Motion planning with PRM and RRT

In this assignment, you will implement motion planners for two systems: a planar robot arm (with state given by the angles of the joints), and a steered car (with state given by the x,y location of a point on the robot, and the angle that the robot makes with the horizontal).

The instructions for this assignment are briefer, because by now you should have a sense of the type of questions that are interesting to explore in a report. Although I do not ask specific questions here, a good report, for example, might compare different values of *k* for a k-PRM, explore different environments for RRT, and discuss.

Take a look at these movies with a few sample animations of the simulation and planning systems you will write.

PRM for the arm robot

You will plan motions for 2R, 3R, and 4R planar arms. The base of the arm will be at location 0, 0, and the joint angles are measured counter-clockwise as described in class. For simplicity, each link on the arm will be represented by a line segment (the sample movies show the links as rectangles; that would be an extension).

Kinematics

First, you'll need to write a simple simulator for the arm that computes the locations of end points of the links for a configuration given by angles $\theta_1 \ldots \theta_n$. You may wish to take a look at the slides for the robotics lectures if you have forgotten how to do this.

I'd recommend also writing some drawing code at this point, so that you can view configurations. You should separate the drawing code and the code that computes the kinematics, since the kinematics are useful in the motion planning algorithm, and you don't want to slow the planner down by drawing graphics each time.

For graphics, you can use cs1lib, which you probably installed at the beginning of the class (see main course web page). Keep the graphics simple.

Obstacles and collision detection

There are obstacles in the workspace; you could make them either small discs or small squares for simplicity. See the movies for an example, and set up some interesting test worlds. Notice in the movies that we all the arm to have overlapping segments; we do not worry about self-collision. This is intentional; you can imagine that the links are at slightly different z-heights.

The motion planner needs to be able to detect if at some configuration (given by angles) there is a collision between the arm and the obstacles. You may use the Shapely library to check for collisions between polygons. (You are very much permitted to discuss the details of how to do this on Piazza, look up ideas online, etc. Just cite!)

Test your collision detection using the graphics by setting some configurations, and having the screen turn red for collision configurations.

Probabilistic Roadmap (PRM)

For the arm robot, you write a Probabilistic Roadmap planner, as described here. You should implement the *k-PRM* version, in which the local planner only attempts to connect a vertex to its nearest k neighbors.

There are two phases in the planner:

1. Roadmap generation, in which a graph is created
2. Query phase, in which specific paths from start to goal are created.

Start by generating the roadmap. To do this, you'll need a sampling method to create the vertices of the graph in the configuration space, a collision detection method to check if these vertices are legal, and a local planner that can attempt to connect vertices.

You'll need to also find the *k* nearest neighbors of a new vertex to connect to with the local planner. A simple (but inefficient) solution is to just loop over all previous vertices, using the local planner to check distances, and then sorting the resulting array.

A note on angular distances

I prefer to use radians for distances. Notice that the distance between the angle 0 is very close to the angle 6.27, since the circle wraps around. There are two things to be careful of when computing angular distance:

1. Are the angles you are working with in some reasonable range? If you keep rotating a link by adding a small amount to the angle, you might pass 2π. Be careful that you choose a good convention for angles and stick to it (some choose $0..2\pi$; others use negative angles for the bottom half of the circle).

2. Consider two points on the unit circle. There are two ways to go on the circle between those two points: clockwise and counter-clockwise. The correct distance is the minimum of the two. Notice that if the distance in one direction is d, then the distance in the other direction is $2\pi - d$.

I suggest writing a nice angular distance function and testing it in isolation, since it will be very useful for both the PRM and the RRT.

Query phase; testing

You should generate some interesting maps. Perhaps a "forest" of small rectangular or disc-shaped obstacles would be interesting. Show pictures of plans generated for your robot in your report. You do not need to animate anything; a good picture would perhaps show all the configurations on a single figure, illustrating how the arm moves from start to goal configuration.

Rapidly exploring random tree for a car-like mobile robot

For the car, your task is to construct a Rapidly exploring Random Tree RRT to plan motions for a car-like robot.

Assume the robot is a point, and the configuration of the robot is (x, y, θ), the location of the center of the disc, and the heading of the robot. Assume that the wheels are arranged like those of the tricycle discussed in lecture.

The controls for the robot are v and ω, the forward velocity of the robot and the angular velocity of the robot. If $v = 1$ and $\omega = 0$ for some period of time, the robot drives forwards, where "forwards" is defined by the heading of the robot θ. Both x and y will change based on the value of θ.

For the mobile robot, I have provided code that computes a new configuration after taking one of six actions for a short period of time: forwards ($v = 1$, $\omega = 0$), backwards ($v = -1$, $\omega = 0$), forwards turning counter-clockwise ($v = 1$, $\omega = 1$), backwards turning counter-clockwise ($v = -1$, $\omega = 1$), forwards turning clockwise, backwards turning clockwise). Take a look at `display_planar.py` to see how to use the functions from `planarsim.py`.

A few notes about the implementation. First, a planar robot has a configuration (x, y, θ). However, `planarsim.py` internally does most of its calculations using what are called homogeneous coordinate transform matrices. These matrices encode a rotation matrix and a translation vector that rotate and translate from the origin to some frame described by (x, y, θ). *You do not need to use these matrices, since there are functions to compute simple x, y, θ values from a matrix.*

You should also notice that the simulator gives you back some sampled trajectory: a list of matrices computed across the total duration of the trajectory at some timestep. You might use these sampled points to do collision detection, or you might just want to use the final configuration. (Side note. Notice that even though the simulator does some sampling, this is just for convenience and has nothing to do with how configurations are computed – the simulator will be just as accurate (and faster) if you reduce the number of samples.)

Possible extensions

- Read at least one paper on the RRT or one paper on the PRM. Briefly describe the main results of each paper in a previous work section of the report.

- Implement a variation of PRM or RRT; for example, toggle-PRM, medial axis PRM, or you are feeling particularly adventurous and have quite a bit of extra time, discrete RRT for multi-robot planning or RRT*. (Either of the last two would be worth 5 points.)

- Finding nearest neighbors by looping over all points is expensive. A better solution would use some sort of Approximate-Nearest-Neighbor (ANN) algorithm to find the neighbors in logarithmic or amortized constant time. You might try the nearest neighbor module in scikit.

Rubric

- Robots models and kinematics

- PRM: 5 points
- RRT: 5 points
- Report: 5 points
- Extensions: 5 points