EE183DA

Team Poseidon

Lab 3: Trajectory Planning

Authors

Anastasia Sukhorebraya-Beck

Arash Jalili Kalhori

Nathan March

Yuci Shen

Table of Contents

Introduction and Lab Overview

In this lab, we built a Rapidly-exploring Random Tree (RRT) to find a path from an arbitrary ignition position to a final position in the presence of obstacles. We used a nonholonomic planner to capture the true physical motion of the car in a 2-dimension grid world. Check out the github listed in the appendix to see the implementation of these functions.

Symbols and Conventions

a. Symbols

C = configuration space
C_{obs} = object space in configuration space
V = set of vertices in C-space
E = set of edges in C-space
s = state
a = action
s' = next state
x = x-postion
y = y-position
θ = angle

b. Assumptions

Each node in our graph contains the state of the object represented by the three parameters shown below as well as the nodes parent.

$$s = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

For this lab C_{obs} is a set of rectangular obstacles that give a 0 transition probability when s' is an element of C_{obs}.

Trajectory Planning Simulation

a. C-Space Representation

For tools we used python for coding and the pygames library for the visualization tool. We created a window and filled it with black to represent the free space available

to be traversed. The list of obstacles is represented by red rectangles. To set the start and end point you click any black part of the screen. The first click represents the starting point and the second click represents the goal state. The radius of the initial and final point represents the radius of the car at any node along the RRT tree trajectory.

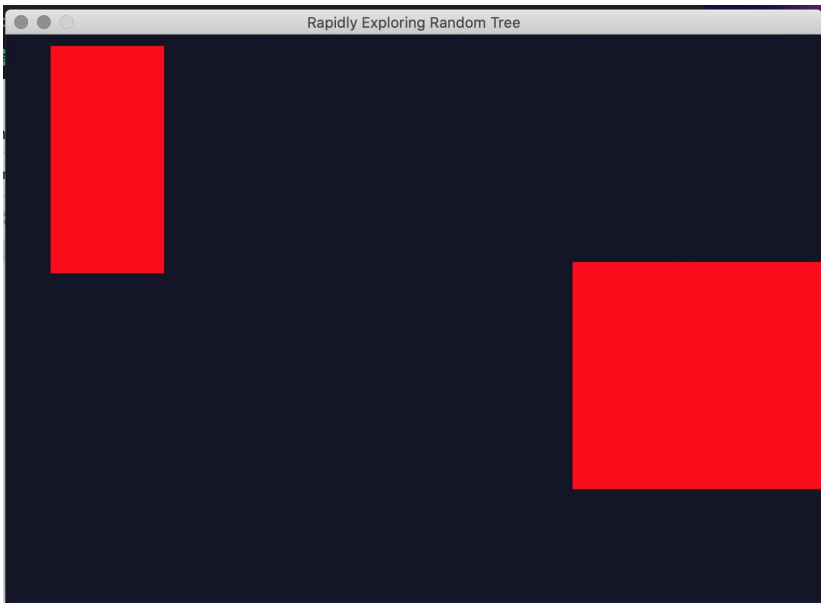


Figure 1. Robot Environment

Figure 1 represents the environment for our robot. The red rectangles are obstacles representing C_{obs}. The black space is the C space of our robot.

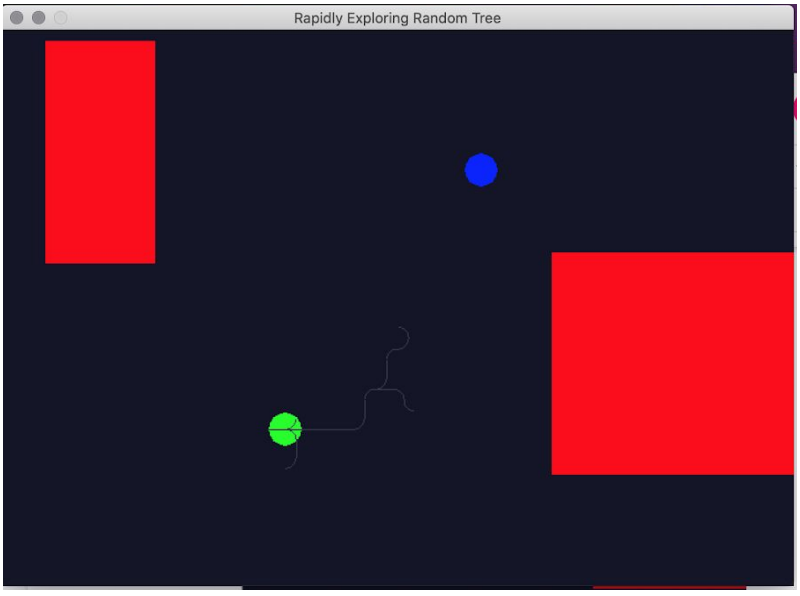


Figure 2. First Steps of RRT

The start state of our car is the green dot with the radius of the circle representing the radius of our car. First we place the green dot in an arbitrary location. Then once we set the goal state, shown by the blue dot, we can see that the RRT starts forming from the green dot expanding outwards to eventually hit the blue dot.

