COSC 76 Project 7: Motion Planning

Seungjae Jason Lee

Introduction

Objective

The main objective of this project is to implement robot motion planning through different algorithms. First, we aim to implment robot arm motion planning by using Probabilistic Roadmap(PRM). Then, we aim to implement car-like mobile robot with Rapidly Exploring Random Tree(RRT).

Game Rule

The rule for arm robot motion planning is that there are robot arms that have 1 joint, 2 joints, and 3 joints. By default, a robot's arm starts at (0,0). Then, the location of the rest the arm is determined by angles, measured counter-clockwise, at each joint and fixed lengths of arm components. Thus, our configuration is a vector that consists of angles of joints. The objective of the problem is to find a path from start configuration to end configuration using PRM while avoiding obstacles.

The rule for car-like mobile robot is that there is a robot that behaves like a car. It can only move forward, backward and turn sideway but not move directly sideway. The objective of the problem is to find a path from start configuration to end configuration using RRT while avoiding obstacles. The configuration is a vector with x coordinate, y coordinate, and theta, which decides where the front of the robot/car is.

Code Design/ Building the Model

For the first part, I had to build the whole model from the scratch, which involves implementing kinematics, obstacles collisions, and PRM. For the second part, our professor kindly provided some codes and thus, I only had to implement RRT.

PRM Class

First, I implemented PRM class in PRM.py. This class uses PRM algorithm to solve the robot arm motion planning. The following methods are implemented in the class:

1. `__init__(arm_lengths, obstacles)` : initializes the class by taking fixed length of robot arms and a list of list of points for obstacles(polygon).
2. `config_to_coord(self,config)` : takes a configuration, which is a vector of angles in degrees. Then, the function converts angles into a list of coordinates of joints of an arm.
3. `PRM_setup(self,n,k)` : takes n, which is a number of sample tries, and k, which is used for k-nearest neighor. This method builds vertices and edges of configurations and returns them. Detailed Implementation will be explained later.
4. `PRM_query(self, init_config, final_config, k, V, E)` : based on the given vertex and edge, it looks for a path to connect from init_config to final_config. To find a shortest path, I used astar_search from project 2 with a bit of modification.
5. `KNN(self,q,V,k)` : finds the k nearest neighbors. I used scikit-learn library to find k nearest neighbor, which counts as an extension and will be explained later.
6. `delta(q, neighbor_q)` : this method checks if two configurations are close enough to be treated as a continuous motion. Helper method used to build edge. I took euclidean distance and checked if it is less than 50. If so, returned True. Else, returned False.
7. `euclidean(self,v1,v2)` : this method measures the euclidean distance of two vectors and returns the distance.
8. `check_collision_free(self, q)` : checks if any of the arm is in collision with the obstacles (polygon). I used shapely's `disjoint` method to check if two objects are crossing each other.
9. `check_on_plane(self,q)` : returns False if any of the coordinate of robot is negative.

As mentioned before, I also modified astar_search and implemented it in the class. The following methods are for astar_search to find shortest path:

1. `get_successor(self, q, E)` : This method loops over edges and see if there are any edges that connects 'q'. If so, it returns a list of connected configurations.

2. `backchain(self,node)` : This method is a helper function for astar_search. Once astar_search reaches the goal state, backchain method finds the actual path by visiting parent nodes.

3. `astar_search(self, init,final,V,E)` : finds the shortest path from init to final with given vertices and edges.

In addition to these methods, I made solution class and AstarNode class to implement astar_search. For the demonstration, I made a video of it.

RRT Class

FOr RRT, I made my robot into a round disk with small radius. As for the space, I used 50*50 space for demonstration. The following are some of the methods that I implemented for the algorithm.

