Lab #6 Report: Path Planning and Pure Pursuit for Competitive Autonomous Navigation

Team #1

Nick Dow
Dane Gleason
Sadhana Lolla
Sienna Williams

RSS Spring 2023

May 1, 2023

1 Introduction

Dane Gleason
The overall goal of our robot is to race other robots on a closed loop track, completing the race in the fastest time while operating completely autonomously. Previous labs have built up to this with various components necessary to complete this goal. Most recently, Lab 5 developed a Monte Carlo Localization (MCL) algorithm to determine the approximate position of the robot in its environment. Using the odometry output of the MCL, this lab provides the final critical component of functionality: autonomous navigation using path planning and a pure pursuit controller.

Autonomous navigation is a key last component as it inherently satisfies our goal of autonomy, fills remaining functionality gaps, and provides means for long-term goal seeking. By automating navigation, the robot will be able to independently race as desired and perhaps even more importantly, the possibility of optimization is introduced. Being that our robots primary functionality is to drive, path planning is an obvious next step in our progression and brings us closer to a competitive robot for the final challenge. Finally, for any autonomous system it is important to define goals, such that the system has some criteria to operate within. Path planning provides these goals in the form of real-world waypoints that necessarily go beyond simple control.

Our implementation of autonomous navigation requires two basic components: path planning and a controller. The path planner uses the environment map,

starting location, and a goal location to define a trajectory between the starting and goal locations. Path planning is widely implemented through a variety of well-known algorithms, with trade-offs between optimality and computation speed as well as trajectory smoothing methods. Our solution uses an optimized A* algorithm to balance each of these trades. The A* algorithm outputs a sufficiently optimized trajectory which is then used as an input for our controller. Our solution uses a pure pursuit controller as required by Lab 6. The controller must issue necessary control for the car to follow the trajectory. For pure pursuit, this can be divided into defining what section to follow and how to follow that section, for which we used our MCL solution and repurposed parking controller. With functional path planning and control, we were able to integrate these into autonomous navigation.

Finally, we tested our path planning and pure pursuit controller in simulation, on the robot, and in carefully designed experiments to quantify and improve performance for the final challenge. Experiments include qualitative evaluation such as in evaluation of path optimality with comparison to a true optimum as well as quantitative evaluation such as the error measured as absolute minimum distance from the desired trajectory. Combining Lab 6 with our previous work including wall following, visual servoing, and localization, will allow us the building blocks to integrate into a final solution capable of racing our robot autonomously.

2 Technical Approach

Nick Dow

The path planning module required two core components to be implemented: the path generation algorithm and the pure pursuit algorithm. The path planning algorithm presented an open problem that we could solve with a variety of algorithms, each with their own advantages and drawbacks. These algorithms can be broadly categorized into two groups, sampling-based algorithms and search-based algorithms. Considering the potential of each algorithm and the search space we were exploring, we decided to implement A* for our path-planning algorithm. Pure pursuit is a bit more straightforward, as we follow the path generated by the planner, although there is some significant logic in that simple objective. In addition, there are some controls that we implemented to make the pure pursuit more robust to slightly faulty plans, such as one that takes us too close to wall.

2.1 Path Planning

2.1.1 Comparison of Path Planning Algorithms

Sadhana Lolla

The first step to planning and following a path is being able to find the short-

est path from two points on a given map. To this end, we considered several different path planning algorithms and evaluated their speed and accuracy. We looked at sampling-based algorithms, which probabilistically sample potential paths that could lead to the goal, and search based algorithms search a defined space progressively until a path is found. Sampling allows a greater exploration of the path space, but is not guaranteed to return a path. Search based path planning is more likely to return a path, but is more computationally expensive and is less likely to be optimal.

Search-based algorithms: When considering search-based algorithms, we examined A*, Breadth First Search (BFS), and Dijkstra's algorithms. In terms of asymptotic complexity, Dijkstra's algorithm and A* have the same time complexity, since the heuristic we use for A* has an $O(1)$ complexity (Euclidean distance): the complexity is $O(E \log V)$, where E is the number of edges and V is the number of nodes. In this graph, since every node has exactly four neighbors, there are $O(4 * V) = O(V)$ edges, so the complexity becomes $O(V \log V)$. We also considere BFS, which has a runtime of $O(E + V) = O(V)$. BFS has the best asymptotic complexity– however, it's important to note that in practice, the heuristic speeds up A*, which is not captured in the asymptotic complexity– A* most likely will expand far fewer nodes than either BFS or Dijkstra's algorithm.

