ROS Workshop
July 2024

Er. Athira Krishnan R
athirakrishnanr94@gmail.com

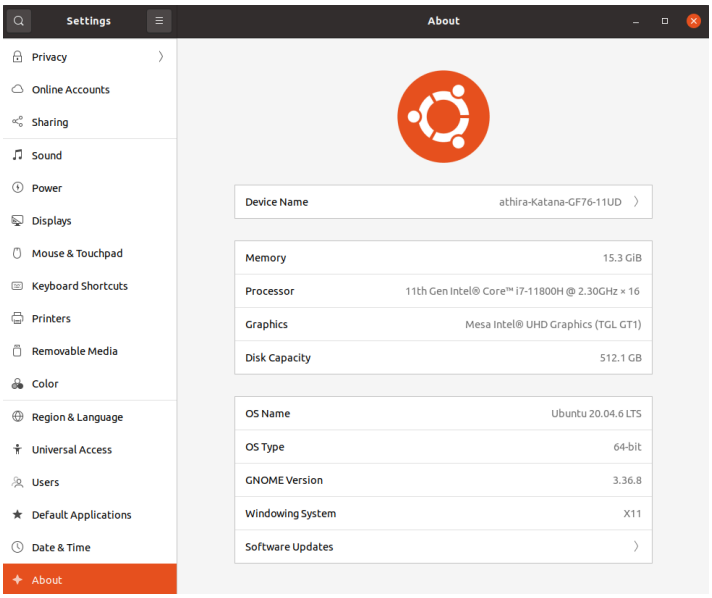
Prerequisites

To start with we need a machine with Ubuntu OS installed. I would recommend to use Ubuntu 20.04 for this set of tutorial.

System requirements can be listed as follows.

- Ubuntu 20.04 machine with,
    - Free space/ storage 200GB. (To kickstart minimum 100GB is required.)
    - GPU support is mandate to load robot models. (Preferably 4GB)
    - Minimum 4GB RAM.

To verify the Installtion pls check Seetings-> About. YOu should see something similar.



Code Editor

- Code editor helps you to ensure that your scripts abide by the standards (PEP8). This include comments, indentation,etc. It also helps in debugging the scripts. VS code editor is the recommended editor. Alternative would be Sublime editor. Once VS code is installed add the plugins for C++, Python and ROS support respectively. To open the editor from a folder in a terminal type

```
code .
```

ROS Installation

Steps to install ROS 2 is officially released in, https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html. But in this section I would walk you through the procedure and the steps in detail to avoid and confusions or set up error.

To set up the locale in a terminal copy paste the command below,

```
locale  # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale  # verify settings
```

To enable ROS2 repository, use the command below,

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

To add the keys for access,

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o
```

```
/usr/share/keyrings/ros-archive-keyring.gpg
```

Now to add them to your source list,

```
echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]
http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME)
main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

To update the repositories and upgrade the machine ,

```
sudo apt update
Sudo apt upgrade
```

To install ROS and its dependent packages,

```
sudo apt install ros-foxy-desktop python3-argcomplete
```

Install development tools,

```
sudo apt install ros-dev-tools
```

In your terminal source

```
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh
source /opt/ros/foxy/setup.bash
```

In ~/.bashrc make entry

```
source /opt/ros/foxy/setup.bash
```

Test the installation

- Open a new terminal and run the following commands,
    - ros2 run demo_nodes_cpp talker
- Open another terminal and run the command,
    - ros2 run demo_nodes_py listener

You should see the `talker` saying that it's `Publishing` messages and the `listener` saying `I heard` those messages. This verifies both the C++ and Python APIs are working properly. Now you are ready to get started.

Basics of ROS

ROS 2 is a middleware based on an anonymous publish/subscribe mechanism that allows for message passing between different ROS processes.

At the heart of any ROS 2 system is the ROS graph. The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate.

The transition from ROS1 to ROS2 offers a multitude of benefits for robotics developers and researchers. The **improvements in performance, real-time capabilities, security, and cross-platform compatibility** make ROS2 a compelling choice for building advanced robotic systems.

Graph Elements

- Nodes: A node is an entity that uses ROS to communicate with other nodes.
- Messages: ROS data type used when subscribing or publishing to a topic.
- Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
- Discovery: The automatic process through which nodes determine how to talk to each other.

Nodes

A node is a participant in the ROS graph. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to Topics. Nodes can also provide or use Services and Actions. There are configurable Parameters associated with a node. Connections between nodes are established through a distributed discovery process. Nodes may be located in the same process, in different processes, or on different machines.

Client Libraries

ROS client libraries allow nodes written in different programming languages to communicate. There is a core ROS client library (RCL) that implements common

functionality needed for the ROS APIs of different languages. This makes it so that language-specific client libraries are easier to write and that they have more consistent behavior.

The following client libraries are maintained by the ROS 2 team:

- rclcpp = C++ client library
- rclpy = Python client library

Discovery

Discovery of nodes happens automatically through the underlying middleware of ROS 2. It can be summarized as follows:

1. When a node is started, it advertises its presence to other nodes on the network with the same ROS domain (set with the ROS_DOMAIN_ID environment variable). Nodes respond to this advertisement with information about themselves so that the appropriate connections can be made and the nodes can communicate.
2. Nodes periodically advertise their presence so that connections can be made with new-found entities, even after the initial discovery period.
3. Nodes advertise to other nodes when they go offline.

Nodes will only establish connections with other nodes if they have compatible Quality of Service settings.

Getting started

ROS2 Workspace

In ROS 2, a workspace serves as the central point for organizing and developing your robot software. It's a directory that houses all the different software packages, data files, and configuration scripts related to your specific robot project.

Steps to create our workspace:

Open a terminal, and type these commands:

```
mkdir -p ~/ros2_ws/src

cd ~/ros2_ws/

colcon build

echo "source ~/ros2_ws/install/setup.bash" >> ~/.bashrc
```

That's it! You have now created a ROS 2 workspace.

ROS2 Package

A ROS 2 package is the fundamental building block of robot software in ROS 2. Each package contains a specific piece of functionality or capability, like motor control, sensor processing, or communication with other systems.

```
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_python <package_name>
cd ~/ros2_ws
colcon build
```

To check whether a new package got registered. Either open a new terminal window or source the bashrc file.

```
source ~/.bashrc
ros2 pkg list

sudo apt-get update -y

sudo apt-get upgrade -y
```

To install some ROS2 helper packages,

```
sudo apt-get install ros-foxy-gazebo-ros
sudo apt-get install ros-foxy-gazebo-ros2-control
sudo apt-get install ros-foxy-joint-state-publisher-gui
sudo apt-get install ros-foxy-moveit
sudo apt-get install ros-foxy-xacro
sudo apt-get install ros-foxy-ros2-control
sudo apt-get install ros-foxy-ros2-controllers
sudo apt-get install libserial-dev
sudo apt-get install python3-pip
pip install pyserial
```

colcon is the primary command-line tool for building, testing, and installing ROS packages.

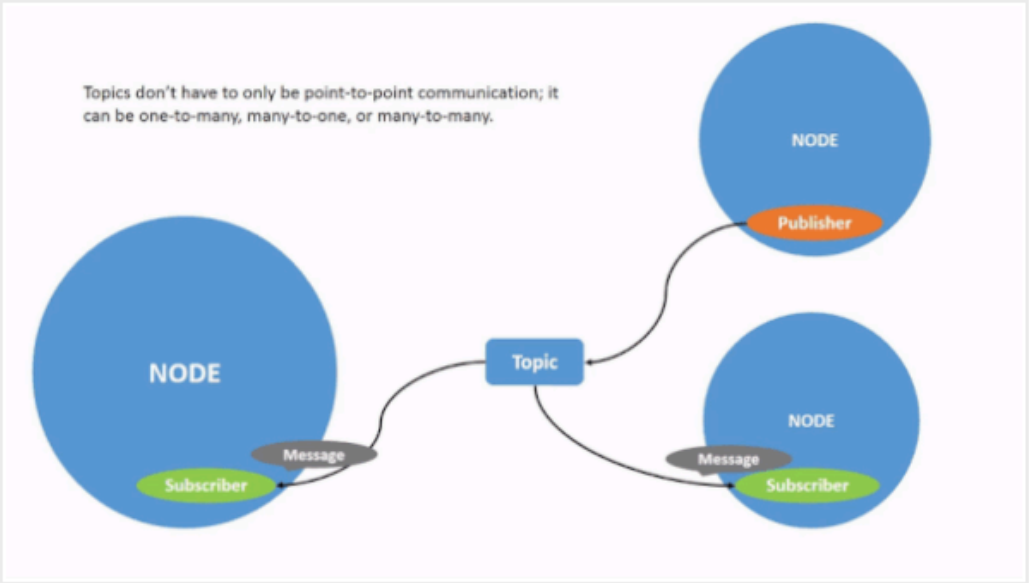
In your terminal window, type the following command:

```
echo "source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash" >>
~/.bashrc
```

Publishers and Subscribers

So, the publisher is the one who shares the information, the subscriber is the one who listens and learns, and the topic is the special channel they use to talk to each other. This

way, all the robot's helpers can work together and keep it safe and aware of its surroundings, just like a superhero team!



Inside the python package you created above, there should be another folder with the same name. Create 2 python file inside that folder publisher.py and subscriber.py and paste this code in publisher.py.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    rclpy.init(args=args)
```

```
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
class MinimalSubscriber(Node):

        def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
        String,
        'topic',
        self.listener_callback,
        10)
        self.subscription  # prevent unused variable warning

        def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)
def main(args=None):
        rclpy.init(args=args)
        minimal_subscriber = MinimalSubscriber()
        rclpy.spin(minimal_subscriber)
        # Destroy the node explicitly
        # (optional - otherwise it will be done automatically
        # when the garbage collector destroys the node object)
        minimal_subscriber.destroy_node()
        rclpy.shutdown()
if __name__ == '__main__':
        main()
```

In package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
```

```
  <name>basics</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="athira@todo.todo">athira</maintainer>
  <license>TODO: License declaration</license>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
        <build_type>ament_python</build_type>
  </export>
  <exec_depend>rclpy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
</package>
```

In setup.py

```
from setuptools import setup


package_name = 'basics'
setup(
   name=package_name,
   version='0.0.0',
   packages=[package_name],
   data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
   ],
   install_requires=['setuptools'],
   zip_safe=True,
   maintainer='athira',
   maintainer_email='athira@todo.todo',
   description='TODO: Package description',
   license='TODO: License declaration',
   tests_require=['pytest'],
   entry_points={
        'console_scripts': [
```

```
                'talker = basics.publisher:main',
                'listener = basics.subscriber:main'
        ],
},
)
```

rosdep install --from-paths src --ignore-src -r --rosdistro <distro> -y

colcon build

Turtlesim

You can start the main application by simply executing two of its nodes. The package name you need in this case is turtlesim and the nodes you need to start are turtlesim_node and turtle_teleop_key. Make sure to source ROS 2 and run these nodes in two separate terminals.

```
ros2 run turtlesim turtlesim_node

ros2 run turtlesim turtle_teleop_key
```

To observe the turtle and its features,

```
ros2 node list

ros2 topic list -t

ros2 topic info /turtle1/cmd_vel

ros2 interface show turtlesim/msg/Pose

ros2 service list

ros2 interface show turtlesim/srv/Spawn

ros2 interface proto turtlesim/srv/Spawn
```

To interact with the turtle,

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 5,y: 5,theta: 0}"

ros2 service call /reset std_srvs/srv/Empty

ros2 service call -r 0.5 /spawn turtlesim/srv/Spawn "{x: 5,y:
5,theta: 0}"

ros2 param set /turtlesim background_r 125

ros2 action send_goal /turtle1/rotate_absolute
turtlesim/action/RotateAbsolute {'theta: -1.57'} --feedback
```

Transform Library TF2

Here we would discuss the details of transforms with the TF2 library. It will introduce Dynamic Transform Broadcasters, Transform Listeners and Static Transform Broadcasters.

For this part also we will use the Turtlesim again. This time, we will insert a second turtle into the simulation. As you drive the first turtle around with keyboard teleop, the second turte will follow it closely. In order to do this, the second turtle needs to know where the first one is, w.r.t. its own coordinate frames. This can be easily achieved using the TF2 library.

1. Dynamic TF Broadcaster

Our aim is to continuously broadcast the current position of the first turtle. Create an ament python package with dependencies on tf2_ros.

