EE106A: Lab 3 - Forward Kinematics/Coordinate Transformations*

Fall 2022

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

Introduction

Coordinate transformations are one of the fundamental mathematical tools of robotics. One of the most common applications of coordinate transformations is the forward kinematics problem. Given a robotic manipulator, forward kinematics answers the following question: Given a specified angle for each joint in the manipulator, can we compute the orientation of a selected link of the manipulator relative to a fixed world coordinate frame or a frame attached to another point on the robot?

---

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

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 the prelab to write the forward kinematics map for one of Sawyer's arm, then you'll compare your results against some of ROS's built-in tools. You'll also learn a bit more about `tf2`, a useful 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.

1 Forward kinematics

As discussed in lecture, the forward kinematics problem involves finding the configuration of a specified link in a robotic manipulator relative to some other reference frame, given the angles of each of the joints in the manipulator. In this exercise, you'll write your own code to compute the forward kinematics map for an example robot's arm.

Note: These parameters correspond to an old robot named Baxter. They will **NOT** work on Sawyer.

1.1 Kinematic functions

Assuming you have completed the prelab, you may leverage code that you have already written in `kin_func_skeleton.py`, provided both partners can explain the code.

1.2 Set up your workspace

Create a workspace called `lab3` in your `~/ros_workspaces` directory. Refer to Lab 1 if you need to review how to do this.

In the src folder inside your `lab3` workspace, create a package called `forward_kinematics` which depends on `rospy` and `sensor_msgs`. Instructions on how to do this are also in Lab 1. Remember to run `catkin_make` to initialize and build your workspace and run and `source devel/setup.bash` so your new workspace is on the `$ROS_PACKAGE_PATH`. 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.git`. You can clone it by running

```
git clone https://github.com/ucb-ee106/106a-fa22-labs-starter/tree/main/Lab3
```

Keep the `joint_ctrl` folder aside and move the other files into the `src` folder inside your `forward_kinematics` package. We also highly recommend you make a **private** GitHub repository for each of your labs just in case.

1.3 Writing the forward kinematics map

Writing the forward kinematics map for a serial chain 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 as a function of the joint angles)

Task 1: Using the code from the prelab and referring to the textbook (available on bCourses) if necessary, write a Python function that computes the coordinate transformation between the base and tool frames for the robot arm pictured below (steps 3-7 above). Your function should take an array of 7 joint angles as its only argument and return the 4x4 homogeneous transformation matrix $g_{st}(\theta)$. Refer to Figure 1 for the parameters of the example robot arm. The only other parameter you should need is the 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.

Note: Copying the information into Python from the diagram below can take a while, so we have done it for you in `example_forward_kinematics.py`.

