keeping a smooth presentation. During our presentation, unfortunately our videos didn't play as we were presenting on a staff member's computer. Although unexpected, our team handled it well and we continued the presentation, both staying informative and keeping within the time limit.

Matthew: On the technical side, I learned that it is important to develop an accurate visualization system for as much data as possible when debugging a new, complex system with many data points. I spent a lot of time trying to figure out what was wrong with the coordinate conversion, and tried to develop an ASCII visualization tool but it did not work. I also learned to be prepared for documentation to be incomplete or misleading. The order of transformations specified in the map documentation did not end up being correct. For communication, I got practice speaking up and presenting at a loud volume, even in a room with a lot of background noise and distraction. For teamwork, I realized that I should have incorporated my teammate's suggestion to ask for a loaner laptop when mine stopped connecting to the internet. I'm currently working on getting that fixed.