```
ros2 pkg create --build-type ament_python tf --dependencies tf2_ros rclpy
pip3 install scipy
```

Create an executable broadcaster.py

```
#! /usr/bin/env python3

import rclpy
import sys

from geometry_msgs.msg import TransformStamped
from rclpy.node import Node
from scipy.spatial.transform import Rotation as R
from tf2_ros.transform_broadcaster import TransformBroadcaster
from turtlesim.msg import Pose

class DynamicBroadcaster(Node):

    def __init__(self, turtle_name):
        super().__init__('dynamic_broadcaster')
        self.name_ = turtle_name
        self.get_logger().info("Broadcasting pose of : {}".format(self.name_))
        self.tfb_ = TransformBroadcaster(self)
        self.sub_pose = self.create_subscription(Pose, "{}/pose".format(self.name_),
self.handle_pose, 10)

    def handle_pose(self, msg):

        tfs = TransformStamped()
        tfs.header.stamp = self.get_clock().now().to_msg()
        tfs.header.frame_id="world"
        tfs._child_frame_id = self.name_
        tfs.transform.translation.x = msg.x
        tfs.transform.translation.y = msg.y
        tfs.transform.translation.z = 0.0

        r = R.from_euler('xyz',[0,0,msg.theta])

        tfs.transform.rotation.x = r.as_quat()[0]
        tfs.transform.rotation.y = r.as_quat()[1]
        tfs.transform.rotation.z = r.as_quat()[2]
        tfs.transform.rotation.w = r.as_quat()[3]

        self.tfb_.sendTransform(tfs)

def main(argv=sys.argv[1]):
    rclpy.init(args=argv)
    node = DynamicBroadcaster(sys.argv[1])
```

```
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass

    node.destroy_node()
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```

To adjust the entry points, in setup.py

```
entry_points={
        'console_scripts': [
                'broadcaster = follower.broadcaster:main',
        ],
},
```

Build the package. Now to test the script,

```
ros2 run turtlesim turtlesim_node
ros2 run follower broadcaster turtle1
```

To verify the working in a terminal,

```
ros2 run tf2_ros tf2_echo turtle1 world
```

2. Dynamic TF Listener

To create a TF listener, create another file listener.py and copy the script.

```
#!/usr/bin/env python3

import sys
import math

from geometry_msgs.msg import Twist

import rclpy
from rclpy.node import Node
```

```
from rclpy.qos import QoSProfile
from tf2_ros.transform_listener import TransformListener
from tf2_ros.buffer import Buffer
from tf2_ros import LookupException

class TfListener(Node):

    def __init__(self, first_turtle, second_turtle):
        super().__init__('tf_listener')
        self.first_name_ = first_turtle
        self.second_name_ = second_turtle
        self.get_logger().info("Transforming from {} to {}".format(self.second_name_,
self.first_name_))
        self._tf_buffer = Buffer()
        self._tf_listener = TransformListener(self._tf_buffer, self)
        self.cmd_ = Twist ()
        self.publisher_ = self.create_publisher(Twist,
"{}/cmd_vel".format(self.second_name_),10)
        self.timer = self.create_timer(0.33, self.timer_callback) #30 Hz = 0.333s

    def timer_callback(self):
        try:
            trans = self._tf_buffer.lookup_transform(self.second_name_, self.first_name_,
rclpy.time.Time())
            self.cmd_.linear.x = math.sqrt(trans.transform.translation.x ** 2 +
trans.transform.translation.y ** 2)
            self.cmd_.angular.z = 4 * math.atan2(trans.transform.translation.y ,
trans.transform.translation.x)
            self.publisher_.publish(self.cmd_)

        except LookupException as e:
            self.get_logger().error('failed to get transform {} \n'.format(repr(e)))

def main(argv=sys.argv):
    rclpy.init(args=argv)
    node = TfListener(sys.argv[1], sys.argv[2])
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()
```

Add the script to setup.py and build the package. To test the working,

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: "turtle2"}"
ros2 run follower broadcaster turtle2
ros2 run follower listener turtle1 turtle2
```

URDF

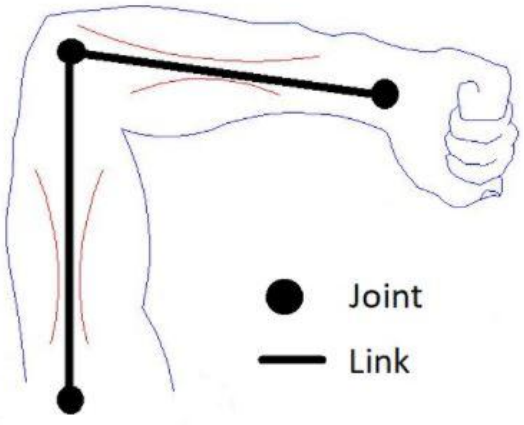
A URDF (Universal Robot Description Format) file is an XML file that describes what a robot should look like in real life. It contains the complete physical description of the robot.
The body of a robot consists of two components:

1. Links
2. Joints

Links are the rigid pieces of a robot. They are the “bones”.

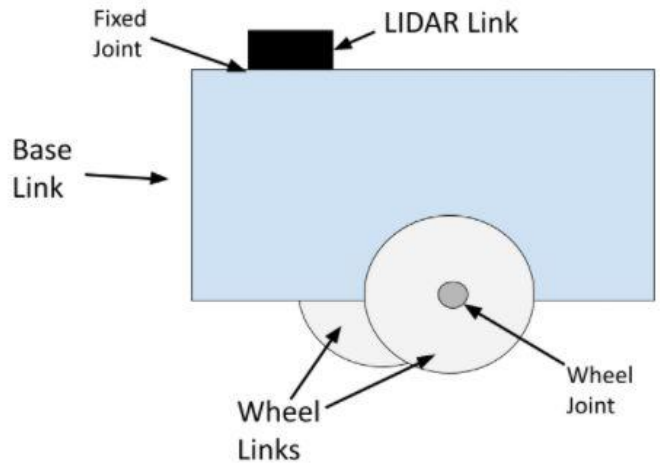
Links are connected to each other by joints. Joints are the pieces of the robot that move, enabling motion between connected links.

Consider the human arm below as an example. The shoulder, elbow, and wrist are joints. The upper arm, forearm and palm of the hand are links.



For a robotic arm, links and joints look like this.

For a mobile robot with LIDAR, links and joints look like this:



The wheel joints are revolute joints. Revolute joints cause rotational motion. The wheel joints in the photo connect the wheel link to the base link.

Fixed joints have no motion at all. You can see that the LIDAR is connected to the base of the robot via a fixed joint (i.e. this could be a simple screw that connects the LIDAR to the base of the robot).

```
sudo apt install ros-foxy-joint-state-publisher-gui
sudo apt install ros-foxy-xacro
ros2 pkg create --build-type ament_cmake basic_mobile_robot
#Create some extra folders with the following names.

mkdir config launch maps meshes models params rviz worlds

colcon build
```

Create a new file named **basic_mobile_bot_v1.urdf**.

```
gedit basic_mobile_bot_v1.urdf
```

Create the URDF File

I am going to open up a terminal window, and type the following command to go to the directory where my URDF file will be located.

cd ~/ros_ws/src/basic_mobile_bot/model

Add your URDF file to this folder. For example, you can add a URDF file like this:

```
<?xml version="1.0" ?>
<robot name="basic_mobile_bot" xmlns:xacro="http://ros.org/wiki/xacro">

  <!-- ****************** ROBOT CONSTANTS ******************************  -->
  <!-- Define the size of the robot's main chassis in meters -->
  <xacro:property name="base_width" value="0.39"/>
  <xacro:property name="base_length" value="0.70"/>
  <xacro:property name="base_height" value="0.20"/>

  <!-- Define the shape of the robot's two back wheels in meters -->
  <xacro:property name="wheel_radius" value="0.14"/>
  <xacro:property name="wheel_width" value="0.06"/>

  <!-- x-axis points forward, y-axis points to left, z-axis points upwards -->
  <!-- Define the gap between the wheel and chassis along y-axis in meters -->
  <xacro:property name="wheel_ygap" value="0.035"/>

  <!-- Position the wheels along the z-axis -->
  <xacro:property name="wheel_zoff" value="0.05"/>

  <!-- Position the wheels along the x-axis -->
  <xacro:property name="wheel_xoff" value="0.221"/>

  <!-- Position the caster wheel along the x-axis -->
  <xacro:property name="caster_xoff" value="0.217"/>

  <!-- Define intertial property macros  -->
```

```
  <xacro:macro name="box_inertia" params="m w h d">
          <inertial>
          <origin xyz="0 0 0" rpy="${pi/2} 0 ${pi/2}"/>
          <mass value="${m}"/>
          <inertia ixx="${(m/12) * (h*h + d*d)}" ixy="0.0" ixz="0.0" iyy="${(m/12) * (w*w +
d*d)}" iyz="0.0" izz="${(m/12) * (w*w + h*h)}"/>
          </inertial>
  </xacro:macro>

  <xacro:macro name="cylinder_inertia" params="m r h">
          <inertial>
          <origin xyz="0 0 0" rpy="${pi/2} 0 0" />
          <mass value="${m}"/>
          <inertia ixx="${(m/12) * (3*r*r + h*h)}" ixy = "0" ixz = "0" iyy="${(m/12) * (3*r*r +
h*h)}" iyz = "0" izz="${(m/2) * (r*r)}"/>
          </inertial>
  </xacro:macro>

  <xacro:macro name="sphere_inertia" params="m r">
          <inertial>
          <mass value="${m}"/>
          <inertia ixx="${(2/5) * m * (r*r)}" ixy="0.0" ixz="0.0" iyy="${(2/5) * m * (r*r)}"
iyz="0.0" izz="${(2/5) * m * (r*r)}"/>
          </inertial>
  </xacro:macro>

  <!-- ****************** ROBOT BASE FOOTPRINT ************************** -->
  <!-- Define the center of the main robot chassis projected on the ground -->
  <link name="base_footprint"/>

  <!-- The base footprint of the robot is located underneath the chassis -->
  <joint name="base_joint" type="fixed">
          <parent link="base_footprint"/>
          <child link="base_link" />
          <origin xyz="0.0 0.0 ${(wheel_radius+wheel_zoff)}" rpy="0 0 0"/>
  </joint>

  <!-- ********************** ROBOT BASE ********************************* -->
  <link name="base_link">
          <visual>
          <origin xyz="0 0 -0.05" rpy="1.5707963267949 0 3.141592654"/>
          <geometry>
          <mesh filename="package://basic_mobile_robot/meshes/robot_base.stl" />
          </geometry>
          <material name="Red">
          <color rgba="${255/255} ${0/255} ${0/255} 1.0"/>
          </material>
```

```
        </visual>

        <collision>
        <geometry>
        <box size="${base_length} ${base_width} ${base_height}"/>
        </geometry>
        </collision>

        <xacro:box_inertia m="40.0" w="${base_width}" d="${base_length}"
h="${base_height}"/>

