SIT310 – Task 5

Navigation Stack I

Overview

This task focuses on using the ROS navigation stack to control your robot.

Task requirements

a. Ensure you have already completed tasks 1-4 and assessments 1 and 2.
b. To complete this task, you will need to use ROS on the raspberry Pi as we will be using your physical robot
c. If you didn't attend the lecture this week, then either watch the lecture online and/or review the PowerPoint slides on the unit site.
d. Read the task instructions
e. Complete the activities in the task guide
f. Review your progress with your lab tutor or cloud tutor.

Task guide

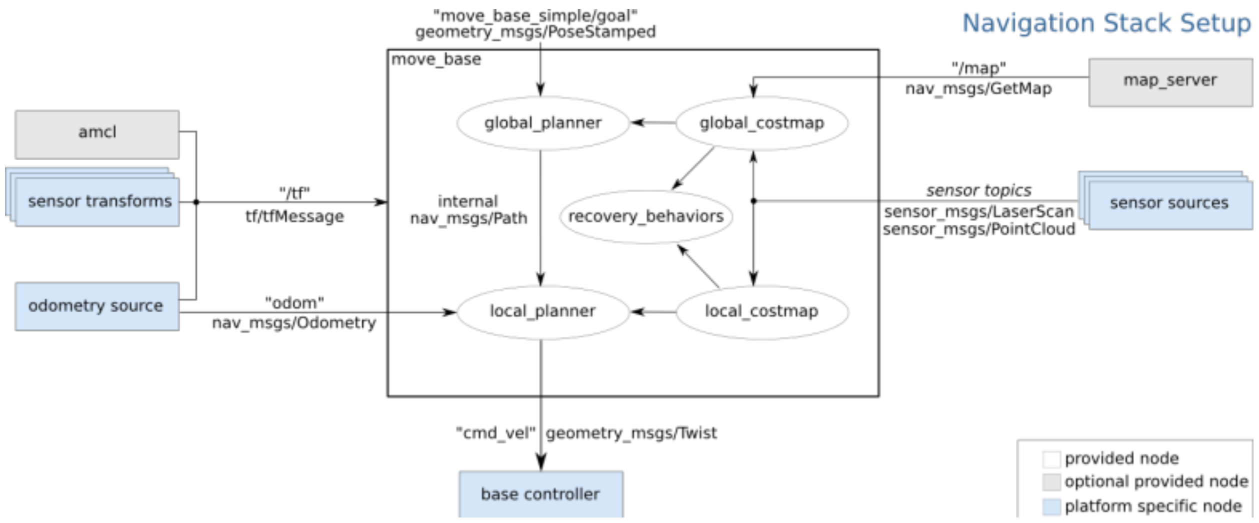
So far, we have created some custom code for the robot to publish sensor messages and move based on published messages. We have also written a custom ROS package that takes sensor messages and publishes messages to request the robot move (the scared robot). This is fine, but the power of ROS is in the vast range of frameworks which provide advanced support for controlling robots. One such framework is the Navigation framework. In this lab and the next one you will learn use the navigation stack to control your robot.

1. Navigation stack

    1.1. Details of the navigation stack can be found here: http://wiki.ros.org/navigation

    1.2. The following diagram shows an example full setup using the navigation stack. It is complicated, so we will build it up bit by bit.

    1.3. This is a bit tricky to understand as written, so we will break it down into smaller parts. The remainder of this tutorial will concentrate on setting up the transform tree of the Zumo in ROS, next week we will focus on publishing sensor information.



2. The Transform library

2.1. The first stage of implementing the ROS navigation stack is to use the transformation library which lets ROS keep track of the robot at all times. It also allows modules to transform the robots coordinates at any time, effectively moving the robot.

2.2. The library we will be using is *tf2* - the second-generation ROS transform library. The is a description of tf2:

"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. tf2 keeps track of all these frames over time, and allows you to ask questions like:

- Where was the head frame relative to the world frame, 5 seconds ago?
- What is the pose of the object in my gripper relative to my base?
- What is the current pose of the base frame in the map frame?

tf2 can operate in a distributed system. This means all the information about the coordinate frames of a robot is available to all ROS components on any computer in the system. Tf2 can operate with a central server that contains all transform information, or you can have every component in your distributed system build its own transform information database."

