16.405 Lab 6 Report

Team 2: Grace, Akila, Hoang, Quang, Noah

Editors: Grace and Akila

April 2022

1 Introduction - Grace

The goal of this lab is to write a robust algorithm that would allow the robot to plan and follow a path through a known environment. In the specific case of lab 6, the robot would plan its path through the Stata basement. This goal was accomplished by creating two different algorithms: one which would plan a path using a given map and one that would follow the created path. These two algorithms were then combined and implemented to the robot.

This lab builds on previous labs by continuing to require that data be transformed so everything is in the same reference frame. It also relies on the localization method that was created in lab 5 so that the robot knows where it is located within its environment. Not only does the path planning method build on previous labs, but it is crucial for the final challenge. The robot will likely have to use these methods to plan a path through the city in the final challenge and to navigate the Johnson track. Since it is a time based challenge, it will be important to find the fastest possible path (such as a path with the fewest stop signs).

This lab report will detail how each algorithm was created and implemented, then evaluate the robot's performance.

2 Technical Approach

This lab was approached with the idea of creating two separate algorithms that would then be combined into one functioning piece of code to implement. The path planning algorithm utilized the A* algorithm to find the optimum path, and the path following algorithm used pure pursuit. This section details the specifics of how each algorithm was implemented.

2.1 Path Planning

2.1.1 Preprocessing Data - Grace

Before the path planning algorithm could be used, all of the data had to be transformed so that the map, start point, and end point were all in the same reference frame. Specifically, the start and goal points are given in coordinates, whereas the map was given in pixels. Therefore, the map has to be converted to coordinates before running the algorithm to plan a path. After the path has been planned, the method will be reversed to convert coordinates back to pixels.

The way the pixels were converted to coordinates was through rigid transformations. In this transformation, only the x and y coordinates were transformed due to the fact that the robot stays on the ground, so the z coordinate is not as important. First, each pixel was labelled with a (u,v) coordinate. From here, they were multiplied with the map resolution in order to scale the pixel to the correctly sized map. This gives the initial pose to put into the transformation equation. In order to get the new transformation, the initial pose is substituted into the following equation:

$$\begin{bmatrix} x \\ y \\ . \\ . \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 & map.x \\ sin\theta & cos\theta & 0 & map.y \\ 0 & 0 & 1 & map.z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} res * u \\ res * v \\ . \\ . \end{bmatrix}. \tag{1}$$

In this equation, map.x, map.y, and map.z are given in the map.origin.position part of the message. Similarly, θ, the map resolution, u and v are also given in the odometry. Plugging these in this gives a new u and v, labelled x and y, in the same reference frames as the start and goal points for the algorithm to use. After the algorithm generates a path, these coordinates are converted back by reversing the above equation.

2.1.2 A* Algorithm - Akila

When choosing our planning algorithm, we considered the tradeoffs between search (A*) and sample (RRT) based algorithms. Specifically, while search algorithms may take longer than sampling algorithms, our planning will run offline, making a slow runtime less of an issue. Additionally, the search based algorithms are more reliable than sampling algorithms to get an optimal solution rather than a quick or unusual inefficient path.

The A* search algorithm was selected for its ability to reach an optimal solution regardless of situation (assuming an admissible and consistent heuristic is used). In this case we used Euclidean distance which is admissible because it is the shortest path between points and therefore can't be an overestimate, and it is a consistent because all of distances are calculated in the same manner.

A* search is a smart search algorithm that uses heuristics to guide its decisions. Specifically, the following distance heuristic is used:

$$f = g + h \tag{2}$$

Here, "g" represents the distance from the start to the current node and "h" represents the distance from the current node to the goal node (employing the heuristic). Specifically "h" is:

$$h = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{3}$$

These combine to define the cost "f" of the path by taking a certain route. The goal is to find the "least cost" path (i.e. optimal).

