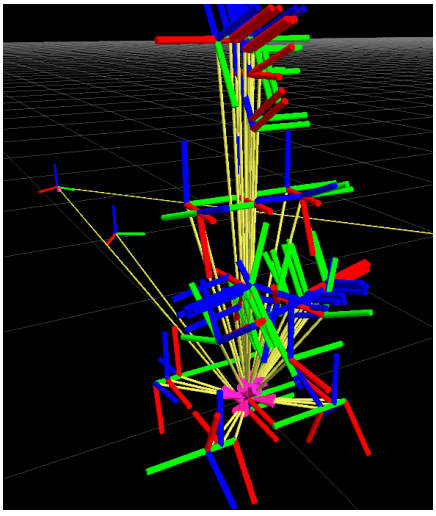
Robotic Systems Engineering Lab 04

This document contains examples and exercises on writing code for forward kinematics problems and the ‘tf2’ library. This lab features two manipulators, the ‘OpenManipulator’ and the ‘Robotis Manipulator-H’.

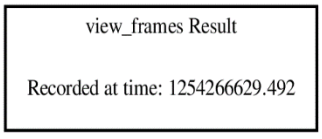
When doing tasks with a robot it is crucial that the robot be aware of where it is itself as well as where the rest of the world is in relation to itself. Having that information, the programmer could simply request from a library what is the vector that they need to perform a specific task, eg moving a gripper of a robot with respect to its main body, in order to grasp some target. This is where the ‘tf’ library comes in.

The ‘tf’ library

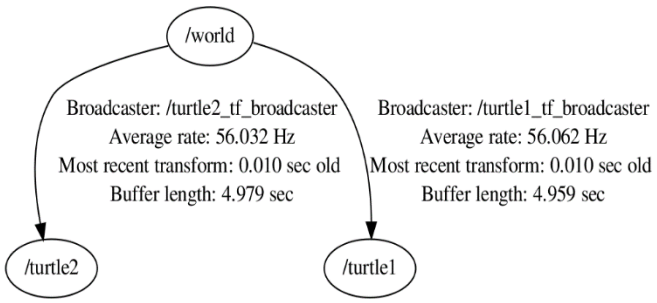
The ‘tf’ library was designed to provide a standard way to keep track of coordinate frames and transform data within the entire system such that individual component users can be confident that the data is in the coordinate frame that they want without requiring knowledge of all the coordinate frames in the system. ‘tf’ maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc, between any two coordinate frames at any desired point in time. In a few words, ‘tf’ is a library for keeping track of coordinate frames



The ‘tf’ library is most closely related to the concept of a scene graph (a common type of data structure used to represent a 3D scene for rendering) and can be separated into two different parts. The first part is disseminating transform information to the entire system. The second part of the library receives the transform information and stores it for later use. The second part is then able to respond to queries about the resultant transform between different coordinate frames.



Essentially, the library provides a transform between two coordinate frames at a requested time. Its design mainly consists of two modules. The ‘Broadcaster’ module takes care of the distribution, whereas the ‘Listener’ module is performing the reception and the queries for the transforms.



- *'tf' is closely represented by a tree*

Transforms and coordinate frames can be expressed as a graph with the transforms as edges and the coordinate frames as nodes. However, with a graph, two nodes may have multiple paths between them, resulting in two or more potential net transforms introducing ambiguity to the problem. To avoid this, we limit the 'tf graph' into a 'tf tree'. This is why, although the 'tf' library can be closely associated with a scene graph, it is most closely represented by a tree that is designed to be queried for specific values asynchronously, has the benefit of allowing for dynamic changes to its structure and each update is specific to the time at which it was measured.

- *'tf' stamp*

To be able to operate, all data which is going to be transformed by the 'tf' library must contain two pieces of information: the coordinate frame in which it is represented and the time at which it is valid. These two pieces of data are referred to as a 'Stamp'. Data which contains the information in the 'Stamp' can be transformed for known data types.

- *Broadcaster/Listener modules*

The Broadcaster module was designed very simply. It broadcasts messages every time an update is heard about a specific transform with a minimum frequency. The Broadcasters send updates periodically whether they have changed or not. The Listener collects the values into a sorted list and when queried can interpolate between the two nearest values. Because the Broadcaster send transforms regularly, the Listener does not ever assume the presence of a coordinate frame into the future. That's why it is crucial to always choose the appropriate communication frequency.

