Lab 6 Report:
Following A* and RRT-Generated Paths using Pure Pursuit for Autonomous Vehicle Indoor Racing

Team 10

Cynthia Cao
Trey Gurga
Grace Jiang
Toya Takahashi
Jonathan Zhang

RSS

April 27, 2024

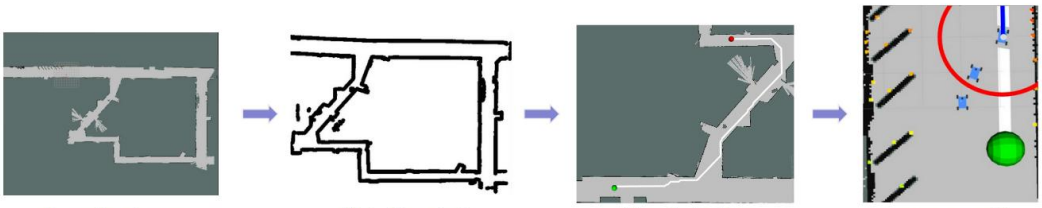
Contents

1 Introduction

(Authors: Jonathan Zhang)

A key task for mobile autonomous robots is efficient and safe path planning. From robotic arms that need to perform complex medical tasks to search and rescue drones that need to quickly and safely navigate dangerous environments, the ability to generate and follow paths is crucial to controlling and executing autonomous motions. This holds true for our robot as well, which needs to be able to generate paths and follow trajectories on its own in order to succeed at City Driving in the final challenge. In this lab, our group implemented a path planning algorithm for our robot in the stata basement. Given a known static environment map, a start pose, and an end pose, our robot can successfully generate an optimal trajectory and follow it without crashing.

Our implementation of path planning can be split into two main modules, a path generating algorithm and a path following algorithm. Our path generating algorithm uses graph algorithms on a grid representation of the map to generate optimal paths within the map that avoids obstacles. Both a searched-based algorithm, A*, and sampling-based algorithm, RRT, were considered, but experimental trial runs with both showed that A* was better suited for our metrics of fast run time and shortest distance. Our path following algorithm takes the output of our path generating algorithm, and allows our robot to follow along any planned path. It utilizes localization from Lab 5 to understand where it is relative to the map, and implements pure pursuit to follow the path once it has that knowledge. Combining the two, our robot can successfully move to planned locations on a map without crashing.



In the rest of this paper, we go more in depth about our implementations of our path generating algorithm and path following algorithms, before analyzing experimental results, both in simulation and the real world, and concluding with a summary and reflection of our work. In the technical approach, we discuss the main technical problems of our implementation, explain our A* and RRT path planning algorithms and how we chose between them, and go into our tuning of the pure pursuit model for path following. In experimental analysis, we cover the quantitative and qualitative metrics we used to measure the performance of our algorithms for both simulation and real life. Finally in our conclusion, we recap the main problem, purpose, and our work, and list future improvements that we can make.

2 Technical Approach

(Authors: Cynthia Cao, Toya Takahashi, Grace Jiang, Trey Gurga, Jonathan Zhang)

The main technical problem we aimed to solve was to obtain a level of abstraction between a simple inputted command of desired location and the output controls necessary for the robot to get there. We break down this complex problem by splitting it into two main modules, a path generating module and a path following module. The path generating model will take the input and map and return

a path that successfully moves through the map while avoiding obstacles. The path following model will take this returned output of the path generating model as its input, and then output the drive commands necessary to follow it. The two together will be able to successfully allow the robot to get to its desired location.

2.1 Path Planning

For effective path planning, we need algorithms that can generate safe and short paths to the goal as quickly as possible. Two path planning algorithms were implemented for this lab: A* and RRT. A* searches through a discretized representation of the environment in order to find an optimal path. On the other hand, RRT randomly samples the space, generating and pruning a tree of potential configurations until the goal space is reached. Our representation of the environment as a search space, as well as the implementations and evaluations of these two algorithms, are expanded on below.

2.1.1 Environment Representation

(Author: Jonathan Zhang)

Before we can implement our algorithms, we must first come up with a environment representation for us to run those algorithms on, so that the path generated would correspond to a path in the real world. In order to do this, we use a unit grid system, where every unit/pixel in the map corresponds to a point in the map, which we achieve through transformation matrices. The exact ratio for this is a 1x1 unit cell in our grid converts to a 5x5cm square in the real world. With this grid, we can convert paths our algorithm generates to paths in the real world.

In addition to our transformation matrices, we also erode and dilate obstacles on the map. This is necessary to account for the fact that our robot itself has a non-negible width, and cannot drive on a unit cell directly next to obstacles. In order to account for this, we need to make the walls and obstacles on the map that we feed into our path generating algorithm bigger so that we can't generate paths that would transform to points 5cm away from obstacles. In order to achieve this, we first erode with a 3x3 matrix to reduce some noise, and dilate with a 17x17 matrix to account for the width of the car. After this, our path generating algorithms will be able to generate paths our car can follow without it intersecting with the wall.

The whole conversion process can be seen in Fig. 1. With this new map representation, we can implement our algorithms A* and RRT.

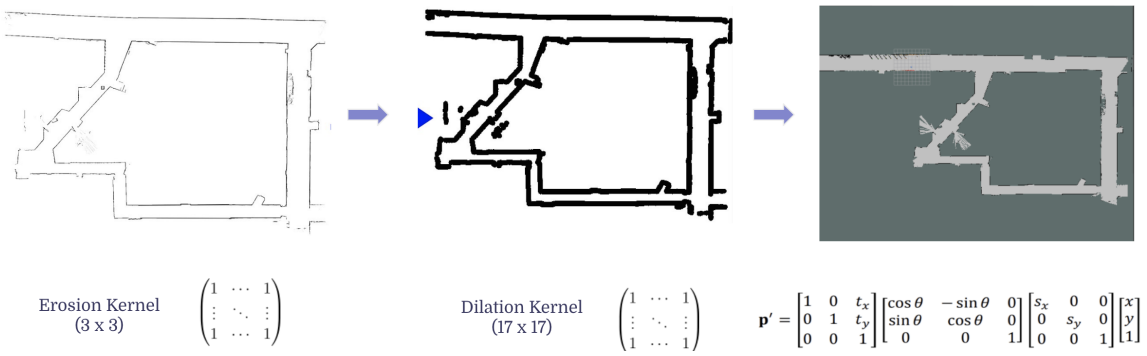


Fig. 1: **The grid map is both eroded and dilated before being fed into our algorithm, and the returned output goes through a translation and rotation matrices to convert to the real world.** The exact erosion kernel used is a 3x3 matrix and the dilation kernel is a 17x17 matrix to account for the width of the robot.