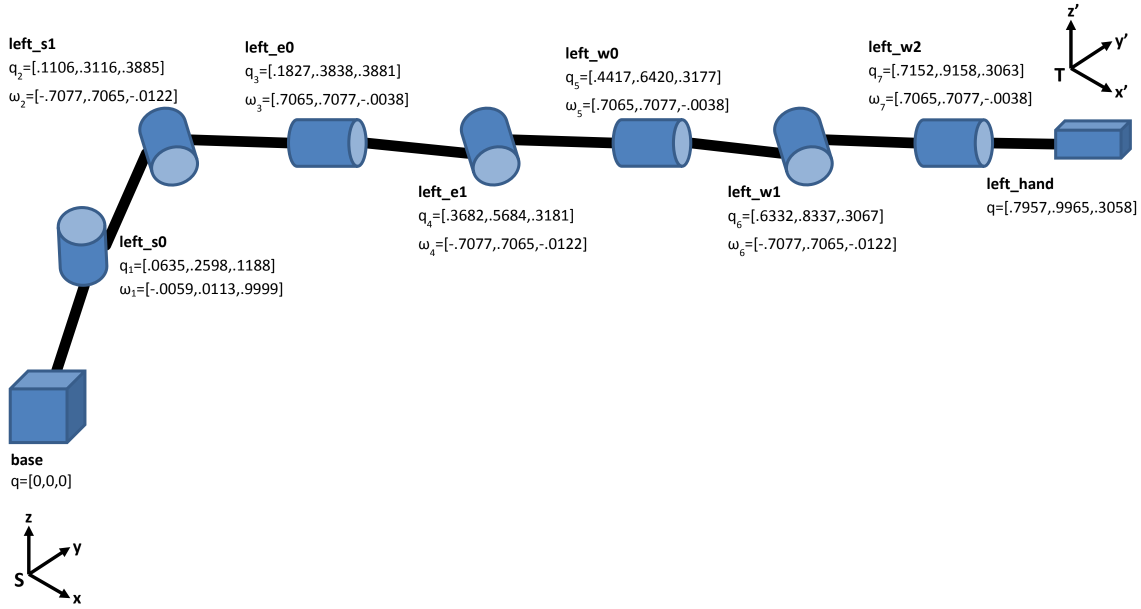

Figure 1: Example robot arm parameters (Baxter).

1.4 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.

Before we compare our software with built-in ROS functionality, we will need to disconnect from the Sawyer robot so that it does not publish data and override our recorded messages. We will do this by editing the ./bashrc file in your main folder. Comment out the line:

```
#export ROS_MASTER_URI=http://[robot].local:11311
```

and then close out all terminals.

To view our recorded data, we'll use a new tool called `rosbag`, which allows you to record and play back all the messages published on a set of topics, in order to test pieces of your software. We recorded a set of data from the example robot while we moved its left arm around. Find the `baxter.bag` file (from `lab3_resources.zip`), start in a separate terminal:

```
roscore
```

and open a new terminal, then play the file with

```
rosbag play baxter.bag
```

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

Try `rostopic echo`-ing 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`, you can figure out what joint angles correspond to example robot's left arm. (Hint: names starting with 'left_' correspond to the 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.py` that receives the 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. 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?

Hint: You will need to edit the second function of `example_forward_kinematics.py`.

1.5 Writing a tf Listener

`tf` is more than just a command line utility. It's a powerful set of libraries that you can use to find transforms between different frames on your robot. You'll be writing a listener node using `tf2`, which is the newer, supported version of `tf`. The `tf2` package is ROS independent, so you need to import `tf2_ros`, which contain ROS bindings of the various `tf2` functionalities. You can import it in your code with the following line:

```
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 the following line:

```
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 the following:

```
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 also take in a target frame and a source frame as command line arguments (the Python library `sys` might be helpful to look at). Please also note that 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 `tf` data discussed above and ensure that the outputs are the same. Note: you do not have to format your output the same way, but the position and orientation should be the same.

Checkpoint 1

Submit a checkoff request at `https://tinyurl.com/fa22-106alab`. 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

---

2 Make Sawyer move

Important: For this section, make sure that you connect back to Sawyer by uncommenting the line:

```
export ROS_MASTER_URI=http://[robot].local:11311
```

in your ./bashrc file.

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`, before you begin. **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.**

To create your workspace, make a folder called `lab3_sawyer` with a subfolder `src` in your `ros_workspaces` folder. From within the new `src`, run `catkin_init_workspace`, then from within `lab3_sawyer`, run `catkin_make`.

To set up your environment, 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/
```

Use the following line to ssh into one of the Sawyer robots:

```
./intera.sh [name-of-robot].local
```

(where `[name-of-robot]` is either `azula`, `alice`, `amir`, `ada`, or `alan`) in your folder to set up your environment for interacting with Sawyer, then run `source devel/setup.bash` so your new workspace is on the `$ROS_PACKAGE_PATH`.

Sawyer has the interface package `intera_interface`. Don't forget to check that you have this package imported! There is an important detail: Sawyer only has one arm! This means that whenever you try to move an arm on Sawyer, it must be the **right** one.

Move the `joint_ctrl` folder into the `src` folder. Build and source in the home directory of your `lab3` workspace as appropriate. 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 node (python file) in the `src` folder of `joint_ctrl` in your `lab3_sawyer` workspace. What dependencies will be needed? (Hint: Include `intera_examples` as a dependency.) 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, which should be 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
```

as appropriate 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/fa22-106alab`. 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
