EE106A: Lab 7 - Path Planning*

Fall 2022

Goals

By the end of this lab, you should be able to:

- Use MoveIt to plan paths using the ROS action server and the MoveIt Commander Wrapper.
- Plan and execute paths with obstacles and orientation constraints on Sawyer.
- Understand the difference between open-loop and closed-loop control.
- Implement a trajectory-tracking controller on Sawyer.

Contents

1 Planning with MoveIt **2**
- 1.1 Using the MoveIt GUI 2
  - 1.1.1 Basic planning 2
- 1.2 Using the action server interface 3
  - 1.2.1 Test a simple action client 3

2 Planning on the Sawyer robot **4**
- 2.1 Path planning 4
- 2.2 Back to MoveIt Commander 4
  - 2.2.1 Planning with obstacles 5

3 Controlling Sawyer **6**
- 3.1 Feed-Forward Control 6
- 3.2 Closed Loop Control 7

Introduction

In this lab, you'll explore two heavily-connected fields of robotics: path planning and control.

In Lab 5, you explored the inverse kinematics functionality available for the Baxter/Sawyer robot. Inverse kinematics can produce joint configurations that position a robotic manipulator at a specified end effector position, but it doesn't tell us how to *move* the manipulator between a specified start and end position. The problem of choosing a joint trajectory that moves a manipulator between a given start and end configuration while obeying a set of constraints is the *path planning* problem.

*Developed by Jewook Ryu and Aryaman Jhunjhunwala Fall 2022, Valmik Prabhu, Philipp Wu, Nandita Iyer, and Ravi Pandya, Fall 2018, referencing material from Aaron Bestick and Austin Buchan, Fall 2014.

In Lab 3, you moved Sawyer's arm between two positions by simply changing each of the joint angles linearly from the start to the end state. While this worked for a very simple case, it's generally not a good strategy, since it doesn't give us any control over the path the manipulator takes. In particular, there is no way to ensure the manipulator avoids collisions with itself or other objects in the environment. Path planning algorithms attempt to solve this problem.

The MoveIt package you used for inverse kinematics in Lab 5 also includes a variety of powerful path planning functionality — in fact, to move between your specified end effector positions, MoveIt was using this functionality behind the scenes. In this lab, you'll get acquainted with path planning on an actual Sawyer robot.

When a robot is given a path, it still needs to execute it in the real world. This is the domain of control. In Lab 4, you implemented a simple proportional controller to direct the turtlebot to an AR tag. In this lab, you'll be implementing a PID controller, the most-used controller in industry, to make the robot's end effector track the trajectory.

1 Planning with MoveIt

MoveIt's path planning functions are accessible via ROS topics and messages, and a convenient RViz GUI is provided as well. In this section, you'll learn how to use MoveIt's planning features via both of these interfaces.

1.1 Using the MoveIt GUI

In this section, you'll use MoveIt's GUI to get a basic idea of what types of tasks path planning can accomplish.

1. Begin by creating a new workspace called `lab7` in your `ros_workspaces` directory. Add a `src` directory inside `lab7`.

2. Next, inside `lab7/src`, create a package named `planning` that depends on `rospy`, `roscpp`, `std_msgs`, `moveit_msgs`, `geometry_msgs`, `tf2_ros`, `baxter_tools`, and `intera_interface`.

Our starter code for this lab can be found at https://drive.google.com/drive/folders/1PLx3uvtRoFSmoXv4JsOrnGoWJO9QilC
We also highly recommend you make a private GitHub repository for each of your labs just in case.

1. If you already have this repo cloned, you can simply navigate to it and run `git pull` to grab the latest updates.

   (a) If not, you can first clone it by running

```
git clone https://github.com/ucb-ee106/106a-fa22-labs-starter.git
```

2. Inside `planning`, create a directory called `launch` and copy the `baxter_moveit_gui_noexec.launch` file from the starter code into the new folder. Don't forget to build and source your workspace!

1.1.1 Basic planning

1. Use `roslaunch` to run `sawyer_moveit_gui_noexec.launch`. The MoveIt GUI should appear with a model of the Sawyer robot. If the robot position doesn't seem accurate, hit "Reset" in the lower left corner.