All of these algorithms are single-source shortest path algorithms, so they are all single-query algorithms. Since a path is recomputed every time a start and end pose is specified, all of these algorithms are also dynamic.

Sampling-based algorithms: For sampling-based algorithms, we consider Rapidly-Exploring Random Trees (RRT), RRT*, and Probabilistic Roadmaps (PRM).

PRM is a multi-query algorithm, which means that it retains its graph structure between queries and therefore is quite fast at returning paths between different start and end points in the graph when necessary. However, if the graph space is large, as it is in our case, building this graph representation running PRM will take a long time since no path is being targeted, so the entire graph is explored. This also means that if the map changes for some reason (ie a new obstacle), the entire algorithm must be rerun, so this is not dynamic.
On the other hand, both RRT and RRT* are single-query algorithms and also adapt to dynamic changes in the map between runs.

Discussion of Tradeoffs: We quickly decided that we would prefer a single-query algorithm to a multi-query one, since we often just create one path and traverse it. Therefore, we opted not to use PRM. Within the search-based algorithms, we chose to use A* since the heuristic makes it particularly attractive for our use case, where an admissible heuristic (Euclidean distance) is readily available. Between A* and RRT*, we chose to use A* because we are given an

easily discretizeable space to which A* applies well. However, given more time, we'd also like to try RRT* since it is quite fast as well.

2.1.2 A* Search Algorithm

Sadhana Lolla
We decided to use an optimized version of A* to effectively balance the tradeoff of speed and accuracy. A* is a heuristic based search algorithm that is not always guaranteed to return the shortest path between two points, but explores nodes in the order of their distance from the goal. This heuristic ensures that we somewhat greedily explore nodes, which increases efficiency.

We first convert the continuous map space to the discretized pixel space, where every pixel is a node in the graph. We then start exploration at the start node and consider a node's neighbors as the nodes directly above, below, and to the left and right of it. At every node, we consider all of its neighbors if they are valid (within the bounds of the graph and do not have an obstacle). We add these neighbors to a priority queue that is keyed by the length of the shortest path through the neighbor added to the Euclidean distance between every neighbor and the final goal position. At every subsequent step, we pop off the priority queue and repeat until we've reached the goal. It is critical to note here that if there is a path, A* will definitely find it.

2.1.3 Map Dilation

Sienna Williams
In order to make the planned path feasible for the actual robot to drive, we had to ensure that the path did not go too close to any walls or other barriers. This was done through map dilation which increases the thickness of the walls of the map. When the map wall's thickness is increased the planned path is much further from the wall. This procedure does make the path less optimal, but this is necessary to protect the expensive components of the robot. Without map dilation, the safety controller of the robot could interfere with the robot's driving and end up sacrificing even more time during the final challenge. Figure 1 shows a map that has been dilated by a 10 pixel radius.

The map was given as an input in the form of an Occupancy Map that gives values between 0 and 100 that is the probability of there being a barrier in that pixel. There are also values of -1 for pixels that have an unknown occupancy. The map was first transformed to have only values of 0 or 1. The original values of less than 50 were set to 0 and values greater than or equal to 50 as well as values of -1 were set to 1. This ensured that pixels with a high probability of having a barrier as well as unknown pixels were treated as barriers to ensure the safety of the robot. Then the function ndimage.binary_dilation was applied to the array and this converted all neighbors within a 30 pixel radius of a 1 value into additional 1s, hence increasing the thickness of the barriers.

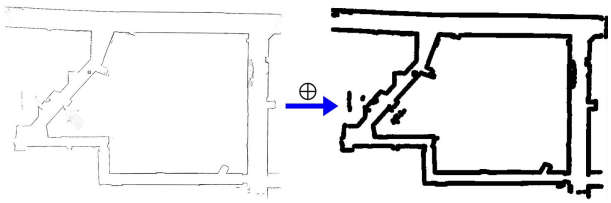

Figure 1. A map that has been dilated with a 10 px radius.

2.2 Pure Pursuit

Nick Dow

The pure pursuit module has two main problems to solve when following the path given by the path-planning algorithm: knowing what part of the path it should follow and how to follow that part of the path. We solve the first problem we begin by using odometry input from our particle filter localization module from Lab 5. Then we sort the trajectories of the path by their distance from the robot. Taking the closest trajectory, we increment through each trajectory of the path until we have an intersection with our lookahead distance. This goal point is solution to what part of the path to follow, so now we solve how to follow this point. With some safety controls built in prevent the robot from getting too close to the wall, we do simple proportional control to follow this point.