b. Rectangular Shape Consideration

To adapt this function to support a true rectangular shape we would need to modify the collision function described later to check overlap between two rectangles instead of the overlap between the current node radius and the rectangular obstacles. We would also need to ensure that the bounding box extends past the wheels to avoid collision. We chose not to implement the rectangular shape of the robot body because this would lower the amount of error leeway we have.

c. One Time Step Next State

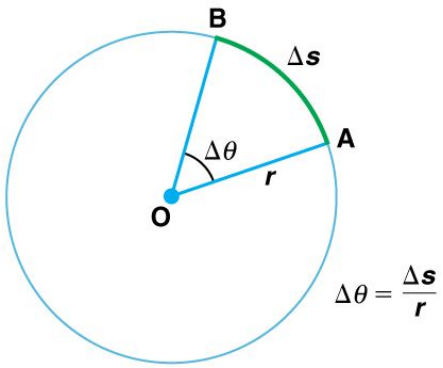
To get the next state after one-time step for this nonholonomic system we simplified the problem by not considering all input PWM control combinations, but four simplified PWM control combinations, we can reduce the computation time and generate a path more quickly. The four car maneuvers we considered were driving forward, backward, turning right, and turning left. Based on the x, y, and theta state we are considering, we calculated the future state using our system dynamics. Then after calculating each final position for each maneuver we chose the action that minimized the distance between the node and the point in question. The control inputs for the commands describe can be represented by the table below:

Action	PWM Left	PWM Right
Forward	0	180
Backward	180	0
Turn Left	60	180
Turn Right	180	60

Table 1. PWM mappings to commands

Table 1 describes the control input for any command that we send to the car. Each node in our path stores the command to get to that node, so we can easily convert these list of commands to control inputs with the table above.

For Left and Right turns, we can't simply draw a line between two points, and must calculate the start/end coordinates and use them to draw an arc.



Given Values
- Start Point A [x_a, y_a, theta_a]
- Turn Angle d_theta
- End Angle theta_b
- Radius R

Find
- End Point B [x_b, y_b, theta_b]

Step 1: Find the end orientation. 0 rad points to the right, and PI/2 rad points up.
- For Right Turns: theta_b = theta_a - d_theta
- For Left Turns: theta_b = theta_a + d_theta

Step 2: Create a correction factor
a. Right Turns … $\phi = 0$
b. Left Turns … $\phi = PI/2$

Step 3: Find the center coordinates using the starting coordinates and the vector AO
- A_tangent = < cos(theta_a + ϕ), sin(theta_a + ϕ) >
- AO = < Rsin(theta_a + ϕ), Rcos(theta_a + ϕ) >
- Center = [x_a + Rsin(theta_a + ϕ), y_a + Rcos(theta_a + ϕ)]

Step 4: Find the end coordinate using the center coordinates and the vector OB
- B_tangent = < cos(theta_b + ϕ , sin(theta_b + ϕ) >
- OB = < -Rsin(theta_b + ϕ), -Rcos(theta_b+ ϕ) >
- Endpoint = [center_x - Rsin(theta_b + ϕ), center_y - Rcos(theta_b + ϕ)

Step 5: Get the start/end angles for pygame.draw.arc
- angleA = theta_a + d_theta + ϕ
- angleB = theta_b + d_theta + ϕ
- startAngle is the smaller angle, and endAngle is the larger

Step 6: Get the rectangle dimensions for pygame.draw.arc
- height = width = radius*2
- rec_x = center_x - radius

- rec_y = center_y - radius

d. Collision Free Check

In order to check that the state remains collision free based on the next state we leveraged the pygame libraries built in collidepoint function inside the rectangle class. Since all out obstacles are rectangles, we just checked that the next state we chose did not collide with any rectangles in C_{obs}.

In part 3(g) we show how we optimized that model by checking every point in the trajectory of a potential step to make sure that the step chosen will not cut through an obstacle. This rarely affects an optimal path for a maze with large solid obstacles. But if we had smaller obstacles, this would be an even more obviously useful upgrade.

e. Closest Point to Random Point

Determining the closest point to a point outside the graph required us to calculate the distance between every node *V* in the graph and the single other target point. We implemented this using Euler Distance between the two points.

$$dist(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

f. RRT Planner

By combining the above functions, we can create the full RRT planner and calculate the trajectory from start to end. Below is the pseudocode for combining the following functions:

1. Random Point

   Generate a random point in C space that is not in C_{obs}.
2. Closest Node

   Find the state in nodes *V* that is closest to the random chosen point.
3. Calculate Next State Without Collision

   First calculate the end position of each of the four maneuvers: forward, backward, left, right. Then, pick the action that minimizes the distance to the random point. Note that we are picking that minimum distance where the next state is also in C space and not in C_{obs}
4. Add State to Graph

   Add the node to the graph containing the next state from the best command with an edge from the closest node to the random point to the next state.
5. Continue until Goal State

   Continue adding nodes to the graph until the next state reaches the goal state. Then backtrack, adding every edge from each node to its parent

node until the start node. The result will be the nonholonomic RRT path from start state to goal state.

g. Robust Trajectories

To create more robust trajectories we need to account for the fact that even if a line might not end points inside an object, its trajectory can still be inside.

