ROB521 Lab 2 Report
Planning and Navigation

Feb.27, 2024

PRA0104 Group 4

Jingzhou Liu (1005906405)
Aoran Jiao (1006075373)
Yicheng Zhang (1005680568)

1. Part A: Simulation

The objectives of this section are to implement RRT and RRT*, then execute both of these algorithms on the willow garage map in simulation. The outputs of RRT and RRT* are demonstrated in Figure 1 below, where the found paths are highlighted in green.

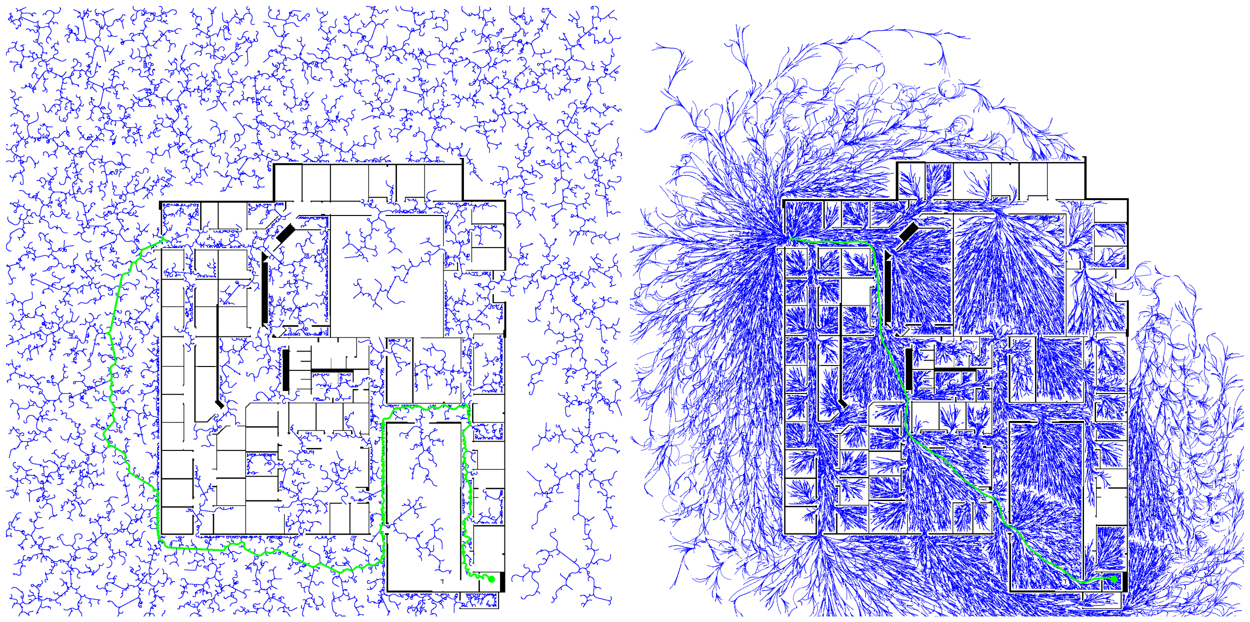

Figure 1: A discovered path of RRT (*left*) and a discovered optimal path of RRT* (*right*) are shown in green and the resulting trees are shown in blue.

1.1 Task 1: Collision Detection

Given N 2D points representing the positions of the robot, the objective is to return a collision flag array of size N. This is accomplished by first determining all of the occupied pixels in the provided occupancy map via the *points_to_robot_circle* function, then checking which pixels have an intensity value of 0, which indicates the presence of an obstacle. Note that for each 2D position, *points_to_robot_circle* returns its surrounding pixel locations within a given radius, representing the size of the robot.

1.2 Task 2: Simulate Trajectory

The *simulate_trajectory* function simulates driving a non-holonomic robot towards a goal 2D point location given a 2D robot pose (x, y, θ). The function utilizes *robot_controller* and returns a collision-free trajectory consisting of various waypoint poses.

robot_controller defines a collection of candidate linear and angular velocities and rolls out a trajectory for *timestep* number of seconds for each of these candidate velocities via the function *trajectory_rollout*. *robot_controller* then selects the linear and angular velocity combination that yields the closest position to the goal position.

trajectory_rollout takes a list of velocity candidates and the current robot pose and forward simulates for *timestep* number of seconds in accordance with the following robot kinematics:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

where v and ω are the linear and angular velocities. Each point in the trajectories are then checked for collision, with only the collision-free part of a trajectory is saved.

1.3 Task 3: RRT Planning Algorithm

The RRT algorithm follows the following steps:

1. Sample a point using *sample_map_space*
2. Find the closest node in the existing tree using *closest_node*
3. Simulate driving towards the point from the closest node using *simulate_trajectory*, obtaining a new node
4. Ignore the new node if it is already in the tree using *check_if_duplicate*
5. Add the new node to the tree
6. Return if the new node is close to the goal location, otherwise repeat Steps 1-5

1.3.1 Sampling Map Space

The function *sample_map_space* implements biased sampling, where there is a 3% chance the point is sampled near the goal location. Otherwise, the point is sampled uniformly within the map.

1.3.2 Finding the Closest Node

The function *closest_node* returns the closest node to a provided location by first constructing a *KDTree* using SciPy with all existing nodes and use *KDTree.query* to find the closest node. We choose to use *KDTree* due to its superior runtime complexity of *O(n logn).*

1.3.3 Duplicate Node Checking

The function *check_if_duplicate* checks if a given node is already in the tree by comparing the robot poses described by each node.

1.4 Task 4-7: RRT* Planning Algorithm

The RRT* algorithm follows the following steps:

1. Sample a point using *sample_map_space*
2. Find the closest node in the existing tree using *closest_node*
3. Simulate driving towards the point from the closest node using *simulate_trajectory*, obtaining a new node
4. Ignore the new node if it is already in the tree using *check_if_duplicate*
5. Compute the cost-to-come of the new node via invoking the *cost_to_come* function
6. Add this new node to the tree

7. Query the *KDTree* for all nodes within a *ball_radius* from the new node location using *KDTree.query_ball_point*
8. Find the node with the lowest cost within the *ball_radius* and rewire to the new node if the cost is lower than the existing connection from the closest node
    a. Attempt to connect each candidate node within the *ball_radius* to the new node via *connect_node_to_point*
    b. Compute the cost-to-come of this new connection for each candidate node
    c. Select the candidate node with the lowest cost-to-come
    d. Rewire the best candidate node to the new node if its cost is lower than that of the closest node
9. For all candidate nodes within the *ball_radius*, attempt connecting the new node to each of them. Rewire the connection to each of the candidate nodes from the new node if the trajectorhy from the new node yields a lower cost-to-come. Update the cost-to-come of the children nodes of any candidate nodes that are rewired via *update_children*.
10. If the new node is close to the goal location, add it to a list of potential goal nodes
11. Determine the goal node with the lowest cost-to-come.
12. Repeat Steps 1-11 until a sufficiently optimal path is found.

