Lab 6: Autonomous Navigation - Path Planning and Following

Team # 9

Artemis Pados
Selinna Lin
Min Khant Zaw
Reng Zheng

6.4200 RSS

April 24, 2025

1 Introduction - Selinna Lin/Reng Zheng

In previous labs, our racecar moved in space using the help of physical cues. For Lab 3: Wall Follower, we used LIDAR to measure the distance to the wall and, with the help of PID control, moved in space by tracking the wall. In Lab 4: Visual Servoing, we instead used computer vision to determine the relative position of an orange line in space. However, both methodologies implemented by these labs are problematic in scenarios where there are no terrain features one can follow for navigation.

We therefore build upon Lab 5: Localization to implement a path planning algorithm. Localization of the racecar allows us to accurately locate ourselves in space without using explicit physical markers to guide navigation, enabling a wider array of paths. We use this, along with a prescanned map of our driving range, the Stata basement, to run a pathfinding algorithm that provides the racecar with a set of waypoints in space to follow, forming a path. Once we have this set of waypoints, we can combine it with a the *Pure Pursuit* path-following algorithm to move along the computed trajectory, utilizing localization to determine the deviation from the car's path.

As the goal of path planning is to complete the route as quickly as possible, our success metric is the speed at which the racecar can travel between two points. Choosing between deterministic and probabilistic path-finding algorithms, we settled on the deterministic A* algorithm, which, with the right heuristics, is

guaranteed to produce the shortest viable path between two points in space.

Additionally, our *Pure Pursuit* had free variables that need to be tuned to ensure the racecar optimally followed the trajectory. Therefore, we also measured the cross-track error, or deviation from the preplanned path, of the racecar in runs, in order to ensure that it was faithfully following the optimal path instead of performing erratic movements between waypoints and quantify any errors in path-following.

2 Technical Approach

2.1 Localization - Selinna Lin

We implemented Monte Carlo Localization (MCL) to estimate the racecar's 2D pose in our prescanned Stata basement map. MCL maintains particles representing hypotheses of the car's poses, and alternates between predicting motion (motion model) and evaluating sensor readings (sensor model).

- **Map**: We are given a 2D occupancy map of the Stata basement.
- **Initialization of Particles**: We initialize particles around the known start pose with a Gaussian spread.
- **Motion Model**: We ingest wheel odometry $\Delta x, \Delta y, \Delta\theta$ into the motion model, adding tuned zero-mean Gaussian noise such that it's enough to represent real-world noise and uncertainty, but not too much that the particles diverge.
- **Sensor Model**: The sensor model simulates LiDAR beams for each particle via raycasting and computes the likelihood $p(z \mid x, m)$ as a weighted mix of hit, short, max, and random components $(\alpha_{\text{hit}}, \alpha_{\text{short}}, \alpha_{\text{max}}, \alpha_{\text{rand}})$, using a precomputed range–measurement lookup table. These weights/probabilities are passed into the particle filter for resampling particles.
- **Particle Filter**: The particle filter assigns each particle a weight proportional to its sensor likelihood, normalizes weights, and resamples with replacement, shifting the mean average pose of these particles toward high-probability regions.
- **Publishing Pose**: The mean average pose of the resampled particles is published as the estimated transform between the /map frame and frame for the car's expected base link, which is /base_link_pf for the simulator and /base_link for the real racecar.

Both the motion model and sensor model update the particles in separate callbacks (odometry and LiDAR, respectively). In our tests without synchronization, simultaneous access to the particles caused erratic localization - most likely

from corrupted particle weights. To prevent these race conditions from happening, we used a single threading lock for both updates, which helps to ensure that each update of the motion model and reweighing of the sensor model will complete atomically before the other one takes over.

2.2 Trajectory Planner (RRT) - Artemis Pados

In our autonomous navigation system of Lab 6, we implemented the Rapidly-exploring Random Tree (RRT) algorithm that enables our car to find a collision-free path (if it exists) from a start pose to a goal pose within an occupancy grid map by using sampling-based strategies. Including coordinate transformation utilities to convert between world and map frames using a known transform matrix derived from the map's metadata, our implementation of RRT begins by creating a tree, initialized at the start pose. In each iteration, a random point is sampled from the free space. The algorithm finds the nearest node in the tree to the sampled free point, then steers a fixed step size (0.5 meters in our case) toward it to create a new node. If the new node is free, it is added to the tree and linked to its parent. This process repeats until the tree grows close enough to the goal pose (tolerance of 0.3 meters in our implementation), at which point the goal is connected and the tree is backtracked to construct the full path.