Note this will not be as useful in case where all the obstacles are much bigger than the travel distance because as a path hits an obstacle, in the next iteration that node will not be expanded further into the obstacle. So even if we hit an obstacle, it will not be a part of the optimal path. After Improvement #1, for the trajectories (left and right) we will use a list of points inside the bounding box to check that C_{obs} and bounding box of the potential step's trajectory do not share any points..

Improvement #1

Before checking every point in the trajectory for a possible collision we have the following edge cases:

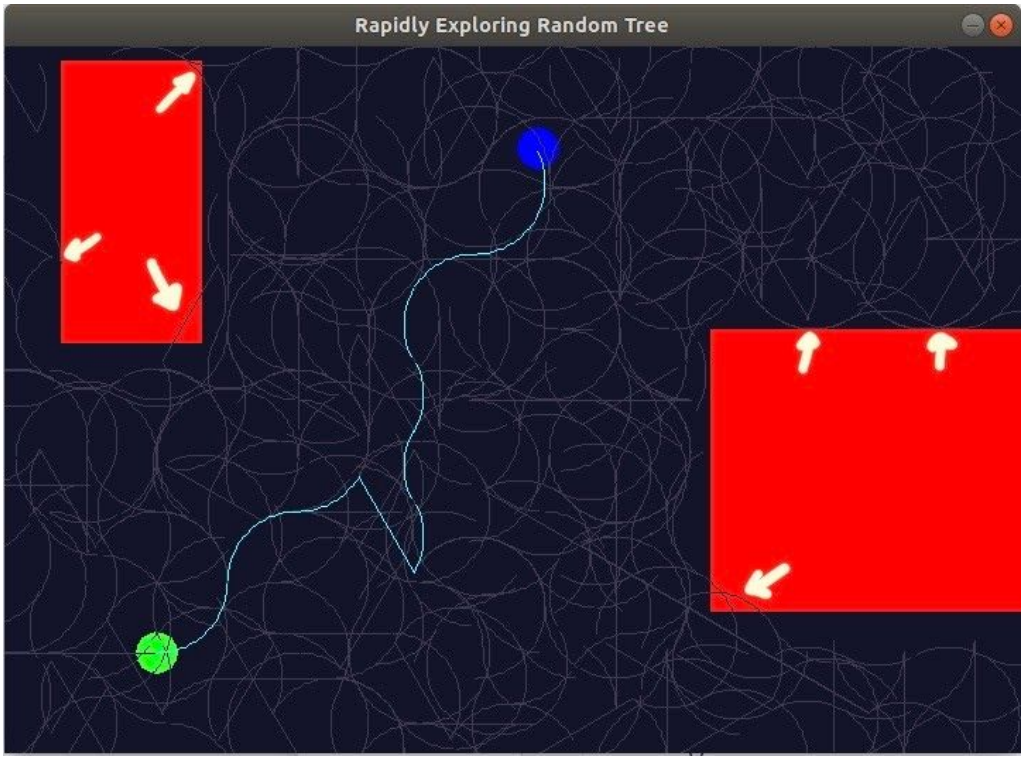


Figure 3. Path Search w/ Turn Collisions

Commentary for the **Figure 3** below explains in more detail why this happens and how we address it. This issue is caused by the initial logic we used to evaluate which actions were legal and which nodes were the best.