Initially, this implementation treated each pixel as a node. However, to speed up the path planning process several pixels were clustered together as a node. This reduced the map resolution making the search space smaller and therefore quicker to explore. Additionally, because it uses Euclidean distance as the heuristic, it means that it cuts corners when navigating obstacles as seen in 1 below:

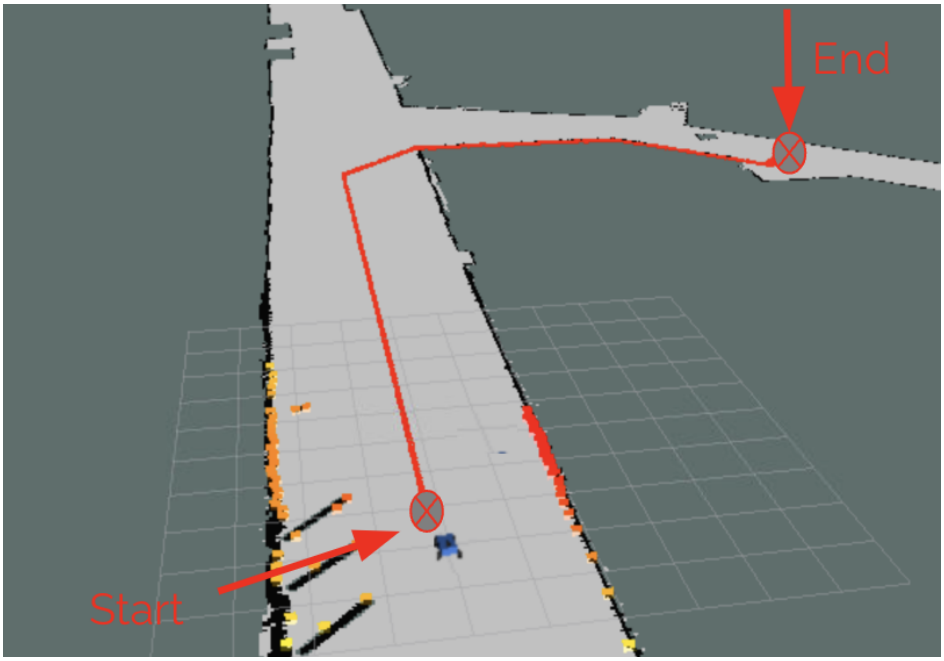

Figure 1: A* Algorithm using Euclidean distance causes the robot to hug the edges of the obstacles

When deployed into the real world, this proximity to the wall would be a big safety hazard so instead we included a wall dilation to give the illusion of a larger obstacle. Then, the planner would account

for this buffer when optimizing the path and when the buffers are removed during deployment in the real world, the robot would be remain a safe distance away from any obstacle. 2 below is a simulated run with the inclusion of the buffer, producing a more realistic and safe path:

Figure 2: A* Algorithm using Euclidean distance with wall dilation to add a buffer around obstacles

2.2 Pure Pursuit - Hoang

After determining the fastest path through a given map, the robot needs to follow that path. This was done by using pure pursuit. The main goal of pure pursuit is to find steering angle of the car such that the car can follow the given path. There are 4 steps to find this steering angle that will be detailed in this section:

- Calculate perpendicular distance between segments of path and the car.
- Find the lookahead point.
- Calculate the steering angle of car.
- Publish commands to the car.

In order to begin pure pursuit, the perpendicular distance between segments of path and the car need to be calculated. This is done by using the location of the car from the localization algorithm written in Lab 5 and the given segments of the path which are provided by the path planning algorithm. These calculations are used to find the closest path segment for the car to navigate to. Figure 3 below visualizes this process. Distances d1 and d2 provide the perpendicular distance between the car and the segments. From these calculations, the car determines that segment 1 is the closest segment, setting a goal point to navigate toward.

