EECS 598: Motion Planning
Winter 2019
Homework Assignment #3
Due 3/13/2019 at 1:29pm

Rules:

1. **All homework must be done individually, but you are encouraged to post questions on Piazza.**
2. No late homework will be accepted.
3. **You must show all your work to receive credit for your solutions.**
4. The goal of this homework is to learn how to use RRTs for practical problems and to compare the performance of variants of RRT.
5. Submit your code along with a pdf of your answers in a zip file to Gradescope.

Software

1. Download the HW3 template here.

Implementation

In openrave, python is useful to interface with algorithms, but the algorithms themselves are implemented in C++ as openrave plugins (this is done for speed). For this assignment you will need to create a plugin for openrave in C++. Please see here for a tutorial on how to create a plugin skeleton.

Use the `openrave-createplugin.py` script to create a plugin. This script will also make a test file in python that interfaces with the plugin. You can use the code from this test file to send commands from a python script to your C++ plugin in openrave.

The following implementation problem should be done starting from the HW3 python template. Only edit what is inside the `### YOUR ... HERE ###` blocks. Feel free to look around the internet and the openrave installation for example code for reference, but you should implement your own code from scratch unless explicitly stated otherwise. Sharing code is not allowed. Since there is already an RRT planner in openrave, you may not use any of the code from that implementation, we will check to make sure there is no code copying.

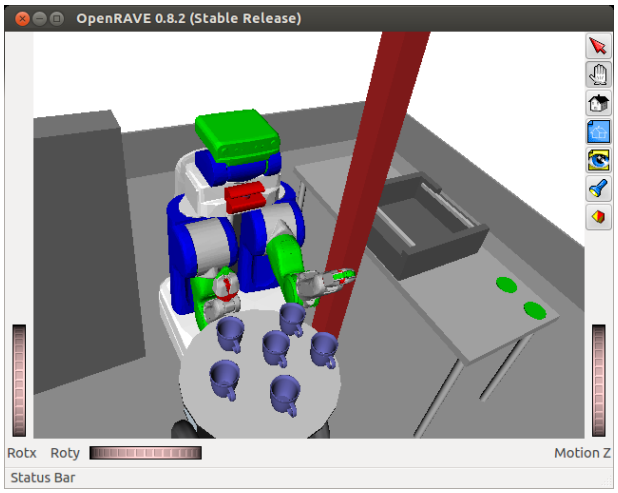
Put your openrave plugin code in the same directory as the template. Include the plugin files, python files, and `hw3.env.xml` in your `zip`.

1. (10 points) Create a header file with the following classes:

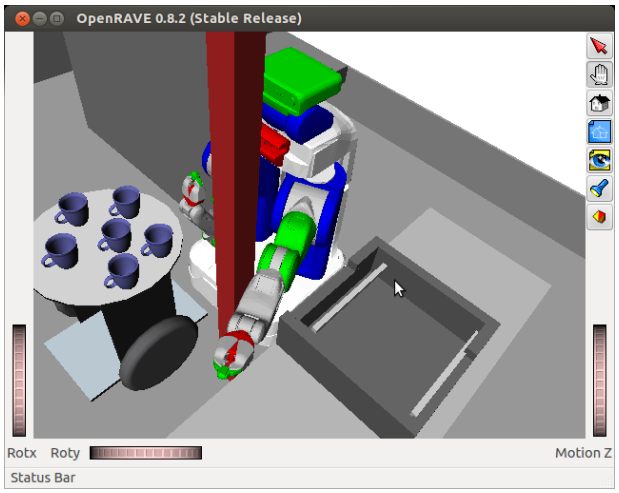
- `RRTNode`
  - This should store the configuration. I recommend something like `std::vector<float> _q`
  - This node should also have a way to access its parent node (e.g. through a pointer)
  - Add any other variables and functions you see fit
- `NodeTree`
  - This should store a set of nodes. I recommend something like `std::vector<RRTNode*> _nodes`
  - Include methods to add, delete, and get nodes from the node set
  - Include a method that returns the path of nodes from the root to a node with a given index (including that node in the path)
  - Add any other variables and functions you see fit

2. (10 points) Implement a nearest-neighbor function that finds the closest node in a `NodeTree` to a given configuration.

3. (50 points) Make a copy of `hw3.py` called `hw3_rrt.py`. Implement the RRT-Connect algorithm in C++ to search for collision-free paths in the 7DoF configuration space of the robot's arm. The algorithm should start from the current configuration of the robot's arm and find a path to a specified goal configuration. The algorithm should take as input a goal configuration (sent from a python script to your plugin).



