Lab 6 Report: Path Planning

Team #10

Diego Delarue
Jack Lewis
Jason Lee
Vlada Petrusenko

RSS (6.4200)

April 27, 2023

1 Introduction (author: Diego Delarue)

After having implemented localization in the car, we now have the ability to know where the car is on the map. The next step is to use this information and a set goal point to take the car to said goal point. In order to do this, we split up the task into three steps:

1. Path Planning: Finding an efficient path from an initial pose to a goal.
2. Pure Pursuit: Taking a path to get to a given goal.
3. Integration: integrate Lab 5's particle filter with path planning and pure pursuit to get the car to move to a given location in a map.

We chose to split the problem of getting the car to move to a desired goal in this way in order to be able to work concurrently in path planning and pure pursuit before integrating.

For path planning, the main question to ask is which path-finding algorithm to use. The algorithms we considered were:

1. Breadth First Search (BFS)
2. Dijkstra's Algorithm
3. A*
4. Rapidly-exploring Random Tree (RRT)

Given the desired characteristics of having a near-optimal path, short computation time, and achievable path given our pure pursuit, we found that A* fits the best to address this problem. Even though some technical difficulties did not allow for our implementations to be fine-tuned using the car, A* showed a lot of promise finding a good path in an acceptable amount of time.

For pure pursuit, the big issue to address was a way to get the car to follow a given path. What we found to be the best solution was to create a stream of odometry messages containing velocity and pose to get the car to follow a path made of points on the map. This involved some calculations to estimate the car's path given a pose and velocity and gave promising results in simulation.

Overall, the individual modules to get the car to go to a desired location in a given map got done. However, further integration with the real life car is still in progress.

2 Technical Approach

2.1 Path Planning (author: Jack Lewis)

2.1.1 Framework

The path planning ROS node receives data about the world through an OccupancyGrid, which divides the world into a grid and reports how likely each cell is to be occupied. We use this grid directly for our pathfinding algorithms, only allowing paths that stay only in cells we are sure are not occupied. We also add a buffer around all obstacles by marking all of the cells in a radius of any occupied cell also occupied. This prevents us from choosing paths that go through gaps the car couldn't fit through or would cause the car to clip a corner or otherwise hit an obstacle.

2.1.2 Algorithms Considered

We started out with a simple BFS that considered the four adjacent cells to every cell and returned the path of fewest cells from the cell containing the start point to the cell containing the end point. We also randomize the order in which it considers the four neighbors to stop the car from always choosing to go fully north and then fully east in a section of the map in which it would be fastest to go directly northeast.

Next, we considered both Dijkstra's and A*, which both actually look at the distance between nodes, solving the diagonal problem from BFS. These algorithms both look at the shortest path to the cells they consider, but A* looks at cells it thinks will be more likely to get it closer to the goal faster. To do this, A* sorts the paths by the sum of their current length and the Euclidean distance from the current end of the path to the goal cell.

We also considered RRT, which was our only sampling-based algorithm (as opposed to the others, which are search-based). RRT chooses a random point in the space (for us, a random cell in the OccupancyGrid), finds the closest cell we already have a path to, and connects them if there are no obstacles in the way (for us, if all of the cells along the line between them are not occupied). Once it chooses a cell close enough to the goal cell, it follows the parent pointers back to the start cell to determine the path.

2.1.3 Choosing an Algorithm

To help choose an algorithm, we tested all of them on a path through a large portion of the Stata basement.

Figure 1: (left to right) BFS, Dijkstra's, and A* paths through the basement, found in 0.71s (BFS), 79.97s (Dijkstra's), and 54.92s (A*)

BFS was consistently the fastest, but even with the neighbor randomization, the paths rarely deviated from the eight (inter)cardinal directions and tended to wander around a lot. It never deviated far from the optimal path, but there were certainly places to improve. Dijkstra's and A* were both much better, giving near-optimal paths (though they were likely still held back slightly by only being able to choose one of eight directions for each step), with A* finding the path much faster due to its prioritization of likely helpful paths.

