Lab 6 Report: Path Planning

Team 1

Vasu Kaker
Ben Evans
Emmanuel Anteneh
Johlesa Orm
Michelle Zbizika

6.4200/16.405 (RSS)

April 27, 2024

1 Introduction

Author of section: Ben Evans

The goal of this lab was to plan a path and execute controls to follow that path to its conclusion. Specifically, we implemented both sampling based planning as Rapidly-exploring Random Tree (RRT), and search based planning as Breadth First Search (BFS) algorithms in order to plan a trajectory. A pure pursuit algorithm was then implemented in order to follow that trajectory.

Given a map of it's environment, the robot was able to use the localization techniques from a previous lab (with an approximate location) in order to estimate its exact location. Using that starting point, a desired ending point, and the map, the robot used a planning algorithm to find a possible trajectory. The localization algorithms were further used to maintain an up to date location as the robot followed the trajectory using the pure pursuit algorithm.

The technical problem presented here encompasses a number of the tasks required for autonomous vehicles such as self driving cars, or logistics robots moving items through a warehouse. This problem then fits nicely within the framework created by previous labs which tackled problems with applications in the same sphere. Line following, wall or terrain following, and collision avoidance also all have clear applications in self driving cars and many other autonomous vehicles.

2 Technical Approach

2.1 Path Planning

2.1.1 Image Dilation/Erosion

Author of section: Michelle Zbizika

The obstacles were represented by an occupancy grid, which was a 2d array corresponding to x, y values in pixels, with value 0 if the space was free at 100 if the spot was occupied. In order to account for the real world size of the robot, we increase the size of the representation of the obstacles by the radius we desire for clearance around the robot.

2.1.2 Search Based

Author of section: Emmanuel Anteneh

Primary search based planning algorithms include BFS and A*. We implemented BFS to find an optimal path from our initial location to a given goal location. Our BFS implementation discretized the search domain into a grid space search space. Upon receiving a real world starting position and ending position, we translate them into the pixel representation of our grid space. We then initiate a standard BFS algorithm where a queue of potential paths to the goal state are built by repeatedly branching one step out from the end of the oldest generated path, discarding any paths containing coordinates that are occupied (have a value of 100 or -1 in the occupancy map). Once a path to the goal state is found, the coordinates are translated back into real world space and are used as the trajectory to be followed.

2.1.3 Sampling Based

Author(s) of section: Ben Evans, Michelle Zbizika

Primary sampling based planning algorithms include PRM (Probability Road Maps) and RRT (Rapidly-exploring Random Trees). An additional optimization of RRT is RRT*. We ultimately implemented two versions of RRT as well as one of RRT* in addition to BFS.

Our initial implementation of RRT was used for much of our early testing. This largely followed the standard algorithm as seen in *Sampling Based Algorithms for Optimal Motion Planning*(1), with one minor modification. This algorithm generates a search tree with the starting point as the root. Samples are taken randomly from the free space of the map, and the tree is extended towards the sample by a new node branching out a fixed distance from the nearest existing node. The new node is only added if there is found to be no collision on the path back to it's parent. The goal point is sampled with some elevated probability to bias the growth of the tree in its direction. This sampling and tree growth proceeds until a tree node is added at the goal point. At that time a path from the goal is traced back to the root through the parents of each respective node. The additional modification used for our implementation was to begin by checking if a collision free path existed directly between the start and goal points.

Unfortunately this first implementation of RRT was found to be somewhat slower than expected and occasionally failed to find a solution. When a solution was found, it was usually quite sub optimal, though this is often expected of RRT. Further review determined that the algorithm was implemented without a means of limiting the distance of extension for new nodes. A revised version of the RRT algorithm was then developed to address this limitation of the first. This new revision changed the data structure of the search tree from a linked list to an array for faster performance, as well as reducing the number of function calls and optimizing the collision checking algorithm. Finally, this revision was implemented with the intent to develop into the asymptotically optimal RRT* algorithm. To that end it included variables which are not strictly necessary for basic RRT, such as the cumulative cost to reach each node from the root. This second implementation of RRT offered significant improvements, with much faster performance and highly reliable solution finding. Despite these benefits, the optimality of the generated solution paths remained concerningly low.