2.3. If you would like to understand this further, there are resources to read:

- Information about tf2: http://wiki.ros.org/tf2
- A summary of the design of tf2: http://wiki.ros.org/tf2/Design
- An academic paper describing the original tf library: http://wiki.ros.org/Papers/TePRA2013_Foote?action=AttachFile&do=view&target=TePRA2013_Foote.pdf
- Today we will be following some specific parts of the tf2 tutorials form here: http://wiki.ros.org/tf2/Tutorials

3. Tf2 demonstration

This section will take you through a demonstration of the power of tf2

3.1. Run the demo

- First start by opening a terminal and installing the tf2 packages for the turtle program:

```
$ sudo apt-get install ros-kinetic-turtle-tf2 ros-kinetic-tf2-tools ros-kinet
ic-tf
```

- Make sure you have roscore running:

```
$ roscore
```

- Run the demo. You will see two turtles. A still turtle, and a turtle that moves towards the other.

```
$ roslaunch turtle_tf2 turtle_tf2_demo.launch
```

- As before, run the program that can control the turtle with your keys:

```
$ rosrun turtlesim turtle_teleop_key
```

- Have some fun trying to get away from the chasing turtle!   But what is happening?

- This demo is using the tf2 library to create three coordinate frames: a world frame, a turtle1 frame, and a turtle2 frame. This tutorial uses a **tf2 broadcaster** to publish the turtle coordinate frames and a **tf2 listener** to compute the difference in the turtle frames and move the second turtle to follow the other.

3.2. Examine what is happening

- We can use tf2 tools to have a look at what is happening.

- The view_frames program creates a diagram of the frames being broadcast over ROS

- Run the following, then quickly switch back to the key control program and move the robot for 5 seconds.

```
$ rosrun tf2_tools view_frames.py
```

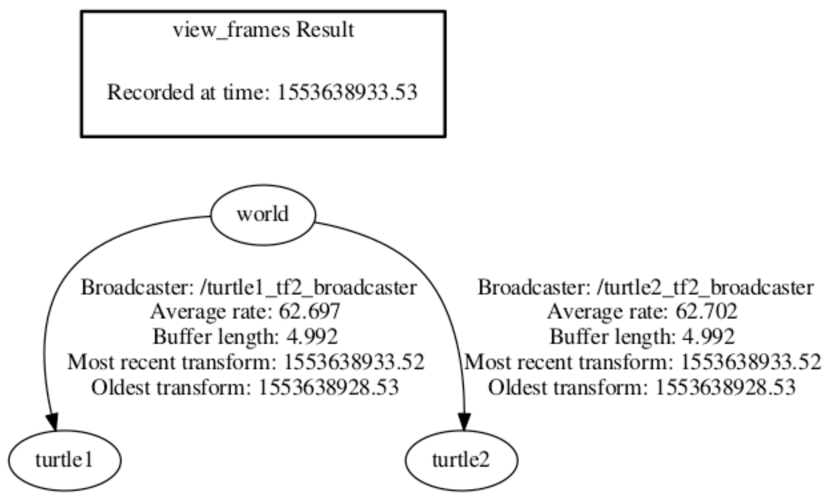
- You will see the following:

```
Listening to tf data during 5 seconds...
Generating graph in frames.pdf file...
```

- Here a tf2 listener is listening to the frames that are being broadcast over ROS and drawing a tree of how the frames are connected. To view the tree, you will need a pdf viewer, install one and then view the pdf, as follows:

```
$ sudo apt-get install evince
$ evince frames.pdf
```

- You will see the following:


view_frames Result
Recorded at time: 1553638933.53
world
Broadcaster: /turtle1_tf2_broadcaster
Average rate: 62.697
Buffer length: 4.992
Most recent transform: 1553638933.52
Oldest transform: 1553638928.53
Broadcaster: /turtle2_tf2_broadcaster
Average rate: 62.702
Buffer length: 4.992
Most recent transform: 1553638933.52
Oldest transform: 1553638928.53
turtle1
turtle2

- Here we can see the three frames that are broadcast by tf2 the world, turtle1, and turtle2 and that world is the parent of the turtle1 and turtle2 frames. view_frames also report some diagnostic information about when the oldest and most recent frame transforms were received and how fast the tf2 frame is published to tf2 for debugging purposes.

