**TF (TRANFORMATION OF FRAMES)**

* **What is TF ?**

TF stands for transformation library in ROS.

TF is a package that lets the user keep track of multiple coordinate frames over time. TF maintains the relationship between coordinate frames in a tree structure buffered in time

It performs computation for transformations between frames.

It allows find the pose of any object in any frame using transformation.

A robot is a collection of frames attached to its different joints.

Frames are defined in every joints and components of the robot.

(frames are fixed points which are used for the reference or as the reference points)

A robotic system typically has many 3D coordinate frames that change over **time**, such as a world frame, base frame, gripper frame, head frame, etc. tf keeps track of all these frames over time.

* **Writing a tf broadcaster (Python)**

Create a new ros package in your existing catkin\_ws workspace named as learning\_tf.

$ catkin\_create\_pkg learning\_tf tf roscpp rospy turtlesim

First order Dependencies of the package are: tf , roscpp , rospy and turtlesim. Can see it using rospack command.

* Build your new package before you can roscd:

$ cd %YOUR\_CATKIN\_WORKSPACE\_HOME%/

$ catkin\_make

$ source ./devel/setup.bash

Let's first create the source files. Go to the package we just created:

$ roscd learning\_tf

$ mkdir nodes

$ touch nodes/turtle\_tf\_broadcaster.py

$ chmod +x nodes/turtle\_tf\_broadcaster.py

Now, write the following code in the above mentioned file (turtle\_tf\_broadcaster.py)

#!/usr/bin/env python3

import roslib

roslib.load\_manifest('learning\_tf')

import rospy

import tf

import turtlesim.msg

def handle\_turtle\_pose(msg, turtlename):

br = tf.TransformBroadcaster()

br.sendTransform((msg.x, msg.y, 0),

tf.transformations.quaternion\_from\_euler(0, 0, msg.theta),

rospy.Time.now(),

turtlename,

"world")

if \_\_name\_\_ == '\_\_main\_\_':

rospy.init\_node('turtle\_tf\_broadcaster')

turtlename = rospy.get\_param('~turtle')

rospy.Subscriber('/%s/pose' % turtlename,

turtlesim.msg.Pose,

handle\_turtle\_pose,

turtlename)

rospy.spin()

Now create a launch file for this demo. With your text editor, create a new file called **launch/start\_demo.launch**, and add the following lines:

<launch>

<!-- Turtlesim Node-->

<node pkg="turtlesim" type="turtlesim\_node" name="sim"/>

<node pkg="turtlesim" type="turtle\_teleop\_key" name="teleop" output="screen"/>

<node name="turtle1\_tf\_broadcaster" pkg="learning\_tf" type="turtle\_tf\_broadcaster.py" respawn="false" output="screen" >

<param name="turtle" type="string" value="turtle1" />

</node>

<node name="turtle2\_tf\_broadcaster" pkg="learning\_tf" type="turtle\_tf\_broadcaster.py" respawn="false" output="screen" >

<param name="turtle" type="string" value="turtle2" />

</node>

</launch>

Now you're ready to start your own turtle broadcaster demo:

$ roslaunch learning\_tf start\_demo.launch

You should see the turtlesim with one turtle.

Now, use the tf\_echo tool to check if the turtle pose is actually getting broadcast to tf:

$ rosrun tf tf\_echo /world /turtle1

* **Writing a tf listener (Python)**

$ touch nodes/turtle\_tf\_listener.py

$ chmod +x nodes/turtle\_tf\_listener.py

Now, write the following code in the above mentioned file (turtle\_tf\_listener.py)

#!/usr/bin/env python3

import roslib

roslib.load\_manifest('learning\_tf')

import rospy

import math

import tf

import geometry\_msgs.msg

import turtlesim.srv

if \_\_name\_\_ == '\_\_main\_\_':

rospy.init\_node('turtle\_tf\_listener')

listener = tf.TransformListener()

rospy.wait\_for\_service('spawn')

spawner = rospy.ServiceProxy('spawn', turtlesim.srv.Spawn)

spawner(4, 2, 0, 'turtle2')

turtle\_vel = rospy.Publisher('turtle2/cmd\_vel', geometry\_msgs.msg.Twist,queue\_size=1)

rate = rospy.Rate(10.0)

while not rospy.is\_shutdown():

try:

(trans,rot) = listener.lookupTransform('/turtle2', '/turtle1', rospy.Time(0))

except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):

continue

angular = 4 \* math.atan2(trans[1], trans[0])

linear = 0.5 \* math.sqrt(trans[0] \*\* 2 + trans[1] \*\* 2)

cmd = geometry\_msgs.msg.Twist()

cmd.linear.x = linear

cmd.angular.z = angular

turtle\_vel.publish(cmd)

rate.sleep()

This function returns two lists. The first is the (x, y, z) linear transformation of the child frame relative to the parent, and the second is the (x, y, z, w) quaternion required to rotate from the parent orientation to the child orientation.