The paths generated by our RRT implementation tend to be jagged and non-smooth, as seen in Fig. 1, due to the randomized nature of the node expansion process. Since each new node is added by steering a fixed distance from the nearest existing node toward a randomly sampled point, the resulting path is composed of discrete straight-line segments that change direction abruptly at each node. Additionally, because the tree is built incrementally based on random samples rather than a global cost function, the final path may include unnecessary detours or inefficient turns even if a more direct route exists. Subsequent processing, such as trajectory smoothing or optimization, could be applied to make RRT more amenable to real-world driving.

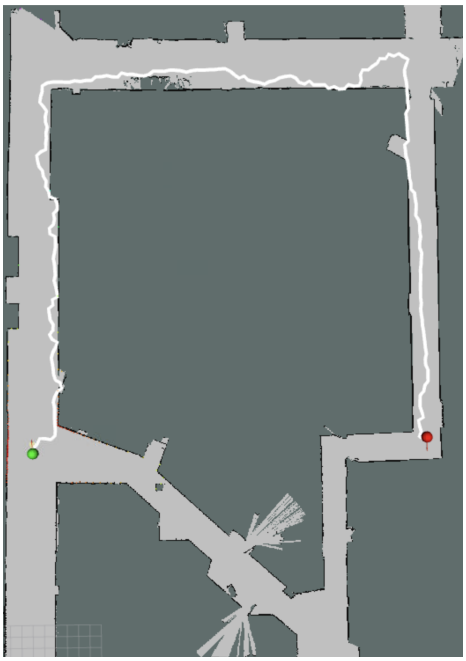

Fig. 1: The figure illustrates a jagged and suboptimal path (as expected) generated by the RRT algorithm on a mapped environment starting from the green start pose and ending at the red goal pose.

2.3 Trajectory Planner (A*) - Min Khant Zaw

In addition to implementing RRT, we also implemented a search-based path planning algorithm, A*. We used the same transform matrix from RRT to convert the coordinates between world and map frames. A* algorithm iteratively checks the neighboring nodes of the current node, which is the start node for the first iteration. For each node x, it calculates:

- $c(x)$: the cost from the start node to node x
- $h(x)$: the estimated cost from each node x to the goal node, known as the heuristic

A* guarantees an optimal path if the heuristic never overestimates the actual cost to the goal. We choose Euclidean distance as our heuristic because it never overestimates the true cost in a grid with uniform movement cost, thus satisfying the admissibility condition required for A* to guarantee the optimal path. Euclidean distance is calculated using the Pythagorean Theorem:

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

The total estimated cost for each node x is then given by:

$$f(x) = c(x) + h(x)$$

The algorithm chooses the node with the minimum $f(x)$ to explore next neighbors. This process continues until the goal node is reached. Finally, A* reconstructs the optimal path by tracing back from the goal to the start using the recorded information of where each node comes from.

Unlike the paths generated by RRT, the paths generated by A* tend to be straightforward and smooth, as seen in Fig. 2, due to the use of a deterministic, cost-based expansion method that prioritizes the nodes with the lowest estimated total cost to the goal. However, we can see that our A* algorithm at this stage generates paths that are too close to the wall, which will cause our car to crash.

Fig. 2: The figure illustrates a path generated by the A* algorithm on a mapped environment starting from the green start pose and ending at the red goal pose.

To avoid collision with walls and obstacles, we added buffer zones around them by expanding the occupied region in the grid. While this solves the problem of generating paths too close to walls and obstacles, A* generates a path that goes all the way around the map when the goal lies in a narrow corridor, as seen in Fig. 3. This behavior is due to the fact that the buffer zones may completely block the narrow free space, causing A* to generate a path around the map or

fail to find a path at all.

Fig. 3: The figure illustrates a path generated by the A* algorithm on a mapped environment starting from the green start pose and ending at the red goal pose. Since the buffer zones block the narrow corridor, A* generates a path around the map.

