项目二：URDF 移动机器人及 MoveIt!机械臂控制

第一部分： 建立移动机器人

1. Introduction to the Project

By doing this project, you will get to know how to use URDF (Universal Robot Description Format) to create a 5 degree-of freedom (DOF) robot arm mounted on a robot chasis, how to use XACRO (Extensible Markup Language Macros) to modularize the design of reusable robot components, how to describe the physical characteristics of a robot such as its colors, inertia and frictions, how to spawn a robot into a Gazebo world, and how to use Gazebo ROS plugins to simulate a robot in the Gazebo world, e.g., moving a robot in the Gazebo world w.r.t. its motion in ROS RViz.

Meanwhile, we will learn the useful motion control tool, namely MoveIt!, which aims to implement the inverse kinematics of a robot arm.

You will gain the following knowledge if you successfully complete this project.

(1) Understand the difference between URDF, XACRO and SDF (simulation description format).
(2) Be able to use URDF to build a simple robot, with the capability of spawning the robot into a tailor designed Gazebo world.
(3) Be able to control the robot motion in Gazebo using Gazebo ROS plugins, and control the motion of a robot arm using MoveIt!.

2. Software Setup

To achieve ROS integration with Gazebo, we need certain dependencies that would establish a connection between both and convert the ROS messages into Gazebounderstandable information. We also need a framework that implements real-time-like robot controllers that help the robot move kinematically. The former constitutes the gazebo_ros_pkgs package, which is a bunch of ROS wrappers written to help Gazebo understand the ROS messages and services, and while latter constitutes the ros_control and ros_controllers packages, which provide robot joint and actuator space conversions and ready-made controllers that control position, velocity, or effort (force). You can install them through these commands:

```
$ sudo apt-get install ros-melodic-ros-control

$ sudo apt-get install ros-melodic-ros-controllers
$ sudo apt-get install ros-melodic-gazebo-ros-control
```

We will be using the hardware_interface::RobotHW class from ros_control as it already has defined abstraction layers and joint_trajectory_controller and diff_drive_controller from ros_controllers for our robot arm and mobile base, respectively.

3. Build the Robot Base

Let's begin by modeling our robot base. ROS understands a robot in terms of URDF. URDF is a list of XML tags that contains all of the necessary information of the robot. Once the URDF for the robot base is created, we shall bring in the necessary connectors and wrappers around the code so that we can interact and communicate with a standalone physics simulator such as Gazebo. Let's see how the robot base is built step by step.

3.1 Robot Base Prerequisites

To build a robot base, we need the following:

- A good solid chassis with a good set of wheels with friction properties
- Powerful drives that can help carry the required payload
- Drive controls

In case you plan to build a real robot base, there are additional considerations you might need to look into, for instance, power management systems—to run the robot efficiently for as long as you wish—the necessary electrical and embedded characteristics, and mechanical power transmission systems. What can help you get there is building a robot in ROS. Why, exactly? You would be able to emulate (actually, simulate, but if you tweak some parameters and apply real-time constraints, you could definitely emulate) a real working robot, as in the following examples:

- Your chassis and wheels would be defined with physical properties in URDF.
- Your drives could be defined using Gazebo-ros plugins.
- Your drive controls could be defined using ros-controllers.

Hence, to build a custom robot, let's consider the specifications.

3.2 Robot Base Specifications

Let's consider the following specifications for our robot base:

- Size: Somewhere within 600 x 450 x 200 (L x B x H, all in mm)
- Type: Four-wheel differential drive robot
- Speed: Up to 1 m/s
- Payload: 50 kg (excluding the robot arm)

3.3 Software Parameters

Now that we have the robot specifications, let's learn about the ROS-related information we need to know of while building a robot base. Let's consider the mobile robot base as a black box: if you give it specific velocity, the robot base should move and, in turn, give the position it has moved to. In ROS terms, the mobile robot takes in information through a topic called /cmd_vel (command velocity) and gives out /odom (odometery). A simple representation is shown as follows:

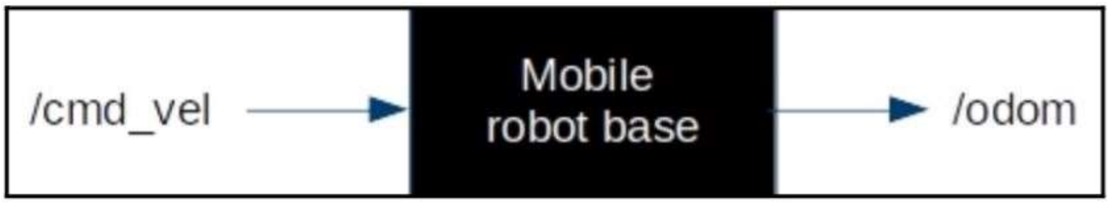


Figure 1: The mobile robot base as a blackbox

3.4 ROS Message Format

/cmd-vel is of the geometry_msgs/Twist message format. The message structure can be found at http://docs.ros.org/melodic/api/geometry_msgs/html/msg/Twist.html. /odom is of the nav_msgs/Odometry message format. The message structure can be found at http://docs.ros.org/melodic/api/nav_msgs/html/msg/Odometry.html. Not all the fields are necessary in the case of our robot base since our robot is a 2 degrees of freedom robot.

