E160 – Autonomous Robot Navigation

Lab 5

Probabilistic Road Map (PRM) Motion Planning

INTRODUCTION

Given a robot's location in a known environment, a motion planning algorithm can be used to construct a collision-free trajectory that connects a start configuration to a goal configuration. Then, the robot can follow the trajectory to safely arrive at the goal location.

In this lab, you will implement a single-query Probabilistic Road Map (PRM) motion planning algorithm. At a high level, the algorithm generates a randomly expanding roadmap, consisting of nodes and edges to connect the start and goal configurations. More specifically, the algorithm roots the road map with a node at the start configuration, then randomly expands the road map by adding new edges and nodes until one of the new nodes has a collision-free edge connecting it to the goal configuration.

BACKGROUND

An example single-query PRM algorithm is:

1. Add start configuration c_{start} to $R(N,E)$
2. Loop
3. Randomly Select New Node c to expand
4. Randomly Generate new Node c' from c
5. If edge e from c to c' is collision-free
6. Add (c', e) to R
7. If c' belongs to endgame region, return path
8. Return if stopping criteria is met

The key steps are step 3, 4 and 7. To save time, Kindel et. al.'s grid cell based weighted sampling scheme has been coded for you to help in step 3. For step 5 and 7, a collision-checking algorithm has been coded for you.

NOTE: For debugging the motion planner, the estimated states are set to exactly equal the actual states in simulator mode (e.g. `self.state_est = self.state_odo`). This way you won't deal with any residual localization problems.

EXPERIMENTS

Download the most recent version of the base code for lab 5. The main control loop `update()` in `E160_robot.py` now contains the following functions:

```
def update(self, deltaT):
        # get sensor measurements
        self.encoder_measurements, self.range_measurements =
            self.update_sensor_measurements(deltaT)

        # update odometry
        delta_s, delta_theta =
            self.update_odometry(self.encoder_measurements)

        # update simulated real position, find ground truth for simulation
        self.state_odo = self.localize(self.state_odo, delta_s, delta_theta,
            self.range_measurements)

        # localize with odometry
        self.state_est = self.state_odo

        # to out put the true location for display purposes only.
        self.state_draw = self.state_odo

        # call motion planner
        self.motion_plan()
        self.track_trajectory()

        # determine new control signals
        self.R, self.L = self.update_control(self.range_measurements)

        # send the control measurements to the robot
        self.send_control(self.R, self.L, deltaT)
```

Note that the `self.localize()` function return the location of the robot in simulation mode. Also, your trajectory tracking code isn't required (I have provided one) but your point tracking code is required. ****Copy all necessary code for these functions from the previous lab 4 to your new lab 5 code. ****

The robot mode and control mode in E160_environment are set to:

```
self.robot_mode = "SIMULATION MODE"
self.control_mode = "AUTONOMOUS CONTROL MODE"
```

In `"AUTONOMOUS CONTROL MODE"`, clicking a point on the gui will update the `self.state_des` in the robot class, and the `self.motion_plan()` will build `self.trajectory` with a sequence of points (E160_state) for the robot to follow to its destination. A new variable `self.state_curr_des` is introduced to indicate the current point that the robot is tracking to, while `self.state_des` is now used to indicate the final destination.

Once a trajectory is constructed by the `self.motion_plan` function, the `self.track_trajectory()` function will set the desired points (`self.state_curr_dest`) to be tracked by the point tracker to be those nodes of the newly constructed trajectory. Note that the `self.tracj_trajectory()` will iterate through the `self.trajectory` list of state for the robot to follow.

Before you begin, find and understand what is happenining in the functrion `motion_plan(self)` found in the file `E160_robot.py`. This is function is called from the robots main loop and will call the motion planner when required.

For this lab, the code you will modify is located in `self.E160_MP.py`:

1. **Create the start and goal nodes**
   The call from `motion_plan(self)` in `E160_robot.py` is what initiates the motion planning algorithm. Within this function, use the constructor `Node(x, y, parent, children, index)` to set the variable `self.start_node`. Set the `nodeIndex` 0, and `parent` to `None`. These are used later when constructing the trajectory from the PRM.