(a) Start configuration



(b) Goal configuration

You will need to determine a reasonable step size to use. Your algorithm should produce a path in under **3 minutes** on average for the best goal bias (see below). We will run this exact command from the command line to test your code: `python hw3_rrt.py`. We will not run any other commands. We will run your code three times, and the code should find a path in under 3 minutes at least one of those times.

Complete the following:

(a) Run the RRT with goal bias ranging from 1% to 96% in increments of 5%, running the algorithm 10 times for each value of goal bias. Plot the average computation time vs. goal bias value.

Explain why the plot looks the way it does and include the plot in your `pdf`. Select the goal bias that yields the lowest computation time and use that for the remainder of this problem.

(b) Once you have computed the path from start to goal, draw the position of the left end-effector of the robot for every configuration along the path in red in the openrave viewer. You should see that the points along the path are no more than a few centimeters apart. Include a screenshot showing the path you computed in your `pdf`.

(c) Convert your path to a trajectory and execute it with the robots controller. You may do this either in python or in C++. If the robot clips any obstacles during execution, adjust your step size in the RRT so that this doesn't happen (you will need to redo part (a) if you change the step size).

4. (20 points) Implement the shortcut smoothing algorithm to shorten the path. Use 200 iterations. When we run `python hw3_rrt.py`, we should see the execution of the smoothed path in the openrave viewer.

   (a) Draw the original path of the end-effector computed by the RRT in red and the shortcut-smoothed path in blue in the openrave viewer. Include a screenshot showing the two paths for one run of the RRT in your `pdf`.

   (b) Record the path length after each iteration of the algorithm. Create a plot showing the length of the path vs. the number of smoothing iterations executed. Include this plot in your `pdf`.

5. (15 points) Run the RRT and smoothing 30 times and record

   - The computation time for the RRT
   - The computation time for smoothing
   - The number of nodes sampled
   - The length of the path (unsmoothed)
   - The length of the path (smoothed)

   (a) Create a table showing these results and compute the mean and variance for each type of data. Include this, along with a discussion explaining why the results look the way they do in your `pdf`.

   (b) Create a histogram showing how runs of the RRT are distributed in terms of computation time. Matlab's hist function is an easy way to do this. Include this histogram, along with a discussion explaining why the results look the way they do in the `pdf`.

6. (35 points) Copy the HW3 template to a file called `hw3_birrt.py`. Implement the BiDirectional RRT-Connect algorithm. We will run this exact command from the command line to run your code: `python hw3_birrt.py` and no others. When we run this code we should see the algorithm generate a new path, smooth it using shortcut smoothing, and execute the corresponding trajectory on the robot. The code should find a path in under `3 minutes` and we will test it as with the RRT-Connect. You will be able to re-use much of the code you wrote for RRT-Connect in your implementation.

(a) Compare the results of BiDirectional RRT-Connect to your implementation of the RRT-Connect in terms

- The computation time
- The number of nodes sampled
- The length of the path (unsmoothed)

Discuss why you are seeing the results that you see and include the discussion in your `pdf`.

(b) Create a histogram showing how runs of the BiDirectional RRT-Connect are distributed in terms of computation time and compare to the histogram for the RRT-Connect. Discuss the similarities (if any) and differences (if any). Include this discussion and histogram in your `pdf`.

7. Extra Credit (40 points) Copy the HW3 template to a file called `hw3_rrtstar.py`. Implement the RRT* algorithm. The cost of a path is its length. We will run this exact command from the command line to run your code: `python hw3_rrtstar.py` and no others. When we run this code we should see the algorithm generate a new path and execute the corresponding trajectory on the robot. Do not smooth the path. Terminate the code after 15 minutes and output the best path found (or no solution). You will be able to re-use much of the code you wrote for RRT-Connect in your implementation.

(a) Run RRT* 30 times and produce a plot showing average path length vs. computation time. Include the plot and discuss your results in your `pdf`.

(b) Compare the results of RRT* to your results from the RRT-Connect and BiDirectional RRT-Connect in terms

- The number of nodes sampled
- The length of the path (unsmoothed for RRT and BiRRT)
- The length of the path (smoothed for RRT and BiRRT)

Discuss why you are seeing the results that you see and include the discussion in your `pdf`.