1. `__init__(self, delta_t, border, obstacles)` : saves the given input to the class. Also, it forms shapely geometry for boundaries and obstacles.
2. `Run_RRT(self, init,goal, k)` : This method actually finds a path using RRT. As the instruction does not request an optimal path, RRT traverses through the tree as it builds it. Once it finds a configuration close to the final goal, then it finds the path til that node and returns that path. If a path exists, it returns a list of vertices and a list of moves(indices) the robot made.
3. `child_config(self, q_node, move_index)` : finds and returns the configuration of the robot if it moves in the direction given by the input move_index. It is equivalent to successor methods that we implemented in previous projects.
4. `convert_list(self,vertex)` : for RRT, I stored vertices in a node so that I can keep track of the path. This method converts a list of nodes to a list of configurations/vectors.
5. `check_goal_state(self,q1,goal)` : checks if q1 is close enough to goal. Since the configuration for this problem has both displacement and angles, I decide to compare each one of them separately. For x and y coordinates, I said that they are close enough if they are 1 away from the actual goal state. For angle, I said it is close to the goal state angle if the difference is less than 0.6 radians, which is approximately 35 degrees.
6. `random_config(self)` : returns a randomly generated configuration. x,y coordinates are randomly generated within the bound and theta is randomly generated between 0 and 2*pi.
7. `check_collision(self, q)` : checks collision for both obstacles and walls(boundaries). Uses shapely to check if objects collide and returns True if collision occurs.
8. `nearest_vertex(self,q,V)` : finds the nearest neighbor by using scikit-learn KNN with k = 1. Returns the closest vertex as a node.

The two pictures are the result of running rrt.py. We see that the path slightly differs each time.