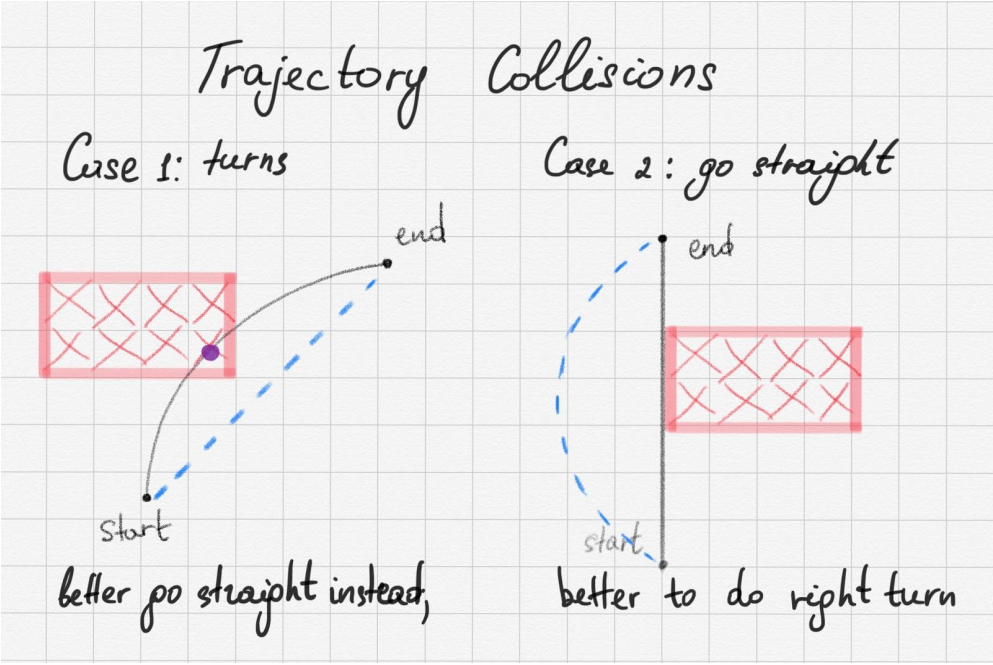


Figure 4. Trajectory Collisions

In Figure 4 Case 1, we are making a turn. Even though start and end positions have no collisions, the trajectory goes through the obstacle box. In Case 1 it is better to follow dashed blue line and go straight.

But the problem is that we choose a step based on which action will bring us closest to the goal, so to the algorithm in Case 1 both turning right and going straight looks equally good.

In Figure 4, Case 2 the car might brush too close against the wall of the obstacle, and the path would be more robust if we turned right.

Solution:

Instead of just checking if start and end positions of a step collide with obstacles, we check if any of the points in that step's trajectory may collide with any of the obstacles on the grid.

To do this we put the trajectory of each possible step into a bounding box and then check that the list of cells in that bounding box does not share any cells with the list of obstacles cells. We must also consider 3 cases (each with 2 subcases).