Because of this issue, we added penalty to the nodes depending on how close they are to the obstacles instead of adding buffer zones around obstacles. We set the safety threshold distance to 0.7 meters and penalized the nodes that are closer than 0.7 meters to the nearest obstacle. With this solution, A* generates a path that goes through the narrow corridor and is far enough from the obstacles for the car to go through without collision, as seen in Fig. 4.

Fig. 4: The figure illustrates a path generated by the A* algorithm on a mapped environment starting from the green start pose and ending at the red goal pose. A* now generates a path far from the obstacle that can go through the narrow corridor.

2.4 Trajectory Follower (Pure Pursuit) - Reng Zheng

The Pure Pursuit algorithm is a proven trajectory follower and is common in robotics tasks involving a pre-computed trajectory consisting of waypoints [1]. Pure Pursuit takes a racecar's position r in space, then looks through the waypoints of the precomputed path until it finds the closest waypoint to the racecar. The algorithm then iterates through the waypoints after the closest waypoint until it finds a waypoint w_t further away than the look-ahead distance l. Then it interpolates between w_t and the previous waypoint w_{t-1} to get the goal point g along the trajectory, which is exactly l away from the racecar.
As Pure Pursuit aims to follow the waypoints as smoothly as possible, which in technical terms means without *jerk*, or changes in acceleration, the algorithm calculates the smallest curve/arc of a circle necessary to get from its current position to g, and then follows that arc. Then, it sets its steering angle to follow this arc, leading to a zero-jerk turn manuever towards the point after it changes its heading.
The arc of the circle is derived in the following manner:

$$\gamma = \frac{r_k - g_k}{l^2} \tag{1}$$

where γ is the curvature of the arc, the sign signaling which direction to turn

into. The derivation follows from the geometry of the problem, further explained in [1]. The steering angle is then calculated by:

$$\delta\theta = \tan^{-1}(\gamma b) \tag{2}$$

where b is the wheelbase length of the racecar, or distance between the two axles of the racecar, as derived by the bicycle model [2]. Now, assuming the speed is sufficiently low so as not to lose traction, we can end up at g just by following the steering angle, minimizing the jerk of the manuever. By iteratively doing the above algorithm until the last waypoint, we can smoothly follow our waypoints until the end of the path using Pure Pursuit.

3 Experimental Evaluation

3.1 Trajectory Planners (RRT and A*) - Artemis Pados/Min Khant Zaw

A* and RRT differ fundamentally in how they approach path planning. A* is a search-based algorithm that operates on a discretized grid and explores the space deterministically. It is both resolution complete (guaranteed to find a solution path if one exists in the discretized grid) and optimal (guaranteed to find the shortest path with an admissible heuristic). However, since A* is resolution complete, it cannot find paths through narrow spaces if the resolution is coarse. Its performance can also degrade in higher-dimensional spaces as it requires exhaustive exploration of the grid.

In contrast, RRT is a sampling-based algorithm that operates in continuous space. It is probabilistically complete, meaning that it will find a solution if one exists given sufficient iterations, but it does not guarantee shortest path optimality. RRT is highly scalable to high-dimensional configuration spaces, making it suitable for robotic systems more complex than our current setup. These theoretical differences shape the respective strengths and limitations of RRT and A* in practical applications.

Fig. 1 and Fig. 2 together qualitatively illustrate how the path planners differ. While A* produces a more direct and smooth path through the narrow corridor, RRT generates a jagged and longer path that loops around the perimeter of the map. This discrepancy arises from the random nature of RRT and its tendency to explore large areas of the free space before reaching the goal.

Table 1 quantitatively compares A* and RRT across three different trials in terms of path length (measured in number of pixels) and planning time (in seconds). We used the A* algorithm without penalty to nodes for this evaluation since we wanted to compare the barebone algorithms of A* and RRT. For A*, the path length is directly computed by counting the number of grid cells along the planned path. For RRT, we approximate the number of pixels by using

the number of waypoints in the path and scaling by a factor of approximately 9.92 pixels per segment, which corresponds to the 0.5-meter step size used in steering divided by the map resolution (0.0504 m/pixel). Overall, the results show that A* consistently finds shorter paths than RRT in all three trials and is also faster in 2 out of 3 cases. We note that the case in which RRT finds a path faster than A* comes from the trial where the optimal path is quite short. This makes sense because random sampling can prove more efficient than explicit search when the start and goal are near each other. Overall, this supports the theoretical expectation that A* returns optimal paths and does so efficiently in low-dimensional, structured environments like ours.