2.2.1 Odometry and Lookahead Intersection

To solve the first problem, the robot starts by finding it's current location in it's environment; we use our lab 5 particle filter to provide localization input to the robot. As a brief refresher, the particle filter maintains a set of particles that are probabilistic "guesses" of what the robot's pose is currently. A sensor model then takes in Lidar data and compares it to virtual Lidar data would be generated by each of particles; it uses that comparison to prune unlikely particles and keeps those that can explain the Lidar scan well.

Our internal representation of the trajectories of the path is an array of poses. For the next step, we generate a set of line segments from adjacent poses in the array. We then sort the line segments by the distance of the closest point on that segment to the robot position using the line-point distance calculation:

$$\text{proj}_{\mathbf{v}}\mathbf{w} = \frac{\mathbf{w}\cdot\mathbf{v}}{\|\mathbf{v}\|^2}\mathbf{v}$$

$$\|\text{proj}_{\mathbf{v}}\mathbf{w} - p\| = \text{distance to line segment}$$

After performing this calculation for all line segments, we find the trajectory with the shortest distance to the robot.

Now that we've found the part of the path that is closest to the robot, we then start our search for the point on the path we follow starting at this line segment. We want to follow the part of the path a certain distance in front of

the robot to make following the path smoother, this parameter is the 'lookahead_distance'. Starting at this segment, we calculate the intersection of a circle of radius 'lookahead_distance' centered on the car's position and the line segment. If an intersection exists, this might be the goal point we wish to follow. However, there are a number of edge cases to this intersection problem:

1. The segment might intersect the circle in two places, and we must resolve which to follow.
2. Most of the segment might be behind the car on the path, so the intersection would be behind the car. This would direct the car in the wrong direction.
3. There might be no intersection at all.

For the first two cases, we resolve them by ignoring the points that are behind the midline of the car; that is, we only consider points that intersect with the 180 degree arc of the the circle centered on the front of the car. If two points exist in front of us, we chose the point closet to the trajectory after the current trajectory. If no points exist in front of us or at all, we repeat this intersection with the next segment in the path.

2.3 Proportional-Integral-Derivative Control

Nick Dow and Dane Gleason

With the goal point calculated, the robot needs a drive command to follow it. The goal point is calculated in the world frame, so we first transform the point into the robot frame via a transformation matrix calculated from the robot odometry. The equation is shown below, with $\mathbf{p}_r$ and $\mathbf{p}_w$ being the goal point in the robot and world frame respectively, $\mathbf{T}_{wr}$ being the transformation matrix from world to robot, and $\mathbf{p}_{ro}$ being the robot's position in the world:

$$\mathbf{gp}_r = \mathbf{T}_{wr}\mathbf{gp}_w - \mathbf{p}_{ro}$$

We calculate the θ offset of the point from the front of the car by taking $arctan(y/x)$. We use this angle as the driving angle for the car. This approach is adapted from our parking controller solution, which used proportional control such that the driving angle was directly proportional to θ. This solution, however, has proven to be insufficient as seen later in experimental evaluation. Accordingly, derivative and integral control has been added to this adaptation to minimize oscillations and steady-state error while following the A* trajectory.

3 Experimental Evaluation

3.1 Path Planner

3.1.1 Test Plan

Sienna Williams
The path planner needed to produce fast, optimal, and feasible paths. The quantitative metrics that we used to determine analyze the path planner were the time that it took to find a path and the percentage of time that the path was infeasible either going through barriers or going too close to them. The qualitative metric that we used to analyze the path planner was how optimal the path looked based on better paths that we could see. This was very subjective because we could not compare our paths with the most optimal path since it could not be easily computed, but our intuition about the optimality of a path is still a reliable measurement.

3.1.2 Test Results

Sadhana Lolla
We tested a variety of paths (pictured) ranging from straight, direct routes to routes that needed to hug a wall, to paths that needed to avoid a variety of obstacles in the map, and very long paths. In all cases, we found that the time taken to generate a path did not take longer than 11 seconds, and with shorter paths often took less than a second, even with obstacles in the path. Example paths with time taken to follow them are shown in Fig 2. We believe that this an acceptable number of seconds, since the path was only generated once at the start of every run in integration. However, if there was a scenario where we needed to generate more paths in real-time, we may want to explore sampling-based approaches that might find these paths faster.

