ROB521 Lab 2

Ben Agro Jerry Chen Connor Lee Cameron Witkowski

PRA0101

Group 3

February 14, 2023

1 Part A: Simulation

1.1 Task 1: Collision Detection

To determine if the robot is colliding when at pose $\mathbf{p}_m = \begin{bmatrix} x & y & \theta \end{bmatrix}^T$ in the map, we first convert that pose into a set of occupancy grid indices denoting the area the robot is covering and then check if any of those grid cells have an obstacle in them. For efficiency, we batch this operation over multiple robot poses.

1.2 Task 2: Simulate Trajectory

1.2.1 Trajectory Rollout

Our trajectory rollout function implements the following equations,

$$x(t) = \begin{cases} \omega t \cos(\theta_0) + x_0 & \text{if} \quad \omega = 0 \\ \frac{v}{\omega} t \sin(\omega t + \theta_0) - \sin(\theta_0) + x_0 & \text{else} \end{cases} \tag{1}$$

$$y(t) = \begin{cases} \omega t \sin(\theta_0) + y_0 & \text{if } \omega = 0 \\ -\frac{v}{\omega} t \cos(\omega t + \theta_0) + \cos(\theta_0) + y_0 & \text{else} \end{cases} \tag{2}$$

$$\theta(t) = \omega t + \theta_0 \tag{3}$$

where ω and v are the given rotational and translational velocities of the robot (in the robot frame), and $\begin{bmatrix} x_0, y_0, \theta_0 \end{bmatrix}^T$ is the initial point in map coordinates. The function returns a trajectory in map coordinates and is batched over (ω, v) pairs for efficiency.

This function has two options for collision checking. The first is to fill any points along the trajectory where the robot collides with `NaN` values (call this `strict` mode). The second is to "stop" the robot the first timestep prior to collision and copy that point to all subsequent timesteps of the trajectory (call this `stop` mode).

1.2.2 Robot Controller

We "simulate" a set of candidate (v, ω) inputs using the trajectory rollout function in `stop` mode, and return the pair that gets closest to the goal, as measured by ℓ_2 distance (θ is not included in the distance measurement).

1.2.3 Simulate Trajectory

Simulate trajectory first finds the best (v, ω) inputs using the `robot_controller` function, and then simulates the resulting trajectory using the `trajectory_rollout` function (in `stop mode`).

1.3 Task 3: RRT Planning Algorithm

1.3.1 Sampling Strategy

On the Willow Garage map: With 95% probability, we sample uniformly from the area of the map containing the Willow Garage building, and with 5% probability, we sample points in the goal area.

On the Myhal map, we sample new points uniformly everywhere.

1.3.2 Closest Node

We compute the closest node using a kd-tree (scipy `cKDTree` implementation).

1.4 Results

A brief description of the RRT algorithm is presented below. The complete code is presented in Figure 1 in Appendix A.

1. Sample a point according to the sampling strategy described in section 1.3.1.
2. Compute the closest node to this point in the current set of nodes.
3. Select vehicle input and simulate the trajectory of the robot driving from the closest node to this new point.
4. In this trajectory, add the last pose reached as a node to the set of nodes, setting its parent to the node the trajectory began from.
5. If the new node is in the goal set, break.
6. Else, go back to step 1 and repeat.

Our results of RRT are presented in fig. 1a.

1.5 Task 4: Re-wiring

To find a trajectory connecting two points, we

1. Relax the heading constraint on the goal point
2. Find the circle that passes through both points and is tangent to the heading at the starting point,
3. Construct the trajectory by adding points along the minor arc of the circumference.

We check each point along the resulting trajectory for collision. When re-wiring nodes, any trajectories that have collisions are rejected.

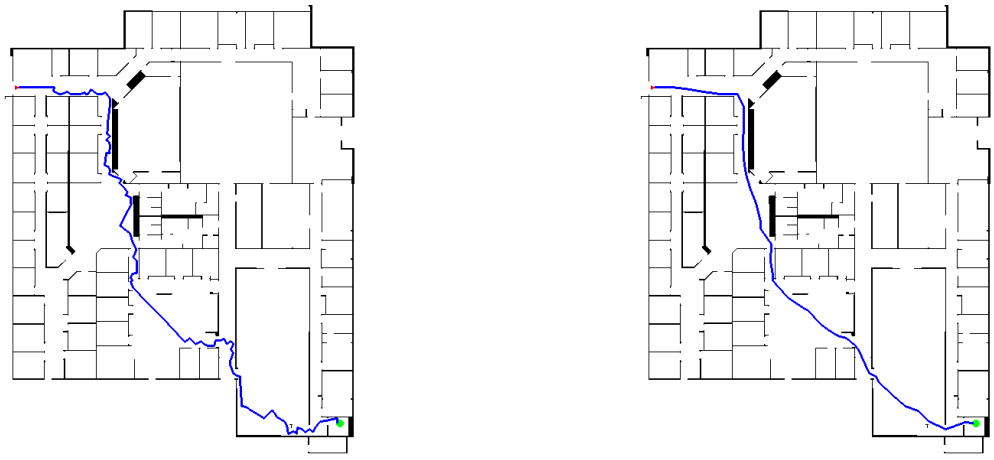
1.6 Task 5: Trajectory Costs

To find the cost of trajectories between two configurations, we sum up the ℓ_2 distances between each waypoint along the trajectory (ignoring heading). This approximates the trajectory length.