1.4.1 Sampling Map Space in RRT*

We modify the function *sample_map_space* used in RRT to implement Informed RRT*. The function behaves as described in Section 1.3.1 before a possible trajectory is found. Once a trajectory is discovered, it samples within an ellipse where the dimensions of the ellipse are dependent on the arc length of the shortest discovered trajectory, in accordance with the Informed RRT* algorithm.

1.4.2 Computing the Cost of a Trajectory

The *cost_to_come* funcion computes the cost given a trajectory consisting of a series of robot poses. The cost is defined as *position_cost + 0.1* orientaion_cost*, where *position_cost* is the arc length of the trajectory and *orientation_cost* calculates the absolute rotational difference in radiance between robot poses along the trajectory.

1.4.3 Connect Node to Point

The function *connect_node_to_point* computes a trajectory that starts from a given robot pose and arrives at a goal point. It accomplishes this by finding the circle that passes through both starting and goal position while having an orientation of 0 in the frame of the starting robot pose. It then computes the trajectory along this circle that goes from the starting position to the goal position and transforms it back to the world frame. Collision checking is also performed along this trajectory, with only collision-free trajectories returned.

1.4.4 Updating Children Nodes' Cost

The function *update_children* updates the cost-to-come of all connected nodes to a given node. It accomplishes this by adding the edge cost to its own cost for its children nodes and then recursively calling *update_children* on its children nodes.

1.5 Task 8: Using Trajectory Rollout for Local Planning

The local planner sends robot velocity commands in order to follow a trajectory consisting of a series of waypoint poses. It follows the following steps:

1. Generates a collection of candidate linear and angular velocities combinations
2. For all velocity combinations, use *trajectory_rollout* to forward simulate *control_horizon* number of seconds, where *control_horizon* is calculated as dividing the distance to the next waypoint by the maximum linear velocity of the robot.
3. Remove all candidate trajectories that have collisions.
4. If all candidate trajecotires have collisions, make the robot move backwards with a linear velocity of -0.1s.
5. Otherwise, select the linear and angular velocity combination that yields the simulated trajectory with the closest final distance to the next waypoint.
6. Repeat Steps 1-5 until the final waypoint is reached.

A video of a simulated robot following the optimal path discovered by RRT* shown in Figure 1 (*right*) is attached.

2. Part B: Real Environment Deployment

Figure 2 exhibits the optimal trajectory found by RRT* on the Myhal map. A video of a simulated robot following said trajectory is attached. In addition, a video of an open-loop rollout of said trajectory deployed on the real robot is also attached.

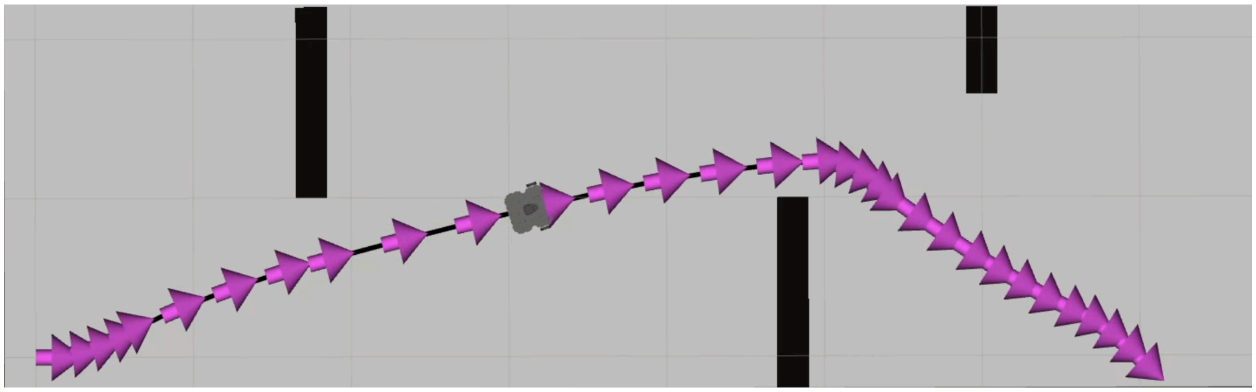

Figure 2: A discovered optimal path of RRT* on the Myhal map used in real-world deployment.

