Worksheet 7 – Introduction to transform libraries and robot model

1. Introduction

The ROS framework has a package to manage the relationships between coordinate frames. This allows the transformation of points, vectors, between any two coordinate frames at any point in time.

http://wiki.ros.org/tf

In this worksheet we will continue to use the tripod depicted in Figure 1. Normally this system is used to make data collection in field tests. That will subsequently be exploited to develop new algorithms that will be migrated to mobile platforms.

Figure 1 – Tripod Sensor Mount.

The tripod possess several sensors:

- 2D Laser scanner with 4 meter range on a tilting platform actuated by a dynamixel servo.
- A thermal camera with a 640x480 resolution and a FOV of 45º x 47º
- A camera with a 1280x720 with a wide FOV of 118º x 69º

n the following exercises we will define the relationships between the different coordinate frames and share them to the ROS framework. First the objective is to define them using the TF C++ implementation. Then we will do the same but with the aid of robot description based on a XML format.

Note: To understand the conventions defined by ROS you should read the following page:

http://www.ros.org/reps/rep-0103.html

2. Transform libraries

Start by creating a ROS package with the needed libraries and packages you need (***roscpp***, ***std_msgs***, ***tf***)

The first objective is to create a ***TF*** broadcaster that will publish to the ROS network the transformation between two coordinate frames. The initial frame is the ***base_footprint*** that represents the floor, then we should publish the height of the tripod at any given moment. Let's imagine that we have a laser that gives us the exact height of the tripod to the floor. This information is passed by the sensor in a topic ***/tripod/height***. You now should create a node that receives this information and publishes a TF from the ***base_footprint*** frame to the ***sensor_frame*** frame (where all the sensors are attached).

So the objective of this first exercise is to subscribe the ***/tripod/height*** whose type is ***std_msgs/Float32*** and publish the transform between ***base_footprint -> sensor_frame***. With your new found ROS proficiency you should now be able to perform this task using only the information on the following tutorial. (Note: the tutorial does almost the exact same task you are required to do)

http://wiki.ros.org/tf/Tutorials/Writing%20a%20tf%20broadcaster%20%28C%2B%2B%29

Now that you've finished your code it's time to test it.

Open a terminal and publish the height you want using the topic aforementioned.

```
$ rostopic pub -r 10 /tripod/height std_msgs/Float32 5
```

Next open RVIZ and subscribe to the TF type remembering that the Fixed Frame should be base_footprint. You can also use the different tools that the TF package has to verify if your transformation is being published correctly.

The next step is to create a subscriber of a TF and print its value to the console. Again there is a very similar tutorial that you should base your implementation on. The transform you are listening now should be ***map*** to ***base_footprint***.

http://wiki.ros.org/tf/Tutorials/Writing%20a%20tf%20listener%20%28C%2B%2B%29

Now to verify that your node is functioning correctly you can use a TF tool.

http://wiki.ros.org/tf#static_transform_publisher

Now open a terminal and place this on the console:

```
$ rosrun tf static_transform_publisher 10.0 0.0 0.0 0.0 0.0 0.0 map base_footprint 100
```

What you are doing is to publish a TF between two frames using the following format:

```
$ rosrun tf static_transform_publisher x y z roll pitch yaw frame1 frame2 period_in_ms
```

Now the final step is to add another frame. Create a function that receives a frame name and publishes a transform between ***base_footprint*** and this new frame with following values **(x: 1, y: 1, z: 1, roll: 0, pitch: 0, yaw: 0)**.

The new frame name should be received in a topic **/new_frame** (***std_msgs/String***). This **TF** should be published at the same rate as the message received. To test it use the rostopic pub function in the console (**Note: If you don't remember how go back to previous worksheets**).

void TfExample::addFrame(std::string name)

3. Introduction to URDF – Universal Robotic Description Format

The TF package simplifies the creation of transformations between coordinate frames. However, as you could ascertain during the previous exercise creating a full description of all the transformations in a Robot would be cumbersome and time consuming.

