ROS-I Training by

TF TUTORIAL

1. Introduction

This tutorial will show you the benefits and the power of the ROS TF tool. It will introduce how to make use of static and dynamic transforms and TF listener.

2. Writing a tf broadcaster

First of all create a new package in your workspace. Name it "learning_tf" with dependencies on tf2, tf2_ros, tf_conversions, rospy and turtlesim and build the package. Inside of the package create a new python node called "tf_broadcaster_example.py" (~/moveit_ws/src/moveit_tutorial/scripts/)

Given below is an example of a ROS node including a TF Broadcaster. The code is based on the simple structure of the node. Feel free to adjust the code structure based on your programming skills. Copy the code to the "tf_broadcaster_example.py".

```python
#!/usr/bin/env python

# imports
import rospy
import tf2_ros
import tf_conversions
import geometry_msgs.msg
from turtlesim.msg import Pose

# node initialization
rospy.init_node("tf_bc_exmpl")

# variable definition
br = tf2_ros.TransformBroadcaster()
t = geometry_msgs.msg.TransformStamped()
r = rospy.Rate(30)
turtle_name = rospy.get_param("~turtle")

# definition of functions
```

```
def handle_turtle_pose(msg):
    t.header.stamp = rospy.Time.now()
    t.header.frame_id = "world"
    t.child_frame_id = turtle_name
    t.transform.translation.x = msg.x
    t.transform.translation.y = msg.y
    t.transform.translation.z = 0.0
    q = tf_conversions.transformations.quaternion_from_euler(0, 0,
       msg.theta)
    t.transform.rotation.x = q[0]
    t.transform.rotation.y = q[1]
    t.transform.rotation.z = q[2]
    t.transform.rotation.w = q[3]

rospy.Subscriber("%s/pose" % turtle_name, Pose, handle_turtle_pose)

while not rospy.is_shutdown():
    br.sendTransform(t)
    r.sleep()
```

2.1. Explanation:

The basic structure of a simple node including Publisher and Subscriber is already well known. So, only the tf specific lines will be explained.

```
import tf2_ros
import tf_conversions
```

Import the essential TF modules. The tf2_ros package provides ROS bindings to tf2. tf_conversions provides the popular transformations.py, which was included in tf but not in tf2, in order to have a cleaner package.

```
br = tf2_ros.TransformBroadcaster()
```

This will initialize a ROS TF Broadcaster, which allows to send transforms from one frame to another one.

```
turtle_name = rospy.get_param("~turtle")
```

This line gets the value of the "turtle" parameter from the parameter server (e.g. turtle1). The ~ prefix prepends the parameter-name with the node's name to use it as a semi-private namespace (e.g. /tf_bc_exmpl/turtle).

```
br.sendTransform(t)
```

This line is the handler function to provide the transform between the "turtle1" (value of the "turtle" parameter) and the "world" frame and publishes it. As it is inside of the while loop and the rate is set to 30 Hz, the transform will be published within this frequency. The *sendTransform* function a StampedTransform, which was set up in the subscriber callback:

- **Header**
    - **Timestamp** (Determine the moment when this transform is happening. This is mainly *rospy.Time.now()* when you want to send the actual transform. This means the transform can change over time to generate a dynamic motion.)
    - **Frame_ID** (The frame ID of the Origin Frame)

- **Child Frame ID** (Frame ID to which the transform is happening)
- **Transform**
  - **Position** in m (X, Y and Z)
  - **Orientation** in Quaternion (You can use the TF Quaternion from Euler function to use the roll, pitch and yaw angles in rad instead)

2.2. Testing the Broadcaster

Create a launch file called tf_examples.launch in your package. Include the turtlesim_node and the turtle_teleop_key node from the turtlesim package. Include the example Broadcaster node in the launch file with private parameter "turtle" of type string. Set the value of "turtle" as "turtle1", and run the launch file.

```
<node name="tf_broadcaster" pkg=" learning_tf" type=" tf_broadcaster_example.py " output="screen">
      <param name="turtle" value="turtle1" type="string" />
</node>
```

Now, use the *tf_echo* tool to check if the turtle pose is already published to tf:

```
$ rosrun tf tf_echo /world /turtle1
```

This should show you the pose of the turtle1 related to the world frame. Now, drive around the turtle using the arrow keys. Make sure to have the terminal in foreground that started the launch file including the keyboard teleop node.

You can use *rqt_tf_tree* to check available TF trees.

```
$ rosrun rqt_tf_tree rqt_tf_tree
```

rqt_tf_tree is a runtime tool for visualizing the tree of frames being broadcasted via ROS. You can refresh the tree simply by the refresh button in the top-left corner of the GUI.
Also rviz can be used to visualize the location of frames. Start rviz.

```
$ rosrun rviz rviz
```

Add a TF visualization element in rviz and set the fixed frame to world. Move the turtle around and follow the location of the turtle1 frame in world.