Appendices

Appendix A: RRT and RRT* Implementation

```
#!/usr/bin/env python3
#Standard Libraries
import numpy as np
import yaml
import pygame
import time
import pygame_utils
import matplotlib.image as mpimg
from skimage.draw import disk
from scipy.linalg import block_diag
from scipy.spatial import cKDTree
from scipy.linalg import inv


def load_map(filename):
   im = mpimg.imread("../maps/" + filename)
   if len(im.shape) > 2:
       im = im[:,:,0]
   im_np = np.array(im)  #Whitespace is true, black is false
   #im_np = np.logical_not(im_np)
   return im_np


def load_map_yaml(filename):
   with open("../maps/" + filename, "r") as stream:
           map_settings_dict = yaml.safe_load(stream)
   return map_settings_dict

#Node for building a graph
class Node:
   def __init__(self, pose, parent_id, cost):
       self.pose = pose # [x, y, theta] of shape (3,)
       self.parent_id = parent_id # The parent node id that leads to this node (There should only
ever be one parent in RRT)
       self.cost = cost # The cost to come to this node
       self.children_ids = [] # The children node ids of this node
       self.pose_traj_to_children = dict() # Maps node_id of children to a a trajectory array of
shape (N, 3)
       return

#Path Planner
class PathPlanner:
   #A path planner capable of perfomring RRT and RRT*
   def __init__(self, map_filename, map_setings_filename, goal_point, stopping_dist):
       #Get map information
       self.occupancy_map = load_map(map_filename)
       self.map_shape = self.occupancy_map.shape
       self.map_settings_dict = load_map_yaml(map_setings_filename)

       #Get the metric bounds of the map
       # [x1 x2]
       # [y1 y2]
```

```
        self.bounds = np.zeros([2,2]) #m
        self.bounds[0, 0] = self.map_settings_dict["origin"][0]
        self.bounds[1, 0] = self.map_settings_dict["origin"][1]
        self.bounds[0, 1] = self.map_settings_dict["origin"][0] + self.map_shape[1] *
self.map_settings_dict["resolution"]
        self.bounds[1, 1] = self.map_settings_dict["origin"][1] + self.map_shape[0] *
self.map_settings_dict["resolution"]
        # [x1 y1]
        # [x2 y2]
        self.bounds = self.bounds.T

        #Robot information
        self.robot_radius = 0.22 #m

        # self.vel_max = 0.26 #m/s (Feel free to change!)
        # self.rot_vel_max = 1.82 #0.2 #rad/s (Feel free to change!)

        self.vel_max = 0.26 #m/s (Feel free to change!)
        self.rot_vel_max = 1.82 #0.2 #rad/s (Feel free to change!)

        #Goal Parameters
        self.goal_point = goal_point #m
        self.stopping_dist = stopping_dist #m
        self.best_goal_node_id = -1
        self.goal_node_ids = list()
        self.best_goal_pose_traj = np.empty((0, 3))
        self.best_goal_pose_path = list()

        #Trajectory Simulation Parameters
        self.timestep = 3.0 #s
        self.num_substeps = 20 #10

        #Trajectory Rollout Options
        vel_grid_res = 7 #7 #4
        rot_vel_res = 9 #7
        num_vel = vel_grid_res*rot_vel_res

        vel_range = np.linspace(-self.vel_max, self.vel_max, vel_grid_res)
        rot_vel_range = np.linspace(-self.rot_vel_max, self.rot_vel_max, rot_vel_res)
        vel_grid, rot_vel_grid = np.meshgrid(vel_range, rot_vel_range, indexing='ij')
        # (num_vel, 1)
        vel_candidates = vel_grid.flatten().reshape(-1, 1)
        rot_vel_candidates = rot_vel_grid.flatten().reshape(-1, 1)

        self.vel_options = np.hstack([vel_candidates, rot_vel_candidates])

        # if there is a [0, 0] option, remove it
        all_zeros_index = (np.abs(self.vel_options) < [0.001, 0.001]).all(axis=1).nonzero()[0]
        if all_zeros_index.size > 0:
            self.vel_options = np.delete(self.vel_options, all_zeros_index, axis=0)

        #Planning storage
        self.nodes = [Node(np.zeros((3,)), -1, 0)]

        #Sampling Parameters
        self.sampling_space_params = dict()
        self.sampling_space_params["goal_range_multiplier"] = 4

        #RRT* Specific Parameters
        self.lebesgue_free = np.sum(self.occupancy_map) * self.map_settings_dict["resolution"] **2
        self.zeta_d = np.pi
        self.gamma_RRT_star = 2 * (1 + 1/2) ** (1/2) * (self.lebesgue_free / self.zeta_d) ** (1/2)
        self.gamma_RRT = self.gamma_RRT_star + .1
        self.epsilon = 2.5

        #Pygame window for visualization
        # self.window = pygame_utils.PygameWindow(
        #     "Path Planner", (2500, 2500), self.occupancy_map.shape, self.map_settings_dict,
self.goal_point, self.stopping_dist)
        real_map_size_pixels = [self.occupancy_map.shape[1], self.occupancy_map.shape[0]]
        self.window = pygame_utils.PygameWindow(
            "Path Planner", (1900, 1900), real_map_size_pixels, self.map_settings_dict,
self.goal_point, self.stopping_dist)
        return



    #Functions required for RRT
    def sample_map_space(self, visualize=0):
        #Return an [x,y] coordinate to drive the robot towards

        sample_goal_range_prob = np.random.rand()
        if sample_goal_range_prob < 0.03: # 0.06
            # samples the goal position
            sample_goal_prob = np.random.rand()
            if sample_goal_prob < 0.5:
                sample = self.goal_point
            else:
                k = self.sampling_space_params["goal_range_multiplier"]
                delta = k*self.stopping_dist * np.random.randn(2,)
                sample = self.goal_point + delta
        else:
            if self.best_goal_node_id == -1:
                # a path has not been found yet
                sample = (self.bounds[1] - self.bounds[0])*np.random.rand(2) + self.bounds[0]
                sample = sample.reshape(2,)
            else:
                # perform informed RRT* sampling
                a = self.sampling_space_params["a"]
                b = self.sampling_space_params["b"]
                center = self.sampling_space_params["center"]
                ellipse_theta = self.sampling_space_params["ellipse_theta"]

                # sample a random point in the ellipse's frame
                # frame origin: ellipse center | orientation: C_3(ellipse_theta)
                angle = np.random.uniform(0, 2*np.pi)
                r = np.sqrt(np.random.uniform(0, 1))
                x = a*r * np.cos(angle)
                y = b*r * np.sin(angle)
                sample_ellipse = np.array([x, y])
                c, s = np.cos(ellipse_theta), np.sin(ellipse_theta)
                w_R_e = np.array([
                    [c, -s],
                    [s,  c],
                ])
                sample = (w_R_e @ sample_ellipse.reshape(2, 1)).reshape(2,) + center
                # self.window.add_point(sample, radius=3, color=(255, 0, 0), update=True)

        if visualize != 0:
            self.window.add_point(sample, radius=3, color=(255, 0, 0), update=True)

        return sample


    def check_if_duplicate(self, pose):
        #Check if point is a duplicate of an already existing node