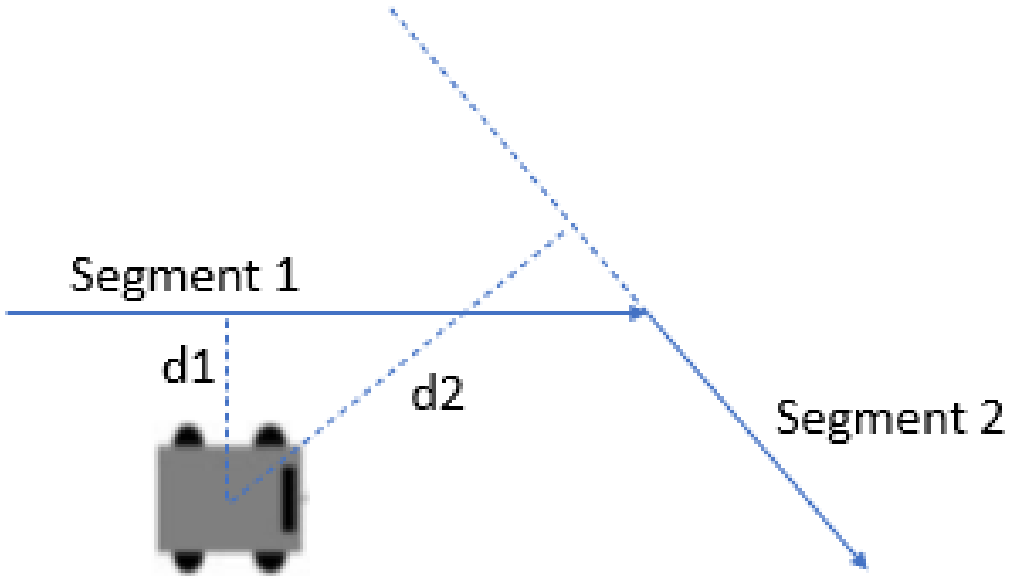

Figure 3: Finding the Closest Segment

After determining the closest segment to the car, pure pursuit finds the lookahead point, which is the intersection point between the closest segment and the circle that has center as the car position and radius as r = 1 meter. One meter was selected as a default starting value to account for the car speed. A visualization of this process can be seen in figure 4. It should also be noted that there are 2 intersection points in this example. The pure pursuit algorithm handles this by choosing the point

that is farthest along in the path. If there are no intersection point, the radius r is gradually increase until there are intersection points.

