Lab 3: Forward Kinematics and Coordinate Transformations*

EECS/ME/BIOE C106A/206A Fall 2021

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 Baxter/Sawyer move to simple joint position goals
- View the sensor and state data published by Baxter using RViz

---

Relevant Tutorials and Documentation:

- Baxter SDK
- Sawyer SDK
- Baxter Joint Position Control Examples
- Sawyer Joint Position Control Examples
- tf2 Tutorials

Contents

*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

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?

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 Baxter's arms, 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 Baxter/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 Baxter/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 one of the Baxter robot's arms.

1.1 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/lab3_starter.git. You can clone it by running

```
git clone https://github.com/ucb-ee106/lab3_starter.git
```

Move the 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.

1.2 Prelab

Make sure you have done the Lab 3 Prelab (implementing the functions in the `kin_func_skeleton.py` file) which was part of Homework 2. For the rest of this lab, you may use your filled-in `kin_func_skeleton.py` file provided that both partners can explain the code.

1.3 Writing the forward kinematics map

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

1. Choose where on the robot to attach the fixed base frame and the moving tool frame.
2. Define $\theta = [\theta_1, \ldots, \theta_n]$ to be the vector of joint angles for an n-degree-of-freedom manipulator, and define $g_{st}(\theta)$ to be the homogeneous transformation from the base frame to the tool frame that results from setting the degrees of freedom to the joint angles in θ.
3. Define a reference "zero" configuration $g_{st}(0)$ for the manipulator at which all entries of θ are 0, and write out the coordinate transformation from the base to the tool frame when the manipulator is in this zero configuration.
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 with respect to 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 $g_{st}(\theta)$, which is now a function of the joint angles.

Task 1: Using the code from prelab and referring to the textbook if necessary, complete the implementation for the `baxter_forward_kinematics_from_angles` function in the `baxter_forward_kinematics.py` file that computes the coordinate transformation between the base and tool frames for the Baxter arm pictured below (steps 3-7 above). The 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 Baxter 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 `baxter_forward_kinematics.py`.

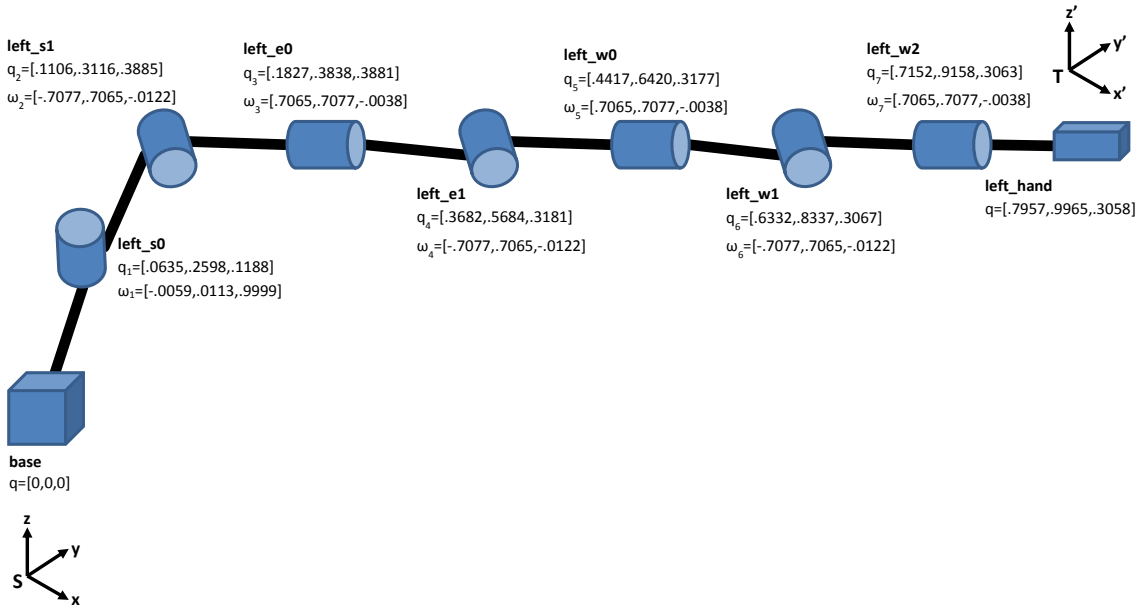

Figure 1: Baxter arm parameters.