3.3. Examining the transforms

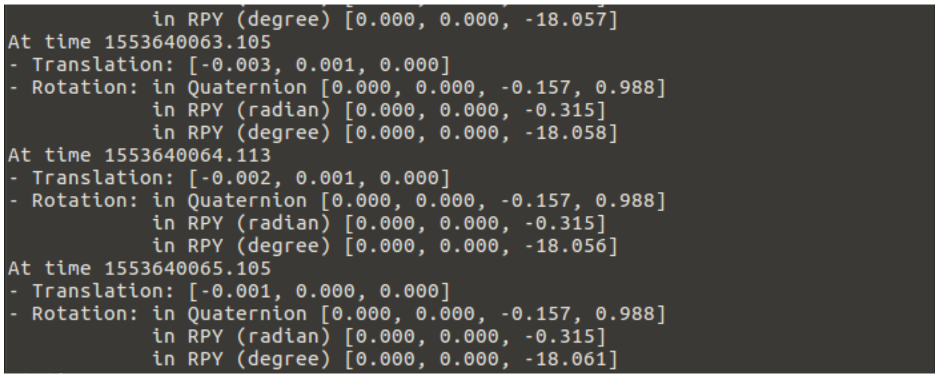
- We can use the program tf_echo to view the transform between any two frames broadcast over ROS.
- Let's look at the transform of the turtle2 frame with respect to turtle1 frame which is equivalent to:

$$\mathbf{T}_{turtle1\_turtle2} = \mathbf{T}_{turtle1\_world} * \mathbf{T}_{world\_turtle2}$$

- Run the following:

```
$ rosrun tf tf_echo turtle1 turtle2
```

- You will see stream of transforms that change as you drive your turtle around.



```
            in RPY (degree) [0.000, 0.000, -18.057]
At time 1553640063.105
- Translation: [-0.003, 0.001, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.157, 0.988]
            in RPY (radian) [0.000, 0.000, -0.315]
            in RPY (degree) [0.000, 0.000, -18.058]
At time 1553640064.113
- Translation: [-0.002, 0.001, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.157, 0.988]
            in RPY (radian) [0.000, 0.000, -0.315]
            in RPY (degree) [0.000, 0.000, -18.056]
At time 1553640065.105
- Translation: [-0.001, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.157, 0.988]
            in RPY (radian) [0.000, 0.000, -0.315]
            in RPY (degree) [0.000, 0.000, -18.061]
```

- You will see that when the turtle2 catches up and eventually settles down, the *Translation* will be 0, because turtle2 is in the exact location as turtle1.
- The *Rotation* values represent the 3D coordinates of the robot. More information here: http://wiki.ros.org/tf2/Tutorials/Quaternions