- *The 'tf2' library*

In this course we are going to be using 'tf2', which is the second generation of the transform library, and as the first generation, lets the user keep track of multiple coordinate frames over time. This lab also features two robots. Manipulator-H for the examples and OpenManipulator for the tasks.

http://emanual.robotis.com/docs/en/platform/manipulator_h/introduction/
http://emanual.robotis.com/docs/en/platform/openmanipulator/

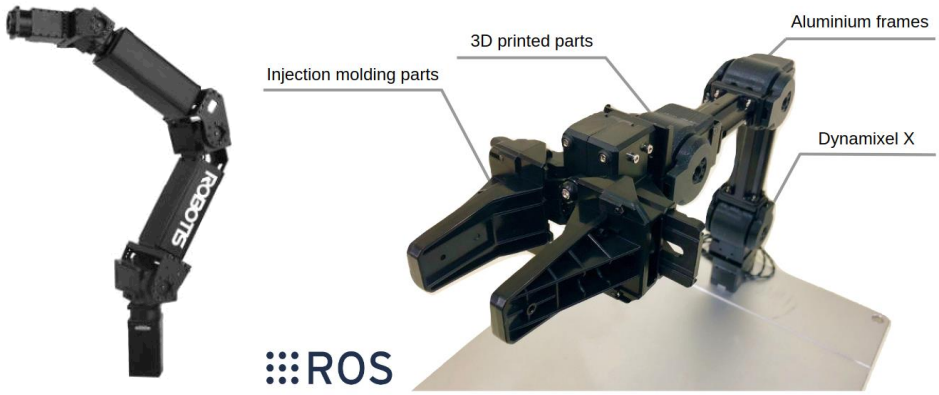


Figure 1. Manipulator-H (left), OpenManipulator (right)

Useful Commands

`rosrun tf view_frames`	-> creates a diagram of the frames being broadcast by tf over ROS
`evince frames.pdf`	-> draws a tree of how the frames that are being broadcast over ROS are connected
`rosrun rqt_tf_tree rqt_tf_tree`	-> a runtime tool for visualizing the tree of frames being broadcast over ROS
`rosrun tf tf_echo [reference_frame] [target_frame]`	-> reports the transform between any two frames broadcast over ROS

Examples

Example 00

In a console type:

```
roslaunch manipulator_h_description manipulator_h_rviz.launch
```

You can see 'Manipulator-H' in 'rviz' along with a GUI for controlling the joint positions. Inside the 'rviz' window enable the options 'Show Names' and 'Show Axes' under 'Displays -> TF' and the option 'All Enabled' under 'Displays -> TF -> Frames'. You can now see the corresponding frame of each joint.

This example uses the 'robot_state_publisher' package which allows you to publish the state of a robot to 'tf'. The package takes the joint angles of the robot as input and publishes the 3D poses of the robot links using a kinematic tree model of the robot.

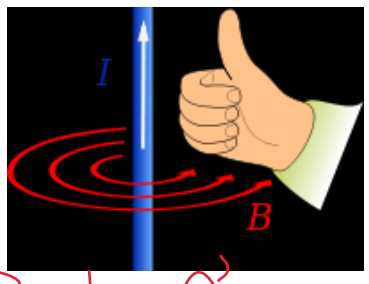
Example 01

In this example, the output is failry similar to the first one, with the only difference that the frames are manually added with our forward kinematics code, and not through the kinematic model that is defined in the 'robot_state_publisher' package. For a deeper understanding, we'll go through the most important bits of the code.

We start by including all the necessary headers and libraries. We then declare the DH parameters of the 'Manipulator-H' robot, as shown in Table 1 and define 'fkine_standard' which is nothing more than a function that computes the Homogeneous Matrix of a joint.

i	a	α	d	θ
1	0	-π/2	0.159	θ_1
2	0.26569	0	0	$\theta_2 - \frac{\pi}{2} + b$
3	0.03	-π/2	0	θ_3 - b
4	0	-π/2	0.258	θ_4
5	0	-π/2	0	θ_5
6	0	0	0.123	θ_6

Table 1. DH parameters of the Manipulator-H robot



Where b = $\text{atan}(\frac{0.03}{0.264})$.

The 'fkine' function is the most important bit of code of this script since the broadcaster is created in it. We create the 'transform' message which expresses a transform from coordinate frame 'header.frame_id' to the coordinate frame 'child_frame_id'. The time stamp of the message is the current time. For each joint we compute the corresponding Homogenous Matrix and convert it into 'Eigen' compatible form. Finally, we simply broadcast the 'tranform' message every time an update is heard, to be used by 'tf'. Pay special attention that we added four lines of code that reverse theta for just the two last frames 4 and 5. We have to do this because in the '.xacro'. file the z axes are reversed.

In the last part of the script, we define the subscriber, which will subscribe to the '/joint_states' topic and invoke fkine as its callback function which will update the new matrices and broadcast to the tf topic. Note that in this example we did not create any listeners.

To run the code, after you've terminated 'rviz' from the previous example, type either of these commands:

```
roslaunch lab04example01 lab04example01cpp.launch
```

```
roslaunch lab04example01 lab04example01py.launch
```

Try to find differences in frame arrangements between the two examples when moving the robot.

Example 02

This example is a bit more specific and its purpose is to showcase how to retrieve information from basic 'tf2' messages. While in the previous example we created a broadcaster, in this one we will create a listener. The package waits for the transformation between the parent frame 'world' and the child frame 'link6' to take place before it prints it out.