 </link>

 <gazebo reference="base_link">
        <material>Gazebo/Red</material>
 </gazebo>

 <!-- ********************** DRIVE WHEELS ******************************* -->

 <xacro:macro name="wheel" params="prefix x_reflect y_reflect">
        <link name="${prefix}_link">
        <visual>
        <origin xyz="0 0 0" rpy="1.5707963267949 0 0"/>
        <geometry>
        <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
        </geometry>
        <material name="White">
        <color rgba="${255/255} ${255/255} ${255/255} 1.0"/>
        </material>
        </visual>

        <collision>
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
        <geometry>
        <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
        </geometry>
        </collision>

        <xacro:cylinder_inertia m="110.5" r="${wheel_radius}" h="${wheel_width}"/>

        </link>

        <!-- Connect the wheels to the base_link at the appropriate location, and
        define a continuous joint to allow the wheels to freely rotate about
        an axis -->
        <joint name="${prefix}_joint" type="revolute">
        <parent link="base_link"/>
```

```
        <child link="${prefix}_link"/>
        <origin xyz="${x_reflect*wheel_xoff} ${y_reflect*(base_width/2+wheel_ygap)}
${-wheel_zoff}" rpy="0 0 0"/>
        <limit upper="3.1415" lower="-3.1415" effort="30" velocity="5.0"/>
        <axis xyz="0 1 0"/>
        </joint>
  </xacro:macro>

  <!-- Instantiate two wheels using the macro we just made through the
        xacro:wheel tags. We also define the parameters to have one wheel
        on both sides at the back of our robot (i.e. x_reflect=-1). -->
  <xacro:wheel prefix="drivewhl_l" x_reflect="-1" y_reflect="1" />
  <xacro:wheel prefix="drivewhl_r" x_reflect="-1" y_reflect="-1" />

  <!-- ********************** CASTER WHEEL ****************************** -->
  <!-- We add a caster wheel. It will be modeled as sphere.
        We define the wheel's geometry, material and the joint to connect it to
        base_link at the appropriate location. -->
  <link name="front_caster">
        <visual>
        <geometry>
        <sphere radius="${(wheel_radius+wheel_zoff-(base_height/2))}"/>
        </geometry>
        <material name="White">
        <color rgba="${255/255} ${255/255} ${255/255} 1.0"/>
        </material>
        </visual>
        <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
        <sphere radius="${(wheel_radius+wheel_zoff-(base_height/2))}"/>
        </geometry>
        </collision>
        <xacro:sphere_inertia m="10.05"
r="${(wheel_radius+wheel_zoff-(base_height/2))}"/>
  </link>

  <gazebo reference="front_caster">
        <mu1>0.01</mu1>
        <mu2>0.01</mu2>
        <material>Gazebo/White</material>
  </gazebo>

  <joint name="caster_joint" type="fixed">
        <parent link="base_link"/>
        <child link="front_caster"/>
        <origin xyz="${caster_xoff} 0.0 ${-(base_height/2)}" rpy="0 0 0"/>
```

```
  </joint>

  <!-- ********************** IMU SETUP ********************************  -->
  <!-- Each sensor must be attached to a link.                           -->

  <joint name="imu_joint" type="fixed">
          <parent link="base_link"/>
          <child link="imu_link"/>
          <origin xyz="-0.10 0 0.05" rpy="0 0 0"/>
  </joint>

  <link name="imu_link"/>

  <!-- ********************** GPS SETUP ********************************  -->
  <joint name="gps_joint" type="fixed">
          <parent link="base_link"/>
          <child link="gps_link"/>
          <origin xyz="0.10 0 0.05" rpy="0 0 0"/>
  </joint>

  <link name="gps_link"/>

</robot>
```

Now go to the meshes folder. Add the following STL files to your meshes folder. A mesh is a file that allows your robot to look more realistic

In Package.xml,

After the <buildtool_depend> tag, add the following lines:

```
<exec_depend>joint_state_publisher</exec_depend>

<exec_depend>robot_state_publisher</exec_depend>

<exec_depend>rviz</exec_depend>

<exec_depend>xacro</exec_depend>
```

Save the file, and close it.

Create the Launch File

Now we want to create a launch file.

I am going to go to my launch folder and create the file. Here is the command I will type in my terminal window.

cd ~/ros_ws/src/basic_mobile_bot/launch

gedit launch_urdf_into_gazebo.launch.py

Type the following code inside the file.

```
import os
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.conditions import IfCondition, UnlessCondition
from launch.substitutions import Command, LaunchConfiguration
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():

  # Set the path to different files and folders.
  pkg_share = FindPackageShare(package='basic_mobile_robot').find('basic_mobile_robot')
  default_launch_dir = os.path.join(pkg_share, 'launch')
  default_model_path = os.path.join(pkg_share, 'models/basic_mobile_bot_v1.urdf')
  robot_name_in_urdf = 'basic_mobile_bot'
  default_rviz_config_path = os.path.join(pkg_share, 'rviz/urdf_config.rviz')

  # Launch configuration variables specific to simulation
  gui = LaunchConfiguration('gui')
  model = LaunchConfiguration('model')
  rviz_config_file = LaunchConfiguration('rviz_config_file')
```

```
use_robot_state_pub = LaunchConfiguration('use_robot_state_pub')
use_rviz = LaunchConfiguration('use_rviz')
use_sim_time = LaunchConfiguration('use_sim_time')

# Declare the launch arguments
declare_model_path_cmd = DeclareLaunchArgument(
        name='model',
        default_value=default_model_path,
        description='Absolute path to robot urdf file')

declare_rviz_config_file_cmd = DeclareLaunchArgument(
        name='rviz_config_file',
        default_value=default_rviz_config_path,
        description='Full path to the RVIZ config file to use')

declare_use_joint_state_publisher_cmd = DeclareLaunchArgument(
        name='gui',
        default_value='True',
        description='Flag to enable joint_state_publisher_gui')

declare_use_robot_state_pub_cmd = DeclareLaunchArgument(
        name='use_robot_state_pub',
        default_value='True',
        description='Whether to start the robot state publisher')

declare_use_rviz_cmd = DeclareLaunchArgument(
        name='use_rviz',
        default_value='True',
        description='Whether to start RVIZ')

declare_use_sim_time_cmd = DeclareLaunchArgument(
        name='use_sim_time',
        default_value='True',
        description='Use simulation (Gazebo) clock if true')

# Specify the actions

# Publish the joint state values for the non-fixed joints in the URDF file.
start_joint_state_publisher_cmd = Node(
        condition=UnlessCondition(gui),
        package='joint_state_publisher',
        executable='joint_state_publisher',
        name='joint_state_publisher')

# A GUI to manipulate the joint state values
start_joint_state_publisher_gui_node = Node(
        condition=IfCondition(gui),
```

```
        package='joint_state_publisher_gui',
        executable='joint_state_publisher_gui',
        name='joint_state_publisher_gui')

# Subscribe to the joint states of the robot, and publish the 3D pose of each link.
start_robot_state_publisher_cmd = Node(
        condition=IfCondition(use_robot_state_pub),
        package='robot_state_publisher',
        executable='robot_state_publisher',
        parameters=[{'use_sim_time': use_sim_time,
        'robot_description': Command(['xacro ', model])}],
        arguments=[default_model_path])

# Launch RViz
start_rviz_cmd = Node(
        condition=IfCondition(use_rviz),
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        output='screen',
        arguments=['-d', rviz_config_file])

# Create the launch description and populate
ld = LaunchDescription()

# Declare the launch options
ld.add_action(declare_model_path_cmd)
ld.add_action(declare_rviz_config_file_cmd)
ld.add_action(declare_use_joint_state_publisher_cmd)
ld.add_action(declare_use_robot_state_pub_cmd)
ld.add_action(declare_use_rviz_cmd)
ld.add_action(declare_use_sim_time_cmd)

# Add any actions
ld.add_action(start_joint_state_publisher_cmd)
ld.add_action(start_joint_state_publisher_gui_node)
ld.add_action(start_robot_state_publisher_cmd)
ld.add_action(start_rviz_cmd)

return ld
```

In rviz folder copy the script shared.

Add the following snippet to CMakeLists.txt file above the if(BUILD_TESTING) line.