Trial	**Path Length (Number of Pixels)**		**Time (s)**	
	A*	**RRT**	**A***	**RRT**
1	570	655	0.414	0.055
2	763	863	0.736	0.999
3	1236	2212	1.081	1.698

Table 1: Comparison of A* and RRT in terms of path length and planning time across three trials. Highlighted cells indicate better performance.

Based on our findings and the theoretical differences between A* and RRT, we concluded that A*is better suited for planning trajectories for our car. This aligns with our expectations since A* and RRT performs better depending on the environment. A* is more suitable for structured indoor maps with known occupancy grids and few dynamic constraints because it uses a deterministic, cost-based exploration strategy, which matches the condition for our car. In contrast, RRT is better for more complex and dynamic environments since it handles vehicle dynamics and constraints more naturally through its random sampling method.

3.2 Trajectory Follower (Pure Pursuit) - Reng Zheng

First, we evaluate the smoothness of our Pure Pursuit algorithm by measuring the steering angle commanded in the output against a baseline. To do so, we used Wall Following as a baseline and had our racecar follow a wall using either Wall Following from Lab 3 or waypoints along the wall. This yielded Fig. 5 which shows that Pure Pursuit, once waypoints are published, has very smooth steering commands with little variance compared to Wall Following, with large jerks (cyan blue line) only occurring when approaching the final waypoint, where rapid corrections are needed to perfectly align with the final point.

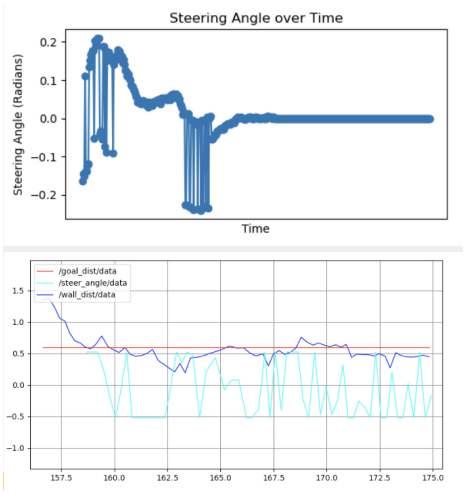

Fig. 5: The top graph illustrates the plot in the steering angle jerk using Pure Pursuit. The bottom graph illustrates the plot in the steering angle jerk using Wall-Following PID versus Pure Pursuit.

We then evaluated the ability of our pure pursuit to follow waypoints accurately. We quantify this with cross-track error, or the error between the trajectory set by the waypoints and our car's position. Because Pure Pursuit is only as good as its localization, we verify cross-track error using the racecar's *perceived* location, rather than actual location, versus the waypoints, and then verified it executed the path by making sure that, in reality, the racecar was at the position corresponding to its last waypoint. This allows us to separate the localization evaluation done in Lab 5 from Pure Pursuit evaluation while also making sure the path is still executed in some fashion. As seen in Fig. 6, conditional on a sufficiently accurate localization, our Pure Pursuit algorithm has minimal cross-track error with a lookahead distance of 1.2 meters going at 1 meter per second, showing the efficiacy of our waypoint following.

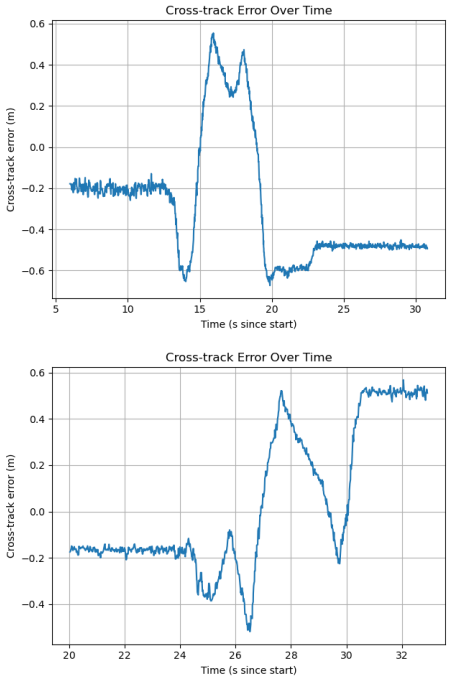

Fig. 6: Two runs depicting cross track error of the run depicted in Fig. 5. We can see our follower has less noise in its tracking than the wall-follower, but has greater transient error when a turn occurs on account of smoothness.