Consequently, ROS also addresses this problem by using a Robot description format (URDF). In the following page there is an introduction to URDF

wiki.ros.org/urdf

And also the specifications of the XML format of URDF

http://wiki.ros.org/urdf/XML

The first step is to understand how ROS normally describes a Robot Model. Usually there is a package with the following name ***robotname_description*** where all the files that contain the description are located. These include the description of all the parts of the robot including actuators, sensors, mechanical components etc. The normal organization of a robot description package is the following:

First the packages are named ***robotname_description***.

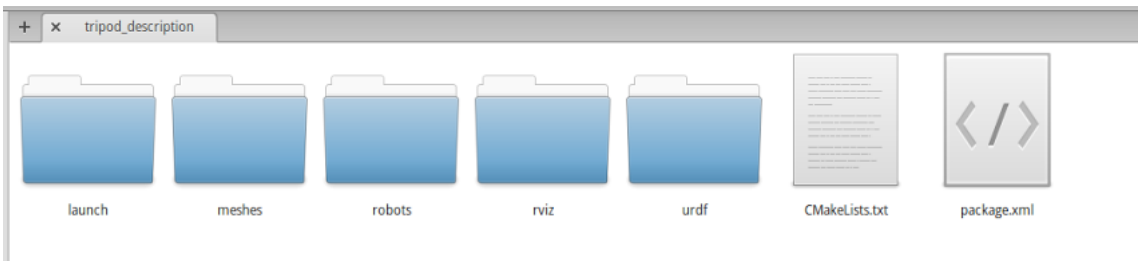

Figure 2- Package example

Inside each folder exists for a specific reason:

launch: You've probably already created launch folders in other worksheets. This folder is where the .launch files will be placed;

meshes: Where the meshes (3D models) of the robot should be placed;

robots: The master files that include the more detailed description of each file;

rviz: This folder is optional but if you create any RVIZ profile you should place it here;

urdf: Where the detailed description of each part of the robot should be placed;

After this brief overview let's go into a more detailed explanation of each one and their dependencies.

Figure 3 – Launch folder

In the launch folder there is a normally a master launch file, i.e. **robot_name.launch** that launches the robot's description so it's available in the ROS framework.

```
<launch>
	<!-- send robot urdf to param server -->
	<arg name="urdf_file" default="$(find xacro)/xacro.py '$(find lesson7)/robots/tripod.urdf.xacro'" />
	<param name="robot_description" command="$(arg urdf_file)" />

	<!-- Use the robot state publisher to send the tf -->
	<node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher" output="screen">
		<param name="publish_frequency" type="double" value="5.0" />
	</node>

	<!-- Only /use_gui = true if there is no other source of joint position information-->
	<node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
		<param name="/use_gui" value="true"/>
	</node>

</launch>
```

Note: All the steps where in following code ***$(find lesson7)*** appears, is to find the lesson7 package if you have a different name you should change it accordingly.

The first part of the code above is to send to the param server the description define in the URDF file. The next step is to use the ***robot_state_publisher*** (http://wiki.ros.org/robot_state_publisher) that will publish the transforms (TF) of fixed joints of the model. The final step is to publish the moveable joints

that are not possible to publish using the only the *robot_state_publisher*. So we use the *joint_state_publisher* (http://wiki.ros.org/joint_state_publisher) that publishes a topic with the position of all the joints (*sensor_msgs/JointState*). That will subsequently be used by the *robot_state_publisher* to send the transform to the ROS network.

Figure 4- Meshes folder

In the meshes folder you should place all the 3D models of the components of the robots. Here you can also create additional folders for the sensors, actuators, etc.

Figure 5 – Robots folder

In the robots folder you should have only the "master" file that calls all the other files that compose your robot. In this case we only have the *tripod_class.urdf.xacro*. But we could have another version of the tripod that included a Kinect and so we could have a *tripod_kinect.urdf.xacro* file.

```
<?xml version="1.0"?>

<!--
- Base : Tripod
- laser : tilting hokuyo
- cameras          : gopro/flir/usb
-->
<robot name="tripod" xmlns:xacro="http://ros.org/wiki/xacro">
        <xacro:include filename="$(find lesson7)/urdf/tripod_library_class.urdf.xacro" />
</robot>
```

In this case we only call the main URDF file.