```
install(
  DIRECTORY config launch maps meshes models params
rviz src worlds
  DESTINATION share/${PROJECT_NAME}
)
```

Save the file and close it.

Build the Package

Go to the root folder.

cd ~/ros_ws/

Build the package.

colcon build

colcon build --packages-select <package_name>

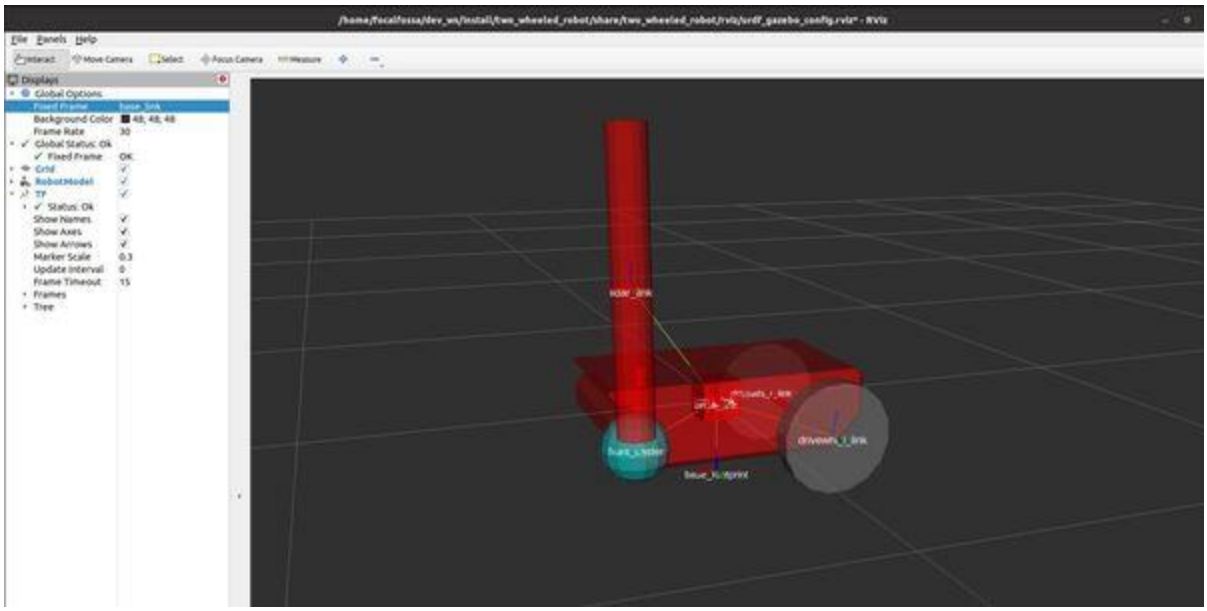
Launch the Launch File

Now let's launch the launch file.

cd ~/ros_ws/

ros2 launch basic_mobile_robot basic_mobile_bot_v1.launch.py

Here is the output:



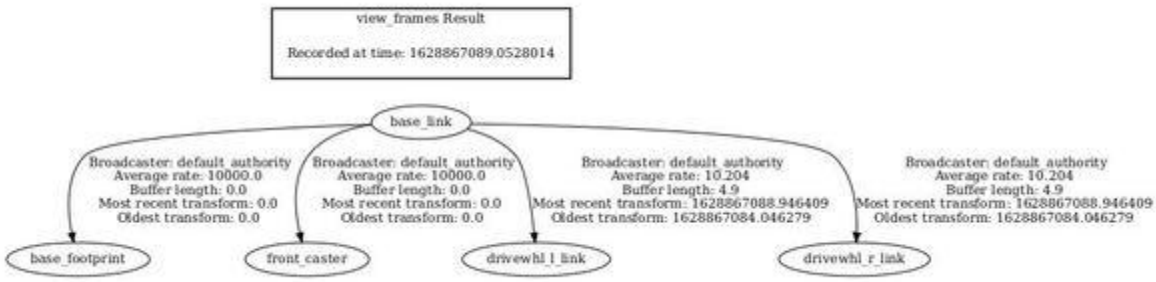
sudo apt install ros-foxy-tf2-tools
Check out the coordinate frames.

ros2 run tf2_tools view_frames.py

In the current working directory, you will have a file called **frames.pdf**. Open that file.

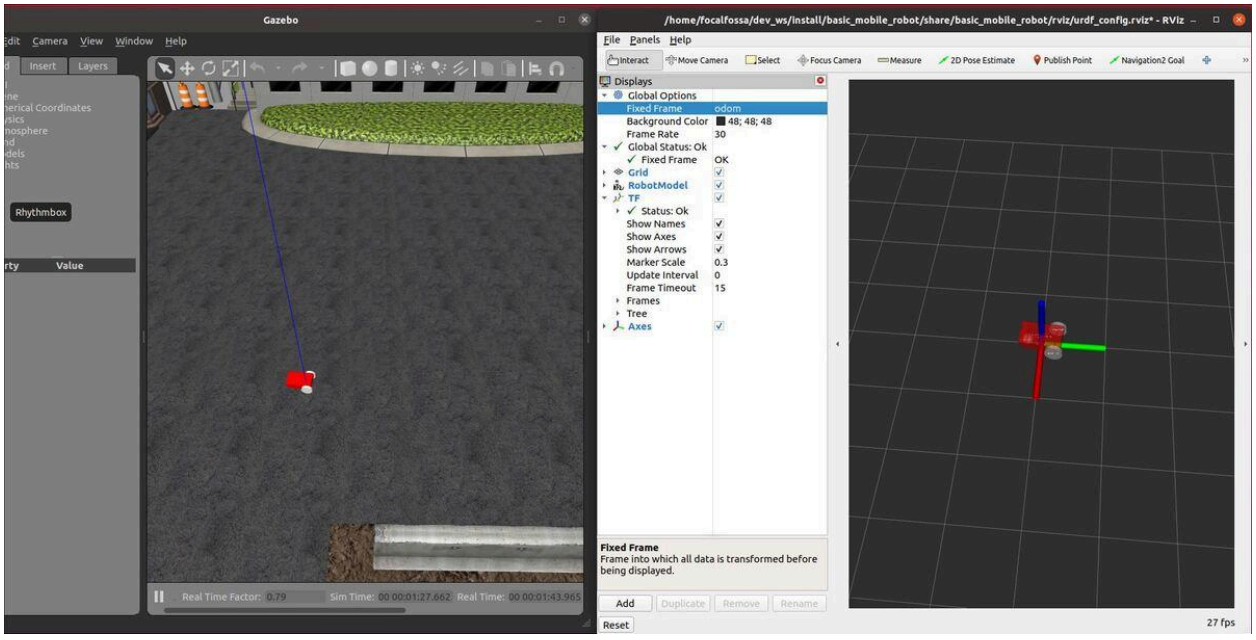
evince frames.pdf

Here is what my coordinate transform (i.e. tf) tree looks like:



Adding Odometry

sudo apt install ros-foxy-gazebo-ros-pkgs



In this tutorial, I will show you how to set up the odometry for a mobile robot. You can get the entire code for this project https://drive.google.com/drive/folders/1dW1fOxWoWvbkt6oxyMGKII7wVY3LXm_V .

Odometry in ROS 2

In robotics, odometry is about using data from sensors to estimate the change in a robot's position, orientation, and velocity over time relative to some point (e.g. x=0, y=0, z=0). Odometry information is normally obtained from sensors such as wheel encoders, IMU (Inertial measurement unit), and LIDAR.

In ROS, the coordinate frame most commonly used for odometry is known as the **odom** frame. Just like an odometer in your car which measures wheel rotations to determine the distance your car has traveled from some starting point, the odom frame is the point in the world where the robot first starts moving.

A robot's position and orientation within the odom frame becomes less accurate over time and distance because sensors like IMUs (that measure acceleration) and wheel encoders (that measure the number of times each wheel has rotated) are not perfect.

For example, imagine your robot runs into a wall. Its wheels might spin repeatedly without the robot moving anywhere (we call this **wheel slip**). The wheel odometry would indicate a further distance traveled by the robot than reality. We have to adjust for these inaccuracies.

To learn all about coordinate frames for mobile robots (including the odom frame), check out this post.

To correct for the inaccuracies of sensors, most ROS applications have an additional coordinate frame called **map** that provides globally accurate information and corrects drift that happens within the odom frame.

The ROS 2 Navigation Stack requires:

1. Publishing of nav_msgs/Odometry messages to a ROS 2 topic
2. Publishing of the coordinate transform from **odom** (parent frame) **-> base_link** (child frame) coordinate frames.

Getting both of these pieces of data published to the ROS system is our end goal in setting up the odometry for a robot.

Simulate the Odometry System Using Gazebo

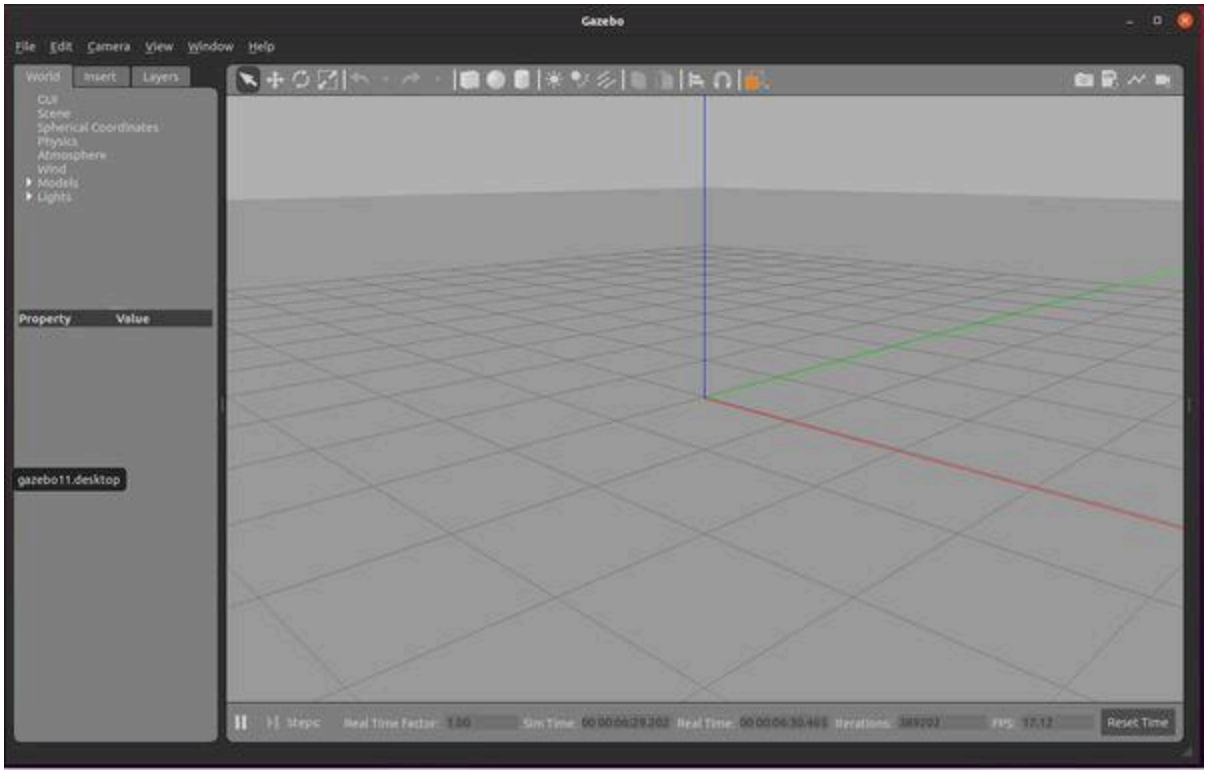
Let's set up the odometry for the simulated robot we created in the last tutorial. We will use Gazebo, an open-source 3D robotics simulator.

Set Up the Prerequisites

Gazebo is automatically included in ROS 2 installations. To test that Gazebo is installed, open a new terminal window, and type:

gazebo

You should see an empty world:



If you don't see any output after a minute or two, [follow these instructions to install Gazebo].

Close Gazebo by going to the terminal window and pressing **CTRL + C**.

Now, open a new terminal window, and install the **gazebo_ros_pkgs** package.

sudo apt-get update

sudo apt-get upgrade

sudo apt install ros-foxy-gazebo-ros-pkgs

The syntax for the above command is:

sudo apt install ros-<ros2-distro>-gazebo-ros-pkgs

....where you replace <ros2-distro> with the ROS 2 distribution that you are using. I am using ROS 2 Foxy Fitzroy, so I use "foxy".

Create an SDF File for the Robot

Later in this tutorial series, we will be using the robot_localization package to enable the robot to determine where it is in the environment. To use this package, we need to create an SDF file.

Why should we create an SDF file instead of using our URDF File? In working with ROS for many years, I have found that URDFs don't always work that well with Gazebo.

For example, when I ran through the official ROS 2 Navigation Stack robot localization demo, I found that the filtered odometry data was not actually generated.

So, we need to use an SDF file for Gazebo stuff and a URDF file for ROS stuff. We will try to make sure our SDF file generates a robot that is as close as possible to the robot generated by the URDF file.

Open a new terminal window, and type:

cd ~/ros_ws/src/basic_mobile_robot/models

If you have colcon_cd set up, you can also type:

colcon_cd basic_mobile_robot

cd models

Create Model.config

Let's create a folder for the SDF model.

mkdir basic_mobile_bot_description

Move inside the folder.

cd basic_mobile_bot_description

Create a model.config file.

gedit model.config

ros2 run tf2_tools view_frames.py

evince frames.pdf

Setup LIDAR for robot

cd basic_mobile_bot_description

Type the following command:

gedit model.sdf

```
<!-- ***************************** LIDAR
*****************************    -->
<link name="lidar_link">
  <inertial>
    <pose>0.215 0 0.13 0 0 0</pose>
    <inertia>
      <ixx>0.001</ixx>
      <ixy>0.000</ixy>
      <ixz>0.000</ixz>
      <iyy>0.001</iyy>
      <iyz>0.000</iyz>
      <izz>0.001</izz>
    </inertia>
    <mass>0.114</mass>
  </inertial>

  <collision name="lidar_collision">
    <pose>0.215 0 0.13 0 0 0</pose>
    <geometry>
      <cylinder>
        <radius>0.0508</radius>
        <length>0.18</length>
      </cylinder>
    </geometry>
  </collision>

  <visual name="lidar_visual">
    <pose>0.215 0 0.13 0 0 0</pose>
    <geometry>
```

```
      <cylinder>
        <radius>0.0508</radius>
        <length>0.18</length>
      </cylinder>
    </geometry>
    <material>
      <ambient>0.0 0.0 0.0 1.0</ambient>
      <diffuse>0.0 0.0 0.0 1.0</diffuse>
      <specular>0.0 0.0 0.0 1.0</specular>
      <emissive>0.0 0.0 0.0 1.0</emissive>
    </material>
  </visual>