1.4 Compare with built-in ROS functionality

Once you think you have finished your forward kinematics function, you'll compare your implementation with with some built-in functions offered by ROS.

To do this, 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. The `baxter.bag` file from the starter code is a recorded set of data from Baxter while its left arm was moved around. Start `roscore` first, then play the file by running

```
rosbag play baxter.bag
```

from the directory that contains the `baxter.bag` file. 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`.

Try `rostopic echo`-ing the `robot/joint_states` topic, which gives the current joint angles of all joints in Baxter'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 Baxter'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 node `forward_kinematics_node.py` that subscribes to the `robot/joint_states` topic and plugs the appropriate joint angles from each message into your forward kinematics map from the last task. Do this by implementing the `baxter_forward_kinematics_from_joint_state` function in `baxter_forward_kinematics.py` and have your subscriber node call that function whenever it receives a new message. Display the resulting transformation matrix in another terminal alongside the `tf` data discussed above. Do you notice any similarities? What do you think the "RPY" portion of the `tf` message is?

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 tinyurl.com/106alab for a staff member to come and check off your work. 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 Baxter/Sawyer move

In this section, you'll explore some of Baxter/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 Baxter/Sawyer.**

To automatically set up the Baxter and Sawyer packages in each new terminal, make sure that your `~/.bashrc` file includes the line

```
source /scratch/shared/baxter_ws/devel/setup.bash
```

If you edited the `~/.bashrc` file, make sure to source it in any existing terminal windows that you plan on using.

To set up your environment for interacting with Baxter, make a shortcut (symbolic link) in the root of your catkin workspace (`lab3`) to the Baxter environment script `/scratch/shared/baxter_ws/baxter.sh` using the command

```
ln -s /scratch/shared/baxter_ws/baxter.sh [path-to-workspace]
```

From the root of the catkin workspace (`lab3`), use the following line to connect to one of the Baxter robots:

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

where `[name-of-robot]` is either `ayrton`, `asimov`, or `archytas`.

To set up your environment for interacting with Sawyer, make a shortcut (symbolic link) in the root of your catkin workspace (`lab3`) to the Sawyer environment script `/scratch/shared/baxter_ws/intera.sh` using the command

```
ln -s /scratch/shared/baxter_ws/intera.sh [path-to-workspace]
```

From the root of the catkin workspace (`lab3`), use the following line to connect to one of the Sawyer robots:

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

where `[name-of-robot]` is either `ada` or `alan`.

Baxter and Sawyer have different interface packages (`baxter_interface` and `intera_interface` respectively), but they are virtually identical. Don't forget to check that you have the correct package imported! The main difference is the obvious one: Sawyer only has one arm! This means that whenever you try to move an arm on Sawyer, it must be the **right** one. On Baxter, you may use either arm.

Open the `baxter_examples` package inside the `baxter_ws` workspace and examine the `scripts/joint_position_keyboard.py` file, which allows you to move the Baxter's limbs using the keyboard. If you are connected to a Baxter, run the program and test its commands. If you are connected to a Sawyer, run the corresponding example in the `intera_sdk/intera_examples` package. Note that you don't need to start `roscore` — it's already running on the Baxter/Sawyer robot itself.

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

Task 4: Create and open a new package called `joint_ctrl` in your `lab3` workspace. What dependencies will be needed? (Hint: Include `baxter_examples/intera_examples` as a dependency.) Make a copy of the `joint_position_keyboard.py` file (from the appropriate package, depending on which type of robot you are using) inside the `joint_ctrl` package, giving 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 left_hand
```

or

```
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 Baxter. 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 tinyurl.com/106alab for a staff member to come and check off your work. At this point you should be able to:

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