        # pose: (3,)

        # constructing a KDTree with the full pose (x, y, theta)
        nodes_KDTree = cKDTree([node.pose for node in self.nodes])
        distance, node_idx = nodes_KDTree.query(pose)
        if np.isclose(distance, 0.0):
            return True
        else:
            return False


    def closest_nodes(self, point, k=1):
        #Returns the index of the closest node
        # point: (2,)

        # re-construct the KDTree:
        nodes_KDTree = cKDTree([node.pose[0:2] for node in self.nodes])
        distance, node_idx = nodes_KDTree.query(point, k)
        return node_idx


    def collision_check(self, points):
        # expects point to be of shape (num_pts, 2)
        # returns a boolean array of shape (num_pts, ), where True means collision

        # (num_pts, num_pts_per_circle, 2)
        occupied_coords = self.points_to_robot_circle(points)
        # occupancy_map = 0 means it is in collision (num_pts, num_pts_per_circle)
        collision_mask_per_point = self.occupancy_map[occupied_coords[:, :, 1], occupied_coords[:,
:, 0]] == 0
        # (num_pts, )
        collision_mask = np.any(collision_mask_per_point, axis=1)
        return collision_mask


    def simulate_trajectory(self, node_i, point_s, visualize=0, frequent_viz_update=True):
        #Simulates the non-holonomic motion of the robot.
        #This function drives the robot from node_i towards point_s. This function does has many
```

```
solutions!
       #node_i is a 3 by 1 vector [x;y;theta] this can be used to construct the SE(2) matrix
T_{OI} in course notation
       #point_s is the sampled point vector [x; y]

       # Return: (N, 3), where N can change depending on the length of the trajectory
       vel, rot_vel, robot_traj = self.robot_controller(node_i, point_s, visualize,
frequent_viz_update)
       return robot_traj


   def robot_controller(self, node_i, point_s, visualize=0, frequent_viz_update=True):
       #This controller determines the velocities that will nominally move the robot from node i
to node s
       #Max velocities should be enforced

       # Idea: perform trajectory rollout on various velocities, select the point closest to
point_s
       point_s = point_s.reshape(1, 2)

       vel_candidates = self.vel_options[:, 0]
       rot_vel_candidates = self.vel_options[:, 1]

       # rollout the trajectories given the velocities
       pose_traj, last_valid_substep, last_valid_poses = self.trajectory_rollout(
           vel_candidates, rot_vel_candidates, node_i.pose, self.timestep, self.num_substeps
       )

       # measure each trajectory's final position error
       last_position_error = np.linalg.norm(last_valid_poses[:, 0:2] - point_s, axis=1)
       best_vel_idx = np.argmin(last_position_error)
       best_vel = vel_candidates[best_vel_idx]
       best_rot_vel = rot_vel_candidates[best_vel_idx]
       best_traj = pose_traj[best_vel_idx, 0:last_valid_substep[best_vel_idx]+1, :]

       if visualize != 0:
           if visualize == 2:
               # draw all paths
               for i in range(pose_traj.shape[0]):
                   for j in range(pose_traj.shape[1]):
                       if not np.all(np.isnan(pose_traj[i, j]) == np.isnan(np.array([np.nan,
np.nan, np.nan]))):
                           self.window.add_point(pose_traj[i, j, 0:2], radius=1, color=(255, 0,
0), update=False)
               # draw all end points
               for i in range(last_valid_poses.shape[0]):
                   self.window.add_point(last_valid_poses[i, 0:2], radius=1, color=(0, 255, 0),
update=False)
           # draw best path
           for i in range(best_traj.shape[0]):
               self.window.add_point(best_traj[i, 0:2], radius=1, color=(0, 0, 255),
update=False)
           if frequent_viz_update:
               self.window.update()
```

```
        return best_vel, best_rot_vel, best_traj


    def trajectory_rollout(self, vel, rot_vel, pose, timestep, num_substeps):
        # Given your chosen velocities determine the trajectory of the robot for your given
timestep
        # The returned trajectory should be a series of points to check for collisions

        # Args:
        # vel: (N, 1)
        # rot_vel: (N, 1)
        # pose: (3, )

        # Returns:
        # an array of shape (num_vel, num_substeps, 3), where points post collision are filled
with np.nan
        # an array of shape (num_vel,) that indicates the last collision-free substep idx
        # an array of shape (num_vel, 3) that represents the last collision-free pose

        # kinematics closed-form solution:
        # x(t) = x_0 + (v/w)*[sin(wt+theta_0) - sin(theta_0)]
        # y(t) = y_0 - (v/w)*[cos(wt+theta_0) - cos(theta_0)]
        # theta(t) = theta_0 + w*t

        num_vel = vel.shape[0]
        x_0, y_0, theta_0 = pose.reshape(3,)

        # (num_vel, 1)
        vel = vel.reshape(-1, 1)
        rot_vel = rot_vel.reshape(-1, 1)

        # (num_substeps, )
        t = np.linspace(0, timestep, num_substeps)

        # simulate trajectories for all timesteps (num_vel, num_substeps)
        x_traj = np.where(
            np.isclose(rot_vel, 0),
            x_0 + vel*t*np.cos(theta_0),
            x_0 + (vel/rot_vel)*(np.sin(rot_vel*t + theta_0) - np.sin(theta_0)),
        )
        y_traj = np.where(
            np.isclose(rot_vel, 0),
            y_0 + vel*t*np.sin(theta_0),
            y_0 - (vel/rot_vel)*(np.cos(rot_vel*t + theta_0) - np.cos(theta_0)),
        )
        theta_traj = theta_0 + rot_vel*t

        # normalize thetas to be between -pi and pi
        theta_traj = (theta_traj + np.pi) % (2 * np.pi) - np.pi

        # (num_vel, num_substeps, 3)
        pose_traj = np.dstack((x_traj, y_traj, theta_traj))

        # (num_vel, num_substeps)
        collision_mask = self.collision_check(pose_traj.reshape(-1, 3)[:, 0:2]).reshape(-1,
num_substeps)
        # set to True for all substeps after the first True (num_vel, num_substeps)
        collision_mask_corrected = np.cumsum(collision_mask, axis=1).astype(bool)
        # (num_vel, num_substeps, 3)
        pose_traj[collision_mask_corrected] = np.array([np.nan, np.nan, np.nan])

        # first substep that encounters a collision
        first_collision_indices = np.argmax(collision_mask_corrected, axis=1)
        # rows that do not have collisions
        no_collision_indices = np.all(~collision_mask_corrected, axis=1)
        # for no collision rows, return the last substep idx
        # for rows with collisions, return the last valid idx
        last_valid_substep = np.where(
            no_collision_indices,
            num_substeps - 1,
            first_collision_indices - 1
        )

        last_valid_poses = pose_traj[np.arange(num_vel), last_valid_substep, :]
        return pose_traj, last_valid_substep, last_valid_poses


    def point_to_cell(self, point):
        #Convert a series of [x,y] points in the map to the indices for the corresponding cell in
the occupancy map