  <sensor name="lidar" type="ray">
    <pose>0.215 0 0.215 0 0 0</pose>
    <always_on>true</always_on>
    <visualize>true</visualize>
    <update_rate>5</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>360</samples>
          <resolution>1.00000</resolution>
          <min_angle>0.000000</min_angle>
          <max_angle>6.280000</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.120000</min>
        <max>3.5</max>
        <resolution>0.015000</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
    <plugin name="scan" filename="libgazebo_ros_ray_sensor.so">
      <ros>
        <remapping>~/out:=scan</remapping>
      </ros>
      <output_type>sensor_msgs/LaserScan</output_type>
      <frame_name>lidar_link</frame_name>
    </plugin>
  </sensor>
</link>
```

Save the file and close it to return to the terminal.

To test use command gazebo and try inserting the model.

cd models

Create a new file named **basic_mobile_bot_v2.urdf**.

gedit basic_mobile_bot_v2.urdf

Inside this file, we will add a link and a joint for the LIDAR. Leave the caster wheel inertial section as-is.

Type this code inside the URDF file.

Save and close the file.

cd launch

gedit basic_mobile_bot_v4.launch.py

Copy and paste this code into the file.

Save the file, and close it.

colcon build

Launch the Robot

Open a new terminal, and launch the robot.

cd ~/ros_ws/

sudo apt-get install ros-foxy-robot_localization

ros2 launch basic_mobile_robot basic_mobile_bot_v4.launch.py

Localization

Now we will use information obtained from LIDAR scans to build a map of the environment and to localize on the map. The purpose of doing this is to enable our robot to navigate autonomously through both known and unknown environments (i.e. SLAM).

The two most commonly used packages for localization are the **nav2_amcl** package and the **slam_toolbox**. Both of these packages publish the **map -> odom** coordinate transformation which is necessary for a robot to localize on a map.

cd launch

touch basic_mobile_bot_v5.launch.py

```
# Author: Addison Sears-Collins
# Date: September 2, 2021
# Description: Launch a basic mobile robot using the ROS 2 Navigation Stack
# https://automaticaddison.com

import os
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription
from launch.conditions import IfCondition, UnlessCondition
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import Command, LaunchConfiguration, PythonExpression
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():

  # Set the path to different files and folders.
  pkg_gazebo_ros = FindPackageShare(package='gazebo_ros').find('gazebo_ros')
  pkg_share = FindPackageShare(package='basic_mobile_robot').find('basic_mobile_robot')
  default_launch_dir = os.path.join(pkg_share, 'launch')
  default_model_path = os.path.join(pkg_share, 'models/basic_mobile_bot_v2.urdf')
  robot_localization_file_path = os.path.join(pkg_share, 'config/ekf.yaml')
  robot_name_in_urdf = 'basic_mobile_bot'
  default_rviz_config_path = os.path.join(pkg_share, 'rviz/nav2_config.rviz')
```

```
  world_file_name = 'basic_mobile_bot_world/smalltown.world'
  world_path = os.path.join(pkg_share, 'worlds', world_file_name)
  nav2_dir = FindPackageShare(package='nav2_bringup').find('nav2_bringup')
  nav2_launch_dir = os.path.join(nav2_dir, 'launch')
  static_map_path = os.path.join(pkg_share, 'maps', 'smalltown_world.yaml')
  nav2_params_path = os.path.join(pkg_share, 'params', 'nav2_params.yaml')
  nav2_bt_path =
FindPackageShare(package='nav2_bt_navigator').find('nav2_bt_navigator')
  behavior_tree_xml_path = os.path.join(nav2_bt_path, 'behavior_trees',
'navigate_w_replanning_and_recovery.xml')

  # Launch configuration variables specific to simulation
  autostart = LaunchConfiguration('autostart')
  default_bt_xml_filename = LaunchConfiguration('default_bt_xml_filename')
  headless = LaunchConfiguration('headless')
  map_yaml_file = LaunchConfiguration('map')
  model = LaunchConfiguration('model')
  namespace = LaunchConfiguration('namespace')
  params_file = LaunchConfiguration('params_file')
  rviz_config_file = LaunchConfiguration('rviz_config_file')
  slam = LaunchConfiguration('slam')
  use_namespace = LaunchConfiguration('use_namespace')
  use_robot_state_pub = LaunchConfiguration('use_robot_state_pub')
  use_rviz = LaunchConfiguration('use_rviz')
  use_sim_time = LaunchConfiguration('use_sim_time')
  use_simulator = LaunchConfiguration('use_simulator')
  world = LaunchConfiguration('world')

  # Map fully qualified names to relative ones so the node's namespace can be
prepended.
  # In case of the transforms (tf), currently, there doesn't seem to be a better alternative
  # https://github.com/ros/geometry2/issues/32
  # https://github.com/ros/robot_state_publisher/pull/30
  # TODO(orduno) Substitute with `PushNodeRemapping`
  #              https://github.com/ros2/launch_ros/issues/56
  remappings = [('/tf', 'tf'),
                ('/tf_static', 'tf_static')]

  # Declare the launch arguments
  declare_namespace_cmd = DeclareLaunchArgument(
    name='namespace',
    default_value='',
    description='Top-level namespace')

  declare_use_namespace_cmd = DeclareLaunchArgument(
    name='use_namespace',
    default_value='False',
```

```
  description='Whether to apply a namespace to the navigation stack')

declare_autostart_cmd = DeclareLaunchArgument(
  name='autostart',
  default_value='true',
  description='Automatically startup the nav2 stack')

declare_bt_xml_cmd = DeclareLaunchArgument(
  name='default_bt_xml_filename',
  default_value=behavior_tree_xml_path,
  description='Full path to the behavior tree xml file to use')

declare_map_yaml_cmd = DeclareLaunchArgument(
  name='map',
  default_value=static_map_path,
  description='Full path to map file to load')

declare_model_path_cmd = DeclareLaunchArgument(
  name='model',
  default_value=default_model_path,
  description='Absolute path to robot urdf file')

declare_params_file_cmd = DeclareLaunchArgument(
  name='params_file',
  default_value=nav2_params_path,
  description='Full path to the ROS2 parameters file to use for all launched nodes')

declare_rviz_config_file_cmd = DeclareLaunchArgument(
  name='rviz_config_file',
  default_value=default_rviz_config_path,
  description='Full path to the RVIZ config file to use')

declare_simulator_cmd = DeclareLaunchArgument(
  name='headless',
  default_value='False',
  description='Whether to execute gzclient')

declare_slam_cmd = DeclareLaunchArgument(
  name='slam',
  default_value='False',
  description='Whether to run SLAM')

declare_use_robot_state_pub_cmd = DeclareLaunchArgument(
  name='use_robot_state_pub',
  default_value='True',
  description='Whether to start the robot state publisher')
```

```
  declare_use_rviz_cmd = DeclareLaunchArgument(
    name='use_rviz',
    default_value='True',
    description='Whether to start RVIZ')

  declare_use_sim_time_cmd = DeclareLaunchArgument(
    name='use_sim_time',
    default_value='True',
    description='Use simulation (Gazebo) clock if true')

  declare_use_simulator_cmd = DeclareLaunchArgument(
    name='use_simulator',
    default_value='True',
    description='Whether to start the simulator')

  declare_world_cmd = DeclareLaunchArgument(
    name='world',
    default_value=world_path,
    description='Full path to the world model file to load')

  # Specify the actions

  # Start Gazebo server
  start_gazebo_server_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(os.path.join(pkg_gazebo_ros, 'launch',
'gzserver.launch.py')),
    condition=IfCondition(use_simulator),
    launch_arguments={'world': world}.items())

  # Start Gazebo client
  start_gazebo_client_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(os.path.join(pkg_gazebo_ros, 'launch',
'gzclient.launch.py')),
    condition=IfCondition(PythonExpression([use_simulator, ' and not ', headless])))

  # Start robot localization using an Extended Kalman filter
  start_robot_localization_cmd = Node(
    package='robot_localization',
    executable='ekf_node',
    name='ekf_filter_node',
    output='screen',
    parameters=[robot_localization_file_path,
    {'use_sim_time': use_sim_time}])

  # Subscribe to the joint states of the robot, and publish the 3D pose of each link.
  start_robot_state_publisher_cmd = Node(
    condition=IfCondition(use_robot_state_pub),
```

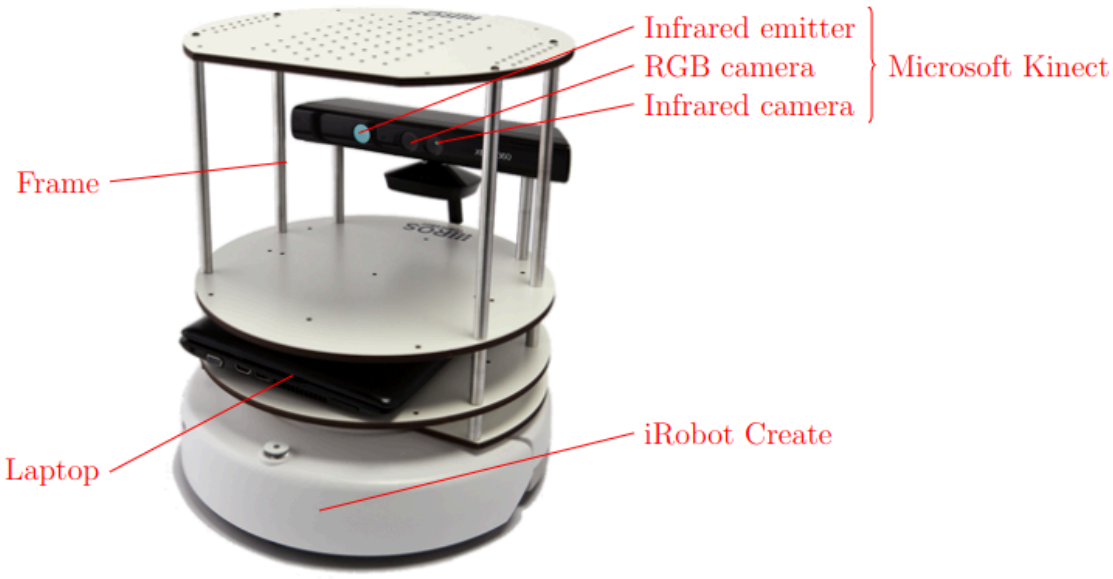
```
  package='robot_state_publisher',
  executable='robot_state_publisher',
  namespace=namespace,
  parameters=[{'use_sim_time': use_sim_time,
  'robot_description': Command(['xacro ', model])}],
  remappings=remappings,
  arguments=[default_model_path])

# Launch RViz
start_rviz_cmd = Node(
  condition=IfCondition(use_rviz),
  package='rviz2',
  executable='rviz2',
  name='rviz2',
  output='screen',
  arguments=['-d', rviz_config_file])

# Launch the ROS 2 Navigation Stack
start_ros2_navigation_cmd = IncludeLaunchDescription(
  PythonLaunchDescriptionSource(os.path.join(nav2_launch_dir, 'bringup_launch.py')),
  launch_arguments = {'namespace': namespace,
                      'use_namespace': use_namespace,
                      'slam': slam,
                      'map': map_yaml_file,
                      'use_sim_time': use_sim_time,
                      'params_file': params_file,
                      'default_bt_xml_filename': default_bt_xml_filename,
                      'autostart': autostart}.items())

# Create the launch description and populate
ld = LaunchDescription()

# Declare the launch options
ld.add_action(declare_namespace_cmd)
ld.add_action(declare_use_namespace_cmd)
ld.add_action(declare_autostart_cmd)
ld.add_action(declare_bt_xml_cmd)
ld.add_action(declare_map_yaml_cmd)
ld.add_action(declare_model_path_cmd)
ld.add_action(declare_params_file_cmd)
ld.add_action(declare_rviz_config_file_cmd)
ld.add_action(declare_simulator_cmd)
ld.add_action(declare_slam_cmd)
ld.add_action(declare_use_robot_state_pub_cmd)
ld.add_action(declare_use_rviz_cmd)
ld.add_action(declare_use_sim_time_cmd)
ld.add_action(declare_use_simulator_cmd)
```