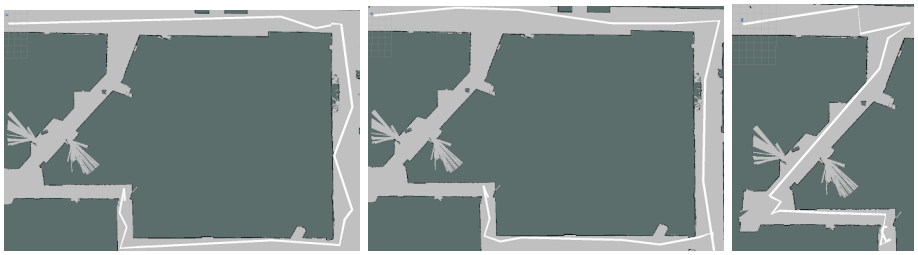

Figure 2: Three RRT paths through the basement, found in (left to right) 54.18s, 3.36s, and 29.80s

RRT was much less predictable in both its paths and the time it took to find them due to its nature as a sampling-based algorithm. While it could find a path very quickly if it got lucky, it would often spend a long time searching through the space before it found a path to the goal. Even after searching for a long time, it would often find very suboptimal paths, especially if they went through more open areas while the optimal path goes through more narrow areas.
Since we are only calculating the path ahead of time and not repeatedly while driving, we are okay with taking a while to find the path, and A* produced the best paths the most consistently while still finding them in a reasonable amount of time, so we chose to use A*.

2.2 Pure Pursuit (author: Jason Lee)

Given a fixed trajectory, the goal of the pure pursuit controller is to drive along the trajectory by navigating to a point a fixed look-ahead distance away. As such, every time we receive odometry data for the car we adjust the steering angle, which at the rate of odometry updates is near continuous. The calculations behind the pure pursuit controller can be split into three separate components: determining the nearest point on the trajectory to the car, finding the look-ahead point on the trajectory a certain distance away, and calculating the steering angle necessary to meet that point.

2.2.1 Nearest Point

From the odometry data, we receive (x, y) coordinates of the position of the car, and treating a trajectory as a connected set of line segments, we're able to calculate the distance between the point and each segment to determine what the closest point on the trajectory is. If u and v are the start and end points of a given segment, the point on the line defined by u and v that is nearest to point P is the projection of P onto $\overrightarrow{uv}$, i.e. the point on the line which would form a perpendicular. However, in most cases this projection point won't lie on the segment itself, rather the line formed by extending the segment past its endpoints. Therefore, we use the parametrization of the line $\overrightarrow{l} = u + t(v - u)$ and restrict $0 \leq t \leq 1$, where t represents the fraction of the way along the segment that the nearest point lies.

Individually calculating these distances for each segment is extremely time-intensive as there are many segments in a trajectory, and calculations would finish much slower than the next odometry reading. To prevent bottlenecks in the system, we use vectorized array methods from the module numpy, which allow us to perform calculations for all segments in batches.

2.2.2 Look-ahead Point

We now calculate a look-ahead point on the trajectory, which is a fixed look-ahead distance away from the robot car. To accomplish this, we begin our search at the previously calculated nearest point. Then, for each succeeding segment along the trajectory, we check to see if there are any intersections between the segment and the circle centered at the car with a radius of the look-ahead distance. We iterate through the segments until such an intersection is found. Using the same parametrization of the line segment as earlier, the intersection forms a quadratic equation w.r.t. t and we once again must make sure solutions to the quadratic equation lie in the range $0 \leq t \leq 1$.

There are a couple edge cases we addressed in our implementation of the pure pursuit controller. Firstly, if the nearest point on the trajectory is further than the specified look-ahead distance, then no points on the trajectory would satisfy the distance constraint. Therefore, we choose to continue driving forward in an attempt to eventually find the trajectory. Additionally, since the intersection quadratic equation can have up to two valid solutions in t, we implemented our algorithm to use the smaller value of t that satisfied our constraints. This corresponds to the point that comes first if a car perfectly followed the trajectory.