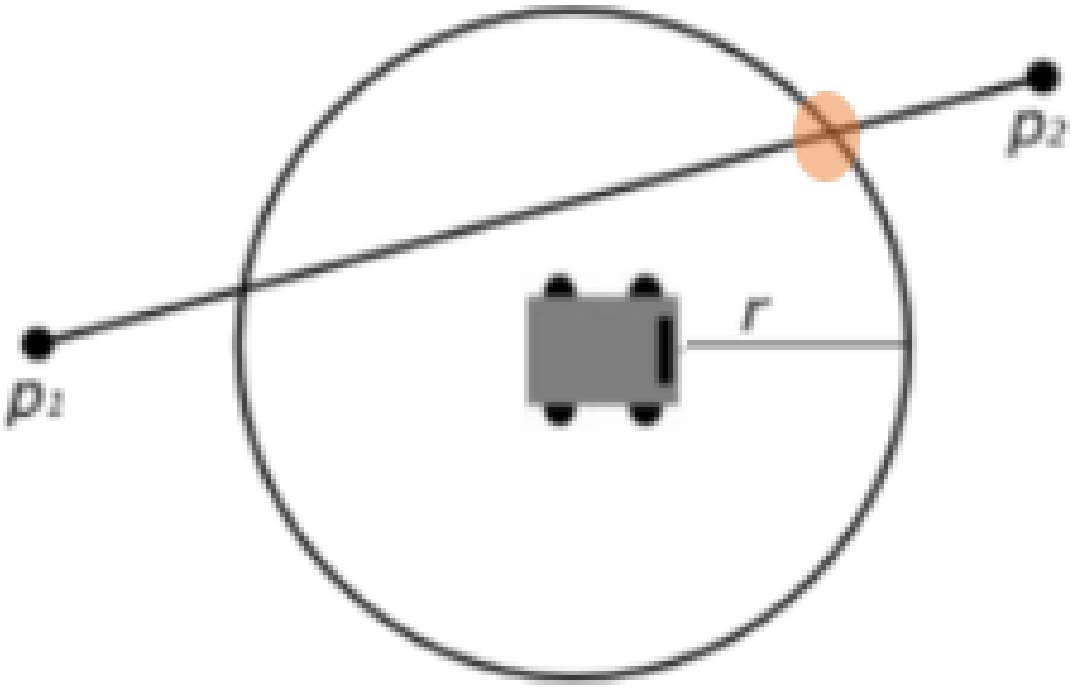

Figure 4: Lookahead point ddetermination

This lookahead point effectively functions as a temporary goal point for the car. The steering angle needs to be adjusted so that the car goes toward the lookahead point. This is done by using trigonometry and geometry. Using a bicycle model to model the car, the steering angle can be found with

$$\delta = tan^{-1}(\frac{2Lsin\alpha}{l_d}), \tag{4}$$

where L is the distance between the front and rear wheels, α the angle between the back wheels and the lookahead point, and l_d the distance between the back wheels and the lookahead point. A visualization of this geometry is provided in figure 5

Once the steering angle is calculated, it is published to the drive command that is sent to the car. If the steering angle is small ($\leq$ 0.01 rad), the car drives at 4 m/s. If the steering angle is big, which means the car is turning, it drives at 1 m/s. This speed difference is so that it gives the car some time make the turn and precent collisions with the wall.

2.3 Integration and Implementation - Akila and Grace

The code was integrated by having the path planner publish the path a channel that the pure pursuit code is subscribed to. This channel connects the two codes and allows the robot to work. The code was then uploaded to the robot. It worked in conjunction with the RViz simulator to set goal points for where the robot should travel to.

3 Evaluation

3.1 Simulation Implementation - Noah

After implementing the algorithms, the robot worked very well in simulation. The robot appeared to follow the planned path very well by observation. In addition to just a qualitative analysis, the

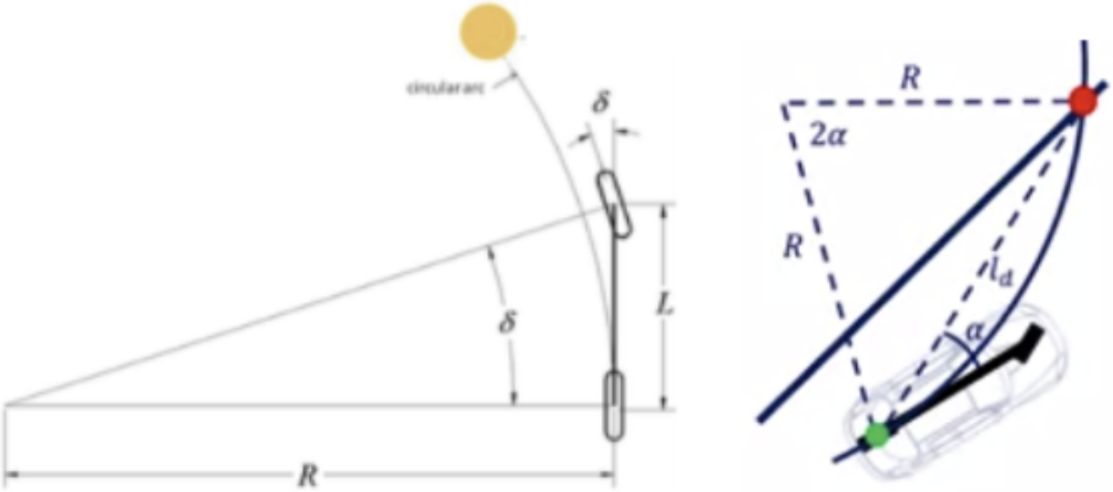

Figure 5: Calculating Steering Angle

simulated robot's performance was also evaluated quantitatively.

The robot was evaluated quantitatively using two metrics: distance error and angle error. First, a new data type called "Error.msg" was created. Creating a new data type gave the team more flexibility in deciding which metrics would be examined. It was decided that analyzing the distance error and angle error made the most sense. Distance error allowed us to measure how far the robot was off its planned course. The difference between the robot's angle and the angle of the line it is pursuing created a way to look at how sharp of a turn the robot is trying to make to get back on course. This is important because, unlike simulation, the physical robot cannot make turns on the dot at whatever angle we specify.