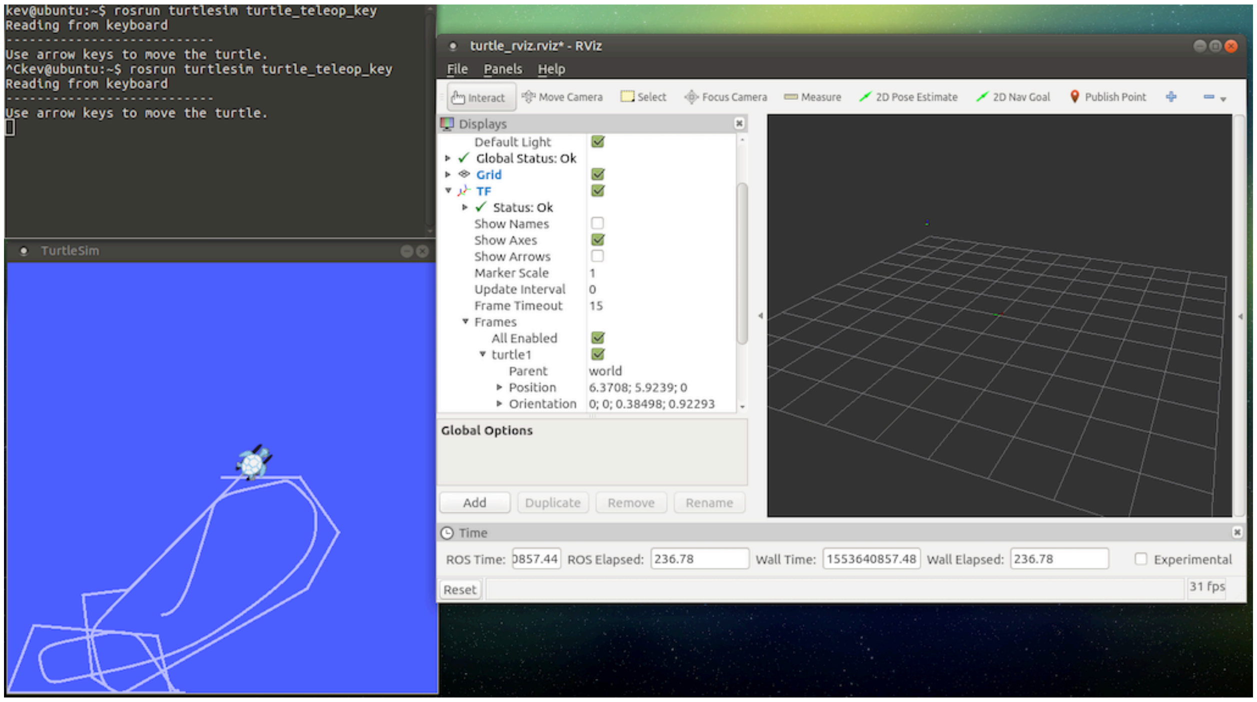
3.4. Visualise the tf2 frames

- rviz is a visualization tool that is useful for examining tf2 frames. It is extremely powerful. There is lots of information here: http://wiki.ros.org/rviz

- Let's look at our turtle frames using rviz. Let's start rviz with the turtle_tf2 configuration file using the -d option for rviz: (make sure you get the quotes correct – these will normally be the ones on the top left of the keyboard)

```
$ rosrun rviz rviz -d `rospack find turtle_tf2`/rviz/turtle_rviz.rviz
```

- You will get a window like the following. You can drive the robot round using the key control program. You will be able to see the robots chase each other in the screen. Expand TF->Frames->turtle1/turtle2/world in the menu to see the frames change.



```
kev@ubuntu:~$ rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
^Ckev@ubuntu:~$ rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

Default Light	✓
Global Status: Ok	
Grid	✓
TF	✓
Status: Ok	
Show Names	☐
Show Axes	✓
Show Arrows	☐
Marker Scale	1
Update Interval	0
Frame Timeout	15
Frames	
All Enabled	✓
turtle1	✓
Parent	world
Position	6.3708; 5.9239; 0
Orientation	0; 0; 0.38498; 0.92293

ROS Time: 0857.44 ROS Elapsed: 236.78 Wall Time: 1553640857.48 Wall Elapsed: 236.78 ☐ Experimental

31 fps

4. Writing transforms for the Zumo robot

4.1. Now that we understand the standard ROS way of writing transforms, we should do the same for the Zumo robot. Also, as we can visualise with rviz we don't need the turtlesim anymore.

4.2. Controlling the robot ourselves

- As we are leaving turtlesim behind, we need to have a way of controlling the robot ourselves without the keyboard controller program.
- We are going to write a simple python keyboard program to generate vmd_vel messages. To get started, we need a python package. Run the following:

