Lab 2: Motion Planning

Robot Autonomy

Prof. Oliver Kroemer

0 Prerequisites

In this lab, you will implement RRT Connect and run it on the real robot to follow a path that avoids a static obstacle.

Hint: You can split up your lab group to work on Sections 1 and 2 in parallel. These two sections do not depend on each other.

0.1 Starting the Robot

Refer to the last lab for detailed instructions.

1. Turn on the robot.
2. Unlock the joints via launching a remote Firefox session on the Control PC.

0.2 Note on Safety

When running a program that commands the robot, please:

1. Stay outside of the robot's workspace (defined by the edges of the table).
2. **Always keep a hand on the e-stop. Under no circumstances should the e-stop be out of reach.**

1 Specifying the Planning Problem

1.1 Launch Control PC

1. Remote launch `franka-interface` on the Control PC by:

   `cd` to `~/Prog/frankapy` is on your machine.

   Run `bash bash_scripts/start_control_pc.sh -i iam-<name> -u student`

1.2 Finding today's lab

This lab uses both utils and lab2 in the robot-autonomy-labs folder.

1. Cd into todays lab:

   `cd` to `~/robot-autonomy-labs`.

1.3 Registering the Obstacle Box

We will first find the dimension and the pose of an obstacle in the workspace. This is usually done with vision, but a quick and easy way is to simply move the robot's end-effector to the vertices of the box obstacle to "register" its size and pose.

1. Place a cardboard box in the workspace like so:

Figure 1: Obstacle Placement

Once you've placed the box, be careful not to move it again! If the box is moved, you'd have to re-register it.

2. Run `python utils/close_gripper.py` to close the gripper.
3. Run `python utils/run_guide_mode.py` to record the positions of either pair of the top two diagonal corners of the box by placing the gripper tip near those corners.
4. Use the recorded positions to compute the dimensions, position, and rotation of the box in the workspace.

   Note:

   x-axis is coming out of the robot base toward the grippers in home pose, and z-axis is up.

   If the box is axis-aligned with the workspace, then rotation angles should all be 0.

   You can treat the bottom of the box 0 on the z-axis.
5. Replace the obstacle box in the `boxes` array in `RRTQuery.py`

1.4 Getting Start and Target Joint Configurations

We will replace `joints_start` and `joints_target` with new ones that are closer to the obstacle.

1. Use `run_guide_mode.py` to record the joint positions for when the end-effector is to the left and to the right of the box:

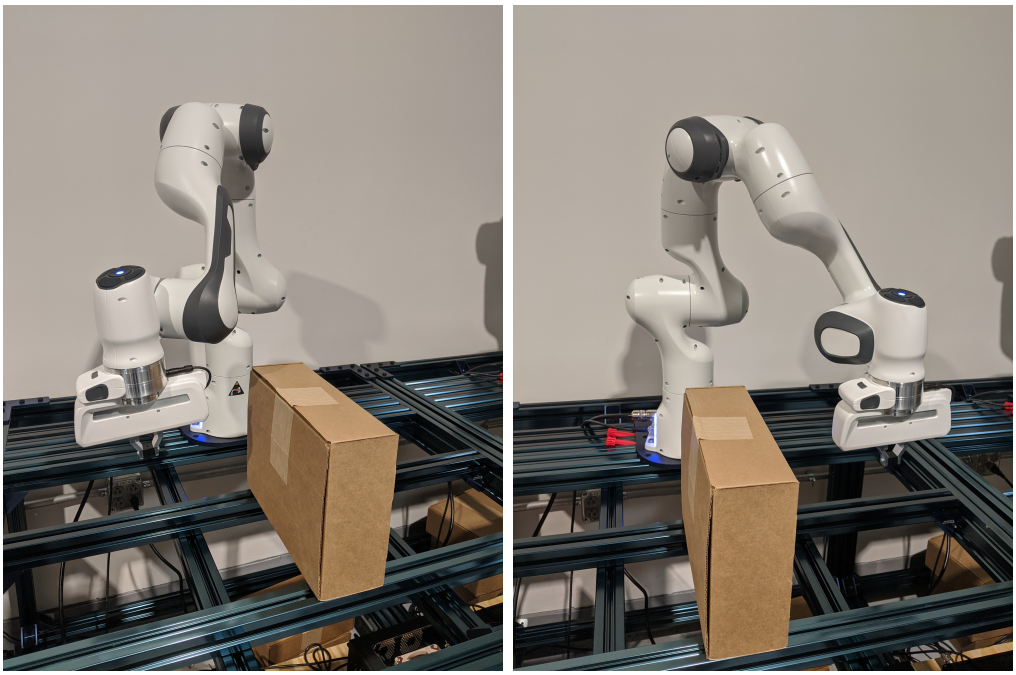

Figure 2: Start (left) and Target (right) Configurations

2. Replace `joints_start` and`joints_target` in `RRTQuery.py` with these recorded joints.

2 Implement RRT Connect

RRT Connect is often faster than RRT in practice because it extends an additional tree from the target node. The search alternates extending each tree until two nodes in the tree are close enough to be connected.

1. Fill in the `TODO` function in `RRTQuery.py`. See Algorithm 1 for details.
2. Run `python RRTQuery.py` to make sure your code runs and is able to find a path. Test this with both using the constraint and not using the constraint. Check the plan in rViz and make sure it looks reasonable. **Please keep a hand on the e-stop while the robot is moving!!!**

Algorithm 1 ConstrainedExtendRRTConnect

Require: T_0, T_1

while True **do**
  $q_{sample} \leftarrow$ SampleValidJoints()
  $q_{near} \leftarrow T_0$.Nearest(q_{sample})
  $q_{new} \leftarrow q_{near} + \min(\Delta q_{step}, \|q_{sample} - q_{near}\|_2) \frac{q_{sample} - q_{near}}{\|q_{sample} - q_{near}\|_2}$
  $q_{new} \leftarrow$ ConstraintProjection(q_{new})
  if InCollision(q_{new}) **then**
    continue
  end if
  T_0.AddVertex(q_{new})
  T_0.AddEdge(q_{near}, q_{new})
  $q_1 \leftarrow T_1$.Nearest(q_{new})
  if $\|q_{new} - q_1\|_2 < d_{connect}$ AND IsSegmentValid(q_{new}, q_1) **then**
    return True, q_{new}, q_1
  end if
  return False, q_{new}, q_1
end while

3 Aruco Tags

1. Launch the azure kinect ros node:
   `roslaunch azure_kinect_ros_driver driver.launch`

2. Launch the aruco tag ros node:
   `roslaunch ~/robot-autonomy-labs/lab2/aruco.launch`

3. Run the command `rosrun rqt_image_view rqt_image_view` and select `/aruco_simple/result` in the drop-down menu in the top left of the window to verify the aruco tag is being detected. The pose of the aruco tag in the camera frame is published on the ROS topic: `/aruco_simple/pose`

4. Add your own aruco state to /Prog/iam-interface/iam-domain-handler/examples/run-state-server.py. You will need to subscribe to the appropriate topic and write your own aruco handler (hint: use the "robot_state_handler()" function for reference). This will be used by the `lab2_custom_behvaior.py` script you edit in step 6.

5. Update the paths AZURE_KINECT_INTRINSICS and AZURE_KINECT_EXTRINSICS in /Prog/iam-interface/iam-domain-handler/examples/lab2_custom_behavior.py (if needed)

6. Alter `lab2_custom_behavior.py` to interact with the block (handle the TODOs).

7. Now close all the terminals and start the iam-interface with:
   `cd ~/iam-construct/`

   `./iam-construct-run.sh iam-<robot-name>.`

8. Start the aruco_ros node again with the command:

   `roslaunch ~/robot-autonomy-labs/lab2/aruco.launch`

9. Navigate to the web-page with the url: `<pc-ip>:9080/javascript/example/index.html`

10. Start the lab2 custom behavior script using the commands:

    `cd ~/Prog/iam-interface/iam-domain-handler/examples/`

    `python lab2_custom_behvaior.py`

11. (optional) Use another aruco tag to add functionality to the script, or combine the RRT planner with the aruco tag(s).

4 Turning Off the Robot

Turn off the robot as instructed in the previous lab:

1. Reset joints
2. Press down on the e-stop to return robot to manual mode (white)
3. Click shutdown on the web interface.
4. Wait until the robot shuts down, the flip the switch on the FCI.