Lab 3 - Forward Kinematics/Coordinate Transformations*

EECS/ME/BIOE C106A/206A Fall 2023

Goals

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

- Compute the forward kinematics map for a robotic manipulator
- Compare your own forward kinematics implementation to the functionality provided by ROS
- Use the powerful functionality of `tf2` in your own ROS node.
- Make Sawyer move to simple joint position goals
- View the sensor and state data published by Sawyer using RViz

---

Relevant Tutorials and Documentation:

- `Sawyer SDK`: `http://sdk.rethinkrobotics.com/intera/API_Reference`
- `Sawyer Joint Position Control Examples` : `http://sdk.rethinkrobotics.com/intera/Joint_Position_Example`
- `tf2 Tutorials`: `http://wiki.ros.org/tf2/Tutorials`

Contents

*Modified by: Mingyang Wang, Fall 2023; Ravi Pandya, Nandita Iyer, Phillip Wu, and Valmik Prabhu, Fall 2018; David Fridovich-Keil and Laura Hallock, Fall 2017; Dexter Scobee and Oladapo Afolabi, Fall 2016; Victor Shia and Jaime Fisac, Fall 2015. Developed by Aaron Bestick, Austin Buchan, Fall 2014.

1 Introduction

You are REQUIRED to finish the Robot Usage Quiz before moving onto any checkpoints with the real robot! `https://www.gradescope.com/courses/568334/assignments/3262884`

Coordinate transformations are one of the fundamental mathematical tools of robotics. As discussed in lecture and homework, one of the most common applications of coordinate transformations is the **forward kinematics** problem: for a robotic manipulator, given a specified angle for each joint, can we compute the orientation of a selected link of the manipulator relative to a fixed world coordinate frame? To a frame attached to another point on the robot?

This lab will explore this question in two parts, which need not be done in order:

- In Part 1, you'll use the code you wrote as part of Homework 2 to write the forward kinematics map for a robot arm, and compare your results against some of ROS's built-in tools. You'll also learn a bit more about `tf2`, a useful ROS package for computing transforms.
- In Part 2, you'll explore Sawyer's basic joint position control functions, and take a quick look at how ROS helps you manage the coordinate transformations associated with all of Sawyer's moving parts.

Before every lab, you should pull the updated starter code. In terminal, navigate to the `~/ros_workspaces` directory and run

```
git pull starter main
```

to pull the lab 3 starter code.

You may have noticed that there are two lab 3 folders: `lab3` and `lab3_sawyer`. The reason for this will become more apparent later.

2 Forward kinematics

In this exercise, you'll write your own code to compute the forward kinematics map for an example robot arm.

Note: These parameters in this section correspond to an older robot named Baxter, a larger robot consisting of two arms as well as a torso. They will **NOT** work on Sawyer, which only has one arm.

2.1 Overview

Writing the forward kinematics map for a robotic manipulator involves the following steps:

1. Define a reference "zero" configuration for the manipulator at which we'll say $\theta = 0$, where $\theta = [\theta_1, \ldots, \theta_n]$ is the vector of joint angles for an n-degree-of-freedom manipulator
2. Choose where on the robot to attach the fixed base frame and the moving tool frame
3. Write the coordinate transformation from the base to the tool frame when the manipulator is in the zero configuration ($g_{st}(0)$)
4. Find the axis of rotation (ω_i) for each joint as well as a single point q_i on each axis of rotation (all in the base frame)
5. Write the twist ξ_i for each joint in the manipulator
6. Write the product of exponentials map for the complete manipulator
7. Multiply the map by the original base-to-tool coordinate transformation to get the new transformation between the base and tool frames ($g_{st}(\theta)$, now a function of the joint angles)

2.2 Writing the Forward Kinematics Map

Inside of `lab3/src`, create a package called `forward_kinematics` which depends on `rospy` and `sensor_msgs`. Move the `example_forward_kinematics.py` and `kin_func_skeleton.py` files into `forward_kinematics/src`.

Task 1: Fill in the `baxter_forward_kinematics_from_angles` function in `forward_kinematics.py` to compute the coordinate transformation between the base and tool frames for the robot arm pictured below. Your function should take an array of 7 joint angles as its only argument and return the 4x4 homogeneous transformation matrix $g_{st}(\theta)$ (steps 3-7 above).