2.1.2 Search-based Algorithm: A*

(Author: Cynthia Cao)

Search-based algorithms systematically search through a discretized representation of the environment to find a path from the start to the goal. We chose to implement on such algorithm, A*, as it can find paths much faster than similar algorithms such as Breadth-First Search and Depth-First Search. Additionally, A* is guaranteed to produce an optimal path given an admissible heuristic, a heuristic that always underestimates the true cost to the goal. This is a valuable characteristic to any path planning algorithm we choose; because we are using these algorithms to race our cars, there is strong motivation to find the shortest path to the goal as possible. We chose the Euclidean distance to the goal from a given point as our A* heuristic, because it will never exceed the true distance to the goal. Further implementation details can be seen in Alg. 1.

2.1.3 Sampling-based Algorithm: RRT

(Author: Toya Takahashi)

Unlike search-based algorithms, sampling-based algorithms explores the search space through random samples rather than discretization of the map, typically leading to quicker runtime. Two popular sampling-based motion planning algorithms are Probabilistic Roadmap Method (PRM), which constructs a randomly sampled graph in a continuous search space to run a shortest-path algorithm, and rapidly exploring random tree (RRT), which grows a tree rooted at the starting position via random samples. While both methods are probabilistically complete, meaning they are guaranteed to find a path from start to goal given enough time, we decided to implement RRT for this lab since PRM often struggles to navigate through narrow passages and it is easier to incorporate robot dynamics for RRT. Another characteristic of RRT is that it is guaranteed to converge to a sub-optimal solution. While an optimal variant of RRT known as RRT* exists, we decided against implementing this algorithm due to its additional time complexity.

Fig. 2 visually shows the steps in our implemented RRT and the tree/path it generates when ran on the Stata basement map. The algorithm begins by initializing a tree with the initial position of the robot. Then, RRT samples the map from the goal region with probability `GoalBias`, which we set to 0.2, and otherwise samples from anywhere in the search space. Next, we find the nearest point

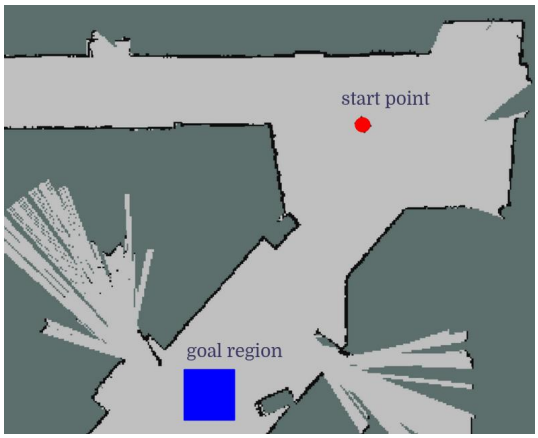
Algorithm 1 A*

```
cost_init ← 0;
path_init ← {x_init};
Q ← {cost_init, path_init}; E ← ∅;
while Q is not empty do
    path_current ← path in Q with min_cost
    x_current ← head of path_current
    if x_current == x_goal then
        return path_current
    else
        remove path_current from Q
    end if
    if x_current in E then
        continue
    end if
    for child of x_current not in E do
        path_new ← path_current + child
        cost_child ← cost_current + EuclidianDistance(x_current, child)
        cost_new ← cost_child + EuclidianDistance(child, x_goal)
        add path_new to Q
    end for
end while
```

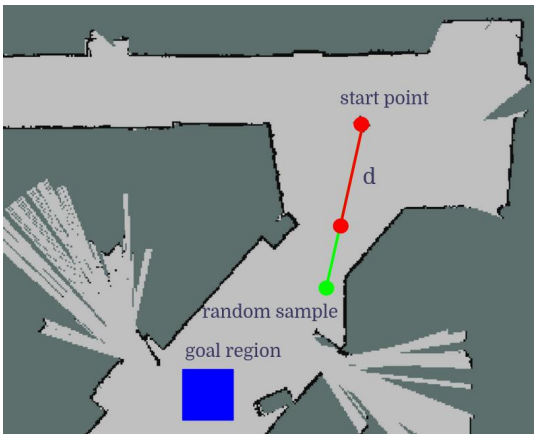
Algorithm 2 Rapidly Exploring Random Tree (RRT)

```
V ← {x_init}; E ← ∅;
while True do
    if Random(0, 1) < GoalBias then
        x_rand ← SampleGoal;
    else
        x_rand ← SampleFree;
    end if
    x_nearest ← Nearest(G = (V, E), x_rand);
    x_new ← Steer(x_nearest, x_new);
    if ObstacleFree(x_nearest, x_new) then
        V ← V ∪ {x_new}; E ← E ∪ {(x_nearest, x_new)};
        if EuclideanDistance(x_new, x_goal) < GoalDist then
            return FindPath(G = (V, E), x_new)
        end if
    end if
end while
```

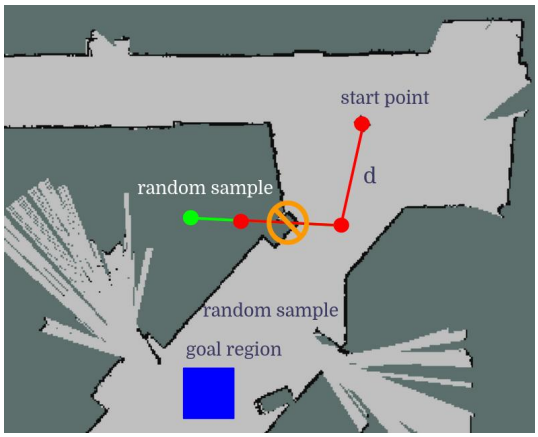
from this sampled point in the tree by iterating through all nodes. Then, the `Steer` function creates a straight path between these two nodes but limited to length d. In our case, d was set to be 40cm which appeared to be a good compromise between runtime and finding paths in narrow passages. Once we ensure this path is obstacle-free by checking if any grid touching the path has a nonzero value in the OccupancyGrid data, we update the tree and repeat the process. Finally, when the euclidean distance between the new point in the tree and the goal position is less than `GoalDist`, the algorithm returns the path by backtracking the parent nodes.