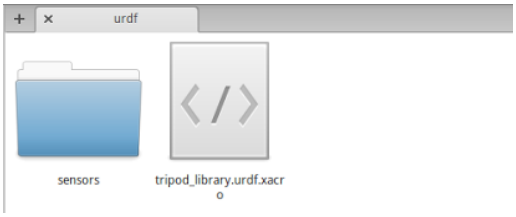

Figure 6 – The URDF folder

Here is where the entire description of the robot is stored. We have a main xacro.urdf file that subsequently includes all the components of the robot. For example the descriptions of the sensors that are inside the folder named sensors.

Figure 7 - Sensors folder

So the basic components of a URDF description are links and joints. A link is a coordinate frame that has a visual, collision definition and physical properties. Where a joint defines the connection between two links (TF). Joints can be fixed or movable (continuous, revolute and prismatic) (Note: For more information go to the URDF tutorial on ROS wiki)

Link example

```
<link name="name">
        <!— Visual definition  -->
        <visual>
                <origin xyz="0 0 0" rpy="0 0 0" />
                <geometry>
                        <box size="0 0 0" />
                </geometry>
        </visual>
        <!— Collision definition  -->
        <collision>
                <origin xyz="0 0 0" rpy="0 0 0" />
                <geometry>
                        <box size="0 0 0" />
                </geometry>
        </collision>
        <!— Inertial characteristics  -->
        <inertial>
                <mass value="0.001" />
                <origin xyz="0 0 0" rpy="0 0 0" />
                <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.000001" iyz="0" izz="0.0001" />
        </inertial>
</link>
```

Joint example

```
<joint name="joint_name" type="fixed">
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <parent link="frame1"/>
        <child link="frame2"/>
</joint>
```

However, there are some shortcuts you can use XACRO that allows you to create macros of XML. The following example is a sensor in this case of a USB camera.

```
<?xml version="1.0"?>
<root xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sensor"
xmlns:controller="http://playerstage.sourceforge.net/gazebo/xmlschema/#controller"
xmlns:interface="http://playerstage.sourceforge.net/gazebo/xmlschema/#interface"
xmlns:xacro="http://ros.org/wiki/xacro">
<property name="M_PI" value="3.1415926535897931" />
<property name="camera_length" value="0.013" />
<property name="camera_width" value="0.057" />
<property name="camera_height" value="0.029" />
<!-- This is commented below and its assumed that the lens is centered -->
<property name="lens_x" value="0.0" />
<property name="lens_y" value="0.0" />
<property name="lens_z" value="0.0" />
<property name="M_PI" value="3.1415926535897931" />
<xacro:macro name="odroid_cam" params="name parent cam_length cam_width cam_height
*origin">
<joint name="${name}_joint" type="fixed">
        <axis xyz="0 1 0" />
        <insert_block name="origin" />
        <parent link="${parent}"/>
        <child link="${name}_link"/>
</joint>
<link name="${name}_link">
        <inertial>
                <mass value="0.001" />
                <origin xyz="0 0 0" rpy="0 0 0" />
                <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.000001" iyz="0" izz="0.0001" />
        </inertial>
        <visual>
                <origin xyz="0 0 ${cam_height/2}" rpy="0 0 0" />
                <geometry>
                        <box size="${cam_length} ${cam_width} ${cam_height}" />
                </geometry>
        </visual>
        <collision>
                <origin xyz="0 0 ${cam_height/2}" rpy="0 0 0" />
                <geometry>
                        <box size="${cam_length} ${cam_width} ${cam_height}" />
                </geometry>
        </collision>
</link>
<!-- go from the base of the sensor to the actual sensor location -->
<joint name="${name}_lens_joint" type="fixed">
        <origin xyz="${cam_length/2} 0.0 ${cam_height/2}" rpy="-${M_PI/2} 0 -${M_PI/2}"/>
        <!-- origin xyz="${lens_x} ${lens_y} ${lens_z}" rpy="-${M_PI/2} 0 -${M_PI/2}"/-->
        <parent link="${name}_link"/>
        <child link="${name}_lens_link"/>
</joint>
```

The basic commands you need to know about XACRO during this worksheet are:

XACRO declaration, has a name and parameters here is an example definition:
```
<xacro:macro name="sensor_xacro" params="name param1 param2 param3 *origin">
```

To define a property you need to create the following line:
```
<property name="property_name" value="X:X" />
```

Then to use the value of a parameter or property value you have to call it inside ${}
```
 <box size="${property_name} ${cam_width} ${cam_height}" />
```

It is also possible to do mathematical operations for example if you want half of the height of the camera, you should do something like this ${cam_height/2} where cam_height is property that you've previously defined.

