Lab 5: Inverse Kinematics*

EECS/ME/BIOE C106A/206A Fall 2022

Goals

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

- Use MoveIt to compute inverse kinematics solutions
- Create a visualization of Sawyer's kinematic structure, as defined in the URDF file
- Open and close Sawyer's grippers programmatically
- Use MoveIt to move 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.

*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 Getting started with Git

Create a new workspace called `lab5` in your `ros_workspaces` directory. Refer to Lab 1 if you need a refresher on how to do this.

Our starter code for this lab is on GitHub for you to clone so that you can easily access any updates we make to the starter code. It can be found at https://github.com/ucb-ee106/106a-fa22-labs-starter. You can clone it by running

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

We also highly recommend you make a **private** GitHub repository for each of your labs just in case.

2 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.

2.1 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. For the first task, we'll use an external tool called Position Kinematics, which offers an inverse and forward kinematics service. You can find these two services by running `rosservice list` and look for:

```
/ExternalTools/right/PositionKinematicsNode/FKService
/ExternalTools/right/PositionKinematicsNode/IKService
```

Later in the lab, we will also use MoveIt to control the Sawyer. 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. There are two ROS file types for kinematic descriptions of robots, the Universal Robot Description Format (URDF) and Xacro (introduces the XML macro language to URDF files). Xacro is the more modern one as it describes the same information more cleanly. We will not examine these xacro files in this lab, but we encourage you to look through them if you plan to write your own URDF files for your final project. The Xacro file for Sawyer is contained in the `sawyer_description` package.

Note: The entire Sawyer SDK, including the `sawyer_description` package, is in the `/opt/ros/eecsbot_ws/src/sawyer_robot/sawyer_description` directory. To find and inspect the Xacro file:

```
roscd sawyer_description
```

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. In a convenient folder, run

```
check_urdf <(xacro
↪  /opt/ros/eecsbot_ws/src/sawyer_robot/sawyer_description/urdf/sawyer.urdf.xacro)

urdf_to_graphiz <(xacro
↪  /opt/ros/eecsbot_ws/src/sawyer_robot/sawyer_description/urdf/sawyer.urdf.xacro)
```

Note: These are two total lines of commands. They are too long, so they must be pushed on to the next line.

Open the PDF file that this command creates. What do you think the blue and black nodes represent? In addition to Sawyer'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 Sawyer's sensing capabilities?

2.2 Compute inverse kinematics solutions

Create a package called `ik` with dependencies on `rospy`, `moveit_msgs`, and `geometry_msgs`. Move the `ik.py` and `fk.py` inside the src folder of your package.

Task 2: Edit `ik.py` to prompt the user to input (x, y, z) coordinates for a gripper configuration, then constructs an IK request and submits the request to the `IKService` service offered by the `PositionKinematics` node, and prints the returned vector of joint angles in the terminal. Then, edit `fk.py` to prompt the user to input a comma-separated list of 7 joint angles and submits the request to the `FKService` service offered by the `PositionKinematics` node. A couple of tips:

1. The `IKService` service takes as input a message of type `SolvePositionIKRequest`, which is complicated. The most important part is the `pose_stamped` field, which specifies the desired configuration of the gripper.
2. The orientation of the gripper 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 Sawyer's grippers pointing straight down.

Test the IK solutions by plugging the returned joint angles into your FK node. Does the transformation match your originally specified gripper position and orientation? Why or why not?

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

Submit a checkoff request at https://tinyurl.com/fa22-106alab for a staff member to come and check off your work. At this point you should be able to:

- Explain what the different parts of the graphical representation of Sawyer's xacro.URDF file represent
- Validate the output of the Position Kinematics IK service by solving IK for three different poses, plugging these returned joint angles back into your Position Kinematics FK client node, and verifying that you get approximately the original pose back
- Explain whether the solutions found by the Position Kinematics IK service are unique, and if not, why not

---

3 Open loop manipulation with Sawyer

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

First, add a `move_arm` package to your `lab5` workspace. (Include `intera_interface` along with the standard `rospy` and `std_msgs` dependencies.) Run `catkin_make` to build your new package.

3.1 Setting up your environment for Rethink robots

Make a shortcut (symbolic link) in the root of your catkin workspace (the `lab5` directory) to the Sawyer environment script `/scratch/shared/baxter_ws/intera.sh` using the command

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

Connect to Sawyer by running the `intera.sh` file as you did in Lab 3, then enable the robot by running

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

(The Sawyer may already be enabled, in which case this command will do nothing. **Note: if you push the e-stop, you will have to run this before moving the arm again.**)

3.2 Recording pick-and-place positions

Now, position the table in front of Sawyer within the reachable workspace. Find an object that will fit in Sawyer's gripper, and place it somewhere on the surface.

You can get the transformation from the base to the gripper by running

```
rosrun tf tf_echo base right_gripper_tip
```

With the robot enabled, grasp the sides of Sawyer's wrist, placing the arm in zero-g mode, where it can be moved easily by hand. (*Note*: If you have never used zero-g 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*: You can try the command "`roslaunch intera_examples sawyer_tuck.launch`" to place Sawyer in a good starting joint 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 Sawyer's gripper pointing straight down in both of these configurations.

3.3 Moving the arm

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 Sawyer's arm between different poses. Run Sawyer's joint trajectory controller with the command

```
rosrun intera_interface joint_trajectory_action_server.py
```

Next, in a new window, start MoveIt with

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

omitting the last argument(s) if your robot does not have a gripper. This command also opens an RViz window; ignore it for now.

Locate the `ik_example.py` file (from the starter code), place it in the `src` directory of your `move_arm` package, make it executable, then build and source your workspace. `ik_example.py` contains an example of using MoveIt to move the Sawyers's arm to a specified pose. Open the file and examine it.

Run the example with (Don't forget to ssh into the robot!):

```
rosrun move_arm 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 Sawyer's movements, moving the arm in compliant zero-g 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.

3.4 Grippers

It's easy to operate Sawyer's grippers programatically as part of your motion sequence. Copy the file `gripper_test.py` to the `move_arm/src/` directory. Try this by running

```
rosrun move_arm gripper_test.py
```

This script calibrates and then opens and closes Sawyer's right gripper.

Task 3: Make a copy of the `ik_example.py` file in the `move_arm/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

Submit a checkoff request at https://tinyurl.com/fa22-106alab for a staff member to come and check off your work. At this point you should be able to:

- Use `tf_echo` to get the current end-effector pose for Sawyer's arm
- 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?