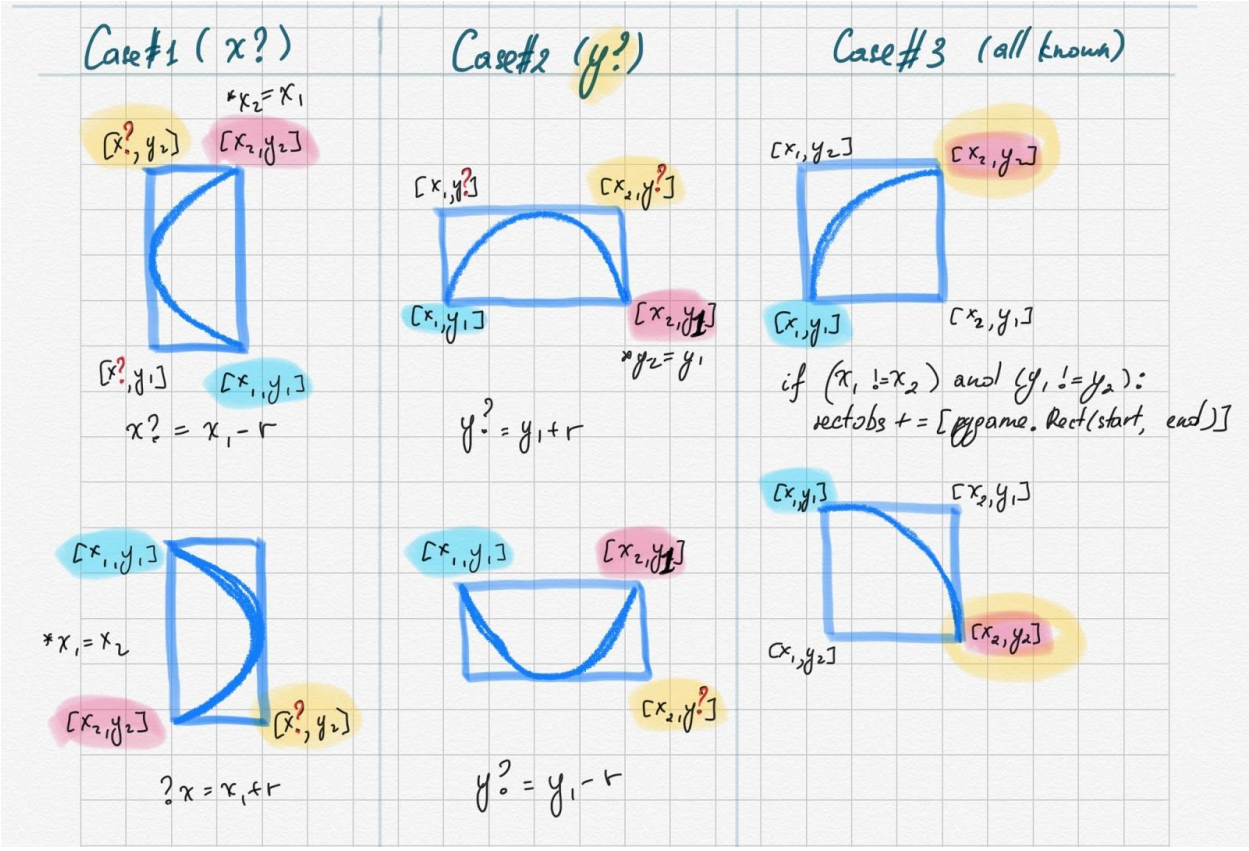


Figure 5. Finding Trajectory Bounding Boxes

To create a bounding box we need x and y positions of the start position (highlighted in light blue) and the opposite corner of the bounding box. Figure 2 shows how to find unknown x_2 and y_2 (highlighted in yellow) given the turn radius, r. Now the algorithm can check for collisions for the entire trajectory and see that those moves are illegal.

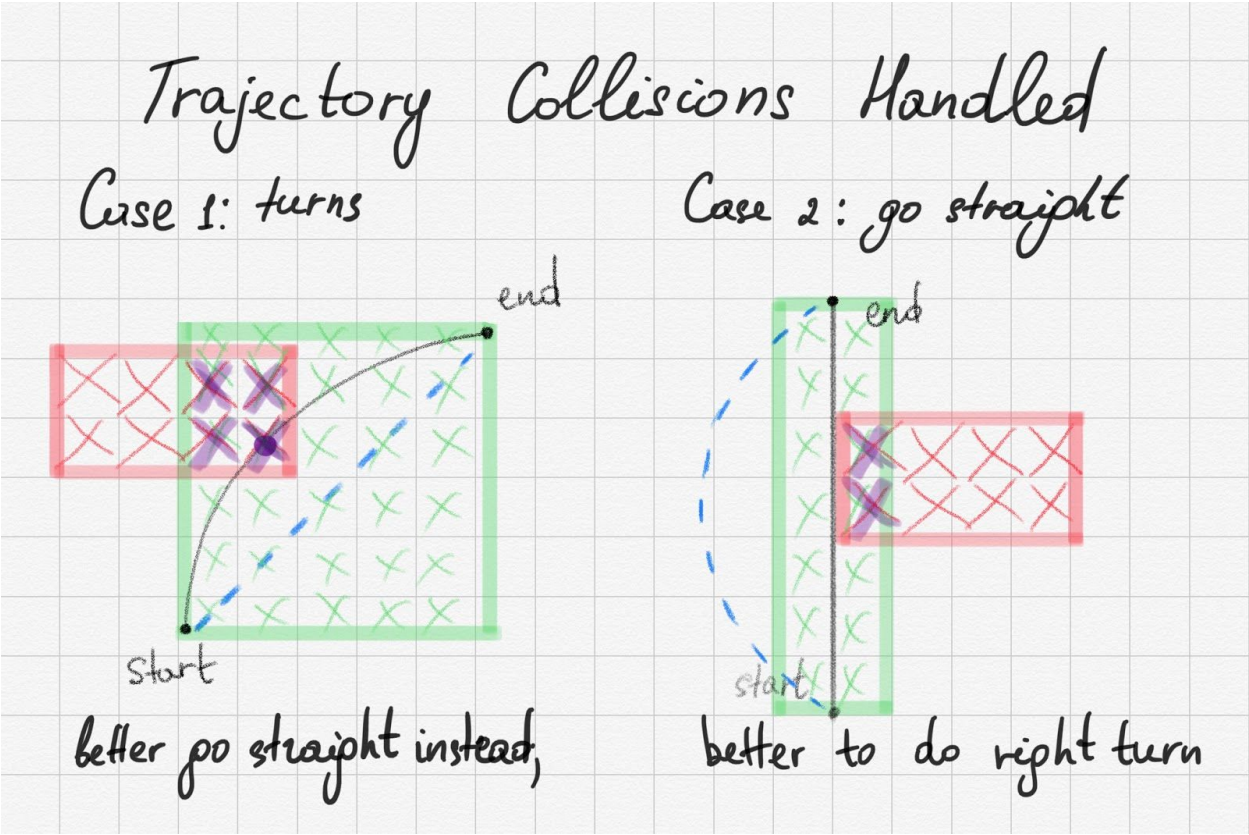


Figure 6. Trajectory Bounding Box v. Obstacles

This gives a more robust algorithm because we're guaranteed that the car will not cut through or brush against an obstacle. This also allows us to find more efficient paths as we can set tighter radius for the car and not worry that it will collide.