        #point is a N by 2 matrix of points of interest
        point = point.copy()
        # origin vector is from map to world coord
        origin_x, origin_y, theta = self.map_settings_dict["origin"]
        # we want world to map coord, so we subtract origin vector first
        point += -np.array([[origin_x, origin_y]])
        # to convert from map coord to map pixel coord, we divide by resolution
        point = point / self.map_settings_dict["resolution"]
        # so far, y = 0 is at the bottom, but in an image y = 0 is at the top
        point[:, 1] = self.map_shape[0] - point[:, 1]
        # round to integer for pixel coordinate (N, 2)
        point = point.astype(int)
        return point


    def points_to_robot_circle(self, points):
        #Convert a series of [x,y] points to robot map footprints for collision detection
        #Hint: The disk function is included to help you with this function
        # points (N, 2)

        # (num_points, 2)
        pixel_coords = self.point_to_cell(points)
        r_px = self.robot_radius / self.map_settings_dict["resolution"]

        rr, cc = disk((0, 0), r_px)
        # A list of pixel coords that draw out a cirlce at the origin (num_px_per_circle, 2)
        circle_px_coords = np.array([rr, cc]).T
        # (num_points, num_px_per_circle, 2)
        circle_points = pixel_coords[:, np.newaxis, :] + circle_px_coords
        # clip any pixel coords outside of the range to within the range
        circle_points = np.clip(circle_points, a_min=[0, 0], a_max=[self.map_shape[1]-1, 
self.map_shape[0]-1])
        return circle_points


    #RRT* specific functions
    def ball_radius(self):
        #Close neighbor distance
        card_V = len(self.nodes)
        return min(self.gamma_RRT * (np.log(card_V) / card_V ) ** (1.0/2.0), self.epsilon)


    def connect_node_to_point(self, node_i, point_f, visualize=0):
        #Given two nodes find the non-holonomic path that connects them
        #Settings
        #node is a 3 by 1 node
        #point is a 2 by 1 point

        pose_i = node_i.pose.reshape(3,)
        x_w_0 = pose_i[0]
        y_w_0 = pose_i[1]
        theta_0 = pose_i[2]

        point_f = point_f.reshape(2,)
        x_w_f = point_f[0]
        y_w_f = point_f[1]

        # find point_f in robot frame
        w_T_r = np.array([
            [np.cos(pose_i[2]), -np.sin(pose_i[2]), pose_i[0]],
            [np.sin(pose_i[2]),  np.cos(pose_i[2]), pose_i[1]],
            [0                ,  0                ,  1.0     ],
        ])
        r_T_w = inv(w_T_r)
        point_f_robot_frame = r_T_w @ np.array([point_f[0], point_f[1], 1.0]).T
        x_r_f = point_f_robot_frame[0]
        y_r_f = point_f_robot_frame[1]

        if np.isclose(y_r_f, 0.0):
            # drive straight since no lateral deviation in robot's frame
            # substeps = np.ceil(d / self.robot_radius).astype(np.int64) + 1
            pose_s = np.array([x_w_f, y_w_f, theta_0])
            pose_traj = np.linspace(pose_i, pose_s, self.num_substeps)[None, ...]
        else:
            # in robot's frame, define a circle that passes through
            # the robot's current position (0, 0) to the point_f in
            # robot's frame. The circle must also have a slope of 0
            # at (0, 0) to match the robot's current heading, causing
            # the circle's center to lie on the y-axis of robot's frame
            traj_radius = (x_r_f**2 + y_r_f**2) / (2*y_r_f)
            traj_center_robot_frame = np.array([0.0, traj_radius, 1]).T
            traj_center = (w_T_r @ traj_center_robot_frame).reshape(3,)[0:2]
            # self.window.add_point(traj_center, radius=3)

            # calculate the linear and angular velocities to achieve traj_radius
            rot_vel = self.rot_vel_max
            vel = traj_radius * rot_vel

            if vel > self.vel_max:
                vel = self.vel_max
                rot_vel = vel / traj_radius

            # calculate time step assuming maximum angular velocity
            theta_f = np.arctan2(
                (x_w_f - x_w_0) / traj_radius + np.sin(theta_0),
                -(y_w_f - y_w_0) / traj_radius + np.cos(theta_0),
            )
            angle_distance = (theta_f - theta_0 + np.pi) % (2*np.pi) - np.pi
            timestep = (angle_distance) / rot_vel

            if timestep < 0:
                timestep *= -1
                rot_vel *= -1
                vel *= -1

            # (num_substeps, )
            t = np.linspace(0, timestep, self.num_substeps)

            # simulate trajectories for all timesteps (num_vel, num_substeps)
            x_traj = np.where(
                np.isclose(rot_vel, 0),
                x_w_0 + vel*t*np.cos(theta_0),
                x_w_0 + (vel/rot_vel)*(np.sin(rot_vel*t + theta_0) - np.sin(theta_0)),
            )
            y_traj = np.where(
                np.isclose(rot_vel, 0),
                y_w_0 + vel*t*np.sin(theta_0),
                y_w_0 - (vel/rot_vel)*(np.cos(rot_vel*t + theta_0) - np.cos(theta_0)),
            )
            theta_traj = theta_0 + rot_vel*t

            # normalize thetas to be between -pi and pi
            theta_traj = (theta_traj + np.pi) % (2 * np.pi) - np.pi

            # (1, num_substeps, 3)
            pose_traj = np.dstack((x_traj, y_traj, theta_traj))


        # (1, num_substeps)
        collision_mask = self.collision_check(pose_traj.reshape(-1, 3)[:, 0:2]).reshape(-1,
self.num_substeps)
        # set to True for all substeps after the first True (1, num_substeps)
        collision_mask_corrected = np.cumsum(collision_mask, axis=1).astype(bool)
        # (1, num_substeps, 3)
        pose_traj[collision_mask_corrected] = np.array([np.nan, np.nan, np.nan])

        # first substep that encounters a collision
        first_collision_indices = np.argmax(collision_mask_corrected, axis=1)
        # rows that do not have collisions
        no_collision_indices = np.all(~collision_mask_corrected, axis=1)
        # for no collision rows, return the last substep idx
        # for rows with collisions, return the last valid idx
        last_valid_substep = np.where(
            no_collision_indices,
            self.num_substeps - 1,
            first_collision_indices - 1
        )

        last_valid_poses = pose_traj[np.arange(1), last_valid_substep, :]

        if visualize != 0:
            for i in range(pose_traj[0].shape[0]):
                self.window.add_se2_pose(pose_traj[0, i], color=(0, 0, 255), length=10)

        return pose_traj[0], last_valid_substep[0], last_valid_poses[0]


    def cost_to_come(self, trajectory_o):
        #The cost to get to a node from lavalle
        # trajectory_o: (N, 3)
        trans_cost_to_come = np.linalg.norm(trajectory_o[1:, 0:2] - trajectory_o[0:-1, 0:2],
axis=1).sum()

        abs_angle_diff = np.abs(trajectory_o[1:, 2] - trajectory_o[0:-1, 2])
        rot_dist_error = np.minimum(np.pi * 2 - abs_angle_diff, abs_angle_diff)