2. In the Displays menu, look under "MotionPlanning" $\implies$ "Planning Request" and check the "Query Start State" and "Query Goal State" boxes to show the specified start and end states.

3. Set the start and goal states for the robot's motion by dragging the handles attached to the end effector. The start state will show in green, and the end state will show in orange.

4. Switch to the "Planning" tab and click "Plan." The planner will compute a motion plan, then display the plan as an animation in the window on the right.

5. If you want to see the complete path of the arm to be displayed, go to the Displays menu, under "Motion Planning" $\implies$ "Planned Path" and select the "Show Trail" option.

Note that execution will not work (for now), because execution has been disabled in the launch file. Later in the lab, when we're working on the real robot, you'll be able to execute paths through the GUI as well.

1.2 Using the action server interface

The MoveIt GUI provides a nice visualization of the solutions computed by the planner, but in a real world system, the start and end states would likely be generated by another ROS node. MoveIt's ROS interface allows you to use the same `move_group` node you used for inverse kinematics in Lab 5 to define environments and plan and execute trajectories on a real robot.

Planning and executing a trajectory would be difficult to coordinate using simple topics and services. Therefore, The planning functionality of the `move_group` node uses a third type of communication within ROS known as an *action server*. The ROS website provides a detailed description.

The `move_group` action server interface allows us to plan and execute trajectories, as well as monitor the progress of a trajectory's execution and even stop the trajectory before it completes.

An action server and client exchange three types of messages as a request progresses:

- **Goal:** Specifies the goal of the action. In our case, the goal includes the start and end states for a motion plan, as well as any constraints on the plan (like obstacles to avoid).
- **Result:** The final outcome of the action. For a motion plan, this is the trajectory returned by the path planner, as well as the actual trajectory measured from the robot's motion.
- **Feedback:** Data on the progress of the action so far. For us, this is the current position and velocity of the arm as it moves between and start and end states.

These messages are exchanged over several topics, as shown in Figure 1. You can use `rostopic echo` to view the topics as you would with normal messages.

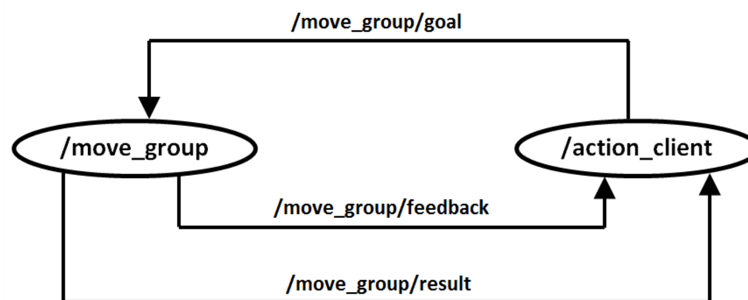

Figure 1: Action server topics

The data contained in these three message types is specified by a `.action` file:

1. Navigate to the `moveit_msgs` package.
2. `cd` into the `/action` subdirectory.
3. Open and examine `MoveGroup.action`

1.2.1 Test a simple action client

1. From the lab7 starter files, move `action_client.py` into the `/src` subdirectory in your `planning` package.
2. Launch `baxter_moveit_gui_noexec.launch` to start the `move_group` node, followed by `python action_client.py`.

   The second file should request a motion plan from the action server, then print the result to the terminal. You should also be able to see an animation of the plan in RViz.
3. `ctrl-c` out of `python action_client.py`.

Modify `action_client.py` so that you can input the start and goal states for the manipulator at the terminal, rather than having them hard coded into the program (This is very similar to what you did in lab 5). Since there are a lot of angles, just make it so you can input **the first four joint angles of both the start and goal states**.

Test this with several combinations of joint angles and comment on the results. Does the planner always return the same result given identical requests? If not, do you have any ideas why?