2.2.3 Steering Angle

Now that we have the point P that we would like to drive towards, we must calculate the steering angle that gets the car from the current position (C) to P when said angle is constantly applied. Because the odometry data has x, y, and θ components, we first calculate the coordinates of P in the reference frame of the car: this involves taking the relative position of P to C and applying a rotation of $-\theta$.

$$P_{new} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} (P - C)$$

From this, we take the element-wise arctan of P_{new} to determine η, the angle to P in the reference frame of the car. Finally, we use the Ackermann model assumption where L is our wheel-base length (experimentally determined to be 0.325 meters) to calculate δ, our desired steering angle.

$$\delta = \arctan \frac{2L \sin\eta}{|P_{new}|}$$

3 Experimental Evaluation (author: Vlada Petrusenko)

For conducting an experimental evaluation of our work we needed to test Path Planning and Pure Pursuit separately, and then evaluate them integrated. That would include simulation and on-car evaluation, however, we were only able to get the simulation data, and we would be able to provide a theoretical plan of

experimental evaluation on the car.

For evaluating Path Planning we first considered different Algorithms and their efficiencies, as more detailed described in Section 2.1.3. One of the evaluating parameters would be the time in which the algorithm successfully finds the path, as well as the overall precision and correctness of the algorithm. While the timing was the constraint that defined Dijkstra to be the least useful in this context, BFS was way more unreliable and "wandering around", so A* ended up being the good middle ground. While it is controversial to argue which metric should weigh more and what is the bottleneck in the overall system, we believe that both of the metrics have to be considered.

Another quantitative metric that we could have considered would be precision and the least square error of the path compared to the actual path. Visually, however, it seems like all of the algorithms successfully find the anticipated path, hence, this metric is not as useful as the timing. On the real car, a similar evaluation would be conducted using rosbags and some "ideal" path, and would be one of the future improvements of the project.

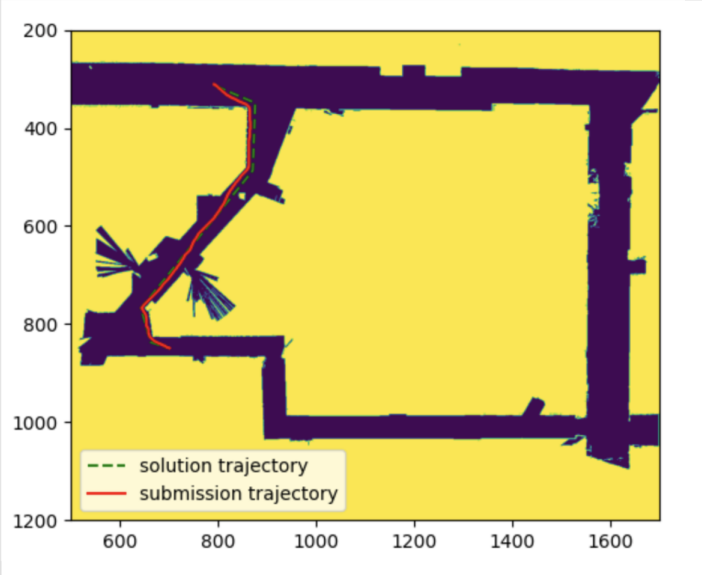

Figure 3: Path Planning simulation, Stata basement

In order to evaluate Pure Pursuit we needed to compare a found or predefined path with the module's results. While it was more challenging than we anticipated, and we were not able to get a quantitative evaluation from the car, a theoretical way to evaluate is the following. The car was positioned at some point on the map, and the initial position was aligned with the actual location.

