Robot Autonomy - Homework 2

Professor: Oliver Kroemer

1. Introduction

This homework will focus on collision avoidance and motion planning discussed in lecture. You will first implement a collision detection algorithm, and then use it to determine if two cuboids are intersecting. You will then implement a Rapidly-exploring Random Tree (RRT) and a Probabilistic Roadmap (PRM), as discussed in the lectures, and use it to plan a collision-free path from a start to target joint configuration. Results from the motion planning algorithm will be visualized in the MuJoCo simulation environment.

2. MuJoCo Set Up (Updated for HW2 using venv)

Creating a python "venv"

To set up MuJoCo,

1. Navigate to the home directory
2. Create a python virtual environment. To do this, run "**`python3 -m venv ./<your_env_name>".`** Replace `your_env_name` with whatever you'd like.
3. Activate this environment by running the activate script within the bin folder of the venv path you provided, i.e. run "**`source ./<your_env_name>/bin/activate`**".
4. Within this environment, run "**`which python`**" and ensure that the python path is pointing to a location in the venv path as a sanity check.
5. Install dependencies using the environments pip package by running,
   a) "**`python -m pip install mujoco"`**
   b) **`"python -m pip install quaternion"`**
   c) **`"python -m pip install matplotlib"`**
6. You can now use the packages from the Python venv to run the homework files. Make sure to activate the venv when you run the files.

3. Cuboid-Cuboid Collision Detection

In the first part of the homework, we will implement a collision detection algorithm for use in the remainder of the homework. Collision avoidance is a critical aspect of robot motion planning, as we want our robots to operate safely around objects, other robots, and humans. A first-order approach for collision avoidance is to define cuboids around objects (also known as rectangular prisms) and detect if these cuboids intersect. In this context, these cuboids are also referred to as bounding boxes. For our collision detection algorithm, we recommend implementing the Separating Axis Theorem to determine when two cuboids intersect. We will only use cuboids to parameterize collision geometry in this assignment. For this assignment, we will parameterize a cuboid through specification of

1) the pose of its centroid, which serves as the cuboid local frame; and
2) the specification of the cuboid's lengths in the local x-, y-, and z-dimensions.

We use the standard roll-pitch-yaw convention for orientation, where the (r, p, y) tuple represents the angle about the body's x-axis (r, roll), y-axis (p, pitch), and z-axis (y, yaw), respectively). The function BlockDesc2Points(H, Dim) we provided to you in RobotUtil.py takes in an object pose (specified by a homogeneous transformation matrix, H) and object dimensions (length(x)/width(y)/height(z)) and returns the bounding box of the object and the axes given by the rotation matrix.

3.1 Part 1 - Implement cuboid-cuboid collision checker

To complete this part of the homework, please implement the Separating Axis Theorem. More specifically, please implement the functions CheckPointOverlap() and CheckBoxBoxCollision() in RobotUtil.py.

Evaluating cuboid-cuboid collision checker: To evaluate your collision checker, for the following cuboids in Table 1, provide a yes/no response for whether the test case cuboid is colliding with the reference cuboid. The reference cuboid is the cuboid defined as follows (i.e., with centroid pose coincident with the world frame):

Origin (m) (x, y, z): (0, 0, 0)

Orientation (rad) (r, p, y): (0, 0, 0)

Dimensions (m) (dx , dy , dz): (3, 1, 2)

Consider cases where cuboid faces are touching in the same plane to be a collision, as well as cases where one cuboid completely encloses another cuboid.

Test Case	Origin (m): (x,y,z)	Orientation (rad): (r,p,y)	Dimensions (m): (dx , dy , dz)
1	(0,1,0)	(0,0,0)	(0.8,0.8,0.8)
2	(1.5,-1.5,0)	(1,0,1.5)	(1,3,3)
3	(0,0,-1)	(0,0,0)	(2,3,1)
4	(3,0,0)	(0,0,0)	(3,1,1)
5	(-1,0,-2)	(.5,0,0.4)	(2,0.7,2)
6	(1.8,0.5,1.5)	(-0.2,0.5,0)	(1,3,1)
7	(0,-1.2,0.4)	(0,0.785,0.785)	(1,1,1)
8	(-0.8,0,-0.5)	(0,0,0.2)	(1,0.5,0.5)

Implement Robot Bounding Boxes: The Franka.py file has been expanded to include collision checking based on the RobotUtil.py functions that you just implemented. The constructor also includes additional parameters defining the arm's bounding boxes. We have given you the bounding boxes for each of the links of the Franka arm in the file Franka.py. The variable self.Cdesc contains the poses (xyz and roll/pitch/yaw) of each of the robot arm bounding boxes that you will use to define the transforms for each link. Make use of these bounding box descriptions and the associated link to forward transform the bounding box for each of the corresponding robot links.