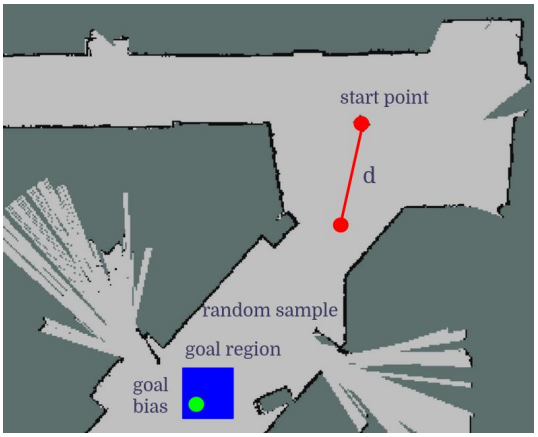
(a) RRT initializes a tree rooted at the robot position (red) and is given a goal region (blue).



(b) Next, it randomly samples from the map (green). The path from the nearest point to the sampled point is limited to a distance $d = 40$cm (red).



(c) Paths which cross an occupied grid are not added to the tree, preventing collision.



(d) RRT samples from the goal region 20% of the time to bias the tree towards the goal.

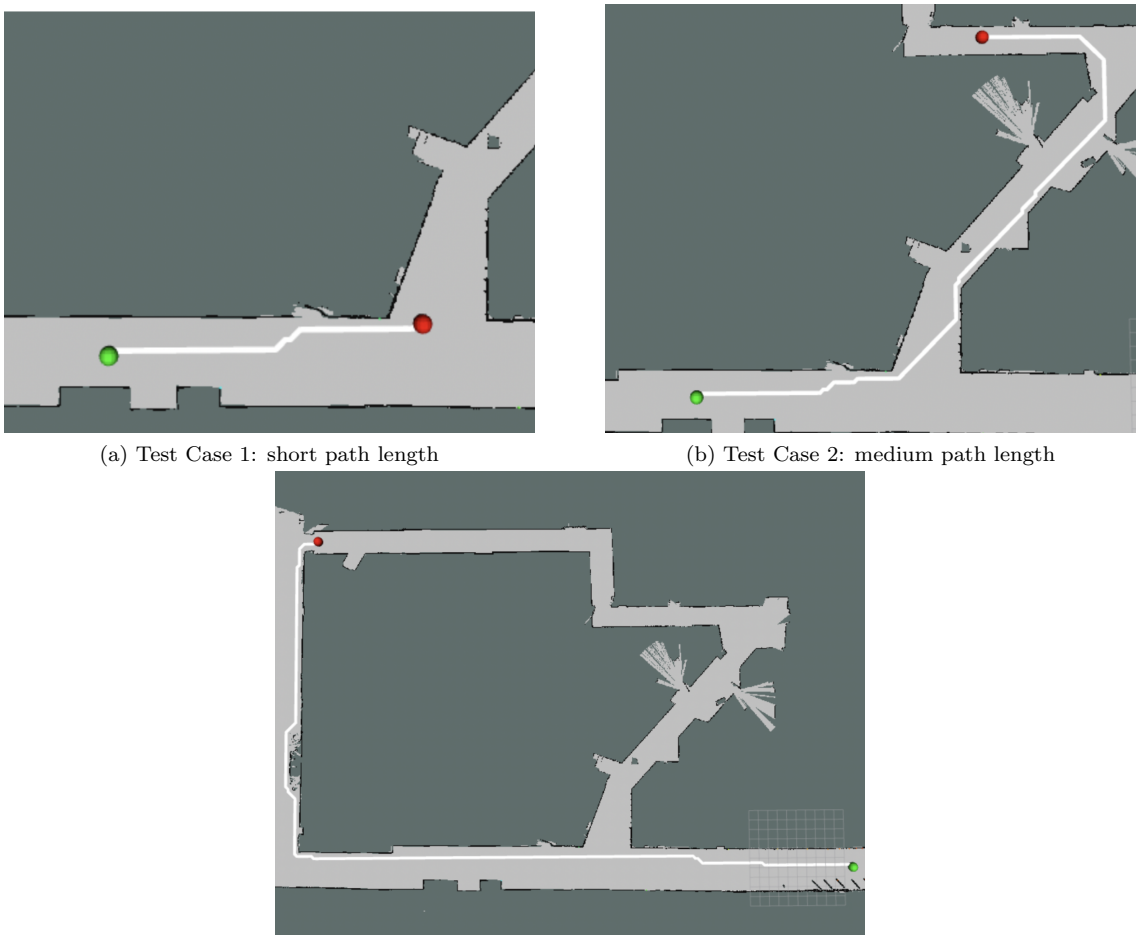
Fig. 2: Our implemented RRT path planner can efficiently find paths through the narrow corridors of Stata basement without collisions by growing a biased tree rooted at the robot towards the goal.

There are modifications we can make to our RRT implementation to improve runtime. First, we can further tune `GoalBias`: increasing this parameter can improve runtime by biasing the tree towards the goal at the expense of searching unexplored areas less. Additionally, we can use a K-D tree, which is a space-partitioning data structure, to store the graph. Using the K-D tree data structure improves the efficiency of finding the nearest node in the tree, which is one of the most time-consuming parts of RRT.

2.1.4 A* and RRT Comparison/Experimental Results

(Author: Grace Jiang)

In simulation, we ran 100 trials of A* and RRT on each of three test cases of varying path lengths and complexities shown in Fig. 3. Across these trials, A* consistently found the same path of the same length, while the length of the path outputted by RRT varied.



(a) Test Case 1: short path length

(b) Test Case 2: medium path length

(c) Test Case 3: long path length

Fig. 3: **Creating three test cases with varying path lengths and complexities proved robust for testing our A* and RRT algorithms.** The green spots mark the start locations of the test cases, and the red spots mark the goal locations. The pictured white paths are those optimal for a discretized grid space, computed by our A* algorithm.

To obtain a useful metric to compare the two algorithms, we averaged across the 100 trials for algorithm runtime and final path length. As mentioned above, our A* algorithm proved to be deterministic, finding a path of an identical length in each of the 100 trials. The final path of RRT varied greatly, sometimes being a close-to-optimal length, and other times being a much longer length, wrapping around a different side of the loop in the Stata basement map to reach the end point.

