EECS 498: Introduction to Algorithmic Robotics
Fall 2020
Homework Assignment #3
Due 10/26/2020 at 2:59pm

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. The goal of this homework is to develop your understanding of motion planning methods and interfacing with openrave. You should use python to implement solutions. You may not use any other language, only python will be accepted.
4. Submit your python files along with a `pdf` of your answers to Gradescope. Do not paste your code into your `pdf`.
5. Remember that copying-and-pasting code from other sources is not allowed.

Questions

1. (15 points) AI book, exercise 3.3
2. (10 points) AI book, exercise 4.1
3. (10 points) LaValle book, Chapter 4 exercise 5
4. (10 points) LaValle book, Chapter 4 exercise 16
5. (10 points) LaValle book, Chapter 5 exercise 18

Software

1. Download and unzip `HW3files.zip`. Run `drawing.py`. You should see the PR2 robot in an environment with a doorway and some tables in the openrave viewer. If you do not see this, openrave is not installed correctly (go back to HW1 and install it). You may also see warnings in the terminal that look like:

   ```
   [WARN] [kinbody.cpp:1504 KinBody::SetDOFValues] dof 30 value is not
   in limits -2.321000e+00<-2.121308e+00
   ```

   These are numerical precision issues and you can ignore them. You may also see errors/warnings related to Coin and SoQt (these are graphics packages). You can ignore these, too.
2. You will need to consult the `openrave.py` documentation frequently to complete this assignment.

Important Concepts in Openrave

Below are important concepts in openrave, some of which may be useful for completing the homework.

- **Locking the environment**: You can grab the lock on the environment data structure (what is displayed in the viewer) using the `with env:` statement (where env is an instance of the Environment class). See the `simplmanipulation.py` example to see how to use this. It is important to use this lock so that other openrave threads don't interrupt what you're doing and change variables you may be working on. Keep in mind that as long as you are inside a `with env:` block, changes you make to the robot (e.g. setting the configuration) may not be visible on the display. It is only after you exit this block and wait some amount of time that the display has the chance to refresh and show what you modified. However, the changes you make inside `with env:` still take place in the data structure. For instance, you can check a collision inside a `with env:` block by setting the configuration and calling check collision. Although you may not see the display update, the robot is actually at the new configuration and the collision check is taking this into account. After you exit the `with env:` block and wait, you will see the robot rendered at the new configuration.

- **Active DOFs**: A robot can have many degrees of freedom (e.g. a humanoid), but you may not want to plan for all of those DOF at the same time. For instance, you may want to drive the mobile base of a robot but not move the robot's arms. In openrave, you can select which DOF you want to plan for by using `robot.SetActiveDOFs(...)` (you can look up this function to see how it works). From then on, you can `robot.SetActiveDOFValues(...)` to set only the DOFs of the robot you care about. For example, `robot.SetActiveDOFs([0,5,6])` will set the 0th, 5th, and 6th DOFs to be active. Then you can do `robot.SetActiveDOFValues([pi/2,pi,0])` which will set the 0th joint to $\pi/2$ radians, the 5th joint to pi radians, and 6th joint to 0 radians.

- **Controllers**: In openrave you often set a configuration for the robot to test something about it. For instance, you set a configuration and then test if it is in collision. However, you don't always want the robot to go there (remember that openrave can directly control a real robot). So we can use commands such as `robot.SetActiveDOFValues(...)` to set a configuration for the robot for testing something out. If we really want the robot to go there, we have to tell that to the robot's controller. For instance, `robot.GetController().SetDesired(robot.GetDOFValues())` will tell the robot to go to whatever configuration is set on it (in simulation the robot moves to this configuration instantly, on the real robot this of course takes time). It is important to note that the controller runs in a different thread from your main code (this is so you can do things while the robot is moving). Thus, if you want to move the robot somewhere, after you tell the controller where to go, you need to wait until the controller is done. This can be done by polling this function: `robot.GetController().IsDone()`. Note that when you execute a planned path, you also need to use the robot's controller and so you need to wait for that to complete by polling the same function.

- **Viewer**: The openrave viewer comes with some handy functionality to allow you to zoom in and out and focus on different parts of the scene as well as move things around. The text below is specific to the Coin viewer. To toggle between panning mode and selecting mode, push `Esc`.

  - Panning Mode: To zoom in/out use the scroll wheel on your mouse. To focus on a part of the scene, click on the viewer window and then push `s`. Then click on the part of the scene you

want to focus on. You can rotate around that point by holding down the left mouse button and dragging the mouse. You can translate the camera by holding down the scroll wheel and moving the mouse.

- Selecting mode: Left click on an object to select it. There will be a green box around the object if it is collision-free and an orange box if it is in collision. Once you select an object drag the mouse to move it around. You can also `Ctrl + Click` on a link of a robot to move that link's joint. When you `Ctrl + Click` a wheel will appear. Drag the wheel with the left mouse button to move the joint.