Figure 7.1. With small decay

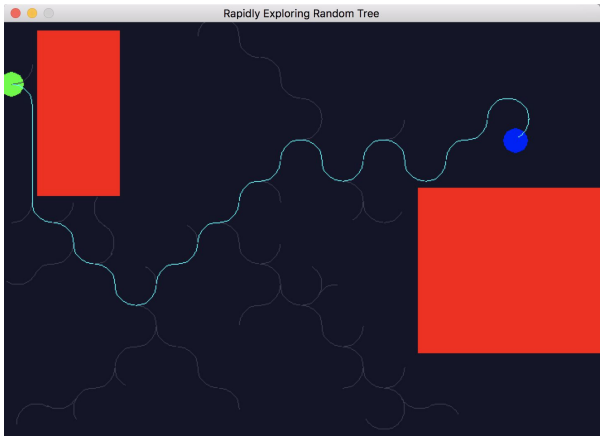


Figure 7.2 With large decay

Now we have another hyperparameter that determines how close a car can be to an obstacle by affecting the size of the bounding boxes for the trajectories. We can see that now all steps that are considered don't cross or touch obstacles, giving a more realistically executable and robust path.

Improvement #2

No matter what step we take - we will be further away from the goal. Which could send us on multiple detour loops even if we choose a radius around the goal object and consider that the path hit the goal if we are within a certain radius from the goal.

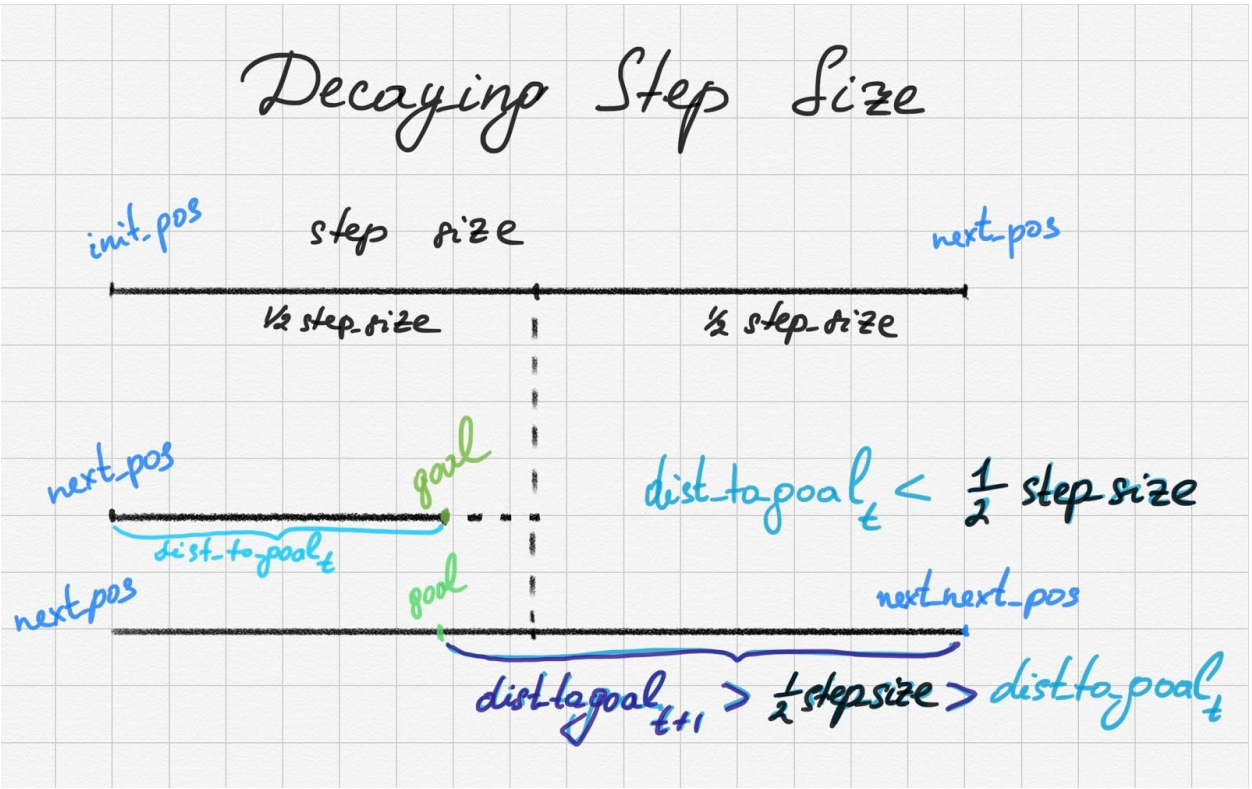


Figure 8. Decaying Step Size

Solution 1: This could be solved by choosing a good constant step size. But if the initial position is far away from the goal it will take a really long time to converge. We also loose precision and can't reach the goal exactly.

Solution 2: It's better to have a decaying step size - as we get closer to the goal, we decrease the step size to prevent bouncing around the goal.