Averaging these path lengths, we found that A* outputs shorter paths than RRT, except for the short test case. Since our RRT implementation is goal-biased and terminates after reaching a goal region, the algorithm quickly terminates before reaching the exact location of the end point. For this test case, the design of RRT allowed it to also have a faster average runtime than A*, though both algorithms terminate on average in less than 0.1 seconds. Our exact numerical results are shown in Tbl. 1.

Table 1: **Averaging across 100 trials for each case, we found that A* finds a path of more optimal distance in less time for the medium and long paths (Test Case 2 and 3).** Highlighted in green are the path lengths of the path computed by A* for the mentioned tests, and highlighted in red are the corresponding path lengths computed by RRT. We see that the computation times corresponding to these path lengths are also less for A*. On average, A* terminates in less than one second for all cases.

Test Case	**1**		**2**		**3**	
	Runtime	**Distance**	**Runtime**	**Distance**	**Runtime**	**Distance**
A*	0.041 s	16.75 m	0.275 s	55.10 m	0.567 s	95.48 m
RRT	0.012 s	16.08 m	1.112 s	71.69 m	0.769 s	109.25 m

For the medium and long test cases, A* algorithm runtimes were less variable than those of RRT. For these cases, a plot of the runtime distributions of A* and RRT can be seen in Fig. 4. In terms of computation speed, A* is a more reliable algorithm than RRT. For any length path, we can be nearly certain that our implementation of A* will run to completion within a fraction of a second, while the same cannot be said for RRT.

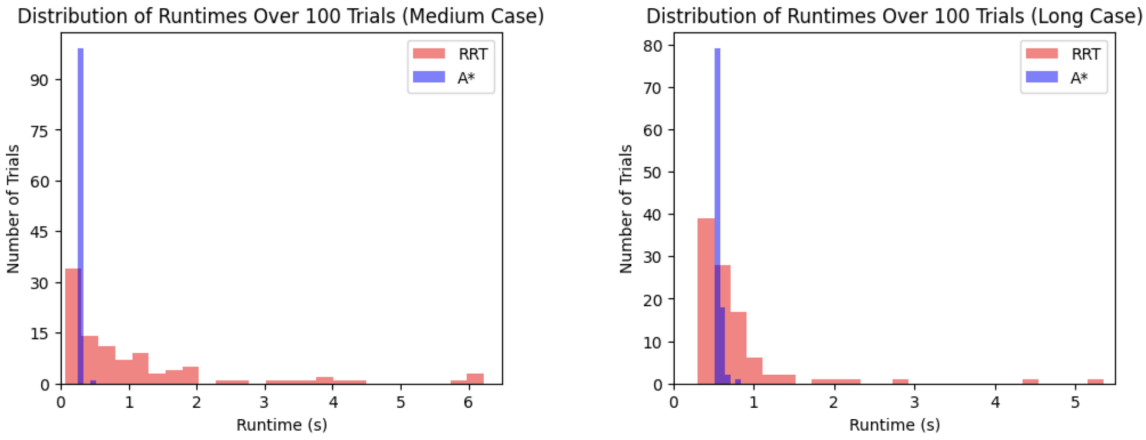


Fig. 4: **RRT has greater runtime variance compared to A*, for both medium and long path lengths.** In the left graph for Test Case 2, RRT has a standard deviation of 1.406s, while A* has an SD of only 0.031s. In the right graph for Test Case 3, RRT has an SD of 0.710s, while A* has an SD of 0.039s.

In summary, with our admissible heuristic of Euclidean distance, A* is guaranteed to find the optimal path based on the given graph, while RRT generates a variety of paths for the same initial and goal positions. Additionally, with the down-sampled map, A* consistently had a quicker runtime than RRT. Due to its optimality and faster runtime, we decided to use A* to generate our path on the robot in the real-world.

2.2 Pure Pursuit

(Authors: Trey Gurga, Cynthia Cao)

Given a set trajectory on the map, our robot must be able to follow it closely and reliably, avoiding corner overshoots and oscillations. Pure pursuit, the algorithm for this task, is based on geometric calculations that dynamically guide the vehicle along a path by continuously adjusting its steering angle.

The key to pure pursuit lies in defining a goal point or lookahead point on the path, which the vehicle aims to reach within a short timeframe. The lookahead distance is the primary tunable parameter; we vary it with changes in the vehicle's velocity and position to make the algorithm adaptive to varying path curvatures and speeds. The algorithm calculates this point by intersecting a circle, centered on the vehicle's current position with a radius defined by the lookahead distance, with the path. The lookahead point is updated at every timestep so that the robot never reaches the lookahead point until it reaches the end of the trajectory.

In our integration with path planning and localization, the algorithm takes the following steps:

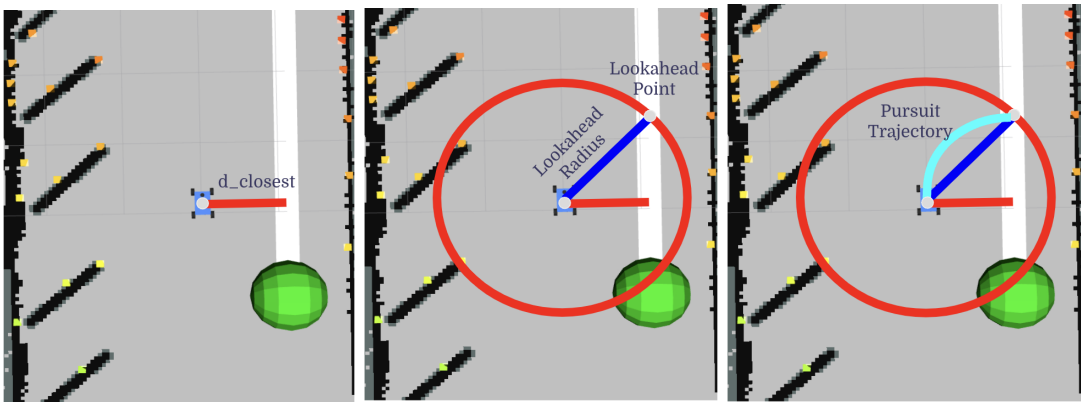
1. A predefined path is manually defined on the map interface or loaded from a trajectory file.
2. Utilizing real-time positional feedback from the vehicle's localization system, the algorithm calculates the closest point on the path to the vehicle. This is achieved through efficient geometric computations, optimizing the process to handle real-time operation constraints.