1.7 Task 6 and 7: RRT*

A summary of our RRT* algorithm is described below. The complete code is presented in listing 2.

1. While a goal is not found or maximum iterations have not been reached:
   (a) Sample a point according to the sampling strategy described in section 1.3.1.
   (b) Compute the closest node to this point in the current set of nodes.



(a) RRT (b) RRT*

Figure 1: Results of RRT and RRT* on the Willow Garage map.

(c) Select vehicle input and simulate the trajectory of the robot driving from the closest node to this new point.

(d) In this trajectory, add the last pose reached as a node `cur` to the set of nodes, setting its parent to the node the trajectory began from.

(e) **Re-wiring 1:**

i. Find nodes {`n`} within a certain radius of `cur`. (using scipy's `query_ball_point`).

ii. For each one of these nearby nodes `n`, try to find a trajectory from `n` to `cur` with `connect_node_to_point` (rejecting any connections that have collisions).

iii. Keep the connection to nearby nodes with the lowest cost (section 1.6), fixing child/parent relationships.

(f) **Re-wiring 2:**

i. Find nodes $\{n\}$ within a certain radius of `cur`.

ii. For each one of these nearby nodes, to find a trajectory from `cur` to `n` with `connect_node_to_point` (rejecting any connections that have collisions).

iii. If the cost to come to `n` through `cur` is less than the current best cost to come to `n`, set `n`'s parent to `cur`, remove `n` from the list of children of its old parent, and update the cost to come of `n` and its children recursively.

(g) If the new node is in the goal set, add it to the list of goals.

(h) Else, go back to step 1a.

Notice that we retain a list of nodes in the goal region (which we search for during a specified number of iterations). The path we retrieve is that to the node in the goal region with the lowest cost to come. See fig. 1b for results. Notice that the path found by RRT* is much smoother and shorter than the path found by RRT.

1.8 Task 8: Local Planner

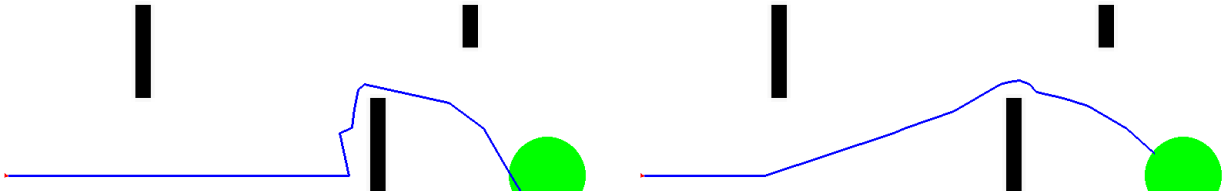
Our local planner works as follows

1. We generate a set of (v, ω) pairs to evaluate.
2. We use the aforementioned trajectory rollout function in `strict` mode to generate the candidate trajectories from these (v, ω) pairs. Trajectories that have collisions are rejected. We dynamically set the rollout time horizon based on the robot's travel time to the goal. This is computed by dividing the ℓ_2 distance of the robot to its current goal by the robot's maximum linear speed (if this value is less than 1s, we use 1s instead).
3. Each valid trajectory is scored based on the ℓ_2 distance from the trajectory endpoint to the goal. If the robot is within some tolerance of the goal, we also include a cost on the robot heading - the absolute value of the smallest angular displacement between the final pose along the trajectory and the goal orientation.
4. We select the trajectory with the lowest cost, execute the corresponding control input, and repeat.

See the attached video.

2 Part B: Real Environment

Results in fig. 2a. See the video attached to the submission.



(a) RRT (b) RRT*

Figure 2: Results of RRT and RRT* on the Myhal map.

3 Appendix A

```
def rrt_planning(self):
i = 0
while True:
    #Sample map space
    print(i)
    point = self.sample_map_space()
    self.window.add_point(point[:-1, 0].copy(), color = (255, 0, 0))
    self.window.check_for_close()

    #Get the closest point
    closest_node_id = self.closest_node(point)[0]

    #Simulate driving the robot towards the closest point
    trajectory_o = self.simulate_trajectory(self.nodes[closest_node_id].point, point, fix_col =
    ↪  True)
    if np.any(np.isnan(trajectory_o)):
        continue

    arrived_to_pt = trajectory_o[-1][:, None]  # (3, 1)

    if self.check_if_duplicate(arrived_to_pt):
        continue
```

```
        #print(f"Arrived to point: {arrived_to_pt}")
        self.window.add_point(arrived_to_pt[:-1, 0].copy())
        assert not self.is_colliding(arrived_to_pt[:-1])


        self.nodes.append(Node(point = arrived_to_pt, parent_id = closest_node_id, cost = 0)) # no cost
        ↪  for RRT
        self.node_pts = np.concatenate((self.node_pts, arrived_to_pt[:2][None]), axis = 0)
        self.nodes[closest_node_id].children_ids.append(len(self.nodes) - 1)
        self.nodes[closest_node_id].traj_to_children[len(self.nodes) - 1] = trajectory_o

        if np.linalg.norm(arrived_to_pt[:-1] - self.goal_point) < self.stopping_dist: # reached goal
            self.goal_nodes[len(self.nodes) - 1] = self.nodes[-1]
            self.best_goal_node_id = len(self.nodes) - 1
            break

        i += 1

    return self.nodes
```