A simulated run was then used to generate data. Figure 6 (below) shows a plot of the distance and angle error over time. A video of this data being generated alongside the robot running in simulation can be seen here. In the plot below, there are periodic large spikes in the distance error, each measured in meters. This happens as a result of the robot turning around corners. When the robot gets to a corner, the line segment it chooses to follow is around the corner rather than directly in front of the robot, causing the spike in distance error. Similarly, there is also a spike in angle error, where the graph takes on errors of around 0.3 radians- or 17°. Nonetheless, because of the car's foresight, it was able to take a smoother trajectory and have a minimal deviation from its planned course. This can be seen in the simulation video linked earlier in the paragraph.
Because the simulation driving error was relatively low, the risk of damaging the physical robot was also low. The large spikes in error for the turns could be remedied by changing how the distance error is evaluated. Thus, we were able to move onto the final part of lab 6: implementing our algorithms on the physical robot.

3.2 Robot Implementation - Grace and Akila

After the code worked in the simulation, the code was transferred to the robot. It worked well by observation, navigating the hallways of Stata well. The start point was wherever the robot was placed, and the goal point was manually set with the "2D Nav Goal" function in RViz. Then the robot was started and left to navigate the hallways on its own, as observed in this link. Similar to lab 5, it was difficult to evaluate the robot numerically. This is due to the fact that, unlike the simulation, it is impossible to identify the "truly correct orientation" of the robot with the tools we are given in this lab. For example, the robot's "absolute" position cannot be identified with a lot of painstaking measurements for every small movement the robot makes. Therefore, the robot's ability to plan a path and then successfully drive to the goal point without any collisions is sufficient.

Some issues that were encountered were that the wifi in the Stata basement is not great, so the robot would sometimes cut out and stall a bit. Initially, the information was also published at too slow of a speed (6 Hz), so the robot would stop every few seconds. After the speed of information being