3. From the closest point, the algorithm determines the lookahead point along the path. This involves calculating the intersection of the trajectory with the lookahead circle.

4. With the lookahead point defined, the required steering angle to navigate the vehicle towards this point is computed using Eq. 3 The steering angle is set such that the path to the lookahead point is not a straight line but an arc for smooth steering.

2.2.1 Design of the Pure Pursuit Algorithm

The Pure Pursuit algorithm operates by identifying a dynamic lookahead point on the path, which is a function of the vehicle's current speed and a predefined lookahead distance. The steps involve calculating the closest point on the trajectory, identifying the lookahead based on the intersection of the trajectory and a circle around the robot with a radius of the lookahead distance, and determining the steering angle to reach the lookahead point along an arc trajectory. These steps ensure close tracking of a trajectory, and are visualized in Fig. 5



(a) Finding Nearest Point (b) Calculating Lookahead Point (c) Calculating Steering Angle

(d) Driving to the Trajectory (e) Following the Trajectory

Fig. 5: **The pure pursuit algorithm allows for effective and accurate trajectory following**: we visualize each step of the process, going from (a) through (e)

Finding the Nearest Point on the Trajectory: The first step in the Pure Pursuit algorithm involves determining the nearest point on the predefined path from the vehicle's current position. This point serves as the starting point for further calculations. Because the trajectory is a discretized set of

points on the map beginning at the start point, finding the closest point eliminates pursuit of points along the trajectory behind the robot. Mathematically the nearest point p_{near} is determined by:

$$p_{near} = \text{argmin}_{p \in \text{Trajectory}} \|p - p_{vehicle}\| \tag{1}$$

where $p_{vehicle}$ is the current position of the vehicle and $\|\cdot\|$ denotes the Euclidean distance. For efficiency, this calculation is vectorized using NumPy operations, allowing for rapid processing even with complex trajectories.

Calculating the Lookahead Point: Once the nearest point is identified, the next step is to calculate the lookahead point $p_{lookahead}$, which is a point on the path at a predefined lookahead distance L_d from the nearest point. This point is determined by finding the intersection of the path with a circle of radius L_d centered at the vehicle's current position. Since the path is represented as a series of points, we utilize the nearest point to find the first point that is greater than or equal to the lookahead distance away from the robot along the trajectory rather than behind the robot. The calculation is done as followed sing Eq. 2 and Alg. 3:

$$D^2 = (x - x_{vehicle})^2 + (y - y_{vehicle})^2 \tag{2}$$

Algorithm 3 Lookahead

Given desired x, y
for each point p_i starting from p_{near} **do**
  `Find` D
  if $D > L_d$ **then**
    `Store` i
  end if
end for
return `min`(i)

This formulation ensures that if there are multiple intersection points on the trajectory, the robot selects the closest point to pursue first, eliminating edge cases where the robot would turn too early or too late.

Computing the Steering Angle: With the lookahead point defined, the final step is computing the steering angle θ required to direct the vehicle towards $p_{lookahead}$. The angle is calculated using the bicycle model approximation, which considers the vehicle's geometry and the angle ϕ between the vehicle's current heading and the line connecting $p_{vehicle}$ to $p_{lookahead}$:

$$\theta = \arctan\left(\frac{2 \cdot L \cdot \sin(\phi)}{L_d}\right) \tag{3}$$

where L is the wheelbase of the vehicle. The angle ϕ is calculated from:

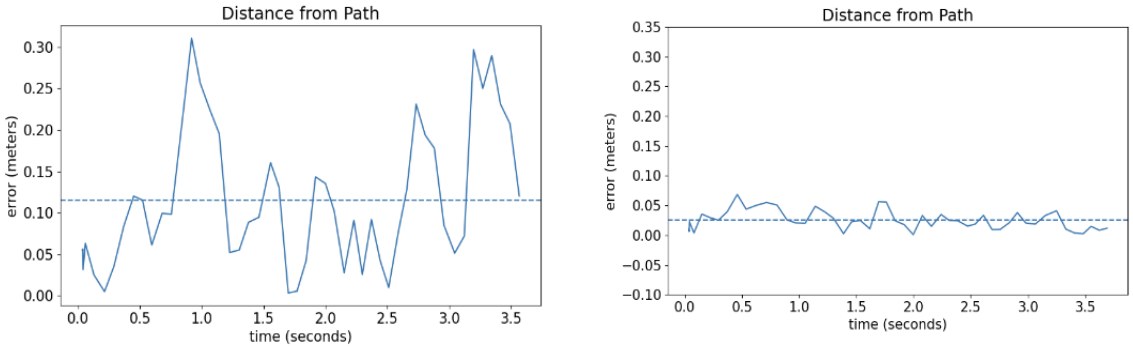
$$\phi = \arctan 2(y_{lookahead} - y_{vehicle}, x_{lookahead} - x_{vehicle}) - \psi \tag{4}$$

where ψ is the current orientation (yaw) of the vehicle, and $\arctan 2$ provides the correct quadrant for the angle.

This approach to the Pure Pursuit algorithm design ensures that the vehicle can dynamically adjust its trajectory at each time step based on current conditions and the defined trajectory, providing robust path tracking capabilities for complex trajectories on the map. With the correct lookahead distance, the robot is able to successfully maneuver through tight corners and remain stable on straightaways.

2.2.2 Experimental Tuning of the Lookahead Distance

The performance of the Pure Pursuit algorithm is highly dependent on the appropriate setting of the lookahead distance. This parameter affects how rapidly the vehicle can adapt to changes in the trajectory. At higher speeds, the robot must have a longer lookahead to avoid oscillation since it is covering more distance in less time. For trajectories with tight curves, a shorter lookahead distance is used to increase the responsiveness of the steering control along the curve to avoid curve overshoot. Conversely, for long straight paths, a longer lookahead distance is beneficial for maintaining stability and smoothness in motion without oscillation, which when implemented on the robot will cause following instability. These adjustments were empirically tested in simulation to find optimal values under various path conditions.