Listing 1: RRT Code

```
def rrt_star_planning(self, max_iters = 4000):
    #This function performs RRT* for the given map and robot
    i = 0
    while len(self.goal_nodes) == 0 or i < max_iters:
        print(i)

        #Sample
        point = self.sample_map_space()
        self.window.add_point(point[:-1, 0].copy(), color = (255, 0, 0))
        self.window.check_for_close()

        #Closest Node
        closest_node_id = self.closest_node(point)[0]

        #Simulate trajectory
        trajectory_o = self.simulate_trajectory(self.nodes[closest_node_id].point, point, fix_col =
        ↪  True)
        if np.any(np.isnan(trajectory_o)):
            continue

        arrived_to_pt = trajectory_o[-1][:, None]  # (3, 1)

        if self.check_if_duplicate(arrived_to_pt):
            continue


        self.window.add_point(arrived_to_pt[:-1, 0].copy())
        assert not self.is_colliding(arrived_to_pt[:-1])
        cost = self.cost_to_come(trajectory_o) + self.nodes[closest_node_id].cost
        self.nodes.append(Node(arrived_to_pt, parent_id = closest_node_id, cost = cost))
        self.node_pts = np.concatenate((self.node_pts, arrived_to_pt[:2][None]), axis = 0)
        self.nodes[closest_node_id].children_ids.append(len(self.nodes) - 1)
        self.nodes[closest_node_id].traj_to_children[len(self.nodes) - 1] = trajectory_o

        i += 1

```

```
        self.min_pt = np.minimum(self.min_pt, arrived_to_pt)
        self.max_pt = np.maximum(self.max_pt, arrived_to_pt)

        #Last node rewire
        kdtree = scipy.spatial.cKDTree(self.node_pts[:-1, :, 0])
        close_idx = kdtree.query_ball_point(arrived_to_pt[:2,0], r = self.ball_radius())
        best_ctc = np.inf
        best_id = closest_node_id
        best_traj = None
        for id in close_idx:
            new_traj = self.connect_node_to_point(self.nodes[id].point, arrived_to_pt)
            if np.any(np.isnan(new_traj)):
                continue
            assert np.allclose(new_traj[0, :2], self.nodes[id].point[:2, 0]),
            ↪  f"{new_traj[0]}\n{self.nodes[id].point}"
            assert np.allclose(new_traj[-1, :2], arrived_to_pt[:2, 0]), f"{new_traj[-1,
            ↪  :2]}\n{arrived_to_pt[:2, 0]}"
            curr_ctc = self.cost_to_come(new_traj) + self.nodes[id].cost
            if curr_ctc < best_ctc:
                best_ctc = curr_ctc
                best_id = id
                best_traj = new_traj

        if best_id != closest_node_id:
            self.nodes[-1].parent_id = best_id
            self.nodes[-1].cost = best_ctc
            self.nodes[best_id].children_ids.append(len(self.nodes) - 1)
            self.nodes[best_id].traj_to_children[len(self.nodes) - 1] = best_traj
            self.nodes[closest_node_id].children_ids.remove(len(self.nodes) - 1)
            del self.nodes[closest_node_id].traj_to_children[len(self.nodes) - 1]


        #Close node rewire
        for id in close_idx:
            new_traj = self.connect_node_to_point(arrived_to_pt, self.nodes[id].point)
            if np.isnan(new_traj).any():
                continue
            assert np.allclose(new_traj[0, :2], arrived_to_pt[:2, 0])
            assert np.allclose(new_traj[-1, :-1], self.nodes[id].point[:-1, 0])
            new_ctc = self.cost_to_come(new_traj) + self.nodes[-1].cost
            if new_ctc < self.nodes[id].cost:
                self.nodes[self.nodes[id].parent_id].children_ids.remove(id)
                self.nodes[id].parent_id = len(self.nodes) - 1
                self.nodes[id].cost = new_ctc
                self.nodes[-1].children_ids.append(id)
                self.nodes[-1].traj_to_children[id] = new_traj
                self.update_children(id)

        if np.linalg.norm(arrived_to_pt[:-1] - self.goal_point) < self.stopping_dist: # reached
        ↪  goal
            self.goal_nodes[len(self.nodes) - 1] = self.nodes[-1]
            if self.best_goal_node_id is None or self.goal_nodes[self.best_goal_node_id].cost >
            ↪  self.nodes[-1].cost:
                self.best_goal_node_id = len(self.nodes) - 1

    return self.nodes
```

Listing 2: RRT* Code