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/fa22-106alab. At this point, you should be able to:

- Plan MoveIt trajectories using both the GUI and action server interfaces.
- View and explain the contents of each action server topic
- Discuss whether the paths found by the MoveIt planner are repeatable and if not, why not

---

2 Planning on the Sawyer robot

As previously mentioned, MoveIt allows you not only to plan paths, but to execute those paths on a real robot. In this section, you'll test this feature on Sawyer.

2.1 Path planning

Create a symlink to `intera.sh` by running

```
ln -s /opt/ros/eecsbot_ws/intera.sh [path-to-workspace]
```

Complete the following steps to run MoveIt on Sawyer. (Remember to run `intera.sh` in each terminal window that will be interacting with the robot.)

1. Sawyer by running

```
rosrun intera_interface enable_robot.py -e
```

2. Start the Sawyer trajectory controller by running

```
rosrun intera_interface joint_trajectory_action_server.py
```

3. Start MoveIt for Sawyer by running

```
roslaunch sawyer_moveit_config sawyer_moveit.launch electric_gripper:=true
```

MoveIt is now ready to compute and execute trajectories on the robot.

2.2 Back to MoveIt Commander

In Part 1 of this lab, you used MoveIt's action server interface directly to compute and execute paths between specified forward-kinematic positions. While this is valuable for understanding what's going on "under the hood," it's cumbersome for normal use For this part of the lab, we'll use the same `moveit_commander` interface as in Lab 5, which provides a simple wrapper around the MoveIt action server.

Copy `path_test.py` and `path_planner.py` from the lab 7 starter to `planning/src`, and examine the two files. Rather than putting all the base MoveIt code into the `path_test.py` script, we have encapsulated it into the `PathPlanner` class inside `path_planner.py`. Make sure to understand both pieces of code before continuing.

Make sure that `path_test.py` is an executable. Run `path_test` using:

```
rosrun planning path_test.py
```

The robot should loop through three poses in series. As the arm moves, pay attention to the orientation of the right gripper. Does it remain at the same orientation throughout the motion?

While end effector orientation during a manipulator's motion is sometimes unimportant, in other cases it can be critical. Examples include moving liquid-filled containers and performing "peg-in-hole" insertion tasks. MoveIt allows you to plan paths with orientation constraints using the same interface as before.

To test this, you'll need to edit `path_test.py`. Edit the file to send an orientation constraint to the planner (uncommenting the code is part, but not all of what you'll have to do). Once you've finished, run the script. Do you see any differences in how the end effector moves? You might also notice that MoveIt fails to find a feasible path more often than before. Why is this?

Once you have this working, run

```
rostopic echo /move_group/feedback
```

while you execute a trajectory to view the feedback produced by the MoveIt action server while the trajectory is executing. What data is included in the feedback messages?

2.2.1 Planning with obstacles

As mentioned in the introduction, path planning algorithms can also solve the problem of planning with obstacles present in the environment around the robot. The `PathPlanner` class contains the `add_box_obstacle` function, which adds a box-shaped obstacle to the planning scene. While it's possible to add more complex shapes, including shapes from STL or OBJ files, it's rarely worth doing so, as more complex geometries will simply increase computation time.

Edit `path_test.py` to add a box representing the table to your scene. For the pose, you may want to try something near

```
X = 0.5, Y = 0.00, Z = 0.00
X = 0.00, Y = 0.00, Z = 0.00, W = 1.00
```

and for the size you might want to try

```
X = 0.40, Y = 1.20, Z = 0.10
```

. You should see a green box appear in the RViz window, at the position you create the object. Now create an artificial "wall" in the air near Baxter or Sawyer's arm.

MoveIt now has a model of both the robot itself and any obstacles around the robot with which computed motion plans must avoid collisions. Try planning a trajectory that moves the robot's arm from one side of the wall to the other. You may want to use the GUI to make it faster. Observe how the computed motion plan changes to avoid the obstacles.