```
$ sudo -H pip install pynput==1.0
```

- Now we want to create a new project for this week. Create a new project as follows.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg lab5 tf2 tf2_ros rospy
```

- Check everything is fine.

```
$ cd ~/catkin_ws
catkin_make
```

- If you are getting fed up of sourcing your development folder setup, then issue the following command to add it to your shell startup script. And open a new shell to activate it.

```
$ echo “source ~/catkin_ws/devel/setup.bash” >> ~/.bashrc
```

- Create a new scripts folder.

```
$ roscd lab5
$ mkdir scripts
$ cd scripts
```

- Create a new file zumo_controller.py. I suggest using the gedit editor as it is easy to use. Add it and use it as follows.

```
$ sudo apt-get install gedit
$ gedit zumo_controller.py
```

- Add the following code. As it is a Python program, check the indentation. A good way of doing this, is by going through the program line-by-line and trying to understand it.

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int8
from geometry_msgs.msg import Twist

rospy.init_node('zumo_controller', anonymous=True)
pub = rospy.Publisher('/zumo/cmd_vel', Twist, queue_size=10)

from pynput import keyboard
import getpass

def on_release(key):
    vel_msg = Twist()
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
    vel_msg.angular.z = 0

    if key == keyboard.Key.up:
        print("Up")
        vel_msg.linear.x = 1
        pub.publish(vel_msg)
    if key == keyboard.Key.down:
        print("Down")
        vel_msg.linear.x = -1
        pub.publish(vel_msg)
    if key == keyboard.Key.left:
        print("Left")
        vel_msg.linear.y = 1
        pub.publish(vel_msg)
    if key == keyboard.Key.right:
        vel_msg.linear.y = -1
        pub.publish(vel_msg)
        print("Right")
    if key == keyboard.Key.esc:
        return False

# Collect events until released
with keyboard.Listener(
        on_release=on_release) as listener:
```

```
        listener.join()
listener.start()
```

- Make the program executable.

```
$ chmod +x zumo_controller.py
```

- You will also need an Arduino program that works with these step sizes. Upload the following to your robot using the Arduino IDE. I advise you save it in your catkin ws for lab5.

```
#define USE_USBCON
#include <Wire.h>
#include <Zumo32U4.h>
#include <ros.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/UInt16.h>
ros::NodeHandle nh;

void ros_handler( const geometry_msgs::Twist& cmd_msg) {
float x = cmd_msg.linear.x;
float y = cmd_msg.linear.y;
if(x == 1.0) forward(100);
if(x == -1.0) backward(100);
if(y == 1.0) left(100);
if(y == -1.0) right(100);
stop();
}
ros::Subscriber<geometry_msgs::Twist> sub("/zumo/cmd_vel", ros_handler);
Zumo32U4Motors motors;
void setup()
{
// Uncomment if necessary to correct motor directions:
//motors.flipLeftMotor(true);
//motors.flipRightMotor(true);
nh.initNode();
nh.subscribe(sub);
}

void forward(int time)
{
motors.setLeftSpeed(100);
motors.setRightSpeed(100);
delay(time);
}

void backward(int time)
{
motors.setLeftSpeed(-100);
motors.setRightSpeed(-100);
delay(time);
}

void left(int time)
{
motors.setLeftSpeed(-100);
motors.setRightSpeed(100);
delay(time);
}

void right(int time)
{
motors.setLeftSpeed(100);
motors.setRightSpeed(-100);
delay(time);
}

void stop()
{
motors.setLeftSpeed(0);
motors.setRightSpeed(0);
}
void loop()
{
nh.spinOnce();
delay(1);
}
```

- To test the new controller program you will need to run the following.

```
$ roscore
$ rosrun lab5 zumo_controller.py
$ rostopic echo /zumo/cmd_vel
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

- Now make sure the robot power is on, and use the keys to control the robot. Note that the keyboard input works anywhere – handy (and annoying if you forget to close the program).
- Examine the topics with 'rostopic echo topicname' to see what is going on.