To run the code, after you've terminated 'rviz' from the previous example, type either of these commands:

```
roslaunch lab04example02 lab04example02cpp.launch
```

```
roslaunch lab04example02 lab04example02py.launch
```

Tasks

Task 01

Solve the Forward Kinematics problem for the 'OpenManipulator' robot based on the picture below. Coding the Forward Kinematics problem is fairly easy because its differs only slightly from 'Example 01'. The challenging part is to actually solve the problem with the correct DH parameters.

Caution!!!

1. **Your 'frame arrangement' must match that of the '.xacro' file. 'Xacro' is just a scripting mechanism that allows more modularity and code re-use when defining a 'URDFL' model. A 'URDF' model is an 'XML' format for representing a robot model (Unified Robot Description Format).**

   The '.xacro' file is located in 'open_manipulator_description/urdf/open_manipulator.urdf.xacro'.

2. **As with the previous lab, code from this Task can be used for the coursework, with the difference that the coursework examines the 'Youbot' and not the 'Manipulator-H'. We remind you that these Tasks are NOT assessed or graded.**

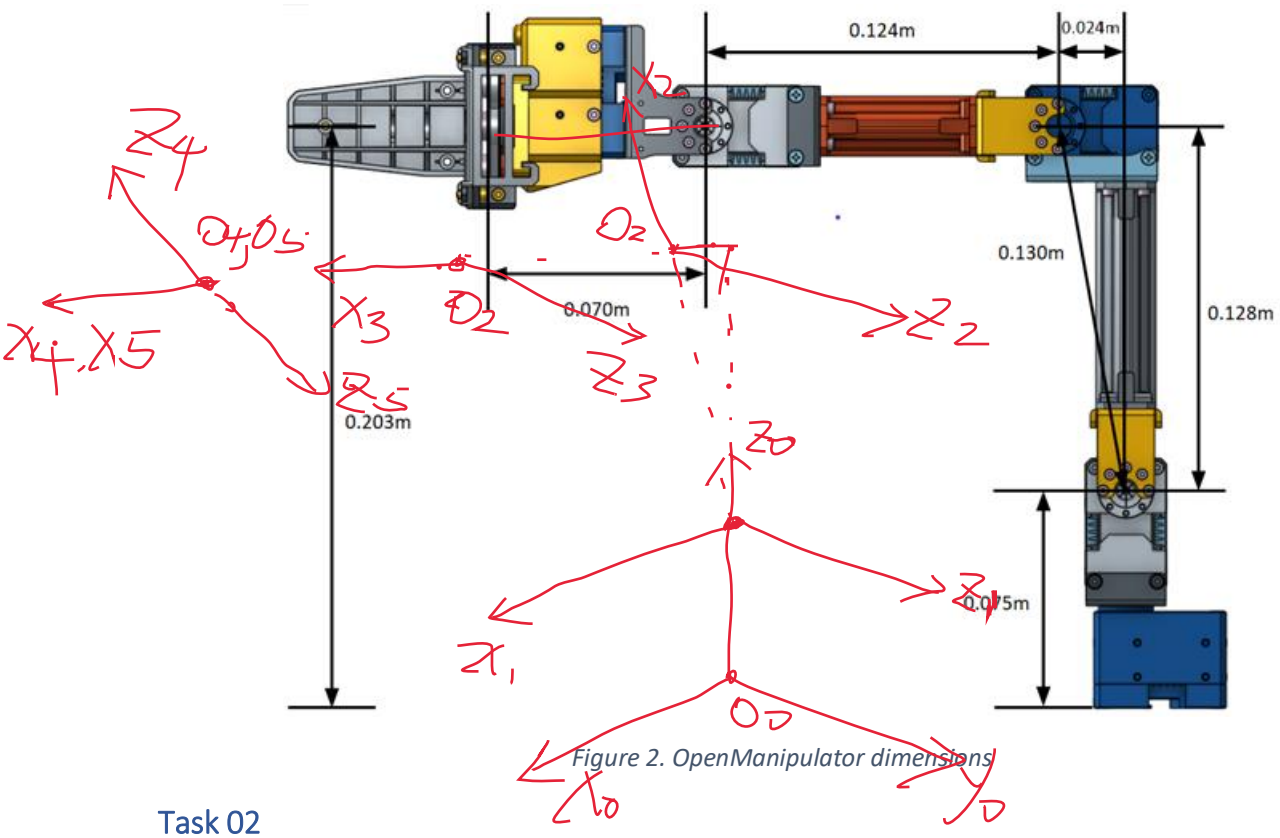


Figure 2. OpenManipulator dimensions

Task 02

In this task you have to create a 'tf2' listener similar to the one shown in Example 02. Only this time you shouldn't print out the transformation directly. You have to create a new frame by applying another transformation, and then broadcast the new frame to 'tf2'.

Resources

- We highly suggest that you go through these tutorials if you're having trouble with completing the tasks or understanding how the 'tf2' library works. http://wiki.ros.org/tf2/Tutorials

  Help on 'Task 02' can be found on Tutorial No.4 'Adding a frame'.

- For an extensive theoretical take on 'tf' library, read: http://wiki.ros.org/Papers/TePRA2013_Foote