```
    ld.add_action(declare_world_cmd)

    # Add any actions
    ld.add_action(start_gazebo_server_cmd)
    ld.add_action(start_gazebo_client_cmd)
    ld.add_action(start_robot_localization_cmd)
    ld.add_action(start_robot_state_publisher_cmd)
    ld.add_action(start_rviz_cmd)
    ld.add_action(start_ros2_navigation_cmd)

    return ld
```

sudo apt install ros-foxy-navigation2 ros-foxy-nav2-bringup

ros2 topic pub /goal_pose geometry_msgs/PoseStamped "{header: {stamp: {sec: 0}, frame_id: 'map'}, pose: {position: {x: 5.0, y: -2.0, z: 0.0}, orientation: {w: 1.0}}}"

Turtlebot



If you don't have turtlebot3 packages, you can install debian packages or from source code.

A. Install debian packages sudo apt install ros-foxy-turtlebot3*

To use the robot in Gazebo simulator,

Open a terminal

If you don't set up ROS Domain ID, then the default ROS_DOMAIN_ID=0.

In this case, we only work with one turtlebot so we can use default ROS Domain ID.

If you want to use different ROS Domain ID, you can perform:

$ export ROS_DOMAIN_ID=11

1.

Set up ROS environment arguments

If you use debian packages,

$ source /opt/ros/foxy/setup.bash

If you use packages in your workspace:

First entering your workspace

$ source install/setup.bash

2. Set up turtlebot model

   $ export TURTLEBOT3_MODEL=burger

3. Set up Gazebo model path

   $ export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:`ros2 pkg \

prefix turtlebot3_gazebo \

`/share/turtlebot3_gazebo/models/

4. Launch Gazebo with simulation world

   $ ros2 launch turtlebot3_gazebo empty_world.launch.py

5. You also can start different world by replacing empty_world.launch.py with turtlebot3_house.launch.py

Control the robot

In this chapter you will learn how to use keyboard or joystick to control robot. In general we will start a ros node that will publish to topic **/cmd_vel**

4.1. Keyboard

1. Open a new terminal

    - Set up ROS environment arguments
    - (Set up ROS_DOMAIN_ID): Only if you set up ROS_DOMAIN_ID in chapter3

Set up turtlebot model

```
$ export TURTLEBOT3_MODEL=burger
```

2. Run a teleoperation node

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

3. If the program is successfully launched, the following output will appear in the terminal window and you can control the robot following the instruction.

Control Your TurtleBot3!

Moving around:

```
   w
a  s  d
   x
```

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)

a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit

Cartographer

The main problem in mobile robotics is localization and mapping. To estimate the position of the robot in an environment, you need some kind of map from this environment to determine the actual position in this environment. On the other hand, you need the actual position of robots to create a map related to its position. Therefore you can use SLAM – Simultaneous Localization and Mapping. ROS provides different packages to solve this problem:

- **2D**: gmapping, hector_slam, cartographer, ...
- **3D**: rgbdslam, ccny_rgbd, lsd_slam, rtabmap...

For ground-based robots, it is often sufficient to use 2D SLAM to navigate through the environment. In the following tutorial, cartographer will be used. Cartographer SLAM builds a map of the environment and simultaneously estimates the platform's 2D pose. The localization is based on a laser scan and the odometry of the robot.

Check if cartographer exists,

$ ros2 pkg list |grep cartographer

If you have already installed, you would see,

cartographer_ros

cartographer_ros_msgs

To install use command,

sudo apt install ros-foxy-cartographer

To start turtlebot,

3.3.1. Simulation in gazebo

Set up turtlebot model

$ export TURTLEBOT3_MODEL=burger

1. Set up Gazebo model path

   $ export GAZEBO_MODEL_PATH=`ros2 pkg \

prefix turtlebot3_gazebo`/share/turtlebot3_gazebo/models/

2. Launch Gazebo with a simulation world

   $ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
3. Run teleoperation node

   $ ros2 run turtlebot3_teleop teleop_keyboard
4. $ ros2 launch turtlebot3_cartographer \

cartographer.launch.py \

use_sim_time:=True
create a map

Hint: Make sure that the Fixed Frame (in Global Options) in RViz is set to "map".

In this way the map is fixed and the robot will move relative to it. The scanner of the Turtlebot3 covers 360 degrees of its surroundings. Thus, if objects are close by to the robot it will start to generate the map.

Teleoperate the robot through the physical world until the enclosed environment is completely covered in the virtual map.

The following hints help you to create **a nice map**:

* Try to drive as slow as possible

* Avoid to drive linear and rotate at the same time

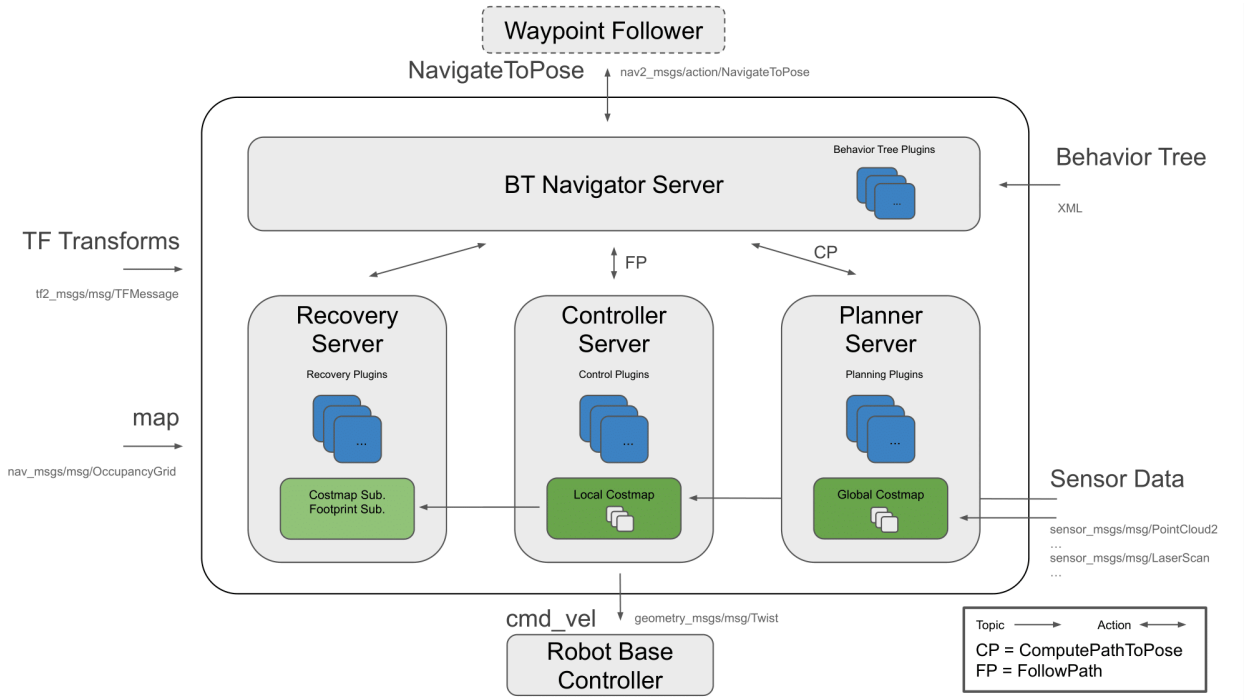
* Do not drive too close to the obstacles

To save the map,

ros2 run nav2_map_server map_saver_cli -f my_map

Navigation

The ROS 2 Navigation System is the control system that enables a robot to autonomously reach a goal state, such as a specific position and orientation relative to a specific map. Given a current pose, a map, and a goal, such as a destination pose, the navigation system generates a plan to reach the goal, and outputs commands to autonomously drive the robot, respecting any safety constraints and avoiding obstacles encountered along the way.



Check the map

1. Check the location of your map

   Once you create a map, you will have two files: “name of your map”.pgm and “name of your map”.yaml

Start the simulated robot

1. Open a terminal
2. Set ROS environment variables

First you need to go into your workspace and source your workspace:

$ source install/setup.bash

   -

Set up Gazebo model path:

```
$ export GAZEBO_MODEL_PATH=`ros2 pkg \
prefix turtlebot3_gazebo`/share/turtlebot3_gazebo/models/
```

set up the robot model that you will use:

```
$ export TURTLEBOT3_MODEL=burger
```

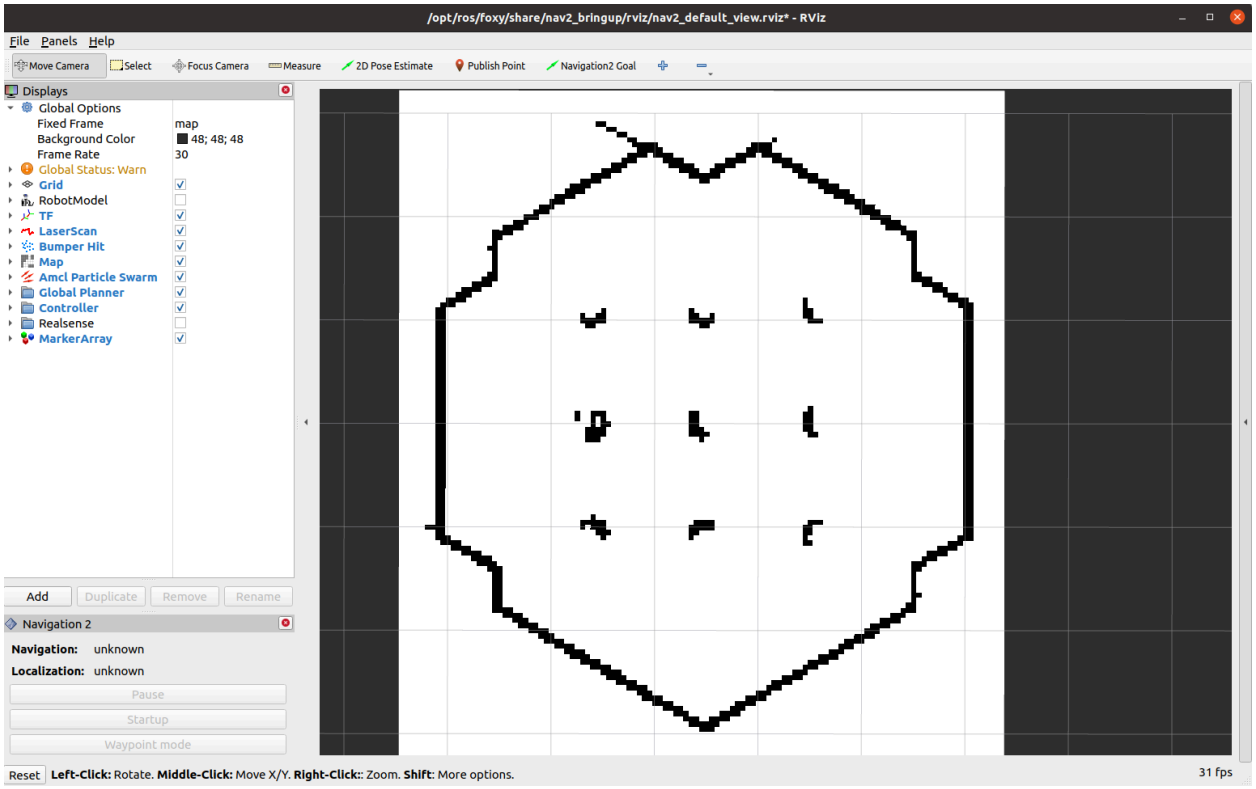
Bring up Turtlebot in simulation

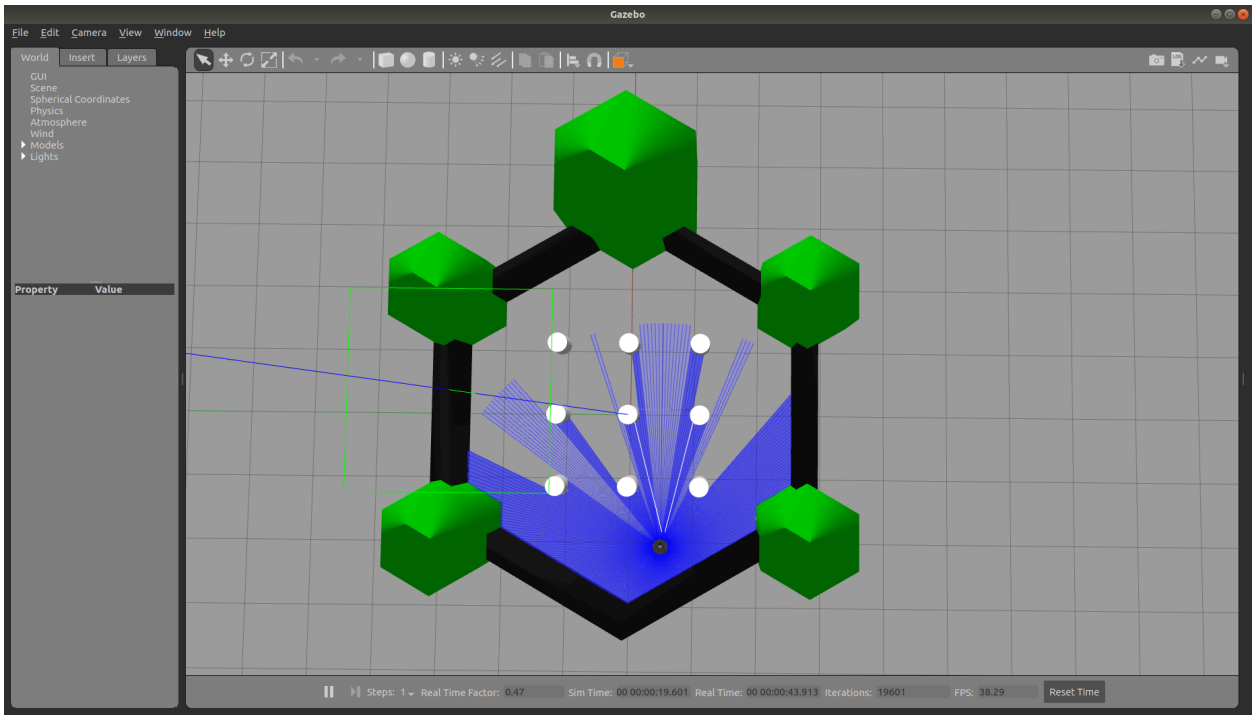
```
# in the same terminal, run
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

