EE106A: Lab 5 - Inverse Kinematics*

Fall 2018

Goals

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

- Use MoveIt to compute inverse kinematics solutions for both of Baxter's arms
- Create a visualization of Baxter's kinematic structure, as defined in the URDF file
- Open and close Baxter's grippers programmatically
- Use MoveIt to move Baxter/Sawyer's gripper(s) to a specified pose in the world frame and perform a rudimentary pick and place task.

Contents

Introduction

In Lab 3, you investigated the *forward kinematics* problem, in which the joint angles of a manipulator are specified and the coordinate transformations between frames attached to different links of the manipulator are computed. Often, we're interested in the *inverse* of this problem: Find the combination of joint angles that will position a link in the manipulator at a desired location in $SE(3)$.

It's easy to see situations in which the solution to this problem would be useful. Consider a pick-and-place task in which we'd like to pick up an object. We know the position of the object in the stationary world frame, but we need the joint angles that will move the gripper at the end of the manipulator arm to this position. The *inverse kinematics* problem answers this question.

This lab has two parts. In Part 1, you'll learn how to use ROS's built in functionality to compute inverse kinematics solutions for a robot. In Part 2, you'll use inverse kinematics to program Baxter/Sawyer to perform a simple manipulation task.

*Developed by Aaron Bestick and Austin Buchan, Fall 2014. Modified by Laura Hallock and David Fridovich-Keil, Fall 2017. Modified by Valmik Prabhu, Nandita Iyer, Ravi Pandya, and Phillip Wu, Fall 2018

1 Inverse Kinematics

An inverse kinematics solver for a given manipulator takes the desired end effector configuration as input and returns a set of joint angles that will place the arm at this position. In this section, you'll learn how to use ROS's built-in inverse kinematics functionality.

1.1 Specify a robot with a URDF

While the `tf2` package — which you examined in Lab 3 — is the de facto standard for computing coordinate transforms for forward kinematics computations in ROS, we have several options to choose from for inverse kinematics. We'll use a package called MoveIt, which provides an inverse kinematics solver and other motion planning functionality.

Both MoveIt and `tf2` are generic software packages that can work with almost any robot. This means we need some method by which to specify a kinematic model of a given robot. The standard ROS file type for kinematic descriptions of robots is the Universal Robot Description Format (URDF). The URDF file for Baxter is contained in the `baxter_description` package. Open the URDF (`baxter.urdf`) and take a look at the data it contains.

Task 1: It's hard to visualize the actual robot by staring at an XML file, so ROS provides a tool that creates a more informative kinematic diagram. Navigate to the folder containing the Baxter URDF, copy the file to your home directory (since you don't have write permissions in its original location), and run

```
check_urdf baxter.urdf
urdf_to_graphiz baxter.urdf
```

on it. Open the PDF file that this command creates. Some of the nodes in the diagram should look familiar from the last lab, but some are new. What do you think the blue and black nodes represent? In addition to Baxter's limbs, what are some of the other nodes included in the URDF, and how might this information be useful if you want to use Baxter's sensing capabilities?

1.2 Compute inverse kinematics solutions

To use MoveIt, you must first start the `move_group` node, which offers a service that computes IK solutions. The file `move_group.launch`, in `lab5_resources.zip`, loads the URDF description of Baxter onto the ROS parameter server, then starts the `move_group` node.

First, create a new workspace called `lab5` and add a package called `ik` with dependencies on `rospy`, `moveit_msgs`, and `geometry_msgs`. Save `move_group.launch` in the new package's `/launch` subdirectory. Then run the launch file with `roslaunch` to start the MoveIt node. MoveIt is now ready to compute IK solutions.

Task 2: Write a node that prompts the user to input (x, y, z) coordinates for an end effector configuration, then constructs a `GetPositionIK` request, submits the request to the `compute_ik` service offered by the `move_group` node, and prints the vector of joint angles returned to the console. You can use the file `service_query.py` in the resources folder as a template for your node. A couple of tips:

1. The `GetPositionIK` service takes as input a message of type `PositionIKRequest`, which is complicated. The most important part is the `pose_stamped` field, which specifies the desired configuration of the end effector.

2. The orientation of the end effector is specified as a quaternion. Since we're not worried about rotation, you can set the four orientation parameters to any values such that their norm is equal to 1. For reference, a value of $(0.0, \pm 1.0, 0.0, 0.0)$ will have Baxter's grippers pointing straight down.

3. Sawyer robots only have a `right_arm` and the gripper's frame is called `right_gripper_tip`. Baxter robots have both a left and right arm, and the gripper frame is called either `left_gripper` or `right_gripper` depending on which arm you use.

Test the IK solutions from MoveIt by plugging the returned joint angles into your Lab 3 forward kinematics function and verifying that you get the originally specified end effector position.(*Note:* you do not need to write a publisher or subscriber for this part, just plug in the joint angles to your lab 3 code manually) Does the IK solver always give the same output when you specify the same end effector position? If not, why not? Any ideas why the solver sometimes fails to find a solution?

Checkpoint 1

Get a TA to check your work. At this point you should be able to:

- Explain what the different parts of the graphical representation of Baxter's URDF file represent
- Validate the output of the MoveIt IK service by solving IK for three different poses, plugging these returned joint angles back into your Lab 3 forward kinematics function, and verifying that you get the original pose back
- Explain whether the solutions found by the MoveIt IK service are unique, and if not, why not

2 Open loop manipulation with Baxter/Sawyer

In this section, you'll use inverse kinematics to program Baxter/Sawyer to perform a simple manipulation task.