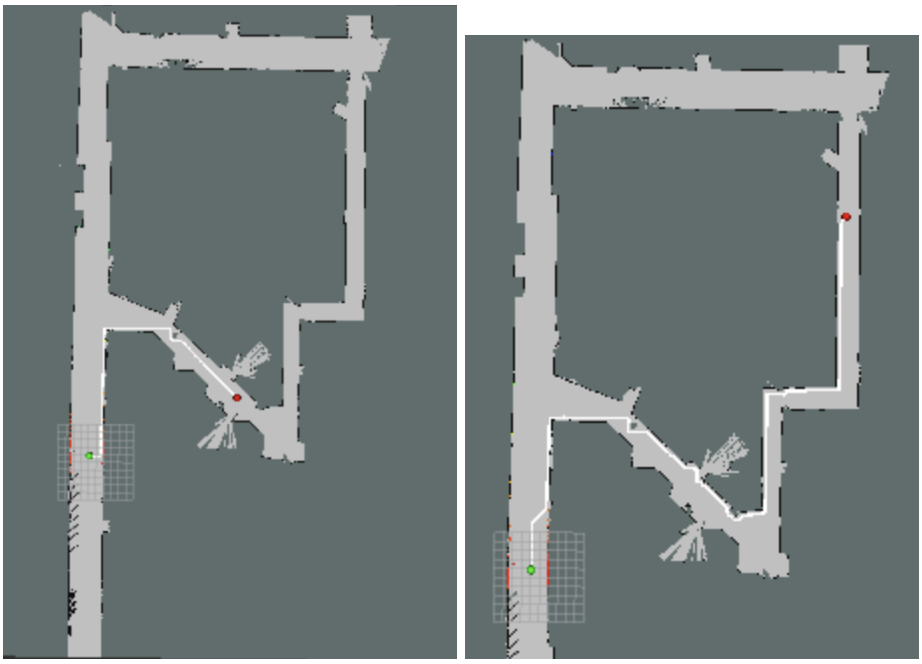

Figure 2. Left: a path with many blocked off obstacles. This path was found in 1.04s. Right: the longest path that we tested. This path took 10.6 seconds to generate.

3.2 Pure Pursuit

3.2.1 Test Plan

Sienna Williams
Pure pursuit outputs drive commands that should lead the robot to follow the path as closely as possible. It aims to minimize the distance that the car is away from the path. To quantitatively analyze its performance, a rostopic called error was created. This topic published the distance difference between the robot and the closest point on the path to the robot at every time step. Some oscillations in this data were expected due to tuning in PID control parameters, but the average error throughout the path gives us an idea of how well pure pursuit is able to follow a path.

3.2.2 Test Results

Sadhana Lolla
We tested our pure pursuit algorithm on both paths that we generated using A*, and using the staff-given paths. We found that in simulation, the error was quite low (as shown in 3) but that it gradually increased as the number of seconds that the pure pursuit was run for increased. We qualitiatively noticed

that the car turned corners quite poorly in the path, which we attribute to gains not being tuned entirely when using PID control. Once the car was off the path, it turned in circles until the path was found again, which may also lead to the spikes in error shown below. We found that even with the turning, the car was not able to find the path very well, which is why the error doesn't decrease and stabilize again. With further tuning and a more robust strategy for staying on the path, we believe that the pure pursuit algorithm will work well regardless of drive scenarios. On gradescope, this algorithm scored a 2.8/3.0, indicating that it was quite close to the staff solution. This means that such oscillations are admissible in the context of pure pursuit.

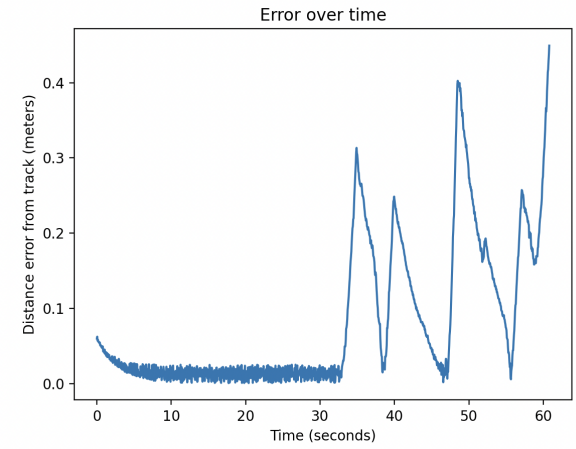

Figure 3. A graph of distance between the car and the path while following it on a path generated by our A* algorithm.