Follow these steps to create a new wall obstacle and plan and execute a path over it:

1. `ctrl-c` out of `rosrun planning path_test.py` and `rviz`
2. Run this to restart `rviz`

   ```
   roslaunch sawyer_moveit_config sawyer_moveit.launch electric_gripper:=true
   ```

3. Switch to "Scene Objects"
4. Determine the size of the object (x,y,z)
5. Press the green '+' button to add the object to the scene
6. Edit the position of the object (x,y,z). You can drag using the GUI as well
7. When you're satisfied with the wall position, click publish

   Note: Before you click publish, make sure that the robot arm is not inside the object you created. You can use the zero-g mode to manually move the arm's position.

8. Switch to the ’Planning’ tab
9. Drag the end effector to the other side of the wall
10. Press Plan, inspect the path to make sure it’s reasonable, and the press Execute

---

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/fa22-106alab. At this point, you should be able to:

- Plan a path with and without orientation constraints
- Identify situations in which path orientation constraints might be useful
- Plan a path that avoids a wall obstacle

---

3 Controlling Sawyer

Now you’ll be replacing MoveIt’s default execution with a controller of your own.

1. Copy `controller.py` from the lab 7 starter files to `planning/src`.
2. Examine the file. `controller.py` implements an open-loop, or feed-forward, velocity controller to follow a provided path (a `moveit_msgs/RobotTrajectory` message, which is returned by MoveIt’s plan method).

You’ll be modifying this file to implement a PD (proportional derivative) and a PID (proportional integral derivative) controller on the robot.

The robot trajectory contains not only the desired position at each point, but also the desired velocity (as the number of path waypoints goes to infinity, this becomes the derivative of the desired position). An open loop velocity controller simply sets the manipulator’s velocity as the desired velocity at the current point in the path.

If the distance between each waypoint in the path is small, and robot executes each command perfectly, we would expect to see the robot’s actual trajectory exactly equal the desired trajectory. However, the world is rarely perfect, and the robot must deal with environmental disturbances, time delays, inaccurate sensors, and imperfect actuators. This is why engineers generally implement closed-loop, or feedback, control. Let’s first analyze the performance of the feed-forward controller.

3.1 Feed-Forward Control

Edit `path_test.py` to use the controller instead of MoveIt’s default execution, then test out the controller. **Warning: Be especially vigilant on the E-Stop when running code with a custom controller. You lose many of the safety and reliability guarantees provided by the internal implementations**. Note that hitting `ctrl-C` should also stop any movement of the robot, but if the robot looks like it will collide with something, use the E-stop first. Also, be sure to check the path in RViz before running hitting Enter. You may want to disable your orientation constraints and move the tables out of the way as well.

After each execution, the controller displays a plot of each joint value over time, with the target in gold and the measured value in blue. How does the performance look in the plot? Use `tf` (either `tf_echo` or do it programmatically) to check the final end effector pose against the goal. How does the open-loop controller compare to the built-in controller?

3.2 Closed Loop Control

Now you'll be implementing closed loop control on these robots. PID (proportional integral derivative) control is ubiquitous in industry because it's both intuitive and broadly applicable. You'll be learning about PID control in class these next few lectures. The control law is:

$$u = u_{ff} + K_p e + K_i \int_0^t e\,dt + K_d \dot{e} \tag{1}$$

where e is the state error $q_d - q$ (where q_d is the desired position/joint angles and q the current position/joint angles) and u_{ff} is the feedforward term (desired velocity). The controller consists of a proportional term which pulls the error towards zero, a derivative term which damps the controller and reduces oscillation, and an integral term which compensates for constant error sources (like gravity).

First, edit `controller.py` to implement a *PD* controller (ignore the integral term) and execute it on the robot. You should be using the gains specified in `path_test.py` and should not need to change them. How do the plots change from when you used the open-loop controller? What could still be improved?

Finally, add the integral term to implement a PID controller and execute it on the robot. Once again, you should not need to change any of the gains. How do the plots change?

Checkpoint 3

Submit a checkoff request at https://tinyurl.com/fa22-106alab. At this point, you should be able to:

- Explain what `controller.py` does. Your TA will point out and ask you to explain individual code blocks.
- Execute paths using the PID controller.
- Discuss the performance of the open loop, PD, and PID controllers against the default controller.
- What do you think K_w is for? Why is it needed?