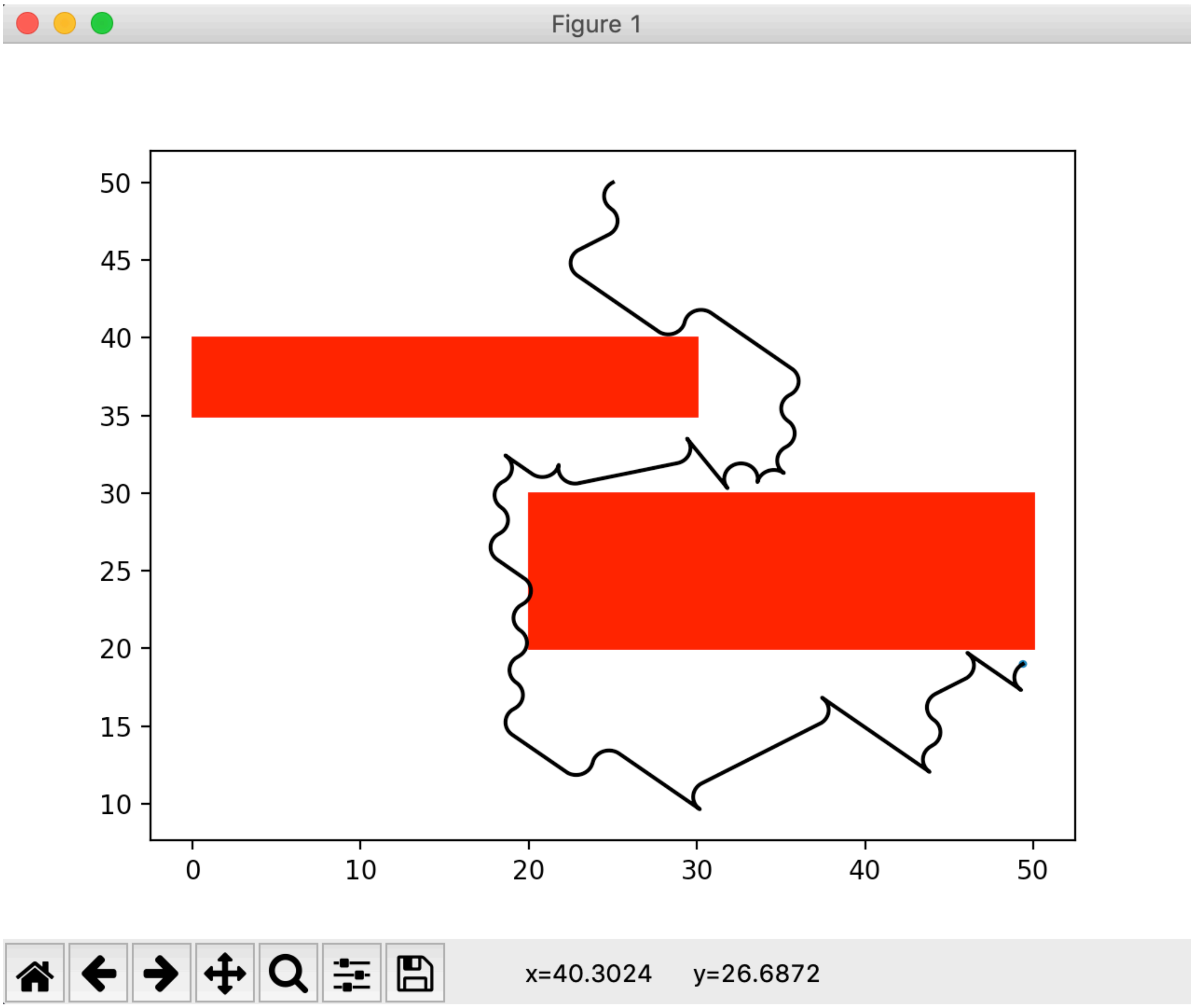
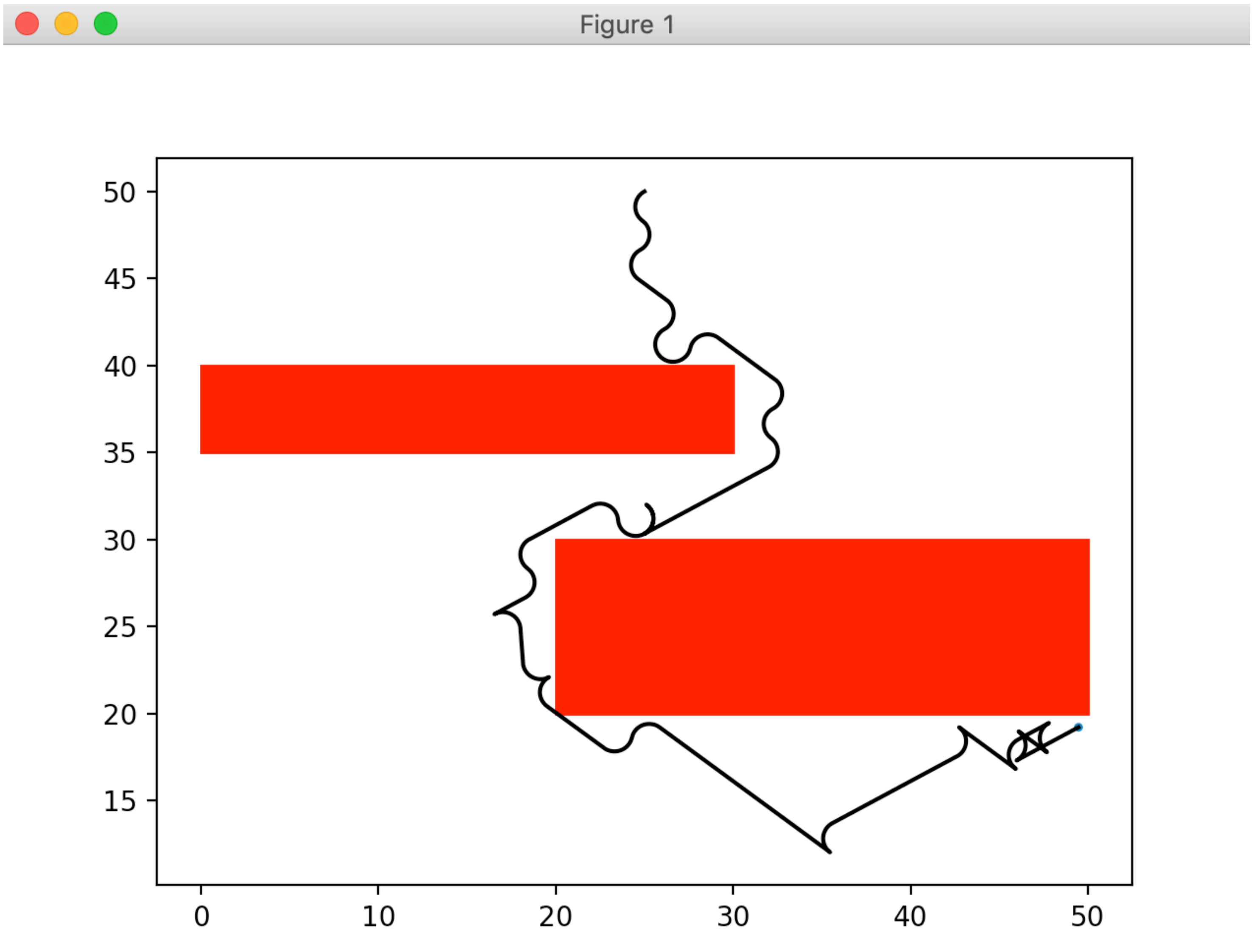