4.3. Calculate the robots Pose and transformation.

- In order to
- A
- Create a new ROS node as follows.

```
$ roscd lab5
$ cd scripts
$ gedit zumo_tf_broadcaster.py
```

- Paste the following. Copy it function by function so you get the Python indentation correct – I suggest using tab.

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

# Because of transformations
import tf_conversions

import tf2_ros
import geometry_msgs.msg
from geometry_msgs.msg import Twist

#for calculating position
import math
from math import sin, cos, pi

x=0
y=0
th=0.0

def handle_zumo_pose(vel_msg):
    global x
    global y
    global th
    br = tf2_ros.TransformBroadcaster()
    t = geometry_msgs.msg.TransformStamped()
```

```
print("x: "+str(x))
print("y: "+str(y))
print("t: "+str(th))

#for rotation, 5 degrees is 0.0872665 rads
if(vel_msg.linear.y==1.0):  th+= 0.0872665
elif(vel_msg.linear.y==-1.0): th-=0.0872665
elif(vel_msg.linear.x==1.0): #this is a movement
    vx = 0.1 #velocity of x, how much we move
    # compute odometry
    delta_x = vx * cos(th)
    delta_y = vx * sin(th)
    delta_th = 0  #no change in angle

    #add the changes to the values
    x += delta_x
    y += delta_y
    th += delta_th

elif(vel_msg.linear.x==-1.0): #this is a movement
    vx = 0.1 #velocity of x, how much we move
    # compute odometry
    delta_x = vx * cos(th)
    delta_y = vx * sin(th)
    delta_th = 0  #no change in angle

    #add the changes to the values
    x -= delta_x
    y -= delta_y
    th -= delta_th


#    y+=vel_msg.linear.y
t.header.stamp = rospy.Time.now()
t.header.frame_id = "world"
t.child_frame_id = "zumo"

#set x,y,z
t.transform.translation.x = x
t.transform.translation.y = y
t.transform.translation.z = 0.0

#set rotation
q = tf_conversions.transformations.quaternion_from_euler(0, 0, th)
t.transform.rotation.x = q[0]
```

```
    t.transform.rotation.y = q[1]
    t.transform.rotation.z = q[2]
    t.transform.rotation.w = q[3]

    br.sendTransform(t)

if __name__ == '__main__':
    rospy.init_node('zumo_tf_broadcaster')
    rospy.Subscriber('/zumo/cmd_vel',
                     Twist,
                     handle_zumo_pose)
    rospy.spin()
```

- Run it check the print statements look correct//

```
$ chmod +x zumo_tf_broadcaster
$ rosrun lab5 zumo_tf_broadcaster
```

- Now, move the robot around, and you should get output as follows. This is the Pose – or location of the robot as it moves forward and then rotates left.

```
x: 0.0
y: 0.0
t: 0.0
x: 0.1
y: 0.0
t: 0.0
x: 0.2
y: 0.0
t: 0.0872665
```

4.4. Visualise the robot with rviz

- As we are now using standard transforms or topics, we can visualize the robot easily
- Leave everything running for now.
- Let's grab the turtle rviz configuration file as it will work fine for our needs

```
$ roscd lab5
$ cp  /opt/ros/kinetic/share/turtle_tf2/rviz/turtle_rviz.rviz zumo.rviz
```

- Now let's run the rviz tool with this

```
$ rviz -d zumo.rviz
```

- You will see a grid with our robot axis in the middle.   Control the robot with your keys as before and you will see it move around.
- In TF->Frames, you will see the world frame and the zumo frames. Observe the Zumo frames changing as we move the robot.

Well, that's all very nice. We have a robot fully controlled by us, publishing its (estimated) position and visualized in a simulator. All this data can be used for navigation or anything you desire.

Next week we will continue to get the Navigation stack working, by getting sensor data working with ROS in a standard way.

5. Additional resources

5.1. Learn about linux

- https://www.computerworld.com/article/2598082/linux/linux-linux-command-line-cheat-sheet.html
-

5.2. Learn about ROS.

- http://wiki.ros.org
- http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom
- http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20broadcaster%20%28Python%29