4 Conclusion - Selinna Lin

This lab marked a significant milestone in building a fully autonomous racecar by integrating localization, motion planning, and control together. We used MCL for pose estimation, A* for trajectory planning, and pure pursuit for trajectory following. The resulting implementation enabled our car to navigate from the starting position to a goal position within a known map of the Stata Basement, while avoiding obstacles along the way.

By comparing RRT and A*, we explored the tradeoffs between searched-based (A*) and sample-based motion planning methods. While A* provided a faster and more optimal paths in our structured environment, RRT is more flexible and scalable in higher-dimensional settings. Meanwhile, the pure pursuit controller was effective at following the path smoothly in real-time, although its performance heavily relied on the quality of both the localization and appropriate tuning of the lookahead distance based on the car's speed.

In the process of tackling issues like particle filter synchronization, jagged path

generation, and various minor bugs, we gained valuable hands-on experience working with both the strengths and limitations of our system. Looking ahead, we're excited to continue refining our approach and improving performance in preparation for the Final Challenge!

References

[1] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-92-01, January 1992.

[2] J. M. Snider, "Automatic steering methods for autonomous automobile path tracking," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-09-08, February 2009.

5 Lessons Learned

5.1 Artemis Pados

This lab provided me valuable technical experience in implementing sampling-based motion planning, particularly the RRT algorithm which is what I focused on. I gained a deeper understanding of how map representations, coordinate transformations, and step-size parameters influence the feasibility and quality of the resulting path. Debugging issues such as map origin alignment and grid indexing took up a lot of my work-time over the past week. This really reinforced to me the importance of precise spatial reasoning in robotics. From a teamwork perspective, this lab emphasized the value of early coordination and dividing tasks effectively. I believe we did a good job in breaking up the technical work of the lab into discrete and independent modules to be able to parallelize work. As the course now transitions away from structured deadlines, I hope that we can maintain the same diligence with task allocation and soft team deadlines.

5.2 Selinna Lin

This lab was quite intensive in terms the amount of technical components we had to complete within a tight timeline, but we made it! I think that struggling through the problems and learning how to tackle issues as they arose was a very valuable experience. Personally, this lab felt like the ultimate integration bomb, since we had to bring everything together from all previous labs into one working system. Often times, issues had to be isolated for effective debugging, which is why unit testing was incredibly important. Testing in simulation was so helpful in identifying and resolving issues before moving to the real car. Not just so, when problems come up during integration, we often have to look back into simulation to pinpoint which component went wrong. Was it the localization? Path planning? Pure pursuit? There's just so many things that could've gone

wrong and I'm glad everything came together in the end. I'd definitely attribute our success to having a cohesive, motivated team where everyone gave it their best!

5.3 Min Khant Zaw

For this lab, I had the opportunity to develop a search-based path planning algorithm, A*, and I learned more deeply about A* algorithm by actually implementing it. It took a while to figure out that the map given to us is rotated by 180°. While working on the lab, we also found some inconsistencies between the occupancy grid representation in the given map data and the description provided in available documentation. We finally decided to ask the TAs about it after spending hours on the issue, and the TA told us that the occupancy grid representation in the given data is indeed different from the documentation. From this experince, I learned that it is important to ask the TAs for help when we are facing issues instead of spending so much time to fix it on our own. As for communication, I think our teamwork and trust have been strengthened over time, and I am looking forward to working on the final challenge with them together.

5.4 Reng Zheng

The main difficulty of the lab was optimizing for speed. For me, things did not feel especially difficult to integrate, perhaps because I have a bit of prior experience with other CI-M courses. However, difficulty increased in the extra credit assignment, where we had to get our robot going as fast as possible in a difficult route. This posed unique challenges in optimizing our code to handle the requirement, as faster movement means more room for error to occur between successive computations. While the performance in the end was shakey, I think the lesson was that it was important to view that extra credit as truly "extra," and have a minimum viable goal in sight first (completion of the lab) as it set expectations without disappointment.