        rot_cost_to_come = np.abs(rot_dist_error).sum() * self.robot_radius

        return trans_cost_to_come + rot_cost_to_come * 0.1


    def update_children(self, node_id):
        #Given a node_id with a changed cost, update all connected nodes with the new cost
        parent_node = self.nodes[node_id]
        for child_id in parent_node.children_ids:
            # (N, 3)
            pose_traj_to_children = parent_node.pose_traj_to_children[child_id]
            edge_cost = self.cost_to_come(pose_traj_to_children)
            self.nodes[child_id].cost = parent_node.cost + edge_cost
            self.update_children(child_id)


    #Planner Functions
    def rrt_planning(self, save_path=True):
        #This function performs RRT on the given map and robot
        #You do not need to demonstrate this function to the TAs, but it is left in for you to
check your work
        while True:
            # Sample map space
            point = self.sample_map_space(visualize=0)

            # Get the closest point
            closest_node_id = self.closest_nodes(point, k=1)

            # Simulate driving the robot towards the closest point
            pose_traj = self.simulate_trajectory(self.nodes[closest_node_id], point, visualize=1)
            # Check if the new node pose is a duplicate and skip if it is
            new_node_pose = pose_traj[-1, :]
            if self.check_if_duplicate(new_node_pose):
                continue

            # Add the last pose in the trajectory as a Node and update closest node's children
            new_node = Node(
                pose=new_node_pose,
                parent_id=closest_node_id,
                cost=0,
            )
            self.nodes.append(new_node)
            new_node_id = len(self.nodes) - 1
            self.nodes[closest_node_id].children_ids.append(new_node_id)
            self.nodes[closest_node_id].pose_traj_to_children[new_node_id] = pose_traj

            # Check if goal has been reached
            if np.linalg.norm(new_node_pose[0:2] - self.goal_point) <= self.stopping_dist:
                self.best_goal_node_id = new_node_id
                self.best_goal_pose_path, self.best_goal_pose_traj = self.recover_path(
                    self.best_goal_node_id, visualize=1
                )
                if save_path:
                    np.save("rrt_path.npy", np.array(self.best_goal_pose_path))
                break

        return self.nodes


    def rrt_star_planning(self, visualize=1, frequent_viz_update=False, max_iteration=40000,
save_path=True):
        #This function performs RRT* for the given map and robot
        iteration = 0
        while True:
            iteration += 1
            # print(iteration)

            # Sample map space
            point = self.sample_map_space(visualize=0)

            # Get the closest point
            closest_node_id = self.closest_nodes(point, k=1)

            # Simulate driving the robot from the closest point to the sampled point
            pose_traj_from_closest_node = self.simulate_trajectory(
                self.nodes[closest_node_id], point, visualize=visualize,
frequent_viz_update=frequent_viz_update
            )

            # Check if the new node pose is a duplicate and skip if it is
            new_node_pose = pose_traj_from_closest_node[-1, :]
            if self.check_if_duplicate(new_node_pose):
                continue
            # Compute the cost-to-come of the new node
            new_node_cost_to_come = self.nodes[closest_node_id].cost +
self.cost_to_come(pose_traj_from_closest_node)

            # Add the last pose in the trajectory as a Node and update closest node's children
            new_node = Node(
                pose=new_node_pose,
                parent_id=closest_node_id,
                cost=new_node_cost_to_come,
            )
            self.nodes.append(new_node)
            new_node_id = len(self.nodes) - 1
            self.nodes[closest_node_id].children_ids.append(new_node_id)
            self.nodes[closest_node_id].pose_traj_to_children[new_node_id] =
pose_traj_from_closest_node


            # Query the KDTree for all node ids within the ball_radius
            nodes_KDTree = cKDTree([node.pose[0:2] for node in self.nodes])
            close_node_ids = nodes_KDTree.query_ball_point(new_node_pose[0:2], self.ball_radius())
            # Remove the new_node id in close_node_indices
            close_node_ids.remove(new_node_id)

            # Find the best node within the ball_radius to connect to new_node
            best_node_id = None
            best_cost_to_come = new_node.cost
            best_pose_traj = None
            for close_node_id in close_node_ids:
                # compute the potential connection between the close_node to the new_node
                potential_pose_traj, last_valid_substep, last_valid_pose =
self.connect_node_to_point(
                    self.nodes[close_node_id], new_node.pose[0:2], visualize=0
                )

                # reject path with collisions
                if last_valid_substep != (potential_pose_traj.shape[0]-1):
                    # cannot connect to new_node due to collision
                    continue

                potential_cost_to_come = self.nodes[close_node_id].cost +
self.cost_to_come(potential_pose_traj)
                if potential_cost_to_come < best_cost_to_come:
                    best_cost_to_come = potential_cost_to_come
                    best_node_id = close_node_id
                    best_pose_traj = potential_pose_traj

            # Re-wire connection to new_node from nodes within the ball_radius
            if best_node_id is not None:
                # there exists a node that yields a lower cost-to-come than the currently wired
node
                # hence needs re-wiring

                # remove existing connection from closest_node_id
                if visualize != 0:
                    existing_pose_traj =
self.nodes[closest_node_id].pose_traj_to_children[new_node_id]
                    for i in range(1, existing_pose_traj.shape[0]):
                        self.window.remove_point(existing_pose_traj[i, 0:2], radius=1,
update=False)
                    if frequent_viz_update:
                        self.window.update()
                self.nodes[closest_node_id].children_ids.remove(new_node_id)
                del self.nodes[closest_node_id].pose_traj_to_children[new_node_id]

                # re-wire to connect from best_node
                new_node.pose = best_pose_traj[-1]
                new_node.parent_id = best_node_id
                new_node.cost = best_cost_to_come
                self.nodes[best_node_id].children_ids.append(new_node_id)
                self.nodes[best_node_id].pose_traj_to_children[new_node_id] = best_pose_traj

                if visualize != 0:
                    for i in range(best_pose_traj.shape[0]):
                        self.window.add_point(best_pose_traj[i, 0:2], radius=1, color=(0, 0, 255),