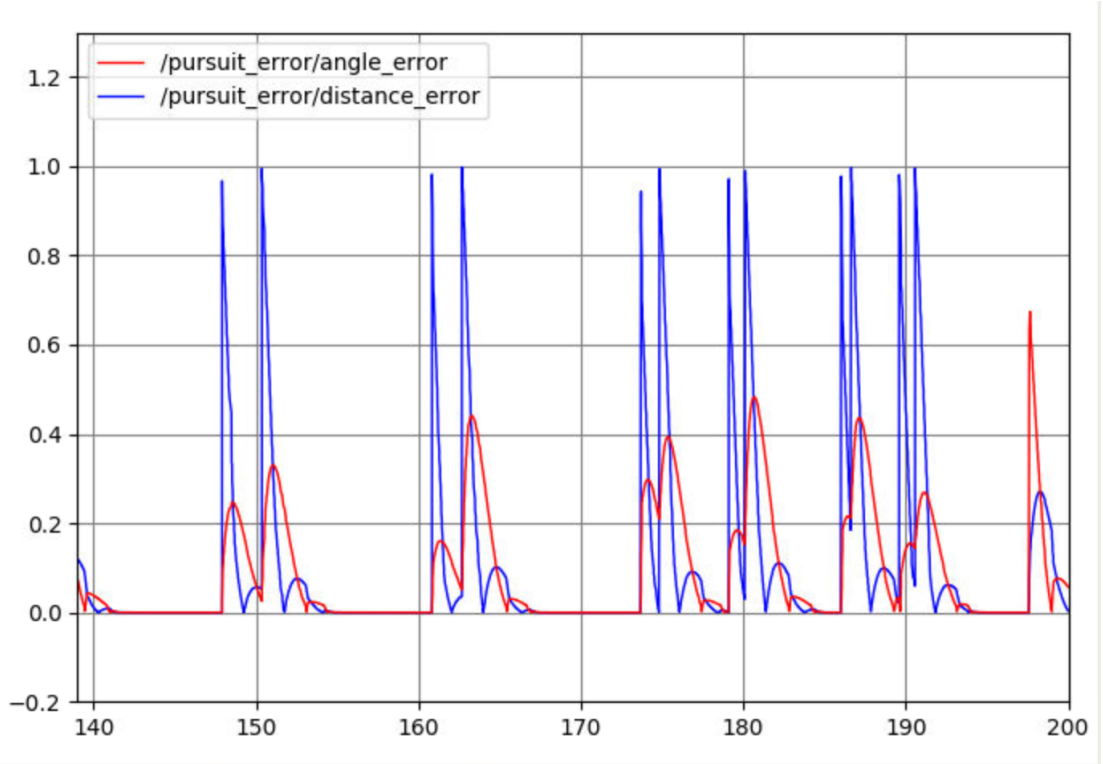

Figure 6: Plot of Distance (m) and Angle Error (rad) over Time (s)

broadcast was increased to 20 Hz, the issue was resolved.

4 Conclusion - Akila

The goals of this lab were to implement a robost path planning algorithm so that our robot could navigate a known environment (in this case the Stata basement). We began by implementing a path planning method using A* search and a path following method using pure pursuit. Both methods were tested in simulation until they worked to our satisfaction. The code was then integrated onto the robot. After being fine-tuned, the robot's performance was evaluated both qualitatively and quantitatively. These algorithmic components will be essential to build up to the final challenge as we navigate through Johnson Track and through the Mini-City.

All of these tests yielded positive results. Future steps would include implementing the RRT algorithm for path planning and comparing its performance to the A* search based method. This would help us examine the tradeoff between speed and discovery of optimal paths. This analysis will be useful as we decide what algorithms to use for each tasks in the final challenge. It is also worth experimenting with how fast the robot can drive without collisions since the speed will be an important factor in the final challenge.

5 Lessons Learned

5.1 Akila

This lab had several difficult components and I realized the importance of dividing work between group members. This also meant we required a lot of communication to complete tasks if people ran into issues. There were several instances where people ran into technical issues and we would all get together to troubleshoot the problem.

On the technical side, I learned how to implement algorithms I was already familiar with (A* and RRT) to real-life problems that would get deployed in physical environments. It was interesting to see how difficult implementation was despite having prior knowledge on the topic. It made me realize that theory is a lot different from practice.

5.2 Grace

In this lab, I learned a lot more about different algorithms that had previously been seen in class and how they become implemented in reality. It was very cool to see how previously done psets could be applied to more than just a fictitious map. It was also just very cool to see the robot navigate the Stata basement all by itself without having to worry (too much) about it hitting a wall.

From a teamwork perspective, I liked how we paired up on the technical challenge so that there was always another person to work with in case one got stuck. It also made debugging a lot easier since someone else could also help review the code for mistakes. Communication was also extremely important since many team members also had exams so it gave us a more realistic idea of when to expect work to be done.

5.3 Hoang

In the technical portion of the lab, I learned about the ideas and implementation of pure pursuit. After readings the slides about pure pursuit, I thought it was fairly simple. However, when I actually implemented, I had to think of smalls details such as what the sign of the steering angle is and how to get the look ahead point given the edge cases.

For the communication aspects, I learned that it is very important to communicate with my team. This week was busy for everyone and we all have midterms and assignments. We kept track of our progress and update the plan every day to accommodate everyone.

5.4 Noah

In this lab, I learned how to more effectively quantitatively evaluate the performance of our robot in simulation. Specifically, I got much better at creating custom message types and using rqt_plot. Though I had done both of these things once before, in this lab I used these more extensively and got much more comfortable working with both custom messages and rqt_plot.

Something else I was reminded of was the challenges in integrating everyone's code. Each week, I get more and more hopeful that this will be a seamless process. Unfortunately, though, it seems that each week we discover new integration challenges. This week, we had to deal with the recurring challenge of not entirely understanding each other's code and a new problem of not publishing to our driving topic frequently enough. Surprisingly, rather than learning something new technically, both of these issues taught me how to be a better communicator and collaborator with my teammates.