Figure 9.1 Constant Step Size

In Figure 6.1, we see that at one point we get really close to the goal. But then because by chance, we didn't get to go straight, we ended up making a detour going further away from the goal before converging.

Figure 9.2 Decaying Step Size

Here we see an improvement due to the step sizes decreasing as we approach.

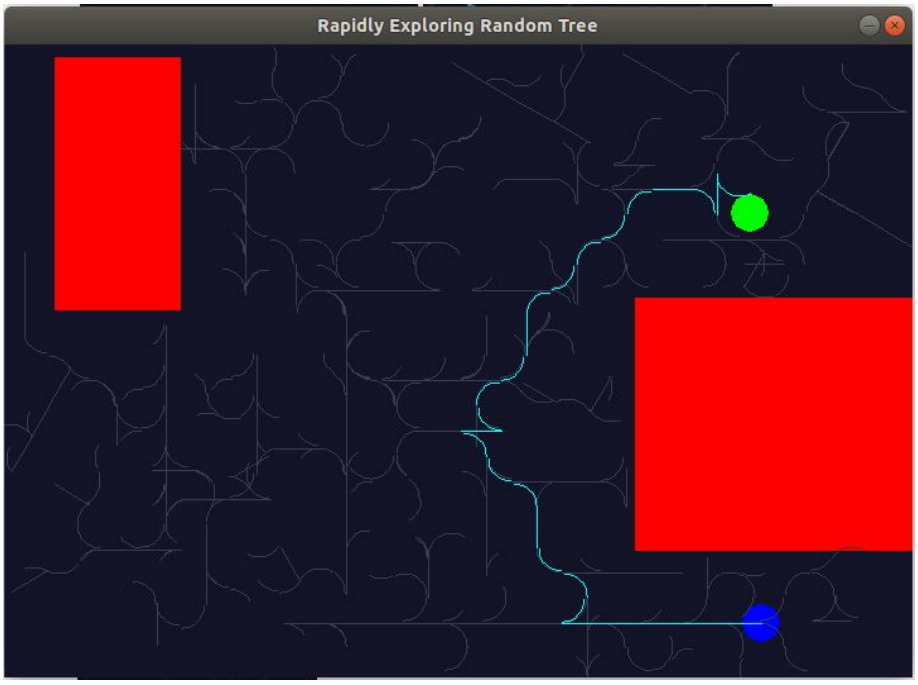


Figure 10. Full map using a decay

RRT* Pseudocode

To find the optimal RRT algorithm between two points much more computation is required. The first change that we must change to our code is to associate some cost with each step. Then whenever we need to add a node we need to be able to get a list of nearby nodes with a certain radius and check if the cost would be reduced if the parent of each node was the new point instead. If so, then rewire the tree so that the parent node becomes the new node instead. Below is some pseudocode of the addition necessary to our code to create the RRT* algorithm:

```
g = graph
x_new = RandomPoint()
If (x_new collides with obstacle) repeat above
x_nearest = nearestNeighbor(g, x_new)
cost(x_new) = distance(x_new, x_nearest)
x_neighbors = get_nearest_neighbors(g, x_new, RADIUS)
For x in x_neighbors:
        If (cost(x_new) + distance(x, x_new) < cost(x):
                cost(x) = cost(x_new) + distance(x,x_new)
                parent(x) = x_new
return g
```

Results

Everything works as expected.

This algorithm returns slightly less efficient path but is a significant improvement in runtime compared to Value and Policy Iteration.

Hence, there is a tradeoff between how smooth and efficient the final trajectory is and the runtime.

In real life if we wanted RRT runtime but a smoother trajectory, we could give higher probability or otherwise prioritize going straight or along the curve with a large radius.

Evaluation

a. PaperBot Experiments

DECAY

- A rate of decay in the step size is a hyperparameter. As such, when it is too high the algorithm can get stuck greatly increasing the runtime, so we only want the step size to decrease when we are a few steps away from the goal.

EPSILON

- Bigger epsilon leads to larger steps and the algorithm converges faster.
- Smaller epsilon finds a smoother and more efficient path.

TURN_RADIUS

- Very small turn radius leads to smooth trajectory.
- Increasing the radius leads to optimal path with fewer steps but that is less smooth.
- Turn radius of more than pi/2 leads to too much wandering and inefficient final trajectory.

GOAL_RADIUS

- Bigger goal radius leads to faster but less precise conversion.
- Goal Radius that is too small compared to the step size could lead to oscillation and detours when the path gets close to the goal. We want to find balance for this parameter.

b. Computational Time

Value iteration algorithm is $\mathcal{O}(|S|^2|A|)$, where $|S|$ is the number of states and $|A|$ is the number of actions. In contrast RRT is O(S^2) because in our case we choose action randomly. However if there are many obstacles and moves are deemed illegal, the upper bound approaches Value Iteration algorithm's runtime.