   Use the `self.addNode` function to add the start node to the PRM.

2. **Random Node Selection**
   The `update_plan` function calls the `self.MotionPlanner` function which contains the main PRM code. Within this function a while loop has been created for you that iterates over possible node expansions. This loop will terminate if the maximum number of iterations is exceeded, or a path was found, (i.e. the PRM successfully connected to the goal node).

   Within the while loop, the first step is the random selection of a node to expand from. The Kindel et. al.'s grid cell based weighted sampling scheme has been coded for you. Called the `self.select_expansion_node()` function and it will return a random node for you to expand on!

3. **Node Expansion**
   For the node expansion, we will use straight line segments. That is, all edges in the PRM will be straight, thereby ignoring dynamic or kinematic constraints.

   First, randomly select a distance and orientation. You can play with the ranges of these random numbers later and see how they affect planner performance.

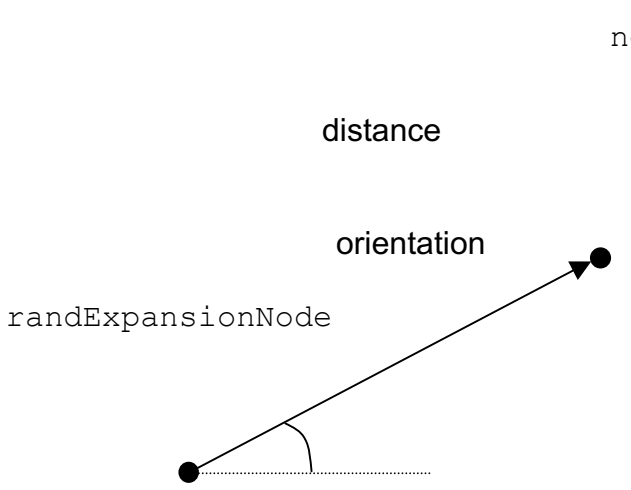

Figure 1: Random expansion to create a new node

Use the distance and orientation, along with the position of the parent node (`expansion_node.x, expansion_node.y`) to determine the location of the new node `new_x` and `new_y`. Using the `Node` constructor, create the `newNode` with this position. Set the `nodeIndex` to be `self.num_nodes`, and the `parent` to be the `expansion_node`.

4. Add new node to PRM

Create an `if` statement that calls `self.check_collision(node1, node2, tolerance)` to determine if the edge connecting the `new_node` to its parent is collision-free. The variable `tolerance` is for the allowable distance between the center of the robot and the wall, (perhaps `self.robot_radius`).

If no collision was found, add the `new_Node` to the PRM using `self.addNode()`, and append the `new_node` to the `children` of the `expansion_node`. This is for the graphing of the PRM!

5. Check for connection to goal

After adding the `new_node` to the PRM, check if the new node connects to the goal node using the collision checker again. If there is no collision between `new_node` and `goal_node`, set the `goal_node.parent` to be the `new_node`, and append `goal_node.parent` to the `children` of `new_node`. Finally, add the `goal_node` to the PRM and set the `path_found` flag to be `true`. Setting this flag will terminate the while loop.

6. Optimize Trajectory Tracker (Optional)

At this point, test your planner on many start/goal locations. You should see the trajectory connect your estimated position with the goal point on the screen. If the screen fills up with white lines, your planner didn't find a solution (maybe your goal destination is on a wall!) Once the trajectory is constructed, the robot will track each node in the trajectory using the `self.track_trajectory()` function.

Also, feel free to make several plans and pick the shortest plan before following it. Another idea is to iterate on the planned path and drop all nodes that have a collision-free line connecting its parent to its child.

DELIVERABLES

1. DEMO!!!

You must demo your working code in simulation (only) to the instructor by midnight on Sunday, April 29th. A live demo is preferred.