For more information you should check the following page:

http://wiki.ros.org/xacro

Also you should start to learn to use the tutorials in ROS wiki. This will allow you in the future to test different packages that you probably need in the final lab work of this module.

Implementation:

First of all download the provided package. The main objective is to create a description of the Tripod.

- The source code already has the launch files and folder organization.
- So you only need to edit and fill out the *tripod_library_class.urdf.xacro*
- Inside the file you will find the parts you need to complete whenever the @TODO tag appears.
- In the end of the worksheet you must be able to see the finalized robot_model (type of rviz message) in RVIZ with all the correct distances between each frame.
- The final step is to create a revolute joint between the servo_link and servo_rotation_link (Note: Don't forget you need to use joint_state_publisher to be able to publish moveable joints)

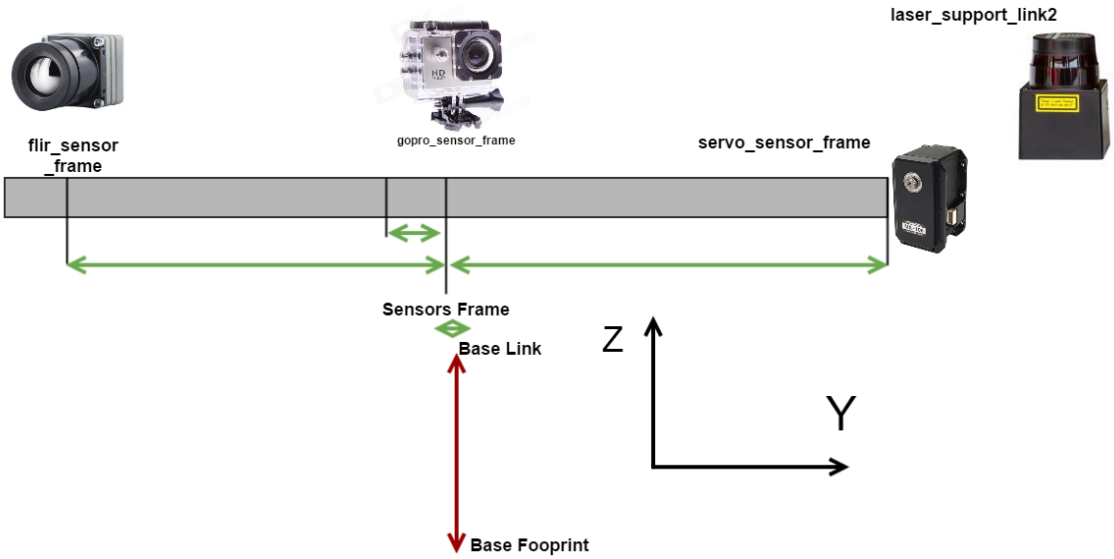

Figure 8- Distances between frames. The arrows depicted in green are the fixed distances that you will need to insert into to your description. While red arrows depict the distances that you can define yourself

Transforms

Frame 1	Frame 2	Distance in (m)
Base Footprint	Base Link	Defined by the user
Base Link	Sensors Frame	X= 0, Y =0, Z= 0
Sensors Frame	flir_sensor_fame	X= 0, Y = -0.173, Z=0
Sensors Frame	camera_sensor_frame	X= 0, Y = -0.053, Z=0
Sensors Frame	servo_sensor_frame	X= 0, Y = 0.095, Z=0

Table 1- Distances between frames

Based on the measurements between the frames, create in the appropriate file a property that defines the offset to the sensors_frame. To do so you will have to understand and complete the xacro sensors_frame. (Note: The provided code has several hints).

The thermal camera dimensions are 22mm x 22mmx 12mm.