Implementation

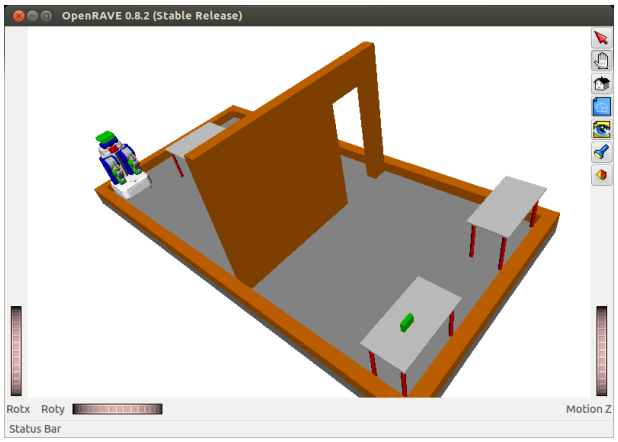
The following implementation problems should be done in python starting from the provided templates. Only edit what is inside the `### YOUR CODE HERE ###` block. Feel free to look around the internet and the openrave installation for example code for reference, especially in the openrave examples, but you should implement your own code from scratch unless explicitly stated otherwise.

1. (5 points) Run the `drawing.py`, it should load the PR2 robot in an environment with several tables and a doorway. Using your mouse to control the viewer, move all the tables to be on the PR2's side of the room. Record their position and rotation (using the display in the GUI). Now edit `drawing.py` to place the tables automatically at the poses you recorded. Include a screenshot of the scene in your `pdf`. Save the code as `tables.py` and include it in your `zip`.

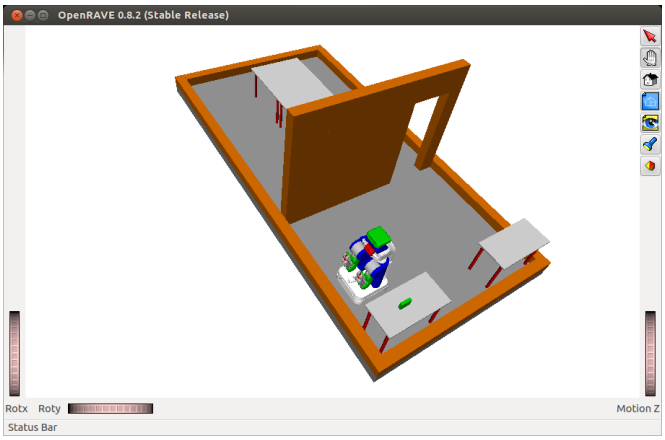
2. (10 points) Edit a clean copy of `drawing.py` to draw a set of red rectangles around the boundaries of every table in the environment. You may use a series of lines to draw the rectangles. Next edit the template to draw 35 blue points in a circle that encompases the environment shown in the scene. Make sure the points are large enough to be easily visible in a screenshot. (see `.../openravepy/_openravepy_0_9/examples/tutorial_plotting.py` for drawing examples). Include a screenshot showing the points and the rectangles in your `pdf`. Save the code as `linespoints.py` and include it in your `zip`.

3. (15 points) Edit a clean copy of `drawing.py` to place the PR2 facing the wall in the center of the environment so that the wall is close to the robot but not touching it. Confirm that the robot is not colliding with the wall by printing out the result of a call to `env.CheckCollision(robot)`. Set the 7 joints of the PR2's right arm as active (all others should be inactive). Inside a `with env:` block, use the `SetActiveDOFValues` function to set a configuration for the arm so that it is pointing straight forward. Call `env.CheckCollision(robot)` to confirm the robot is colliding with the wall in this new configuration and print out the result. Now use the `SetDesired()` function on the robot's controller to command the robot to go to the new configuration. Use `waitrobot()` (which is defined inside `drawing.py`) after `SetDesired()` to ensure the robot reaches the target configuration. Be careful in deciding if `waitrobot()` should be inside or outside the `with env:` block. Explain why you placed `waitrobot()` inside/outside the `with env:`. What would happen if it were placed in the other location and why? Include your answers in your `pdf`. Take a screenshot of the robot in the new configuration in the viewer and include it in your `pdf`. Save the code as `collision.py` and include it in your `zip`.

4. Starting from the `astar_template.py` template, implement the A* algorithm to find the shortest collision-free path for the PR2's base from the start to the goal in the given environment (see figure below). Only edit what is inside the `### YOUR CODE/IMPORTS GO HERE ###` blocks.



(a) Start configuration



(b) Goal configuration

You will be planning for translation of the base in X and Y as well as rotation θ about the Z axis of the robot, for 3 DOF total. If no path exists, the algorithm should print out "No Solution Found." You will need to use a priority queue in the A* algorithm. See the example in `priorityqueue_example.py` to see how to do this in python.

The action cost of moving from node n to node m should be:

$$c(n,m) = \sqrt{(n_x - m_x)^2 + (n_y - m_y)^2 + min(|n_\theta - m_\theta|, 2\pi - |n_\theta - m_\theta|)^2}.$$

The heuristic for node n should likewise be:

$$h(n) = \sqrt{(n_x - g_x)^2 + (n_y - g_y)^2 + min(|n_\theta - g_\theta|, 2\pi - |n_\theta - g_\theta|)^2},$$

where g is the goal configuration.