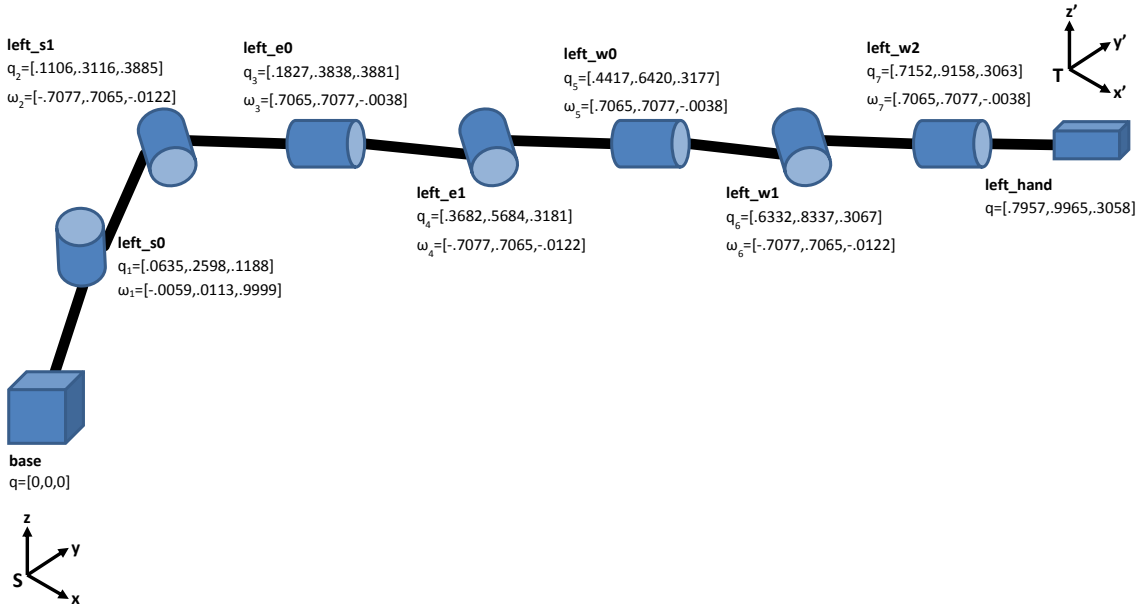

Figure 1: Example robot arm parameters (Baxter).

Copying the information into Python from the diagram above can take a while, so we have done it for you in `forward_kinematics.py`. The only other parameter you should need is the zero-configuration rotation matrix

$$R = \begin{bmatrix} 0.0076 & -0.7040 & 0.7102 \\ 0.0001 & 0.7102 & 0.7040 \\ -1.0000 & -0.0053 & 0.0055 \end{bmatrix}$$

where

$$g_{st}(0) = \begin{bmatrix} R & q \\ 0 & 1 \end{bmatrix}$$

for the appropriate value of q.

2.3 Compare with built-in ROS functionality

Once you think you have your forward kinematics map finished, you'll compare with with some built-in functions offered by ROS. We'll start by using a new tool called `rosbag`.

`rosbag` allows you to record and play back all messages published on a set of topics. You might have noticed a file named `baxter.bag` inside of your `lab3` directory: this file contains a recorded set of data from the example robot while we moved its (left) arm around. We can then use `rosbag` to play back these messages to test pieces of your software!

In terminal, start `roscore`. Then, open a new terminal and play the file with

```
rosbag play baxter.bag
```

A few things to note:

- You can pause playback with the space bar.
- You can view the published messages with the ROS tools like `rostopic list` and `rostopic echo`.
- You can add the `-l` flag to allow the rosbag file to play in a loop.

Try printing the messages being ent over the `robot/joint_states` topic, which gives the current joint angles of all joints in the example robot's left and right arms, as well as those of the head and torso. Using knowledge from `rostopic echo`, can you figure out what joint angles correspond to example robot's left arm?

Next, try running the command

```
rosrun tf tf_echo base left_hand
```

while the bag file is playing. Any ideas about the data that's displayed?

Task 2: Write a subscriber node `forward_kinematics_node.py` that receives messages from the `robot/joint_states` topic, plugs the appropriate joint angles from each message into your forward kinematics map from the last task, and displays the resulting transformation matrix on the terminal.

Hint: You will need to implement the `baxter_forward_kinematics_from_joint_state` function in `forward_kinematics.py` to extract the relevant joint angles from the `robot/joint_states` messages.

Display this in another window alongside the `tf` data discussed above. Do you notice any similarities? What do you think the "RPY" portion of the tf message is?

2.4 Writing a tf Listener

`tf` (transform) is more than just a command line utility; it's a powerful set of libraries designed to manage coordinate transforms between different coordinate frames on your robot. You'll be writing a listener node using `tf2`, which is the newer, supported version of `tf`. You can import it in your code with:

```
import tf2_ros
```

A `Buffer` is the core of `tf2` and stores a buffer of previous transforms. To create an instance of a `Buffer`, use:

```
tfBuffer = tf2_ros.Buffer()
```

A `TransformListener` subscribes to the `tf` topic and maintains the tf graph inside the `Buffer`. To create an instance of `TransformListener`, use:

```
tfListener = tf2_ros.TransformListener(tfBuffer)
```

The function `tfBuffer.lookup_transform(...)` looks up the transform of the target frame in the source frame. The output is of type `geometry_msgs/TransformStamped` (documentation for this type can be found here).

```
trans = tfBuffer.lookup_transform(target_frame, source_frame, rospy.Time())
```

Here are some `tf` exceptions you might want to catch:

```
tf2_ros.LookupException
tf2_ros.ConnectivityException
tf2_ros.ExtrapolationException
```

To catch an exception in Python you can create a try/except block (you might know this format as a try/catch block in most other programming languages). You should consider making a try/except block when using functions such as `lookup_transform` since exceptions can occur often and will crash your program when encountered. With a try/except block, your node will be able to handle exceptions and will not shut down if one occurs. You can write one with the following format:

```
try:
    <code to execute>
except (<exception>, <exception>, . . .):
    <code to execute if an exception occurs>
```

Task 3: Write a tf listener node `tf_echo.py` that duplicates the functionality of the `tf_echo` command line utility. Like the `tf_echo` command, your node should take in a target frame and a source frame as command line arguments.