(a) Deviation from Path with 0.2 Scaling Factor

(b) Deviation from Path with 0.4 Scaling Factor

Fig. 6: **Doubling the scaling factor of the lookahead distance from 0.2 to 0.4 significantly decreases error in path following**: With a scaling factor of 0.2, the car significantly deviates from the desired trajectory, as is visible in the error plot in (a). Doubling this scaling factor to 0.4 significantly mitigates the errors in path following (b)

2.2.3 Performance Evaluation in Simulation

The algorithm's efficacy was evaluated through a series of simulated environments designed to mimic real-world conditions. The performance metrics focused on the accuracy of the path tracking, the responsiveness to changes in the trajectory, and the computational efficiency of the implementation. Results demonstrated that the Pure Pursuit algorithm could effectively follow complex paths with a high degree of precision. The tuning of the lookahead distance played a significant role in adapting to different path curvatures, significantly affecting the overall system performance.

2.2.4 Conclusion

The implementation and testing of the Pure Pursuit algorithm addresses the challenge of real-time path tracking in autonomous navigation systems. The flexibility in adjusting the lookahead distance according to the path characteristics proved crucial in optimizing the tracking performance in simulation and real life.

This approach ensures that the vehicle adheres closely to the prescribed path, adjusting dynamically to real-time changes in its environment and operational parameters. Our implementation is designed for rapid testing and evaluation the algorithm's efficiency and accuracy in different conditions around the map.

3 Real World Implementation

(Authors: Cynthia Cao, Toya Takahashi)

Following evaluation in a simulated environment, the path-planning algorithm of choice, A*, and the tuned pure pursuit algorithm were implemented and tested on our robot. Real-world dynamics necessitated further modifications to both algorithms in order to plan and follow paths while avoiding obstacles such as walls and pillars. Using these updated algorithms, the robot successfully followed a variety of trajectories in State basement at high speeds.

3.1 Path Planning in the Real World

Although A* is guaranteed to be optimal within the grid representation of the environment, the paths generated are not optimal to the real world due to discretization errors in the grid. Furthermore, they are often difficult to drive for a robot that has to account for real world dynamics, especially at higher speeds. As a result, we tested two modifications to A*: 1) rewiring paths using line-of-sight, and 2) adding another set of neighbors to find smoother paths. Ultimately, we chose the second modification for use on our robot.

3.1.1 Rewiring A* Through Line-of-Sight

A* requires a discretized search space such as the pixel grid representation of the State basement that we use. As a result, A* is unable to generate diagonal paths that do not cut through grid spaces at 45 degrees. This results in paths that are not optimal to the real-life environment, which is a continuous search space. However, we can use A* to find a path that approaches the optimal continuous path by rewiring the path when new nodes are added. Specifically, when we evaluate each child node of the current node, we also backtrack as detailed in Alg. 4, checking each previous node in the path until we find the farthest node x from which there is a clear line-of-sight to the child node. Each node in the path between is then removed from the path, and the cost of the child node is updated accordingly. The differences between the two algorithms are shown in Fig. 4.

Algorithm 4 Backtracking

```
Given path_current, cost_current
Given x_current, child
for x in path_current != x_current do
    if ObstacleFree(child, x) then
        remove x from path_current
    else
        cost_current ← cost_x + EuclidianDistance(x, child)
        break
    end if
end for
```

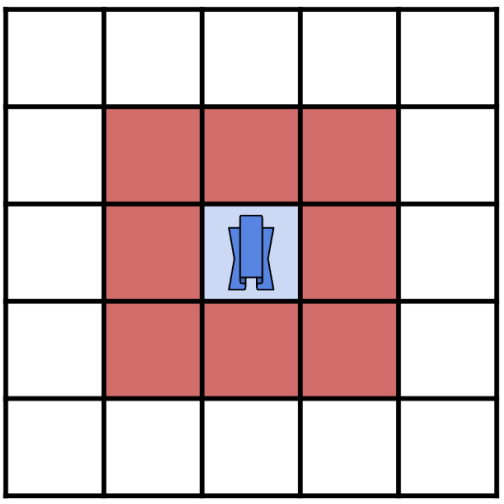
(a) Path Generated by A*

(b) Path Generated Post-Rewiring

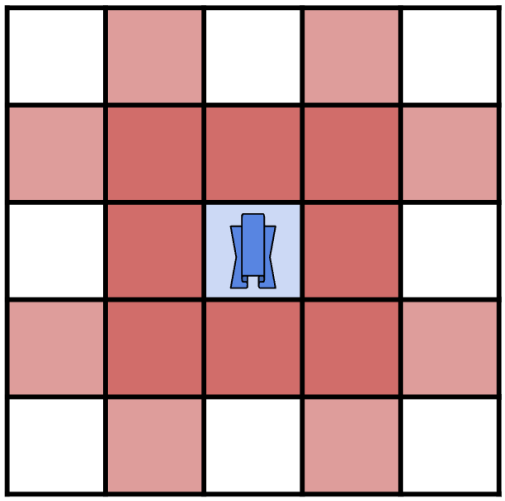
Fig. 7: **The path generated from rewiring A* using line-of-sight (right) is a more optimal path in real life compared to the path generated by A* (left)**. However, some segments of the path fall too close to the wall for the car to easily avoid collisions at high speed, motivating us to test the second modification, which entails adding extra neighbors when sampling the space.

3.1.2 Using "Knight-Move" Neighbors for A*

Another way we improved the path generated by A* was by adding more neighbors to the given graph. Previously, A* expanded to the eight surrounding grid at each time step, but the "knight-move" A* explored additional neighbors as shown in Fig. 8(a) and 8(b). The additional neighbors allowed the algorithm to find paths which are not restricted to 45 degrees increments, decreasing the total distance on the path. Additionally, the runtime of the algorithm is not as heavily compromised as rewiring A* through line-of-sight since no backtracking is necessary. The end result and the improvements made by adding "knight-move" neighbors can be seen in Fig. 8(c) and 8(d).



(a) A* Neighbors



(b) A* Neighbors with the Knight Move

(c) Path Generated by A*

(d) Path Generated with the Knight Move

Fig. 8: **The path generated using the additional neighbors is smoother and shorter than the path generated by the previous A* iteration.** On the race track, this algorithm modification prevented the robot from steering too close to the wall.

3.2 Pure Pursuit in the Real World

Our pure pursuit algorithm also required further refinement to account for dynamics not present in the simulation. The key tuning parameter, the lookahead distance, is modified in two key ways: 1) changing the scaling factor, and 2) changing the lookahead distance based on steering angle.