Hint: Remember to consider the topology of the θ DOF when expanding new nodes.

You do not need to collision-check edges but you must collision-check nodes. You will need to determine reasonable discretizations for each DOF (too small: the algorithm will be very slow, too large: the robot will go through obstacles).

Implement two variants of the A* algorithm:

a. "4-connected" neighbors

b. "8-connected" neighbors

For each variant record the the computation time to find a path and the path cost (using the action cost function defined above) in your `pdf`.

For each variant, also save a screenshot including:

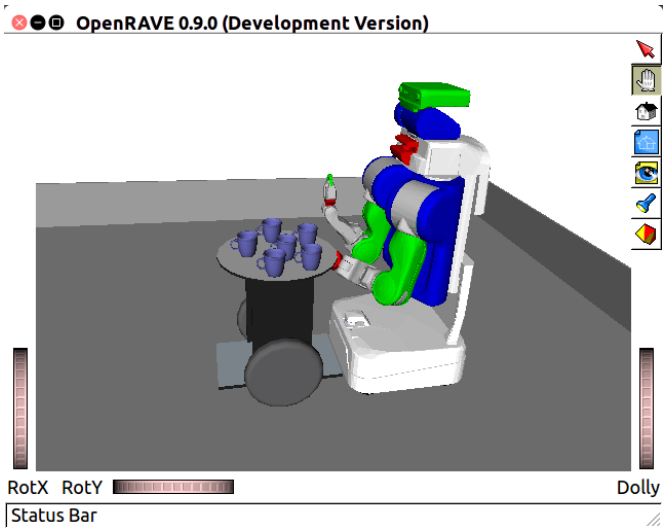
- The computed path drawn in the openrave viewer in black (of course you will not be able to draw the rotation along the path, so just draw the X and Y components of the configurations in the path).
- The X and Y components of the collision-free configurations explored by A* in blue.
- The X and Y components of the colliding configurations explored by A* in red.

Make sure to draw the points above the environment so that you can see them. Include these screenshots in your `pdf`.

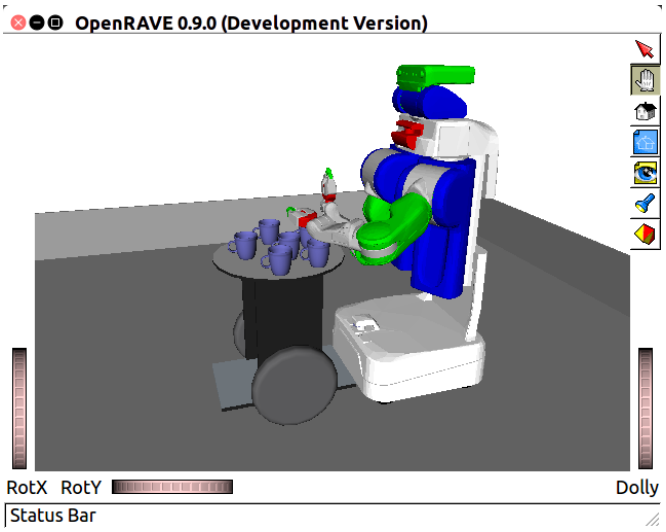
(50 points) To grade your work, we will run the command `python hw3_astar.py` in a folder where we have extracted your source code. We will not run any other command or any modifications of this command. When this command is run, your code should plan using variant (b) and execute a trajectory for the robot and we should see the robot following this trajectory in the openrave viewer. Your code should output the solution in under 10 minutes.

(10 points) Which variant performs better in terms of computation time? Which variant performs better in terms of path cost? Explain why in the `pdf`.

5. (40 points) Implement the RRT-Connect algorithm to search for collision-free paths in configuration space from the current configuration of the robot to a goal configuration. To speed up your code, you need only check collision between the robot and the environment; checking self-collision is not necessary. Put your code in `rrt_template.py` and do not modify code outside the indicated blocks for your code. Use a step size of 0.05rad and a goal bias of 10%. You will be planning for 6 DOF of the PR2 robot's left arm (the 7th DOF, which is the wrist roll joint, is not used).



(a) Start configuration



(b) Goal configuration

Once you have computed the path from start to goal

a. Draw the position of the left end-effector of the robot for every configuration along the path in red in the openrave viewer. You can use the `GetEETransform` function defined in the template. 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`.

b. Convert your path to a trajectory and execute it in the viewer. Verify that the arm does not go through any obstacles.

(20 points) Implement the shortcut smoothing algorithm to shorten the path. Use 150 iterations. Draw the original path of the end-effector computed by the RRT in red and the shortcut-smoothed path of the end-effector in blue in the openrave viewer. Include a screenshot showing the two paths in your `pdf`.

To grade your work, we will run the command `python rrt_template.py` in a folder where we have extracted your source code. When we run `python rrt_template.py`, we should see the execution of the smoothed path in the openrave viewer. We will not run any other command or any modifications of this command. We will run the code three times and your code should output the solution in under 10 minutes at least once.