3.2.3 Integration on the Robot

3.2.4 Test Plan

Sienna Williams
Combining the path planner and pure pursuit, we are able to give the robot a start and end point, it calculates a path between the two and drives along it. This implementation was successfully tested on the robot. The path planning and pure pursuit were qualitatively analyzed through videos of the planning and following of the trajectory of various paths.

3.2.5 Test Results

Sienna Williams
At first there were large oscillations as the robot drove along the path, but these were significantly decreased with the addition of PID control. The robot also had difficulty making turns which we also believe is due to sub-optimal tuning

of the PID parameters which will be improved upon for the final challenge.

Additionally, the location of the 2D Pose Estimate start position was critical to ensuring that our localization could run correctly. It was difficult to make the pose estimate correct as the Stata basement had very few landmarks. An incorrect start position led to our robot turning corners too quickly because of incorrect localization. This can be imporved upon with more testing and trial and error of the start position location.

4 Conclusion

Dane Gleason

With the successful implementation of autonomous navigation through path planning and control, we have synthesized the final piece of the autonomous racecar puzzle. Using an optimized A* algorithm, we were able to create trajectories between two points on a given map while balancing solution optimality and computation speed for a competitive and robust solution. Our implementation of a pure pursuit controller took advantage of our previous Monte Carlo Localization algorithm and an adapted parking controller to follow the trajectory given by the path planner in an fast, yet safe, manner. Our solution was refined in simulation for faster performance, and tested in a variety of real-world experimental trials for quantitative performance evaluation. These trials demonstrated basic automated navigation functionality for Lab 6, though we aim for further improvement and tuning of these algorithms for implementation in the final challenge. Notably, while the performance of our path planning and pure pursuit was quite successful in simulation, performance lacked in robot integration. While the car handled straight stretches, corners and curves were challenging and seem to expose remaining bugs in the real-world implementation as evidenced by large oscillations and loss of path tracking. We are confident moving forward that these performance discrepencies are addressable as backed by our performance in simulation, with time being the limiting factor in this case. Looking forward, we plan to combine the autonomous navigation capabilities established in this lab with the functionality demonstrated in previous labs to deliver a complete and competitive solution to the racecar final challenge.

5 Lessons Learned

5.1 Sienna Williams

I learned that it is important to remember all the little details and to write down how you have fixed errors in case you encounter them again. Our group ran into many of the same errors throughout Lab 6, but wasted time trying to remember how we fixed it previously instead of keeping track of the errors and solutions we have found. I also learned that it can be quite annoying to set up

all the windows that are neccessary for running programs on the robot. It is important to keep our battery charged and router plugged in to avoid having to restart the terminals. Overall, this lab was fun and rewarding once we got everything to work.

5.2 Dane Gleason

I have learned that even with multiple individual parts of the lab working, it is critical to give enough attention to integration of each of these parts. This lesson was reinforced from previous labs, and was especially pertinent with the necessity of dividing tasks in parallel for this lab. I also learned the importance of the project as a bigger picture, with this being the final piece of functionality required for the final challenge, as well as many of the features of this lab being repurposed or adapted code from previous labs. Even though our implementation wasn't as perfect as we might have hoped in terms of performance, I know we are that much more aware of these areas for improvement going into the final challenge and with this, see lots of opportunities for learning through the debugging and integration process going forward. Overall, this lab was challenging but rewarding and I look forward to seeing everything come together soon.

6 Sadhana Lolla

We realized the importance of starting early and keeping each other on track during this lab. Due to unforeseen events such as sickness in addition to travel and exams, we did not start this lab until quite late, which significantly hindered our progress. However, we all spent many hours towards the latter part of the lab making sure that all the parts came together, and I realized the importance of allotting more time for integration. Although all modules worked well in isolation, putting them on the robot proved more challenging than expected, and we did not have time to iterate on this solution as much as we would have liked.

6.1 Nick Dow

I learned value in quick experimentation along with proper documentation of that experimentation. When I was developing the pure pursuit controller I quick iterated through different changes and sometimes would have to repeat experimentation when I forgot how a certain change behaved. I quickly learned to start logging my progress to keep track of what I had tried and not waste work.