The computation of the bounding boxes for a given joint configuration is not given. Complete the missing code in the function CompCollisionBlockPoints().

3.2 Submission

For the collision detection section, submit a pdf page with the following items:

• A list of yes/no responses for whether the cuboids in Table 1 are colliding with the reference cuboid.

• A picture of the robot's bounding boxes for when it is in its home configuration (all angles at 0). Use the PlotCollisionBlockPoints function in Franka.py.

4. Franka RRT Motion Planning

4.1 Implementation

In this section of the homework, you will implement a Rapidly-exploring Random Tree (RRT) motion planner in joint space. This planner should provide a feasible, collision-free path from an initial joint configuration to a given target joint configuration.

Joint Name	Initial	Goal
joint_1	-90°	90
joint_2	-90°	-90°
joint_3	90°	-90°
joint_4	-90°	-90°
joint_5	0	0
joint_6	150°	150°
joint_7	0	0

The motion planner should plan in joint space for joints 1 through 5. We are not concerned with moving the gripper fingers in this assignment, so PRM planning for joint 6 and joint 7 is not necessary.

You will use your cuboid collision detection code from the previous section of this homework to perform collision detection. This includes collisions with the obstacles in the scene and self collisions with the robot's base.

We have given you the bounding boxes for the obstacles in the scene, the robot base, and the camera mount within the RRTQuery.py file. We will consider the robot base and camera mount as obstacles in the environment since we will not be moving the robot base for the homework.

Implement RRT: You should implement your RRT planner to find a collision-free path in RRTQuery.py. This code provides you with the bounding boxes for all the obstacles present in the environment (pointObs and envaxes). As part of the implementation, I recommend expanding the tree using the RRT Connect approach and using a goal bias to guide the tree growth. The Franka.py class now has a function SampleRobotConfig for sampling configurations within the joint limits.

Visualizing RRT Solution: In order to visualize the solution, simply run the RRTQuery.py python file. You should be able to see the robot execute the plan in MuJoCo if a plan has been found.

Implement Path Shortening: The path found by the RRT planner may be a bit winding and unnecessarily long. In the RRTQuery.py file, once a plan has been found, implement path shortening. Remember to check collisions along any shortened paths that you introduce.

4.2 Submission

For the RRT section, submit the following items:

- A video of the Franka executing a plan output by the RRT without path shortening. You can use a screen capturing program or the video recording capabilities within V-REP. If your video is large, please provide a link to it through Google Drive, Dropbox, or some other cloud hosting service so that we may access it later.
- A video of the Franka executing a plan output by the RRT with path shortening.

5. Franka RRT Motion Planning

5.1 Implementation

In this section of the homework, you will implement a Probablistic Roadmap (PRM) motion planner in joint space. This planner should provide a feasible, collision-free path from an initial joint configuration to a given target joint configuration. The scene and query will be the same as the one used for the RRT portion of the homework.

Implement PRM: The PRM is implemented across two files: PRMGenerator.py and PRMQuery.py. The former is meant to be run to create and save the actual PRM, with prmVertices containing the joint configurations for each vertex and prmEdges containing the indices of connecting vertices for each vertex. You will need to modify the file PRMGenerator.py to create the PRM. It may take some tuning of the parameters, but I found a PRM of 1000 vertices to be suitable, with neighbours defined as vertices within a L2 norm distance in joint angles of 2 radians from each other. Note that generating the PRM took approximately 30 min on my laptop.

Querying and Visualizing PRM Plan: The PRMQuery.py file should be complete as it is. The file adds the query points to the graph and performs a best-first search to reach the goal state. It then runs MuJoCo to visualize the output. Note that the query code will always try to extract and execute a plan. If the visualized plan includes a collision with the obstacles, it may indicate that the PRM does not provide enough connectivity or coverage to find a suitable solution.

5.2 Submission

For the PRM section, submit the following item:

- A video of the Franka executing a plan output by the PRM.