3.2.1 Modifying Lookahead Scaling Factor

In the real world, we continue to scale the lookahead distance on the velocity of the robot but double the scaling factor from simulation in order to prevent oscillations along the path during trajectory following. Comparing the results from Fig. 9 and Fig. 10, the improvement in performance is clear.

(a) Timestep 1 (b) Timestep 2 (c) Timestep 3

Fig. 9: **Using the original lookahead scaling factor, the robot oscillates when it tries to follow a path straight forwards**: the different poses of the car in (a), (b), and (c) show deviations from the intended path, which is undesirable behavior for the robot.

(a) Timestep 1 (b) Timestep 2 (c) Timestep 3

Fig. 10: **By doubling the lookahead scaling factor, the robot is now able to easily drive along the straight-line path**: the car maintains a constant pose in the forward direction at all time steps shown, showcasing effective path following.

3.2.2 Modifying Lookahead During Turns

We also prevent the robot from cutting corners during turns and thus running into walls (see Fig. 11). We halve the lookahead distance when the robot is turning which we defined as having a steering angle greater than 8.6°, a number experimentally chosen. With this modification, the robot is able to turn robustly (see Fig. 12) and consistently follow a variety of paths in Stata basement.

(a) (b)

Fig. 11: **Without scaling the lookahead distance, the robot takes a path that would lead to collisions with the wall**: the robot is unable to make it past the turn, as the lookahead distance is too far ahead to prevent it from turning into the wall. There is no check to see if the steering angle crosses a threshold.

(a) (b)

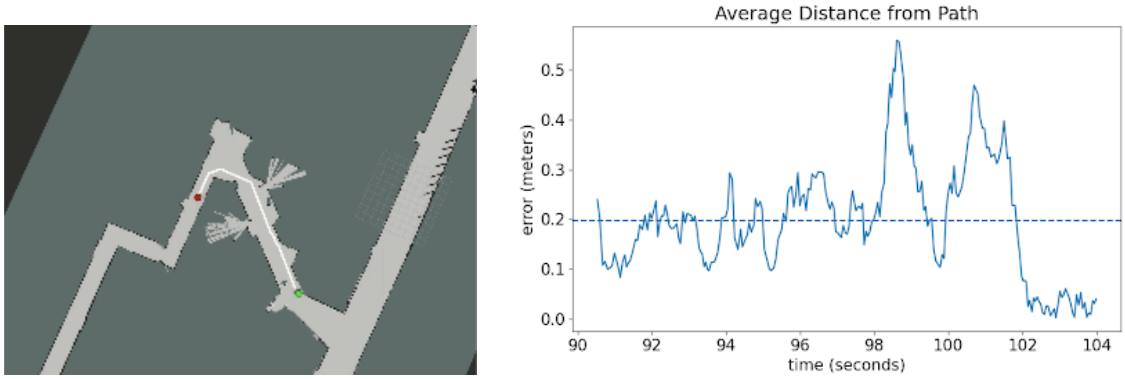
Fig. 12: **The robot avoids collision when lookahead distance is halved during turns**: The threshold of 8.6◦ allows the robot to accurately determine that it is turning, and adjust its lookahead distance to compensate. The robot is able to safely make it around the turn.

3.3 Experimental Results

Two trajectories shown in Fig. 13(a) and 14(a), were chosen in order to evaluate the robot's ability to follow drive through straight hallways and around corners. The robot drove along each path at a constant speed of 2.5 m/s.

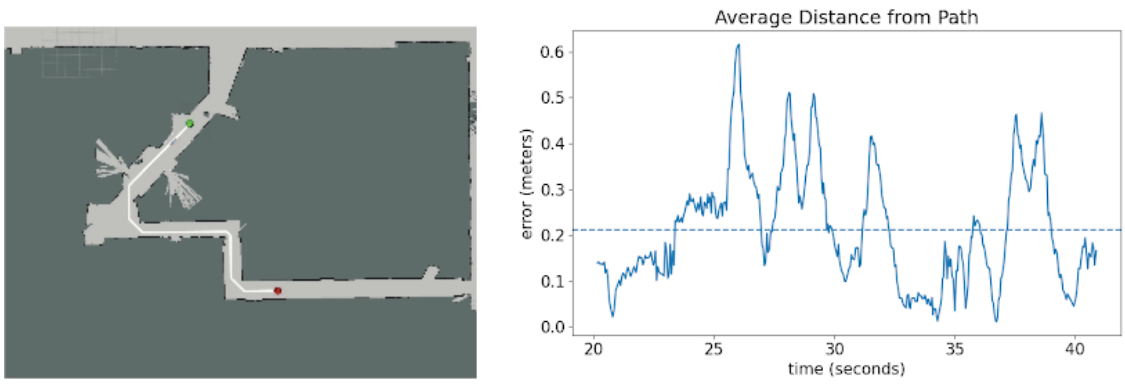
The average error for Track 1 was measured to be 0.198m and for Track 2 was measured to be 0.211m. We plot the exact error in Fig. 13(b) and 14(b). The robot is able to successfully follow planned paths in the real world with little deviation from the track.

Finally, on the race track, the visualized path in RViz that the robot finds using A* with knight-move neighbors is shown in Fig. 15. Our robot successfully drove this track in 25.9 seconds with an in-simulation planning time of 0.62 seconds.



(a) (b)

Fig. 13: **The robot easily follows the trajectory (Track 1) shown with minimal error.** The robot follows the path at 2.5 m/s, with an average error of 0.198m from the planned path.



(a) (b)

Fig. 14: **Along a path with several turns into alternating directions (Track 2), our robot continues to effectively path-follow.** The robot is able to find a path for a more difficult planning problem, and follow it without collisions on in either direction. The error remains low, at 0.215m from the desired trajectory, even with the added complexity in the path.