update=False)
                    if frequent_viz_update:
                        self.window.update()


            # For all nodes within the ball_radius, try connecting to them from new_node
            # and see if the path including new_node yields a lower cost-to-come
            for close_node_id in close_node_ids:
                # compute the potential connection between new_node to a close_node
                potential_pose_traj, last_valid_substep, last_valid_pose =
self.connect_node_to_point(
                    new_node, self.nodes[close_node_id].pose[0:2], visualize=0
                )

                # reject path with collisions
                if last_valid_substep != (potential_pose_traj.shape[0]-1):
                    # cannot connect to new_node due to collision
                    continue


                abs_angle_diff = np.abs(potential_pose_traj[-1, 2] -
self.nodes[close_node_id].pose[2])
                rot_dist = min(np.pi * 2 - abs_angle_diff, abs_angle_diff)
                potential_cost_to_come = new_node.cost + self.cost_to_come(potential_pose_traj) +
10*rot_dist*self.robot_radius
                if potential_cost_to_come < self.nodes[close_node_id].cost:
                    # remove close_node's parent node's information
                    close_node_parent_id = self.nodes[close_node_id].parent_id
                    if visualize != 0:
                        existing_pose_traj =
self.nodes[close_node_parent_id].pose_traj_to_children[close_node_id]
                        for i in range(1, existing_pose_traj.shape[0]):
                            self.window.remove_point(existing_pose_traj[i, 0:2], radius=1,
update=False)
                        if frequent_viz_update:
                            self.window.update()
                    self.nodes[close_node_parent_id].children_ids.remove(close_node_id)
                    del self.nodes[close_node_parent_id].pose_traj_to_children[close_node_id]

                    # re-wire from new_node to close_node
                    self.nodes[close_node_id].parent_id = new_node_id
                    self.nodes[close_node_id].cost = potential_cost_to_come
                    new_node.children_ids.append(close_node_id)
                    new_node.pose_traj_to_children[close_node_id] = potential_pose_traj
                    self.update_children(close_node_id)

                    if visualize != 0:
                        for i in range(potential_pose_traj.shape[0]):
                            self.window.add_point(potential_pose_traj[i, 0:2], radius=1, color=(0,
0, 255), update=False)
                        if frequent_viz_update:
                            self.window.update()


            # Check if goal has been reached
            if np.linalg.norm(new_node_pose[0:2] - self.goal_point) <= self.stopping_dist:
                self.goal_node_ids.append(new_node_id)
                if (self.best_goal_node_id == -1):
                    # setting self.best_goal_node_id for the first time
                    print("SOLUTION FOUND")
                    self.best_goal_node_id = new_node_id
                    self.best_goal_pose_path, self.best_goal_pose_traj = self.recover_path(
                        self.best_goal_node_id, visualize=visualize,
                    )
                    self.update_sampling_space()
                    if save_path:
                        # np.save("rrt_star_path.npy", np.array(self.best_goal_pose_path))
                        np.save("rrt_star_path.npy", self.best_goal_pose_traj[::5, :])

            for goal_node_id in self.goal_node_ids:
                if self.nodes[goal_node_id].cost < self.nodes[self.best_goal_node_id].cost:
                    print("BETTER SOLUTION FOUND")
                    self.best_goal_node_id = goal_node_id
                    if visualize:
                        # Remove previously found path
                        for i in range(self.best_goal_pose_traj.shape[0]):
                            self.window.remove_point(self.best_goal_pose_traj[i, 0:2], radius=3,
update=False)
                        if frequent_viz_update:
                            self.window.update()
                    self.best_goal_pose_path, self.best_goal_pose_traj = self.recover_path(
                        self.best_goal_node_id, visualize=visualize,
                    )
                    self.update_sampling_space()
                    if save_path:
                        # np.save("rrt_star_path.npy", np.array(self.best_goal_pose_path))
                        np.save("rrt_star_path.npy", self.best_goal_pose_traj[::5, :])

            # re-plot the found path for visualization purposes
            if visualize and self.best_goal_node_id != -1 and iteration % 10 == 0:
                for i in range(self.best_goal_pose_traj.shape[0]):
                    self.window.add_point(self.best_goal_pose_traj[i, 0:2], radius=3, color=(0,
255, 0), update=False)
                self.window.add_goal_point(update=False)

            self.window.update()

        return self.nodes


    def update_sampling_space(self):
        # take self.best_goal_pose_traj and pass it through cost to come
        trajectory_length = self.cost_to_come(self.best_goal_pose_traj)
        start_to_goal_vec = self.best_goal_pose_traj[-1, 0:2] - self.best_goal_pose_traj[0, 0:2]

        a = trajectory_length / 2
        c = np.linalg.norm(start_to_goal_vec)/2
        b = np.sqrt(a**2 - c**2)

        center = (self.best_goal_pose_traj[0, 0:2] + self.best_goal_pose_traj[-1, 0:2]) / 2
        ellipse_theta = np.arctan2(start_to_goal_vec[1], start_to_goal_vec[0])

        self.sampling_space_params["a"] = a
        self.sampling_space_params["b"] = b
        self.sampling_space_params["center"] = center
        self.sampling_space_params["ellipse_theta"] = ellipse_theta
        self.sampling_space_params["goal_range_multiplier"] = 1


    def recover_path(self, node_id=-1, visualize=0):
        path = [self.nodes[node_id].pose]
        node_id_path = [node_id]

        current_node_id = self.nodes[node_id].parent_id
        while current_node_id > -1:
            path.append(self.nodes[current_node_id].pose)
            node_id_path.append(current_node_id)
            current_node_id = self.nodes[current_node_id].parent_id
        path.reverse()
        node_id_path.reverse()

        trajectory = np.empty((0, 3))
        for i in range(len(node_id_path)-1):
            curr_node_id = node_id_path[i]
            next_node_id = node_id_path[i+1]
            trajectory = np.vstack((trajectory,
self.nodes[curr_node_id].pose_traj_to_children[next_node_id]))

        if visualize:
            for i in range(trajectory.shape[0]):
                self.window.add_point(trajectory[i, 0:2], radius=3, color=(0, 255, 0),
update=False)
            self.window.add_goal_point(update=False)
            self.window.update()
        return path, trajectory
```