Part 5 (a) on PaperBot Experiments also discusses how hyperparameters affect the test runtime.

c. Obstacle Dynamics

In part 3(g) we show how we optimized that model by checking every point in the trajectory of a potential step to make sure that the step chosen will not cut through an obstacle. This rarely affected an optimal path for a maze with large solid obstacles. But, when run with many small obstacles, a bounding box for the step trajectory was imperative for getting robus final path.

d. Improvements

- Improvements 1 and 2 are detailed in part 3(g).
- Based on experiments, we set the optimal hyperparameters.
- In the future, we can generate a large dataset of starting points, goal points, and obstacles and do a 5 fold cross-validation to optimize hyperparameters to minimize the number of nodes it takes to find an optimal path.

Appendix

https://github.com/nathanzmarch/183DALab3

References

"Chapter 1 – Installing Python and Pygame." *Invent with Python*, inventwithpython.com/pygame/chapter1.html.

Chin, Tim. "Robotic Path Planning: RRT and RRT*." *Medium*, Medium, 26 Feb. 2019, medium.com/@theclassytim/robotic-path-planning-rrt-and-rrt-212319121378.

Chin, Tim. "Robotic Path Planning: RRT and RRT*." *Medium*, Medium, 26 Feb. 2019, medium.com/@theclassytim/robotic-path-planning-rrt-and-rrt-212319121378.

DasSnipezDasSnipez. "Python, Pygame, How to Draw an Arc." *Stack Overflow*, 1 Nov. 1963, stackoverflow.com/questions/20850481/python-pygame-how-to-draw-an-arc.

"HexagonExample." *Pygame.org*, www.pygame.org/project/241.

Pbpf. "Pbpf/RRT-2." *GitHub*, github.com/pbpf/RRT-2.

Pygame :: Anaconda Cloud, anaconda.org/cogsci/pygame.

"PyGame Drawing Basics." *Pygame Drawing Basics*, sites.cs.ucsb.edu/~pconrad/cs5nm/topics/pygame/drawing/.

"PyGame Drawing Basics." *Pygame Drawing Basics*,

sites.cs.ucsb.edu/~pconrad/cs5nm/topics/pygame/drawing/.

Pygame.draw - Pygame v2.0.0.dev5 Documentation,

www.pygame.org/docs/ref/draw.html#pygame.draw.arc.

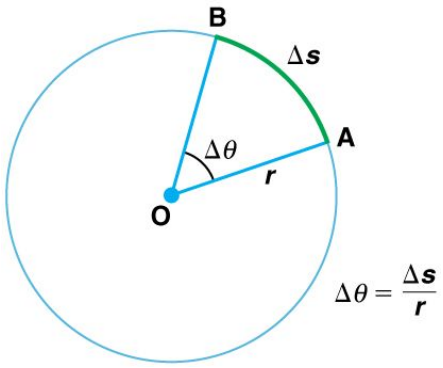
Pygame.gfxdraw - Pygame v2.0.0.dev5 Documentation,

www.pygame.org/docs/ref/gfxdraw.html.

"Python Mathematical Functions." *Programiz*,

www.pr**ogramiz.com/python-programming/modules/math.**

Arc Length Calculations



Given Values
- Start Point A [x_a, y_a, theta_a]
- Turn Angle d_theta
- End Angle theta_b
- Radius R

Find
- End Point B [x_b, y_b, theta_b]

Step 1: Find the end orientation. 0 rad points to the right, and PI/2 rad points up.
- For Right Turns: theta_b = theta_a - d_theta
- For Left Turns: theta_b = theta_a + d_theta

Step 2: Create a correction factor
c. Right Turns … $\phi = 0$
d. Left Turns … $\phi = PI/2$

Step 3: Find the center coordinates using the starting coordinates and the vector AO
- A_tangent = < cos(theta_a + ϕ), sin(theta_a + ϕ) >
- AO = < Rsin(theta_a + ϕ), Rcos(theta_a + ϕ) >
- Center = [x_a + Rsin(theta_a + ϕ), y_a + Rcos(theta_a + ϕ)]

Step 4: Find the end coordinate using the center coordinates and the vector OB
- B_tangent = < cos(theta_b + ϕ , sin(theta_b + ϕ) >
- OB = < -Rsin(theta_b + ϕ), -Rcos(theta_b+ ϕ) >
- Endpoint = [center_x - Rsin(theta_b + ϕ), center_y - Rcos(theta_b + ϕ)

Step 5: Get the start/end angles for pygame.draw.arc
- angleA = theta_a + d_theta + ϕ
- angleB = theta_b + d_theta + ϕ
- startAngle is the smaller angle, and endAngle is the larger

Step 6: Get the rectangle dimensions for pygame.draw.arc
- height = width = radius*2
- rec_x = center_x - radius
- rec_y = center_y - radius