Note: You shouldn't need to create a subscriber for your node. Why do you think this is?

Display your node's output in another window alongside the output of `tf_echo` and ensure that the outputs are the same. You do not have to format your output the same way, but the position and orientation values should be equal.

Checkpoint 1

Submit a checkoff request at `https://tinyurl.com/fa23-106alab`. At this point you should be able to:

- Explain how you constructed your forward kinematics function
- Explain the functionality of your `forward_kinematics` node and demonstrate how it works
- Demonstrate that your `forward_kinematics` node and `tf` produce the same output
- Demonstrate that your `tf_echo` node and `tf` produce the same output
- Finished the robot usage quiz: `https://www.gradescope.com/courses/568334/assignments/3262884`

---

3 Make Sawyer move

In this section, you'll explore some of Sawyer's basic position control functionality. Close all running ROS nodes and terminals from the previous part (including the one running `roscore`). **Additionally, ensure that you have been trained by the course instructors in the proper safety procedures (including use of the e-stop button) and etiquette for running Sawyer.**

Navigate to the `lab3_sawyer` directory. To set up your environment for working with the Sawyer, make a shortcut (symbolic link) to the Sawyer environment script
`/opt/ros/eecsbot_ws/intera.sh` using the command

```
ln -s /opt/ros/eecsbot_ws/intera.sh ~/ros_workspaces/lab3_sawyer/
```

After running `catkin_make`, use the following commands to ssh into the Sawyer robot:

```
source ~ee106a/sawyer_setup.bash
./intera.sh [name-of-robot].local
```

where `[name-of-robot]` is either `azula`, `alice`, `amir`, `ada`, or `alan`. You will have to run these two commands in every terminal window when working with the Sawyers. Then, run `source devel/setup.bash` so your new workspace is on the `$ROS_PACKAGE_PATH`.

`lab3_sawyer` should contain a package named `joint_ctrl`. Run the `joint_position_keyboard.py` script for an example of how to move the Sawyer arm. Note that you don't need to start `roscore` — it's already running on the Sawyer robot itself.

Instead of publishing directly to a topic to control Sawyer's arm (as with turtlesim), the respective SDKs provide a library of functions that take care of the publishing and subscribing for you.

Task 4: Create a new python node in the `src` folder of `joint_ctrl`. Start by making a copy of the `joint_position_keyboard.py` file and give it a new name. Edit your copy so that instead of capturing keypresses, it prompts the user for a list of seven joint angles, then moves to the specified position.

Hint: You might have to call `limb.set_joint_positions()` repeatedly at some interval, say, 10ms, while the robot is in the process of moving to the new position. The `set_joint_positions()` function takes a single argument: a Python dictionary object mapping the names of each joint to the desired joint angles
(e.g., `{'left_s0': 0.0, 'left_s1': 0.53, ..., 'left_w2':1.20}`). Dictionaries are used as follows:

```
# Create an empty dictionary
test_dict = {}

# Add values to the dictionary
test_dict['key1'] = 'value1'
test_dict['a_number'] = 1.024

# Read values from the dictionary
print(test_dict['key1'])
print(test_dict['a_number'])

# Output:
# value1
# 1.024

# You can also create a dictionary with a literal expression
test_dict2 = {'key1': 'value1', 'a_number': 1.024}
```

Test your code with several different combinations of joint angles and observe the results. Once you get your code to work, run the command

```
rosrun tf tf_echo base right_hand
```

and observe the output as you move the robot around. Any ideas what the data represents?

Finally, run

```
export ROS_MASTER_URI=http://[name-of-robot].local:11311
rosrun rviz rviz
```

for the appropriate value of `[name-of-robot]`, as before. The first line above tells RViz to connect to the remote master running on the robot.

Once RViz loads, ensure that `Displays > Global Options > Fixed Frame` is set to `world`. Next, click the Add button and add a RobotModel object to the window so you can see the robot move. Any thoughts as to where RViz gets the data on the robot's position?

Next, add two copies of the Axes object to the display. In the Displays pane of the left side of the screen, set the Reference Frame of one Axes object to `/base` and the other to `/right_hand`. You should see both sets of axes displayed on Sawyer. What do you think the axes represent?
Finally, remove both Axes objects and add a single TF object to the display. What happens?

Checkpoint 2

Submit a checkoff request at `https://tinyurl.com/fa23-106alab`. At this point you should be able to:

- Demonstrate the code you wrote to set Sawyer's joint positions
- Use RViz to display the different state and sensor data topics published by Sawyer
- Explain what the Axes and TF displays in RViz represent