3. Start navigation stack

   Open another terminal, source your workspace, set up the robot model that you will use, then `$ ros2 launch turtlebot3_navigation2 \ navigation2.launch.py \ use_sim_time:=true map:=maps/"you map name".yaml`

If everything has started correctly, you will see the RViz and Gazebo GUIs like this.





Navigate the robot via rviz

- **Step 1: Tell the robot where it is**

  after starting, the robot initially has no idea where it is. By default, Navigation 2 waits for you to give it an approximate starting position.

  It has to manually update the initial location and orientation of the TurtleBot3. This information is applied to the AMCL algorithm.

  This can be done graphically with RViz by the instruction below:

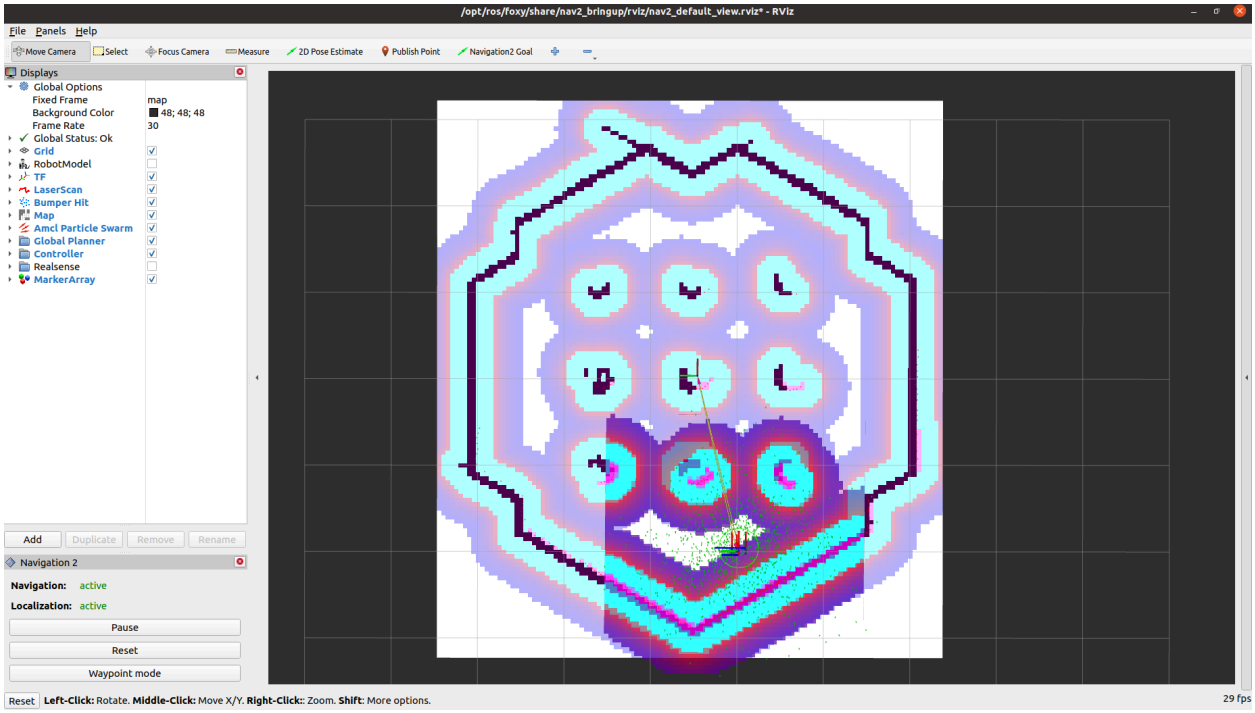
  rviz_initial

  - Click “2D Pose Estimate” button (in the top menu; see the picture)
  - Click on the approximate point in the map where the TurtleBot3 is located and drag the cursor to indicate the direction where TurtleBot3 faces.

- If you don’t get the location exactly right, that’s fine. Navigation 2 will refine the position as it navigates. You can also, click the “2D Pose Estimate” button and try again, if you prefer.

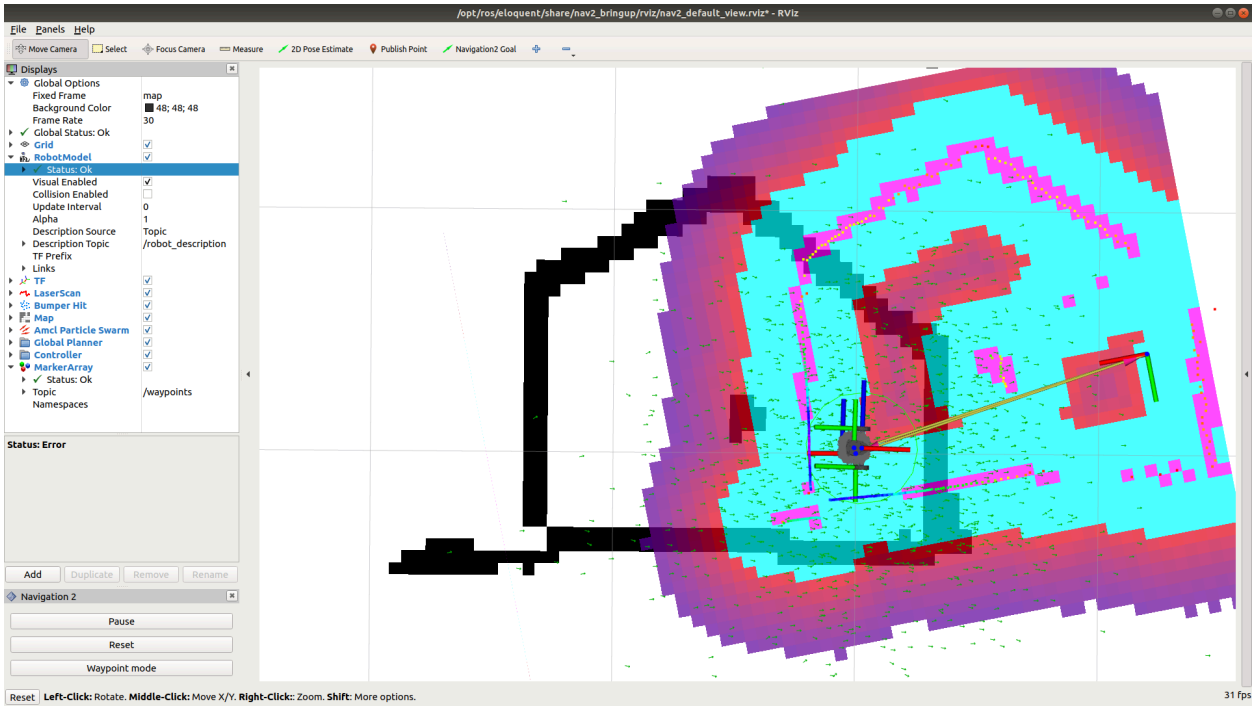
  Once you’ve set the initial pose, the tf tree will be complete and Navigation 2 is fully active and ready to go.

  If you are using simulation with turtlrbot_world map, it will show as below:

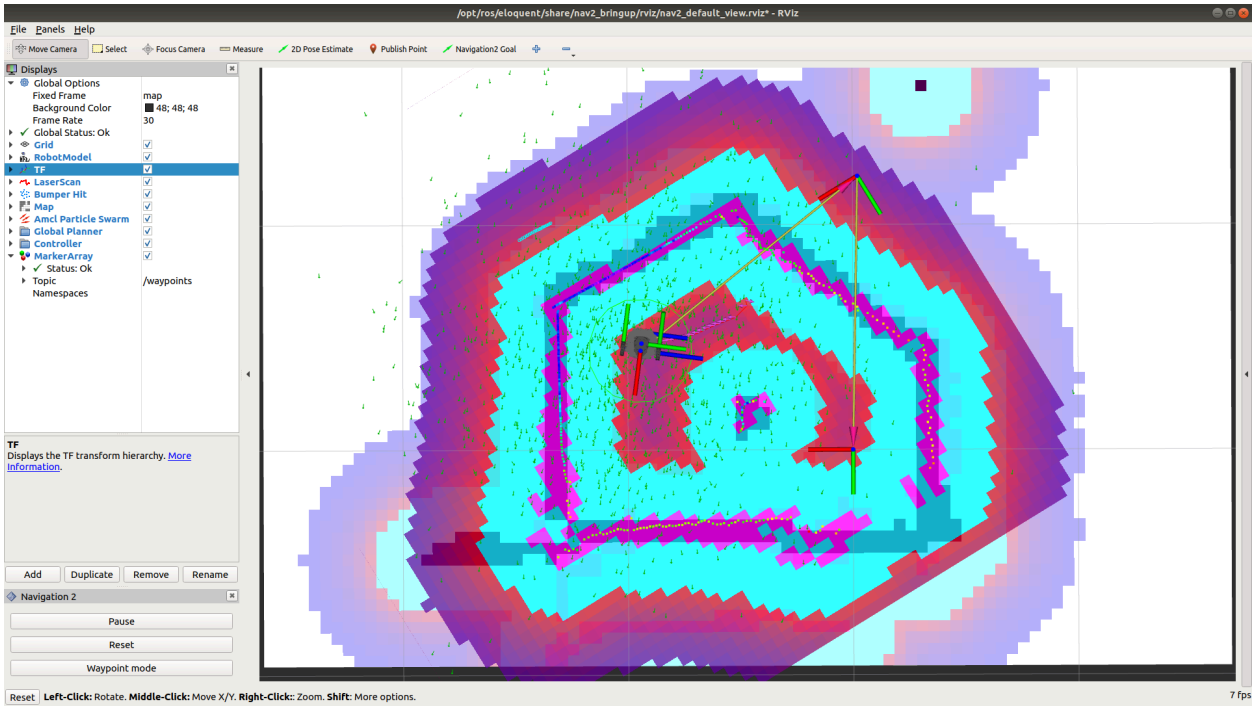


Navigation_is_ready_sim

As soon as the 2D Pose Estimation arrow is drawn, the pose (transformation from the map to the robot) will update. As a result the centre of the laser scan has changed, too. Check if the visualization of the live laser scan matches the contours of the virtual map (Illustrated in the following two pictures! The left one is the wrong robot pose and the right one is right robot pose) to confirm that the new starting pose is accurate.


