Lab 5: Inverse Kinematics*

EECS/ME/BIOE C106A/206A Fall 2023

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

1 Introduction

In Lab 3, you investigated the **forward kinematics** problem: given the joint angles of a manipulator, compute the coordinate transformations between frames attached to different manipulator links. Oftentimes, we're instead interested in the *inverse* of this, or the **inverse kinematics** problem: find the combination of joint angles that will position a link of 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

*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. Modified by Shrey Aeron and Mingyang Wang, Fall 2023

the joint angles that will move the gripper at the end of the manipulator arm to this position.

Before every lab, you should pull the updated starter code. Navigate to the `ros_workspaces` directory and run

```
git pull starter main
```

to pull the lab 5 starter code.

2 Inverse Kinematics

An inverse kinematics solver for a given manipulator takes:

- input: the desired end effector configuration
- output: a set of joint angles that will place the arm at this position.

While the `tf2` package is the de facto standard for computing coordinate transforms for forward kinematics in ROS, we have several options to choose from for inverse kinematics. In this lab, we'll be using two external tools: **Position Kinematics** and **MoveIt**.

2.1 Specify a robot with a URDF

Both `MoveIt` and `tf2` are generic software packages that can work with almost any robot. To achieve this, we need some method 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 a more modern version of URDF, as it is capable of describing 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, which you can find by running `roscd sawyer_description`.

Note: The entire Sawyer SDK, including the `sawyer_description` package, is in the `/opt/ros/eecsbot_ws/src/sawyer_robot/sawyer_description` directory.

Task 1: It's hard to visualize the actual robot by staring at an XML file, so ROS provides a tool that converts a Xacro file into a more informative kinematic diagram. Navigate to your Desktop directory and run the commands:

```
cd ~/Desktop

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)
```

`check_urdf` parses and validates the specified URDF/Xacro file, then `urdf_to_graphiz` parses the file and generates a PDF graph in your current directory. Open and inspect 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?

If you are so inclined, fell free to run `rosrun rqt_tf_tree rqt_tf_tree` to see the full tree!

2.2 Compute inverse kinematics solutions

For the next task, we'll use the external tool `PositionKinematics`, which offers an inverse and forward kinematics service. You can find these two services by running `rosservice list` on the Sawyer and looking for:

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

Task 2: Create a package called `ik` which depends on `rospy`, `moveit_msgs`, `geometry_msgs`, and `intera_interface`. Move the `ik.py` and `fk.py` inside the src folder of your package. You will be implementing these two files to call the `PositionKinematics` inverse and forward kinematics services, respectively.

Edit `ik.py` to:

1. Prompt the user to input (x, y, z) coordinates for a gripper configuration
2. Constructs an IK request and submit the request to the `IKService` service offered by the `PositionKinematics` node
3. Prints the returned vector of joint angles in the terminal

Then, edit `fk.py` to:

1. Prompt the user to input a comma-separated list of 7 joint angles
2. Submit the request to the `FKService` service offered by the `PositionKinematics` node
3. Print out the returned final end-effector position

Some Hints/Information:

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.

Once you think you have a working implementation, 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? Why or why not?
- Any ideas why the solver sometimes fails to find a solution?

Checkpoint 1

Submit a checkoff request at `https://tinyurl.com/fa23-106alab2` 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 Pick and Place

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.

3.1 Planning with MoveIt

On top of inverse kinematics, the `MoveIt` package also includes a variety of powerful path planning functionality. MoveIt's path planning funcitons are accessible via ROS topics and messages, and a convenient RViz GUI is provided as well.

3.1.1 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.

Create a package named `planning` that depends on `rospy`, `roscpp`, `std_msgs`, `moveit_msgs`, `geometry_msgs`, `tf2_ros`, `baxter_tools`, and `intera_interface`. Inside `planning`, create a directory called `launch` and move `sawyer_moveit_gui_noexec.launch` and `planning_context.launch` into the new folder.

- 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.
- 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.
- 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.
- Switch to the "Planning" tab and click "Plan." The planner will compute a motion plan, then display the plan as an animation.
- 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.

3.1.2 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 `move_group` node 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.

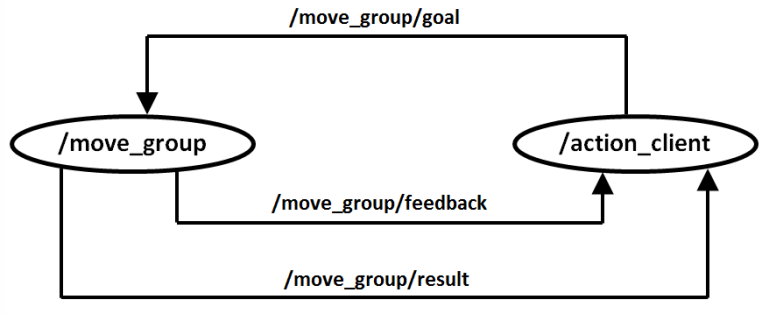

Figure 1: Action server topics

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

- Navigate to the `moveit_msgs` package.
- `cd` to the `/action` subdirectory.
- Open and examine `MoveGroup.action`

3.2 Open Loop Manipulation with Sawyer

In this section, you'll use inverse kinematics to program a Sawyer to perform a simple manipulation task: picking up, moving, and placing an object.

Before getting started, create a `move_arm` package which depends on `intera_interface`, `rospy` and `std_msgs`.

Connect to your Sawyer by running `source~ee106a/sawyer_setup.bash`, and then enable the robot by running `rosrun intera_interface enable_robot.py -e`. Then, run

```
roslaunch intera_examples sawyer_tuck.launch
```

to place Sawyer in a good starting joint configuration.

3.2.1 Recording pick-and-place positions

The table in front of each Sawyer should have a small plastic cube which you will use as our target object. Your task is to record the positions the Sawyer must move to pick up and place the cube.

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

```
rosrun tf tf_echo base right_gripper_tip
```

Grasp the sides of Sawyer's wrist to place the arm in zero-g mode, where it can be moved easily by hand.

- Move the arm to a position where it could grasp the object on the table and record the translation component of the `tf` transform.
- 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.

3.2.2 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. Start Sawyer's joint trajectory controller by running

```
rosrun intera_interface joint_trajectory_action_server.py
```

In a new window, start MoveIt by running

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

This command will also open an RViz window; ignore it for now.

Move `ik_example.py` (from the starter code) to the `src` directory of your `move_arm` package. This file contains an example of using MoveIt to move the Sawyers's arm to a specified pose. Open the file and examine it. Then, run the example with:

```
rosrun move_arm ik_example.py
```

This should prompt you to press enter, after which it will move the gripper to the position (0.5, 0.5, 0.0) with orientation (0.0, 1.0, 0.0, 0.0) (gripper pointing straight down).

Important: Make sure you are watching the trajectories in RVIZ and ensuring that they are safe BEFORE accepting the motion plan!

- Press enter a few times and pay attention to Sawyer's movements, moving the arm in compliant zero-g mode between trials.
- Then, 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.

3.2.3 Grippers

It's easy to operate Sawyer's grippers programatically as part of your motion sequence. Move the `gripper_test.py` file 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 `ik_example.py` file in the `move_arm/src/` directory. Modify your file so that it moves the arm through the series of poses 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/fa23-106alab2` for a staff member to come and check off your work. 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
- 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?