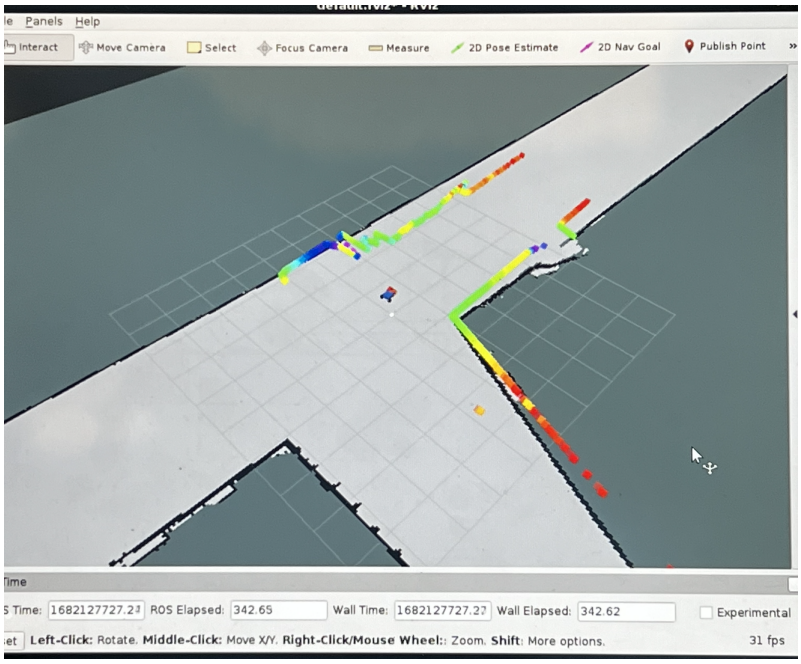

Figure 4: Aligning the car's initial location with the map

Afterward, we were navigating the car using the RViz environment, which generated a path for the robot and was moved along it. For more quantitative evaluation, it would be a good metric to compare the least square deviation of the actual car path as opposed to the planned path. This evaluated the quality of the path planning algorithm, and while simulation and physical environment have the same concept, the real environment can introduce some mechanical errors which would need further adjustment and will be considered in the future work on the project.

4 Conclusion

Overall, our algorithm selection, A*, allowed us to find paths from a given start point, end point, and map in a reasonable amount of time and accuracy. Pure pursuit used the calculated formulas to be able to successfully look ahead for the next points in a path and follow said path until the goal. Even though we were successful in testing these in simulation, we were not able to test them on the car. Difficulties with integration as well as technical difficulties with the car are some the reasons why we are still working on implementing these modules in the car.
As for the future, the integration of the modules on the car is still the of the highest priority. Once this is complete, fine-tuning the parameters to work best

on the real-life car will be done. Further research on path finding algorithms will be done, with special consideration for RRT* since RRT was the fastest algorithm and is combined with our current selection A*. The integration of the other modules for the final challenge will one of the most challenging parts so far, but taking the lessons learned about early integration as well as integrating several modules in the car (particle filter, path-planning, and pure pursuit) will help us be able to successfully get the car to only keep getting better.

5 Lessons Learned

- Jack: I learned a lot about different pathfinding algorithms and the inner workings of a min-heap. I think moving to the "integrate early, improve later" work schedule helped us get a better idea of how the pieces were all supposed to fit together, so I think we will use it again for the final challenge, probably trying to move integration even earlier so we can begin full testing as early as possible.

- Diego: I learned how what might sound like one task can be broken further into components that make the problem significantly easier to address. In this case, getting a car to go to a spot on the map ended up being broken down into localization, path-planning, and pure pursuit. These 3 components could be worked on independently of each other. To still have communication with the team and build in a way that will make final integration easier, we chose to start by pushing simple versions that work and working from there so we can test each other's components. An issue we had found was that integration as last so we did not really know what was happening on the other side. Even though the modules can be worked on independently, integrating early such that everyone build on the same skeleton proved to be helpful

- Jason: As addressed by others, we really recognized the difficulties of integration even when we started with a bare minimum product, and I learned that it's much easier to start from the minimal product and proceed with "stepping stone" improvements that expand the functionality of the product. I also personally learned about the difficulties of relying on remote auto-graders and the importance of getting a local simulation environment running, so we can see the results of personalized tests and pinpoint where errors are occurring.

- Vlada: I learned about the difficulty of parts' integration and unexpected issues that differentiate simulation from the physical environment. I also resonate with the idea that was mentioned above regarding breaking projects into parts and getting an "MVP" of each module to work first, so that they can be developed independently.