With your text editor, open the launch file called start\_demo.launch, and add the following lines:

<launch>

...

<node pkg="learning\_tf" type="turtle\_tf\_listener.py"

name="listener" />

</launch>

Now you're ready to start your full turtle demo:

$ roslaunch learning\_tf start\_demo.launch

You should see the turtlesim with two turtles.

Now run :

$ rosrun rqt\_gui rqt\_gui

This command will show you the tf-tree.

Now run :

$ rosrun rviz rviz -d `rospack find turtle\_tf`/rviz/turtle\_rviz.rviz

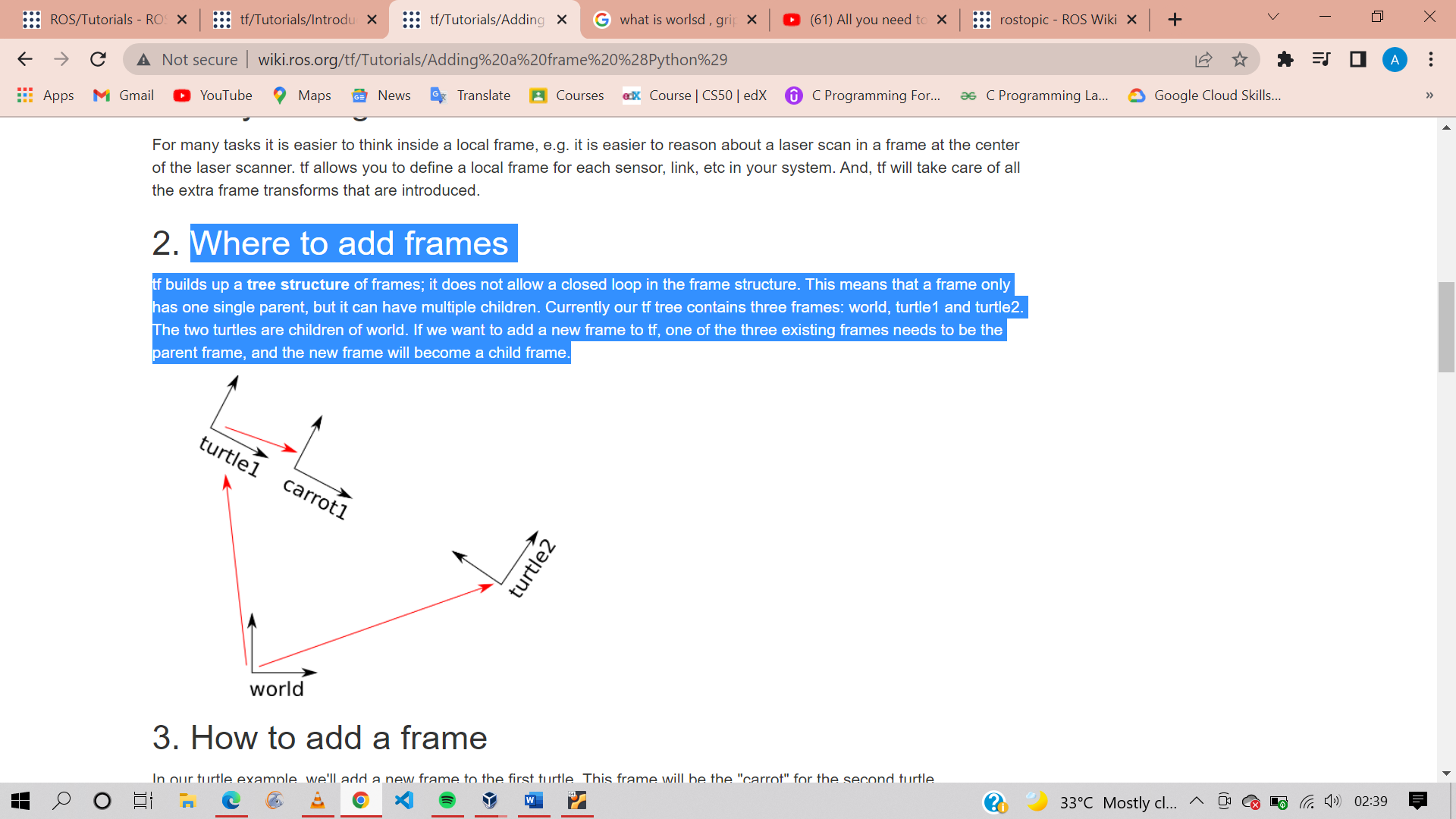
* **Adding a frame (python)**

**Why adding frames**

* For many tasks it is easier to think inside a local frame, e.g. it is easier to reason about a laser scan in a frame at the center of the laser scanner. tf allows you to define a local frame for each sensor, link, etc in your system. And, tf will take care of all the extra frame transforms that are introduced.

**Where to add frames**

* tf builds up a **tree structure** of frames; it does not allow a closed loop in the frame structure. This means that a frame only has one single parent, but it can have multiple children. Currently our tf tree contains three frames: world, turtle1 and turtle2. The two turtles are children of world. If we want to add a new frame to tf, one of the three existing frames needs to be the parent frame, and the new frame will become a child frame.



In our turtle example, we'll add a new frame to the first turtle. This frame will be the "carrot" for the second turtle.

Let's first create the source files. Go to the package we created for the previous tutorials:

$ roscd learning\_tf

$ touch nodes/fixed\_tf\_broadcaster.py

$ chmod +x nodes/fixed\_tf\_broadcaster.py

Now, write the following code in the above mentioned file (fixed\_tf\_broadcaster.py)

#!/usr/bin/env python3

import roslib

roslib.load\_manifest('learning\_tf')

import rospy

import tf

if \_\_name\_\_ == '\_\_main\_\_':

rospy.init\_node('fixed\_tf\_broadcaster')

br = tf.TransformBroadcaster()

rate = rospy.Rate(10.0)

while not rospy.is\_shutdown():

br.sendTransform((0.0, 2.0, 0.0),