To address the concerns regarding path optimality, a version of the RRT* algorithm was then implemented. Rather than write a wholly new algorithm, our implementation of RRT* directly uses the output of our second RRT algorithm. RRT* as described by Karaman and Frazzoli(1) modifies the standard RRT algorithm by rechecking all nodes nearby to any newly added node. In this process it determines, for each node, if there are shorter paths available by traversing back up through a node other than its original parent. Rather than do this rechecking continuously during the building of the search tree, our RRT* implementation takes a completed search tree and iteratively attempts to reduce the cost of the nodes until the costs converge to a minimum. While this approach likely results in a somewhat increased time to return a more optimal path, it does enable the robot to begin moving almost immediately using the less optimal path returned by RRT, and then transition to the more optimal path found by RRT* when that trajectory becomes available.

2.2 Trajectory Following

The predefined trajectory found by the path planner is followed by the robot by implementing pure pursuit and particle filter.

2.2.1 Localization

Author of section: Ben Evans

The localization nodes from lab 5 were used to estimate the location of our robot in real time. This localization protocol was largely developed using the techniques in the text *Probabilistic Robotics*(2)

2.2.2 Pure Pursuit

Author of section: Johlesa Orm

A pure pursuit algorithm was used to control our robot to follow a given trajectory using its estimated location. The algorithm receives an array of points representing the desired trajectory. Once the robot receives a path, it then evaluates its distance from each segment of the path until it finds the closest segment. The nearest segment serves as an anchor for the robot, in order to extrapolate the next possible points it can travel to along the path. (3)

Now that the robot is anchored, it needs to form an area of observation. It does this by forming a circle about its localized point, with a radius of one lookahead distance. The purpose of creating this observation area is to see if and where the circle intersects with the path. At times, the robot may be slightly off the path, and so the concept of a lookahead distance allows the robot to quickly adjust its angle to return to the trajectory when it isn't directly on top of the path. We chose a lookahead distance of 1.0 meters since the path following performed consistently well for lookahead values between 0.5 and 1.5 meters in a combination of straight and curved paths. Outside of this range, we observed that the car often departed from the path or drove in an intense and undesirable oscillatory manner.

Once the intersections along the path are known, the robot is able to find its next goal point along the trajectory. The majority of intersection cases will involve two points. In this case, it is likely that at least one of the intersections will be ahead of the robot, progressing along the path, as desired. Points that fall behind the robot will not be useful for continuing along the path, and so we ignore them. Thus, we filter the intersections for points between $\pm\frac{\pi}{2}$ of the forward angle of the robot. Once we obtain our solution point, we set our robot's steering angle go directly towards it. The process repeats for each timestep on the robot.

2.3 Integration

Author of section: Johlesa Orm

The planned trajectory path was represented as a collection of piecewise points with x and y coordinates. We combine its result with our path following algorithm, along with localization code, so that the robot can follow a set trajectory in real time.

We used the planned trajectory from BFS in our pure pursuit algorithm. Since BFS used pixels in its search for the optimal trajectory, the final path was composed of a large number of piecewise points. A simple corner path could have 512 piecewise points, which would make too many segments and harm the performance of the path follower. To handle the large volume of segments and improve the smoothness of the path following algorithm, we downsized the path to only include points that fell at least 1.0 meters away from each other. This gave us a more accessible piecewise path, scaling down what would've been a 512-segment path to about 24 segments.

An additional component that we implemented in our path following algorithm was dynamic resizing of the lookahead distance. (4) Typically, on sections of a trajectory with high curvature, the robot will do a poor job of following that section of path since its lookahead distance will be too large to capture the intricate bend of the path, and is instead more likely to intersect a part of the path that skips over the curvature. To handle this issue, we check for the curvature of the path starting at the closest segment and looking three segments ahead of it. We use the endpoints of each segment and calculate the circumcenter of the triangle formed by the three points. The distance of the closest segment point to the circumcenter is the radius of curvature, which we then take the reciprocal of to get the curvature itself. Values approaching 0.0 indicate low curvature in the small portion of the path ahead. If the curvature value is above 0.2, we scale the lookahead distance by 0.7 in order to follow the curvature of path more finely. If it is under 0.2, we scale the lookahead distance by 1.5 to try to dampen the frequency and intensity of adjustments to the steering angle.

