Homework 4

Robot Autonomy
CMU 16-662, Spring 2013

TA: Shervin Javdani

1 Introduction

This homework will continue covering motion planning. We'll first extend your last code to work as a bi-directional RRT, then implement a modification to enforce a height constraint. We'll also be doing navigation planning with a differential drive system. Finally, we'll put it all together and figure out how to drive to an object and grasp it.

2 Bi-directional RRT

Implement an bi-directional RRT algorithm in `hw4_birrt.py`. You can use your code from the previous homework to start.

One thing to note - in the previous homework, we made our RRT somewhat greedy by sampling one of the goal configurations on occasion. Without this, our RRT would have explored the space for much longer before connecting. You will need to implement something similar in this homework, and explain what you did.

3 Constrained Bi-directional RRT

We'll implement a simplified version of the CBiRRT algorithm in hw4_constrain_birrt.py. Your initial end-effector pose and goal poses are all at the same height (in z dimension). You must implement a constrained RRT such that the height of the end-effector transform never leaves the initial value by more than one centimeter in any direction. The rotation, X, and Y coordinates can vary.

First, implement the projection function in `project_z_val_manip`. It takes in a start configuration and target height value of the end effector transform, and should output a new configuration where the height is within 1cm of the target value.

Then, add the projection function to your BiRRT code so that it plans paths that satisfy the constraint at all points.

4 Navigation Planning

Now we'll try our hand at some navigation planning. We'll implement a system using the differential drive dynamics from class, ultimately using A* search to find a path.

4.1 Generate Nominal Trajectories

We'll first generate a set of trajectories for which we know we can provide controls. To do so, we'll simulate the differential drive, given by the equation:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} -\frac{\omega_1}{2} r \sin(\theta) - \frac{\omega_2}{2} r \sin(\theta) \\ \frac{\omega_1}{2} r \cos(\theta) + \frac{\omega_2}{2} r \cos(\theta) \\ \frac{\omega_1}{2L} r + \frac{\omega_2}{2L} r \end{bmatrix}$$

(Note: this is a slight correction from the equation derived in class. Namely the $\dot{\theta}$ term). Where θ is the current orientation of the robot, r is the radius of the wheel, and L is the distance from the wheel to the center. Like the derivation, we will assume the wheels are on the centerline of the robot transform. The constants for radius and length to use are given in the code as `WHEEL_RADIUS` and `ROBOT_LENGTH`.

Also note that this derivation assumes the robot drives forward in y, where openrave is setup so driving at $\theta = 0$ means you drive forward in x. Thus, you'll want to use a slightly different version of this.

Now, our task is the generate a set of nominal trajectories by simulating these controls. To do so, you'll need to pick a sequence of controls (e.g. $\omega_1 = \omega_2 = 1$ will drive you forward) for some amount of time. Each nominal trajectory should include one transform representing the difference between the initial and final (call it `transition_transform`), and a list of intermediate transforms (`full_transforms`). The point is to use `transition_transform` during search, and `full_transforms` when executing the trajectory.

The maximum control you can apply to any ω is 1, and the maximum time any nominal trajectory can be is $1s$.

Hint: You'll want to include some short trajectories, and some long ones. This will allow you to quickly make progress towards your goal, as well as fine tune to get exactly to the goal.

Your trajectories must include some that aren't just straight drives or turns in place! Also, to simplify things, I recommend not considering backwards motions (a realistic requirement for robots, since sensors tend to be on the front).

Task 1: Fill in `init_transition_transforms` to generate a set of nominal trajectories.

4.2 Search with your Trajectories

Now you will want to utilize your nominal trajectories (now represented with a relative transform), and utilize A* search to get from your current transform to any goal in goals. As mentioned, you should use `transition_transform` when searching, but when you return the set of all intermediate transforms, you should use `full_transforms` to include *all* the intermediate transforms.

You should have equal cost for using any nominal trajectory - thus, your search will minimize the number of nominal trajectories it uses to get to any goal transform. I used a cost of 1 for any action taken.

To decide when you can terminate, I have provided a function `is_at_goal_basesearch`. You can override the defaults for testing, but your final search algorithm should use the default thresholds - you must be within $2cm$ of the final location, and $\theta = \frac{\pi}{12}$ of the final orientation.

I have provided some functions which can convert from params (x, y, θ) and transforms. Also, note the functions which convert between numpy arrays and dictionary keys have two version - with and without rounding. I recommend using the with rounding for these (though your probably not searching over a regular grid anymore!)

Task 2: Come up with an admissible heuristic to use for your search.

Task 3: Implement `astar_to_transform` to search from the current transform to a *set* of goal transforms using your nominal trajectories. You should be run `run_problem_navsearch` now.

4.3 Drive and grasp

Finally, we'll combine this with an RRT planner for the hand, and navigate to our target object and grasp it. We'll implement a simple scheme for this, where we sample orientations to plan to, see if we can drive to them, and plan an arm trajectory from that orientation. In more detail:

First, fill in `sample_for_grasp` to sample a location near the target object (self.target_kinbody). I made my orientation face the kinbody from that location. Don't use a scheme that works just for this problem - it should be something general.

Next, you'll need to decide if you can actually drive to this location, and if you can pickup the object from that location. You can do so in any way you wish. To help you, I have implemented a wrapper for functions that will timeout after a specified number of sections. There is an example of calling it in `nav_and_grasp`.

Also, I have provided `get_goal_dofs`, a function that will generate a set of goal configurations for the arm for the target object from the robot's current transform.

Task 4: Implement `sample_for_grasp`
Task 5: Implement `nav_and_grasp`. Now you should be able to run `run_problem_nav_and_grasp`.

5 Deliverables and Grading

I've marked the sections that need to be filled in with a `TODO`. Please turn in your code, a pdf writeup, and each of the videos I requested (see below). I expect 8 files - a modified `hw4_birrt.py`, `hw4_constrain_rrt.py`, `hw4_navplan.py`, a writeup in pdf form, and 4 videos. Only one person per group needs to submit, but please make sure everyone's **name and andrewid** is on the pdf.

Writeup items cannot exceed 5 sentences - get your point across quickly!

- In the previous RRT, you would sometimes sample one of your goal configurations, getting to your goal faster than you would otherwise. In your writeup, describe how you connected the two trees for a BiRRT, and in particular, your sampling scheme that gets the trees to connect quickly. (2 pt)
- Implement a BiRRT, and submit a video displaying it working. (3 pts)
- Describe how your `project_z_val_manip` function works. (4 pts)
- Implement your simplified CBiRRT, and submit a video displaying it working. (4 pts)
- In your writeup, describe your set of nominal trajectories for your navigation planner. (5 pts)
- Implement `init_transition_transforms`. (5 pts)
- In your writeup, describe your heuristic. Why is it admissible? (4 pts)
- Implement `astar_to_transform`. Submit a video of your working run of `run_problem_navsearch`. (7 pts)
- Implement `sample_for_grasp`. (1 pt)
- Implement `nav_and_grasp`. Submit a video of your robot driving to the object and grasping it. (5 pts)