/opt/ros/eloquent/share/nav2_bringup/rviz/nav2_default_view.rviz* - RViz
File Panels Help
Move Camera Select Focus Camera Measure 2D Pose Estimate Publish Point Navigation2 Goal
Displays
Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Grid
RobotModel
Status: Ok
Visual Enabled
Collision Enabled
Update Interval 0
Alpha 1
Description Source Topic
Description Topic /robot_description
TF Prefix
Links
TF
LaserScan
Bumper Hit
Map
Amcl Particle Swarm
Global Planner
Controller
MarkerArray
Status: Ok
Topic /waypoints
Namespaces
Status: Error
Add Duplicate Remove Rename
Navigation 2
Pause
Reset
Waypoint mode
Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Zoom. Shift: More options.
31 fps

bad


/opt/ros/eloquent/share/nav2_bringup/rviz/nav2_default_view.rviz* - RViz
File Panels Help
Move Camera Select Focus Camera Measure 2D Pose Estimate Publish Point Navigation2 Goal
Displays
Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Grid
RobotModel
TF
LaserScan
Bumper Hit
Map
Amcl Particle Swarm
Global Planner
Controller
MarkerArray
Status: Ok
Topic /waypoints
Namespaces
TF
Displays the TF transform hierarchy. More Information.
Add Duplicate Remove Rename
Navigation 2
Pause
Reset
Waypoint mode
Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Zoom. Shift: More options.
7 fps

good

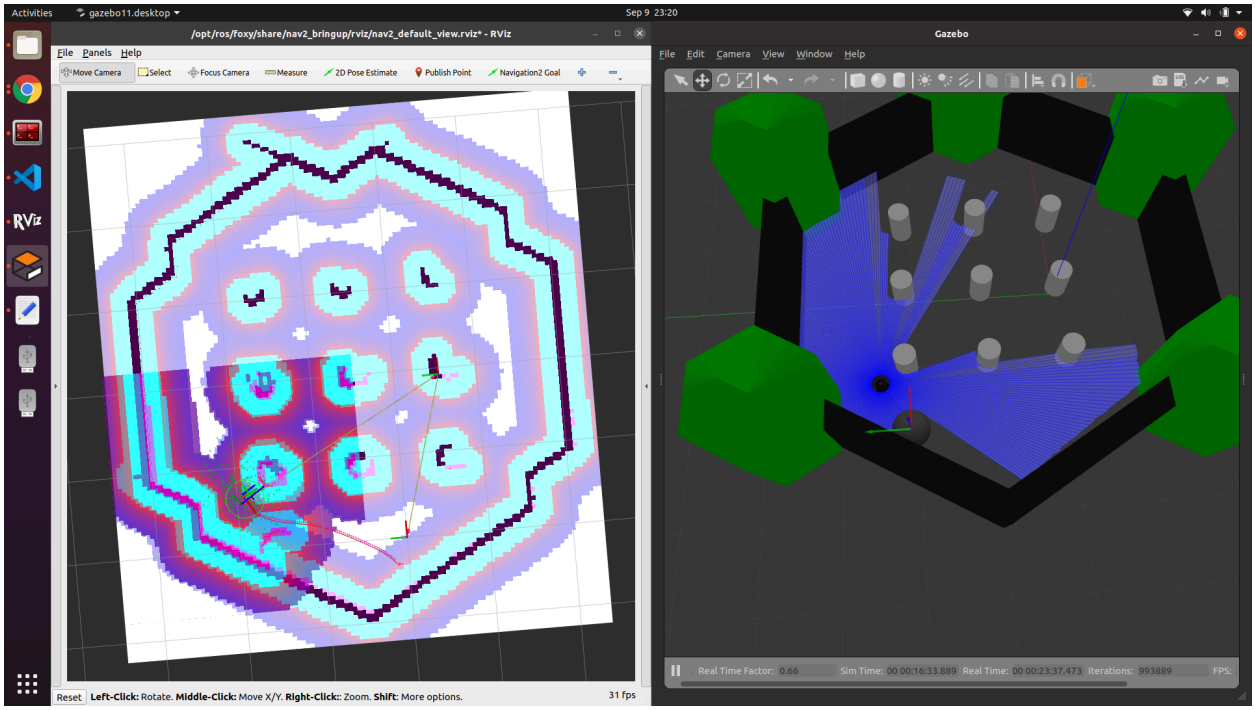
- **Step 2: Give a goal**

  if the TurtleBot3 is localized, it can automatically create a path from the current position to any target reachable position on the map. In order to set a goal position, follow the instruction below:

  - Click the 2D Nav Goal button (also in the top menu)
  - Click on a specific point in the map to set a goal position and drag the cursor to the direction where TurtleBot should be facing at the end
- ***Hint: If you wish to stop the robot before it reaches to the goal position, set the current position of TurtleBot3 as a goal position.***

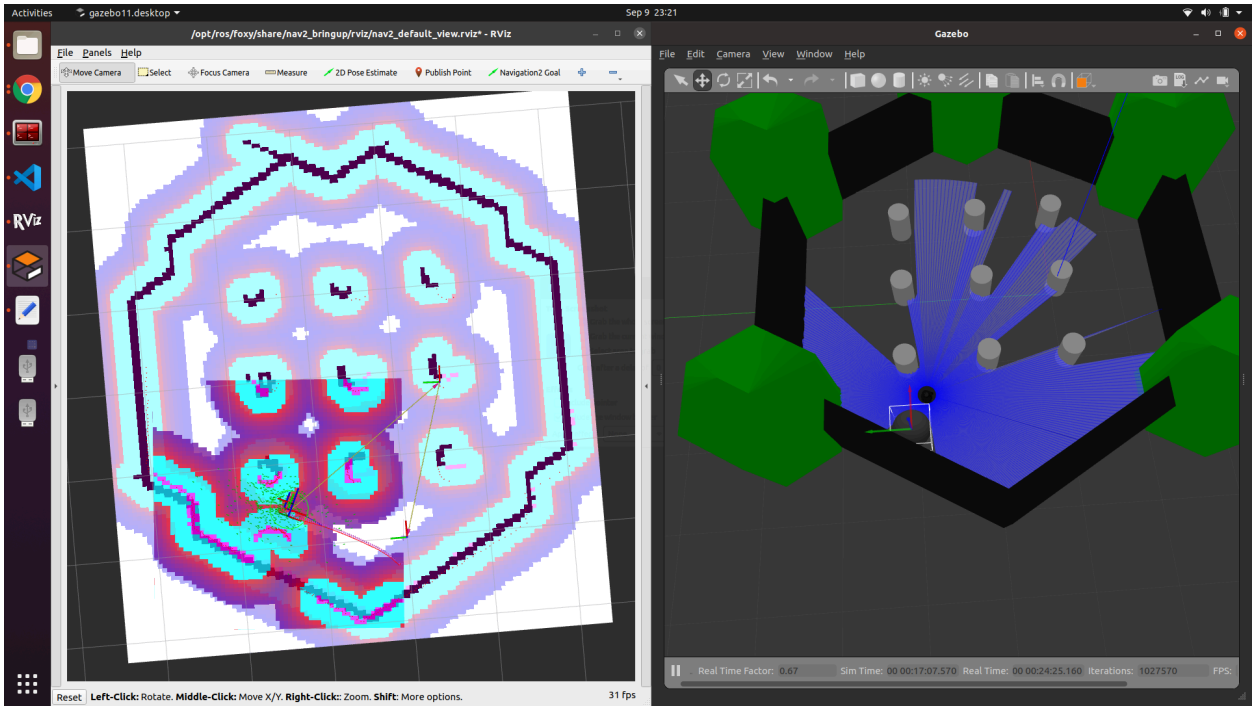
  The swarm of the green small arrows is the visualization of the **adaptive Monte Carlo localization (AMCL)**. Every green arrow stands for a possible position and orientation of the TurtleBot3. Notice that in the beginning its distribution is spread over the whole map. As soon as the robot moves the arrows get updated because the algorithm incorporates new measurements. During the movement the distribution of arrows becomes less chaotic and settles more and more to the robot's location, which finally means that the algorithm becomes more and more certain about the pose of the robot in the map.

- **Step 3: Play around**When the robot is on the way to the goal, you can put an obstacle in Gazebo:



nav_goal_ob_1

You can see how the robot reacts to this kind of situation:



nav_goal_ob_2

Project-Collision Avoidance

```
#!/usr/bin/env python
import rclpy # Python library for ROS
from sensor_msgs.msg import LaserScan # LaserScan type message is defined in
sensor_msgs
from geometry_msgs.msg import Twist #

def callback(dt):
        print '----------------------------------------'
        print 'Range data at 0 deg:   {}'.format(dt.ranges[0])
        print 'Range data at 15 deg:  {}'.format(dt.ranges[15])
        print 'Range data at 345 deg: {}'.format(dt.ranges[345])
        print '----------------------------------------'
        thr1 = 0.8 # Laser scan range threshold
        thr2 = 0.8
        if dt.ranges[0]>thr1 and dt.ranges[15]>thr2 and dt.ranges[345]>thr2: #
Checks if there are obstacles in front and
                                                # 15 degrees left and right (Try
changing the
                        # the angle values as well as the thresholds)
        move.linear.x = 0.5 # go forward (linear velocity)
        move.angular.z = 0.0 # do not rotate (angular velocity)
        else:
        move.linear.x = 0.0 # stop
        move.angular.z = 0.5 # rotate counter-clockwise
        if dt.ranges[0]>thr1 and dt.ranges[15]>thr2 and dt.ranges[345]>thr2:
        move.linear.x = 0.5
        move.angular.z = 0.0
        pub.publish(move) # publish the move object


move = Twist() # Creates a Twist message type object
rospy.init_node('obstacle_avoidance_node') # Initializes a node
pub = rospy.Publisher("/cmd_vel", Twist, queue_size=10)  # Publisher object which
will publish "Twist" type messages
                                # on the "/cmd_vel" Topic, "queue_size" is the size of
the
                                        # outgoing message queue used for
asynchronous publishing

sub = rospy.Subscriber("/scan", LaserScan, callback)  # Subscriber object which will
listen "LaserScan" type messages
```

```
                                    # from the "/scan" Topic and call the "callback"
function
                                # each time it reads something from the Topic

rospy.spin() # Loops infinitely until someone stops the program execution
```