3. Writing a TF listener

TF provides much more tools then just the Broadcaster. A couple of debugging and visualization tools for frames have been introduced recently. Also very powerful is the access to frame transformations. This can be done using TF listener. TF listener solves inverse kinematics. Similar to the command line tool *tf_echo*, TF listener can check the transformation between two frames in nodes. In the following example we will add a second turtle to the turtlesim node. The second turtle should follow the first one. Add the following example code to a python node "tf_listener_example.py" inside of the "learning_tf" package.

```
#!/usr/bin/env python
```

```
# import
import rospy
import math
import tf2_ros
import geometry_msgs.msg

# node initialization
rospy.init_node("tf_lstnr_exmpl")

# variable definition
cmd = geometry_msgs.msg.Twist()
tfBuffer = tf2_ros.Buffer()
listener = tf2_ros.TransformListener(tfBuffer)

# definition of publishers/subscribers/services
turtle_vel = rospy.Publisher("turtle2/cmd_vel", geometry_msgs.msg.Twist,
queue_size=1)

# main program
r = rospy.Rate(30)  #30Hz

while not rospy.is_shutdown():
    try:
        trans = tfBuffer.lookup_transform("turtle2", "turtle1",
rospy.Time(0))
    except (tf2_ros.LookupException, tf2_ros.ConnectivityException,
tf2_ros.ExtrapolationException):
        continue

    cmd.linear.x = 0.5 * math.sqrt(trans.transform.translation.x ** 2 +
trans.transform.translation.y ** 2)
    cmd.angular.z = 4 * math.atan2(trans.transform.translation.y,
trans.transform.translation.x)
    turtle_vel.publish(cmd)
    r.sleep()
```

3.1. Explanation:

```
import tf2_ros
```

Importing tf is necessary to use the tf listener functionalities.

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

A listener has to be initialized first. It utilizes a buffer to keep track of past transforms.

```
    try:
        trans = tfBuffer.lookup_transform("turtle2", "turtle1",
rospy.Time(0))
    except (tf2_ros.LookupException, tf2_ros.ConnectivityException,
tf2_ros.ExtrapolationException):
        continue
```

The listener has to be used inside a try – except block. The listener itself

```
trans = tfBuffer.lookup_transform("turtle2", "turtle1", rospy.Time(0))
```

lookups the transform from "turtle2" to "turtle1" in the actual moment *(rospy.Time(0))* and stores the result in the Transform variable trans which holds Orientation and Position.

```
cmd.linear.x = 0.5 * math.sqrt(trans.transform.translation.x ** 2 +
trans.transform.translation.y ** 2)
cmd.angular.z = 4 * math.atan2(trans.transform.translation.y,
trans.transform.translation.x)
turtle_vel.publish(cmd)
```

The publisher afterwards will make the second turtle move and let it follow the first one with the corresponding mathematics.

3.2. Testing the Listener

Include the listener node to the previous generated launch file "tf_examples.launch" and start it. The lookup will fail for now as we did not spawn the second turtle right now. To do so, a ROS Service call will be used. We will also need to launch a second TransformBroadcaster:

```
$ rosservice call /spawn 2 2 0.2 "turtle2"
$ rosrun learning_tf tf_broadcaster_example.py _turtle:=turtle2
```

This will spawn the second turtle with the initial position x = 2 and y = 2, an orientation of yaw = 0.2 rad and the name "turtle2". Move now the first turtle around and the second turtle should start to follow the first one.

4. Adding static transforms

Another node that is provided by the ROS TF tool is the static_transform_publisher. It can be used to determine static frame transforms, e.g. from a robot base to a sensor devices frame. This one is quite easy to use. Add a virtual camera frame to our first turtle within the recent launch file "tf_examples.launch" by adding the following line:

```
<node pkg="tf2_ros" type="static_transform_publisher"
name="turtle1_cam_frame"  args="0.1 0.0 0.0 -1.57 0.0 0.0 turtle1
turtle_cam" />
```

This will add the frame "turtle_cam" with respect to to the "turtle1" frame. The virtual camera is mounted +0.1 m in x-axis from view of the turtle base and rotated -1.57 rad in yaw. The convention for the static transform publisher arguments is the following order:

X Y Z Yaw Pitch Roll Parent_frame Child_frame publisher_framerate

4.1. Testing the Listener

To verify the location of the added virtual turtle camera, start the generated launch file, use rviz and add the TF visualization element. Set fixed frame to world and move the first turtle around. You should see now both frames – the turtle1 and the turtle1_cam_frame moving in the world.

5. Using TF with UR5

Close all running processes with turtlesim now and start the Gazebo Simulation for the UR5:

```
$ roslaunch ur5_moveit_config_pkg ur5_gazebo.launch
$ roslaunch ur5_moveit_config_pkg ur5_moveit_planning_execution.launch
sim:=true
$ rviz
```

Use your knowledge about TF and publish a topic /actual_tcp_pose within a Pose message from the geometry_msgs. The topic should inform about the actual pose of the TCP ('link_6') related to the robot base link ('base_link').

HINT: Use TF listener to get the required transformation and a publisher with geometry_msgs/Pose to publish the pose.

Additionally attach an object to the end effector of the robot using TF static transform publish. It should be fixed to the end effector with an offset of 0.2 m in z-Axis. The frame id should be "object".

HINT: Use the following line in the launch file

```
<node name="broadcaster" pkg="tf" type="static_transform_publisher"
args="x y z r p y parent_frame child_frame 100" />
```

Fill in the correct names for 'parent_frame' and 'child_frame' respectively and also the correct value for "x y z r p y".