(0.0, 0.0, 0.0, 1.0),

rospy.Time.now(),

"carrot1",

"turtle1")

rate.sleep()

Here we create a new transform, from the parent "turtle1" to the new child "carrot1". The carrot1 frame is 2 meters offset from the turtle1 frame.

Edit the start\_demo.launch launch file. Simply add the following line:

<launch>

...

<node pkg="learning\_tf" type="fixed\_tf\_broadcaster.py"

name="broadcaster\_fixed" />

</launch>

Run the roslaunch command to see the change.

Now run this command, this will show you the changing coordinates of the child and parent frame.

$ rosrun tf tf\_echo turtle1 turtle2

* **Broadcasting a moving frame**

The extra frame we published in this tutorial is a fixed frame that doesn't change over time in relation to the parent frame. However, if you want to publish a moving frame you can code the broadcaster to change the frame over time. Let's change our carrot1 frame to change relative to turtle1 over time.

$ roscd learning\_tf

$ touch nodes/dynamic\_tf\_broadcaster.p

$ chmod +x nodes/dynamic\_tf\_broadcaster.py

Now, write the following code in the above mentioned file (dynamic\_tf\_broadcaster.py)

#!/usr/bin/env python

import roslib

roslib.load\_manifest('learning\_tf')

import rospy

import tf

import math

if \_\_name\_\_ == '\_\_main\_\_':

rospy.init\_node('dynamic\_tf\_broadcaster')

br = tf.TransformBroadcaster()

rate = rospy.Rate(10.0)

while not rospy.is\_shutdown():

t = rospy.Time.now().to\_sec() \* math.pi

br.sendTransform((2.0 \* math.sin(t), 2.0 \* math.cos(t), 0.0),

(0.0, 0.0, 0.0, 1.0),

rospy.Time.now(),

"carrot1",

"turtle1")

rate.sleep()

Note that instead of defining a fixed offset from turtle1, we are using a sin and cos function based on the current time to cause the definition of the frame's offset.

Change the launch file to point to our new, dynamic broadcaster for the definition of carrot1 instead of the fixed broadcaster above:

<launch>

...

<node pkg="learning\_tf" type="dynamic\_tf\_broadcaster.py"

name="broadcaster\_dynamic" />

</launch>

Run the roslaunch command to see the carrot frame as the broadcaster frame.