Appendex B: Trajectory Rollout for Local Planning Implementation

```python
#!/usr/bin/env python3
from __future__ import division, print_function
import os

import numpy as np
from scipy.linalg import block_diag
from scipy.spatial.distance import cityblock
import rospy
import tf2_ros

# msgs
from geometry_msgs.msg import TransformStamped, Twist, PoseStamped
from nav_msgs.msg import Path, Odometry, OccupancyGrid
from visualization_msgs.msg import Marker

# ros and se2 conversion utils
import utils

# path planner
from l2_planning import PathPlanner


TRANS_GOAL_TOL = .1  # m, tolerance to consider a goal complete
ROT_GOAL_TOL = 0.1 #.3  # rad, tolerance to consider a goal complete
TRANS_VEL_OPTS = np.linspace(-0.26, 0.26, 3) #[-0.26, -0.13, -0.025, 0, 0.025, 0.13, 0.26] # m/s,
max of real robot is .26
ROT_VEL_OPTS = np.linspace(-1.82, 1.82, 11)  # rad/s, max of real robot is 1.82
CONTROL_RATE = 5  # Hz, how frequently control signals are sent
CONTROL_HORIZON = 3  # seconds. if this is set too high and INTEGRATION_DT is too low, code will
take a long time to run!
INTEGRATION_DT = .025  # s, delta t to propagate trajectories forward by
COLLISION_RADIUS = 0.225  # m, radius from base_link to use for collisions, min of 0.2077 based
on dimensions of .281 x .306
ROT_DIST_MULT = 1.0  # multiplier to change effect of rotational distance in choosing correct
control
OBS_DIST_MULT = .1  # multiplier to change the effect of low distance to obstacles on a path
MIN_TRANS_DIST_TO_USE_ROT = TRANS_GOAL_TOL  # m, robot has to be within this distance to use rot
distance in cost
PATH_NAME = 'path.npy'  # saved path from l2_planning.py, should be in the same directory as this
file

# here are some hardcoded paths to use if you want to develop l2_planning and this file in
parallel
# TEMP_HARDCODE_PATH = [[2, 0, 0], [2.75, -1, -np.pi/2], [2.75, -4, -np.pi/2], [2, -4.4, np.pi]]
# almost collision-free
TEMP_HARDCODE_PATH = [[2, -.5, 0], [2.4, -1, -np.pi/2], [2.45, -3.5, -np.pi/2], [1.5, -4.4,
np.pi]]  # some possible collisions


class PathFollower():
   def __init__(self):
       # time full path
       self.path_follow_start_time = rospy.Time.now()
```

```
        # use tf2 buffer to access transforms between existing frames in tf tree
        self.tf_buffer = tf2_ros.Buffer()
        self.listener = tf2_ros.TransformListener(self.tf_buffer)
        rospy.sleep(1.0)  # time to get buffer running

        # constant transforms
        self.map_odom_tf = self.tf_buffer.lookup_transform('map', 'odom', rospy.Time(0),
rospy.Duration(2.0)).transform

        # subscribers and publishers
        self.cmd_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
        self.global_path_pub = rospy.Publisher('~global_path', Path, queue_size=1, latch=True)
        self.local_path_pub = rospy.Publisher('~local_path', Path, queue_size=1)
        self.collision_marker_pub = rospy.Publisher('~collision_marker', Marker, queue_size=1)

        # map
        map = rospy.wait_for_message('/map', OccupancyGrid)
        # (1600, 1600)
        self.map_np = np.array(map.data).reshape(map.info.height, map.info.width)
        self.map_resolution = round(map.info.resolution, 5)
        # [-21.0, -49.25, 0.000000]
        self.map_origin = -utils.se2_pose_from_pose(map.info.origin)  # negative because of weird
way origin is stored
        self.map_nonzero_idxes = np.argwhere(self.map_np)

        # collisions
        self.collision_radius_pix = COLLISION_RADIUS / self.map_resolution
        self.collision_marker = Marker()
        self.collision_marker.header.frame_id = '/map'
        self.collision_marker.ns = '/collision_radius'
        self.collision_marker.id = 0
        self.collision_marker.type = Marker.CYLINDER
        self.collision_marker.action = Marker.ADD
        self.collision_marker.scale.x = COLLISION_RADIUS * 2
        self.collision_marker.scale.y = COLLISION_RADIUS * 2
        self.collision_marker.scale.z = 1.0
        self.collision_marker.color.g = 1.0
        self.collision_marker.color.a = 0.5

        # transforms
        self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_link', rospy.Time(0),
rospy.Duration(2.0))
        self.pose_in_map_np = np.zeros(3)
        self.pos_in_map_pix = np.zeros(2)
        self.update_pose()

        # path variables
        cur_dir = os.path.dirname(os.path.realpath(__file__))

        # to use the temp hardcoded paths above, switch the comment on the following two lines
        self.path_tuples = np.load(os.path.join(cur_dir, 'rrt_star_path.npy'))
        # self.path_tuples = np.array(TEMP_HARDCODE_PATH)

        self.path = utils.se2_pose_list_to_path(self.path_tuples, 'map')
        self.global_path_pub.publish(self.path)
        # goal
        self.cur_goal = np.array(self.path_tuples[0])
        self.cur_path_index = 0

        # trajectory rollout tools
        # self.all_opts is a Nx2 array with all N possible combinations of the t and v vels,
scaled by integration dt
        self.all_opts = np.array(np.meshgrid(TRANS_VEL_OPTS, ROT_VEL_OPTS)).T.reshape(-1, 2)

        # if there is a [0, 0] option, remove it
        all_zeros_index = (np.abs(self.all_opts) < [0.001, 0.001]).all(axis=1).nonzero()[0]
        if all_zeros_index.size > 0:
            self.all_opts = np.delete(self.all_opts, all_zeros_index, axis=0)
        self.all_opts_scaled = self.all_opts * INTEGRATION_DT

        self.num_opts = self.all_opts_scaled.shape[0]
        self.horizon_timesteps = int(np.ceil(CONTROL_HORIZON / INTEGRATION_DT))

        self.rate = rospy.Rate(CONTROL_RATE)

        map_filename = "willowgarageworld_05res.png"
        map_settings_filename = "willowgarageworld_05res.yaml"
        self.path_planner = PathPlanner(
            map_filename=map_filename,
            map_setings_filename=map_settings_filename,
            goal_point=self.path_tuples[0, 0:2],
            stopping_dist=0.5,
        )
        self.path_planner.vel_max = 0.26
        self.path_planner.rot_vel_max = 1.82

        rospy.on_shutdown(self.stop_robot_on_shutdown)
        self.follow_path()

    def follow_path(self):
        while not rospy.is_shutdown():
            # timing for debugging...loop time should be less than 1/CONTROL_RATE
            tic = rospy.Time.now()

            self.update_pose()
            self.check_and_update_goal()

            # shrink the control horizon as the robot approaches the target such that
            # it can retain a fast velocity
            dist_from_goal = np.linalg.norm(self.pose_in_map_np[0:2] - self.cur_goal[0:2])
            min_time_to_goal_trans = dist_from_goal / 0.26
            if np.linalg.norm(dist_from_goal) < MIN_TRANS_DIST_TO_USE_ROT:
                abs_angle_diff = np.abs(self.pose_in_map_np[2] - self.cur_goal[2])
                rot_dist_error = np.minimum(np.pi * 2 - abs_angle_diff, abs_angle_diff)
                min_time_to_goal_rot = np.abs(rot_dist_error) / 1.82
            else:
                min_time_to_goal_rot = 0
            control_horizon = min_time_to_goal_trans + min_time_to_goal_rot
            control_horizon = max(control_horizon, 0.75)
            control_horizon = min(control_horizon, CONTROL_HORIZON)

            pose_traj, last_valid_substep, last_valid_poses =
self.path_planner.trajectory_rollout(
                vel=self.all_opts[:, 0:1],
                rot_vel=self.all_opts[:, 1:2],
                pose=self.pose_in_map_np,
                timestep=control_horizon,
                num_substeps=self.horizon_timesteps,
            )

            valid_pose_traj_mask = ~np.isnan(pose_traj[:, -1, :]).any(axis=1)
            valid_pose_traj = pose_traj[valid_pose_traj_mask]

            if valid_pose_traj.shape[0] == 0:
                control = [-0.1, 0]
            else:
                trans_dist_error = valid_pose_traj[:, -1, 0:2] - self.cur_goal[0:2].reshape(1, 2)
                trans_cost = np.linalg.norm(trans_dist_error, axis=1)

                if np.linalg.norm(dist_from_goal) < MIN_TRANS_DIST_TO_USE_ROT:
                    abs_angle_diff = np.abs(valid_pose_traj[:, -1, 2] - self.cur_goal[2])
                    rot_dist_error = np.minimum(np.pi * 2 - abs_angle_diff, abs_angle_diff)
                    rot_cost = np.abs(rot_dist_error)
                else:
                    rot_cost = 0
                final_cost = trans_cost + ROT_DIST_MULT*rot_cost

                best_valid_vel_idx = np.argmin(final_cost)

                control = self.all_opts[valid_pose_traj_mask][best_valid_vel_idx]
                best_valid_pose_traj = valid_pose_traj[best_valid_vel_idx]
                self.local_path_pub.publish(utils.se2_pose_list_to_path(best_valid_pose_traj,
'map'))

            self.cmd_pub.publish(utils.unicyle_vel_to_twist(control))

            # print("Selected control: {control}, Loop time: {time}, Max time: {max_time}".format(
            #     control=control, time=(rospy.Time.now() - tic).to_sec(),
max_time=1/CONTROL_RATE))

            self.rate.sleep()

   def update_pose(self):
       # Update numpy poses with current pose using the tf_buffer
       self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_link',
rospy.Time(0)).transform
       self.pose_in_map_np[:] = [self.map_baselink_tf.translation.x,
self.map_baselink_tf.translation.y,
                                 utils.euler_from_ros_quat(self.map_baselink_tf.rotation)[2]]
       self.pos_in_map_pix = (self.map_origin[:2] + self.pose_in_map_np[:2]) /
self.map_resolution
       self.collision_marker.header.stamp = rospy.Time.now()
       self.collision_marker.pose = utils.pose_from_se2_pose(self.pose_in_map_np)
```

```
        self.collision_marker_pub.publish(self.collision_marker)

    def check_and_update_goal(self):
        # iterate the goal if necessary
        dist_from_goal = np.linalg.norm(self.pose_in_map_np[:2] - self.cur_goal[:2])
        abs_angle_diff = np.abs(self.pose_in_map_np[2] - self.cur_goal[2])
        rot_dist_from_goal = min(np.pi * 2 - abs_angle_diff, abs_angle_diff)
        if dist_from_goal < TRANS_GOAL_TOL and rot_dist_from_goal < ROT_GOAL_TOL:
            rospy.loginfo("Goal {goal} at {pose} complete.".format(
                    goal=self.cur_path_index, pose=self.cur_goal))
            if self.cur_path_index == len(self.path_tuples) - 1:
                rospy.loginfo("Full path complete in {time}s! Path Follower node shutting
down.".format(
                    time=(rospy.Time.now() - self.path_follow_start_time).to_sec()))
                rospy.signal_shutdown("Full path complete! Path Follower node shutting down.")
            else:
                self.cur_path_index += 1
                self.cur_goal = np.array(self.path_tuples[self.cur_path_index])
        else:
            rospy.logdebug("Goal {goal} at {pose}, trans error: {t_err}, rot error:
{r_err}.".format(
                goal=self.cur_path_index, pose=self.cur_goal, t_err=dist_from_goal,
r_err=rot_dist_from_goal
            ))

    def stop_robot_on_shutdown(self):
        self.cmd_pub.publish(Twist())
        rospy.loginfo("Published zero vel on shutdown.")


if __name__ == '__main__':
    try:
        rospy.init_node('path_follower', log_level=rospy.DEBUG)
        pf = PathFollower()
    except rospy.ROSInterruptException:
        pass
```