First, add a `mv` package to your `lab5` workspace. (Include `baxter_tools` or `intera_interface` along with the standard `rospy`, `roscpp`, and `std_msgs` dependencies.) Run `catkin_make` for your new workspace. At the root of your workspace (the `lab5` directory), create a symbolic link to the `baxter.sh` script by running

```
ln -s /scratch/shared/baxter_ws/baxter.sh ~/ros_workspaces/lab5
```

Now, position the table in front of Baxter/Sawyer within the reachable workspace. Find an object that will fit in Baxter/Sawyer's gripper, and place it somewhere on the surface. Connect to Baxter/Sawyer by running the `baxter.sh` file, as in Lab 3, then enable Baxter's joints by running

```
rosrun baxter_tools enable_robot.py -e
```

or Sawyer's joints by running

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

(Baxter/Sawyer may already be enabled, in which case this command will do nothing.) Next, echo the `tf` transform between the robot's base and end effector frames by running

```
rosrun tf tf_echo base [gripper]
```

where `[gripper]` is `right_gripper_tip` if working with Sawyer, or `left_gripper` or `right_gripper` if working with Baxter.

With the joints enabled, grasp the sides of Baxter/Sawyer's wrist, placing the arm in a gravity-compensation mode, where it can be moved easily by hand. (*Note*: If you have never used gravity compensation mode and are having trouble manipulating the robot, **ask an instructor for assistance**. You shouldn't have to use much force to move the robot around!) Move the arm to a position where it could grasp the object on the table and record the translation component of the `tf` transform. (*Hint*: If you're using Baxter, you can try the command "`rosrun baxter_tools tuck_arms.py -u`" to place Baxter in a good starting configuration before placing the gripper near the object.) Next, move the arm to a different position on the table where you'd like to set the object down and record the transform of this location as well. We recommend trying to have Baxter/Sawyer's gripper pointing straight down in both of these configurations.

After recording the positions to which you'd like to move the arm, you'll use the IK and path planning functionality of MoveIt to move Baxter/Sawyer's arm between different poses. Run Baxter's joint trajectory controller with the command

```
rosrun baxter_interface joint_trajectory_action_server.py
```

or Sawyer's using

```
rosrun intera_interface joint_trajectory_action_server.py
```

Next, in a new window, start MoveIt with

```
roslaunch baxter_moveit_config demo_baxter.launch right_electric_gripper:=true
        left_electric_gripper:=true
```

for Baxter, or

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

for Sawyer, omitting the last argument(s) if your robot does not have a gripper. On Baxter, the command above may fail depending on the types of attached grippers; if so, try

```
roslaunch baxter_moveit_config baxter_grippers.launch
```

instead. This command also opens an RViz window; ignore it for now.

Locate the `ik_example.py` file (from `lab5_resources.zip`), place it in the `src` directory of your package, make it executable, and build your workspace. Then source the `devel/setup.bash` file. `ik_example.py` contains an example of using MoveIt to move Baxter's left arm to a specified pose. Open the file and examine it. Change all relevant statements to describe the arm you are controlling (e.g., if you are controlling Sawyer's right wrist, ensure that the relevant lines reference `right_arm` and `right_wrist`.)

Run the example with:

```
rosrun mv ik_example.py
```

The file should prompt you to press enter; it will then move the specified gripper to the position (0.5, 0.5, 0.0) with orientation (0.0, 1.0, 0.0, 0.0) (gripper pointing straight down). Press enter a few times and pay attention to Baxter/Sawyer's movements, moving the arm in compliant gravity compensation mode between trials. Now, try uncommenting the line that only specifies a position target (with no orientation specified). Run this node again and observe the difference in behavior. Which approach is more predictable? You may want to move the table while experimenting with this behavior.

2.1 Grippers

It's easy to operate Baxter/Sawyer's grippers programatically as part of your motion sequence. Open the file `gripper_test.py` and (un)comment the appropriate `import` statement so that you are only importing the gripper class from either the `intera_interface` package or the `baxter_interface` package. Try this by running

```
rosrun planning_baxter gripper_test.py
```

which calibrates and then opens and closes Baxter/Sawyer's right gripper.

Task 3: Make a copy of the `ik_example.py` file in the `mv/src/` directory. Modify your file so that it moves the arm through the series of poses that you recorded earlier using `tf_echo` and attempt to perform a pick and place. You should use code from `gripper_test.py` to open and close the grippers at an appropriate time. Can you make the arm return to the same position, repeatedly, with good enough accuracy to pick up an item?

After completing this task, you might have noticed that open loop manipulation is, in general, difficult. In particular, it's hard to estimate the position of the object accurately enough that the arm can position the gripper around it reliably. Do you have any ideas about how we might use data from the additional sensors on Baxter to perform manipulation tasks more reliably?

Checkpoint 2

Get a TA to check your work. At this point you should be able to:

- Use `tf_echo` to get the current end effector pose for Baxter/Sawyer's arm
- Move Baxter/Sawyer's arm to specified poses using the `moveit_commander` interface
- Perform a pick and place task. This doesn't have to work perfectly (or at all), but at least make an attempt. If pick and place doesn't work, what could you do to improve it?