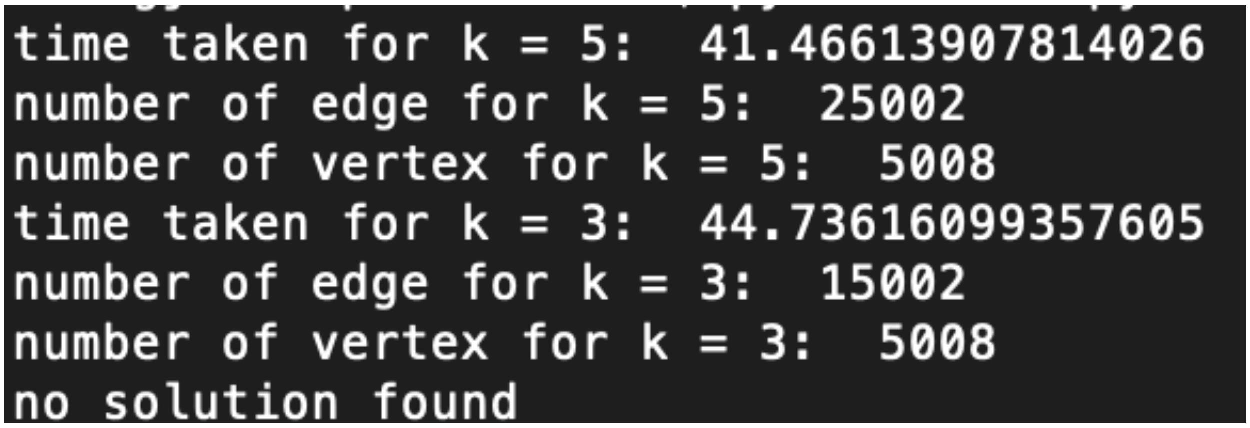
Figure 1
50
45
40
35
30
25
20
15
10
0
10
20
30
40
50
x=40.3024 y=26.6872