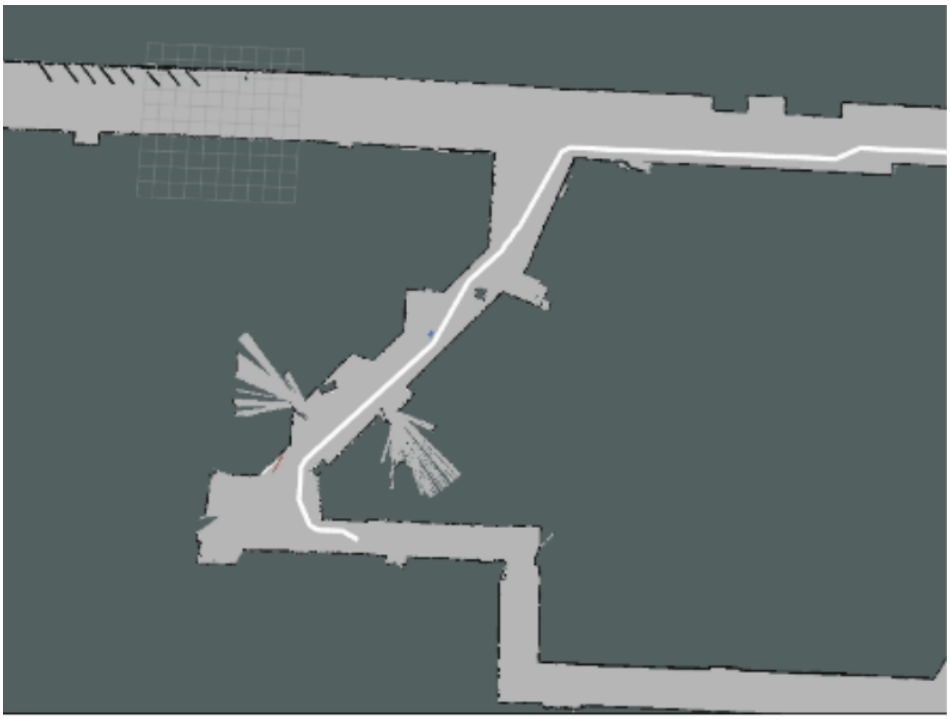


Fig. 15: **Our robot plans the shown race path across Stata basement in 0.62 seconds and follows it to completion in 25.9 seconds.** Our modified A* algorithm creates a smooth race path that avoids steering the robot too close to walls. Running our localization on the robot gives an accurate estimate of where it is along the track, and our pure pursuit model keeps the robot following the planned path.

4 Conclusion

(Author: Cynthia Cao)

This report presents a successful implementation of path following both in simulation and on real-life hardware for the purpose of racing an autonomous vehicular robot in the State basement. Path following is composed of two algorithms: either a search-based or sample-based algorithm to generate the path, and a pure pursuit controller that allows the robot to follow a given trajectory.

We considered two path-planning algorithms, A* (shown in Alg. 1) and RRT (shown in Alg. 2), ultimately finding that A* was better suited for implementation on the robot by nature of its faster runtime and more optimal path-finding (see Tab. 1 and Fig. 4). A* was further modified, as seen in Fig. 7 and 8, for improved performance on real life hardware. A typical pure pursuit implementation was used, where the robot is drives towards a lookahead point on the trajectory that is continuously updated until the goal is reached. We first tuned the lookahead distance in simulation (see Fig. 6, before implementation on real-life hardware (see Fig. 9 and 10). We further tuned the lookahead distance with the steering angle for robustness when turning (see Fig.11 and 12. With these two algorithms, our robot was able to successfully through a variety of tracks, as seen in Fig. 13, 14, and 15.

Potential modifications to improve performance of our path planning include updating our path-planning algorithm to better incorporate the dynamics the of the car. Furthermore, our current

implementation does not consider the pose of the car at either the start or goal; this will need to be implemented for successful completion of City Driving in the final challenge. Furthermore, we can tune the lookahead distance in our pure pursuit algorithm even more finely to be as robust as possible. The speed of our car were capped at 2.5 m/s this lab, but at higher speeds our current implementation may not be as robust, which is something we will have to consider as we move to the final challenge, where our robot can reach speeds of 4 m/s.

The next phase of our work will be to incorporate everything we have learned so far, including using ROS, computer vision, line following, localization, and planning, to implement code that will allow our car to both to race along the Johnson Center Track while remaining safely in its lane, and successfully complete City Driving, where our car will have to localize and path plan while paying attention to stop signs and other visual features. As such, successful implementation of this lab is crucial to our work for the final challenge.

5 Lessons Learned

Cynthia Cao Through this lab, I learned a lot about the designing and evaluating path-planning algorithms and gained a better understanding of how pure pursuit controllers worked. I ended up implementing several variations of A*, which was a learning experience in understanding how variations on certain algorithms could be used in different scenarios. Furthermore, I became more familiar with debugging problems on the robot over the course of the lab, which was a frustrating experience, but very rewarding once our robot successfully drove. Finally, I saw how important being organized and being communicative as a team was, especially when it came to multiple of us implementing different versions of algorithms and solutions to problems that came up.

Trey Gurga ahfahfoewhfuapo

Grace Jiang I gained an understanding of A* and RRT during this lab. Though I'd heard of A* before, I certainly understand it better now; I'd never heard of RRT before and now am aware of it. The experience of looking at path finding algorithm modifications and comparing performance was fun. When looking at data from repeated trials of path-finding algorithms, I felt once again that I like data, especially when it's complete and there are large amounts of it. I also found it satisfying when our robot smoothly sped along the race track after we got everything working together near the lab deadline.

Toya Takahashi From this lab, I learned about how pure pursuit controllers work and implementations of both search-based and sample-based planning algorithms, namely A* and RRT. The lab also allowed me to think of ways to improve generated paths or improve runtime such as adding more neighbors to each node in the graph for A* or adding a goal-bias factor to RRT. Finally, learning to communicate effectively and use git branches in an organized manner was essential for this lab as it had multiple parts including path planning, pure pursuit, and localization.

Jonathan Zhang In this lab, I gained a high level understanding of how path planning works. Just like with lab 5, working hands-on with the search algorithms and pure pursuit allowed me to develop a more in depth understanding of the material taught in lectures. This was especially true with A*, where I was worked on optimizing, implementing, fine-tuning and testing the algorithm.

In this lab, I also gained new experience working with quantitative analysis of results I would normally just look at qualitatively. When comparing A* and RRT, instead of just observing the runtime and looking at the paths, our group used a for loop to run the algorithms 100 times to generate meaningful

numerical data. In this way, we were able to more concretely make decisions about which algorithm would work better for our implementation. Thinking about things this way was super useful practice, and I'll be sure to use it for work I do in the future.

Finally, I also got more practice with parallel coding and feel a lot more confident working with others at the same time because of it. Previously, I would have felt much better just coding things on our own and comparing, but now because of the practice from the labs I've reached a point where I feel coding together is actually more efficient.

(Editors: Cynthia Cao, Trey Gurga, Grace Jiang, Toya Takahashi, Jonathan Zhang)