Overall, these supplements to the path following algorithm were added with the goal of reducing oscillations and making driving much smoother. Downsizing the trajectory had the most positive impact on path following performance. Dynamic resizing of the lookahead distance minimally improved performance by keeping the car on the path more frequently than without resizing. For optimal performance, we still need to find the best curvature threshold and lookahead scalars through further experimentation.
(4)

3 Experimental Evaluation

3.1 Simulation

3.1.1 Path Planning Performance

Author of section:

The main considerations for comparing the performance of the algorithm in simulation were: computation time, which includes initialization time and path computation time in a variety of scenarios, and the optimality of the trajectory, which we measure by the total length of the computed trajectory and the number of turns in the trajectory. The average time for initialization (out of 5 initializations) was 000 seconds for RRT and 000 seconds for BFS. The compute time trajectory distances for various path planning scenarios is summarized in the following graph. Initialization, straight line path, corner

Table 3.1.1 has a caption:

Average path planning computation time.

Scenario	RRT	BFS
Init time	cell5	cell6
Straight line path	cell8	cell9
corner 1	cell5	cell6
corner 2	cell5	cell6
Opposite corners	cell5	cell6
cell1	cell5	cell6

3.2 Real World Testing

Author of section: Johlesa Orm

Unfortunately, we were unable to collect satisfactory data from deployment on the actual robot. What we did observe from the few instances that the algorithm was able to run on the robot was that it oscillated at a much greater magnitude than what was observed in simulation. In addition to further tuning, we plan to investigate other ways to calculate and adjust the final steering angle of the robot once it finds its goal point along the path. We also plan to import additional Python modules, like sympy, in order to speed up computation and improve performance.

4 Conclusion

Section Author: Johlesa Orm

After combining localization, path planning and path following algorithms we were able to achieve satisfactory performance in simulation. What remains is debugging our path following algorithm in order to follow a given trajectory more closely.

5 Lessons Learned

Ben Evans: I was again reminded of the importance of communication between teammates after I acquired some upper respiratory infection in the middle of this lab. Between trying to recover, and trying to avoid sharing the illness, it was quite difficult to stay on top of the state of the team's progress. On the technical side, I thoroughly enjoyed implementing path planning algorithms given relatively few constraints. While I have implemented these algorithms in the past, the context has always been somewhat more structured. It was very fun to figure out optimizations for various elements of the algorithms that I had previously been given as part of specialized python libraries or code provided by an instructor.

Michelle Zbizika: I learned that it's important to have multiple people work on a component so that we can make sure we're implementing something correctly and to be more creative in making the most efficient system.

Emmanuel Anteneh: I found this lab rewarding and frustrating, as it was great to see the final result after working on each component, but was a bit discouraged that we weren't able to get as good of a performance as I was hoping. I think a large part of this was poor time management, and in the future it would be good for our team to allocate time more effectively. Overall though, it was very cool to see our robot follow paths on its own, and I came away from this lab with a deeper appreciation for path planning and pure pursuit.

Johlesa Orm: I enjoyed working on trajectory following for this lab. Implementing the algorithm on the actual robot was quite a struggle and it helped me learn that getting working code onto the robot as soon as possible is helpful for debugging performance issues and finding optimal parameters in a timely manner. I think that had we gotten our simulation code onto the robot a week earlier, and had used the following week in order to debug the real robot issues, I think we would've been in better shape for the lab. The concepts for each of the modules were difficult, and I'm proud of the work that my teammates and I were able to put in this past week.

References

[1] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[2] S. Thrun, W. Burgard, and D. Fox, *Probabalistic Robotics*, 1st ed. Cambridge MA: MIT Press, 2005.

[3] R. C. Coulter *et al.*, *Implementation of the pure pursuit path tracking algorithm*. Carnegie Mellon University, The Robotics Institute, 1992.

[4] RSS_Course_Staff, "Lab 6: Path planning," Available at https://github.com/mit-rss/path$_planning(2024/04/24)$.