Analysis

Different K for PRM

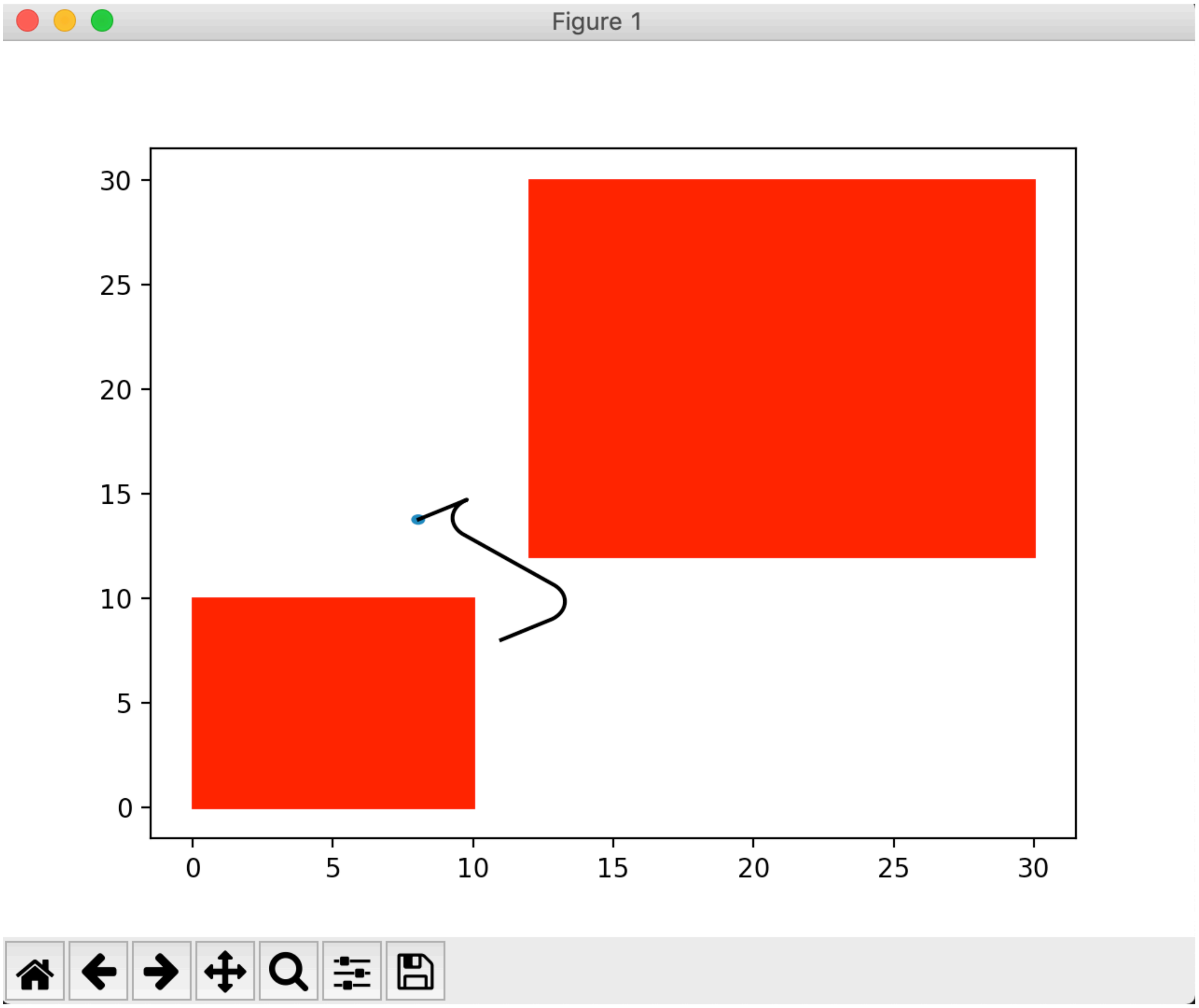
As mentioned in the instruction page, I tried PRM with k = 3 and k = 5 and compared the result. A picture of the result is shown below. We see that there weren't much difference in time. However, we observed a significant difference in number of edges as k = 3 only observes 3 neighbors while k = 5 observes 5 neighbors. Not only that there is a significant difference in amount, the ratio also matches 3:5 for number of edges, which makes sense. Also, since k = 3 has less edges, it could not actually find a solution to the problem. Thus, through this analysis, I observed that if the algorithm cannot find a path, trying higher k could be one of the solution to the problem.



```
time taken for k = 5:  41.46613907814026
number of edge for k = 5:  25002
number of vertex for k = 5:  5008
time taken for k = 3:  44.73616099357605
number of edge for k = 3:  15002
number of vertex for k = 3:  5008
no solution found
```

Testing different durations for RRT

For this analysis, I wanted to check what delta_t(duration) does to the robot planning motion. So, I made a map where there is a small gap for the car to pass by. Then, I tried to run RRT with delta_t = 2 and delta_t = 5. The first did find an answer that I was expecting while the second one could not find an answer. Thus, I concluded that for small narrow and complicated path, it is better to use small delta_t. However, I would make sure that I am not using too small value for delta_t because if we use too small value for a big grid system, the tree may not never reach goal within assigned iteration limit. A picture of the path from delta_t = 2 is shown below.



Extension

Extension 1: For the extension, I used scikit-learn library to implement K- nearest neighbors. It is shown in function called `KNN` for PRM and `nearest vertex` for RRT. I used NearestNeighbors function, which I found from the document. As for the algorithm, I used "ball_tree" option. Based on the research, ball tree is a data structure for high dimensional vectors. So, the structure makes bunch of clusters for each point based on its location. Then, once we try to find closet neighbor, isntead of looking at every single point, we classify our new point into one of the balls/clusters and compare the new point with points within the cluster. Using this algorithm, I was able to observe faster run time.

Extension 2: For the second extension, I came up with a bit modified nearest_vertex method for RRT. Since we are using radians, I noticed that 0.3, which may be small for x,y coordinates of size 50, it is big for radians, which is about 17 degrees. So, to balance out the weight of these numbers, I normalized x,y coordinates by the boundary condition defined by user. So, if the width and height are 50, I divided them by 50. As for radians, I divided by 2*pi to normalize it. Then, I ran KNN to find a nearest neighbor. The code is shown in `nearest_vertex_extension`.

Extension 3: For robot arm problem, we did not require an animation. However, I decided to make one for easier understanding of the algorithm and also for debugging. Under robot*arm*display, I made a class called RobotDisplay, which animates the movement of the arm based on the sequence/path from running PRM. Demonstration is shown in a video attached.