3.5 ROS Controllers

We would define the robot base's differential kinematics model using the diff_drive_controller plugin. This plugin defines the robot kinematics equations. It helps our robot to move in space. More information about this controller is available at the http://wiki.ros.org/diff_drive_controller website.

3.6 Modelling the Robot Base

Now that we have all the necessary information about the robot, let's get straight into modeling the robot. The robot model we are going to build is as follows:

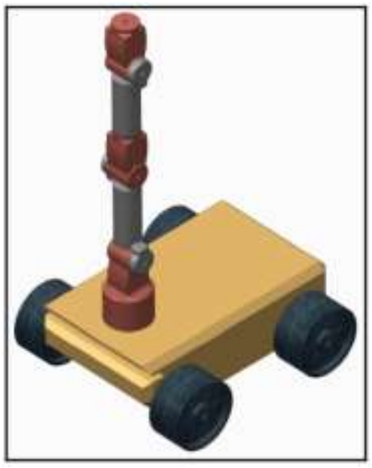


Figure 2: Mobile robot model

There is something you need to know before you come out with thoughts about modeling robots using URDF. You could make use of the geometric tags that define standard shapes such as cylinder, sphere, and boxes, but you cannot model complicated geometries or style them. These can be done using third-party software, for example, sophisticated Computer Aided Design (CAD) software such as Creo or Solidworks or using open source modelers such as Blender, FreeCAD, or Meshlab. Once they are modeled, they're are imported as meshes. The models in this book are modeled by such open source modelers and imported into URDFs as meshes. Also, writing a lot of XML tags sometimes becomes cumbersome and we might get lost while building intricate robots. Hence, we shall make use of macros in URDF called xacro (http://wiki.ros.org/xacro), which will help to reduce our lines of code for simplification and to avoid the repetition of tags.
Our robot base model will need the following tags:

- <xacro>: To help define macros for reuse
- <links>: To contain the geometric representations of the robot and visual information
- <inertial>: To contain the mass and moment of inertia of the links
- <joints>: To contain connections between the links with constraint definitions
- <gazebo>: To contain plugins to establish a connection between Gazebo and ROS, along with simulation properties

For our robot base, there is a chassis and four wheels that are modeled. Have a look at the following diagram for representation information of the links:

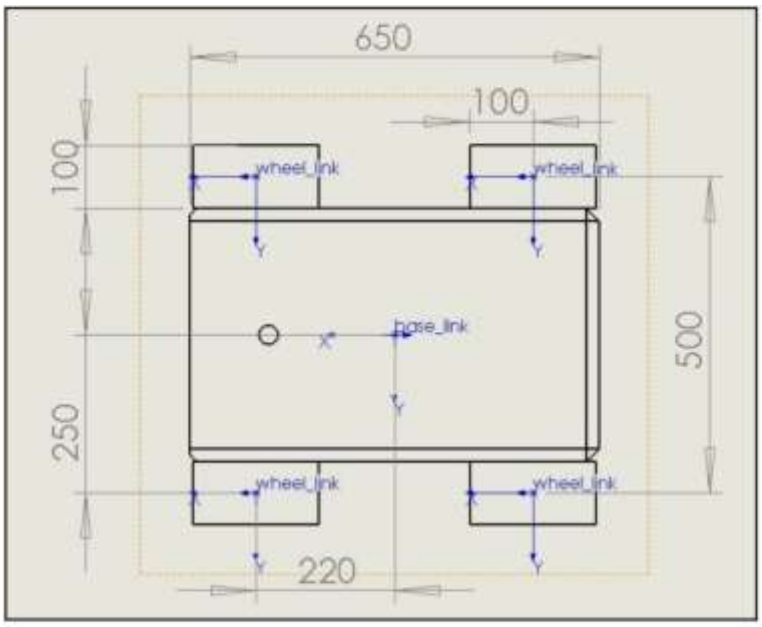


figure 3: The robot base sketch

The chassis is named base_link and you can see the coordinate system in its center. Wheels (or wheel_frames) are placed with respect to the base_link frame. You can see that, as per our representation, the model follows the right-hand rule in the coordinate system. You can now make out that the forward direction of the robot will always be toward the x axis and that the rotation of the robot is around the z axis. Also, note that the wheels rotate around the y axis with respect to its frame of reference (you shall see this reference in the code in the next section).

4. The URDF Modelling Work

Create your new package called my_robot, and compile it even if it is now empty:

```
$ mkdir -p ~/project2_ws/src
$ cd ~/project2_ws/src
$ catkin_init_worksapce
$ catkin_create_pkg my_robot catkin
$ cd my_robot
$ mkdir config launch meshes urdf
$ cd ../..
$ catkin_make
$ source devel/setup.bash
$ echo "source ~/project2_ws/devel/setup.bash" >> ~/.bashrc
```

Download the reference package in this new workspace:

```
$ cd ~/project2_ws/src
$ git clone https://github.com/Zhijun2/robot_description.git
```

Then, copy all the files inside the sub-directory called meshes in the reference package called robot_description to the corresponding meshes subdir of your new package. These files were exported from the CAD software for easily implementing the chasis and arm components.

Enter the urdf subdir of your new project，create a new file called robot_base.urdf.xacro for your chasis.

```
$ cd ~/project2_ws/src/my_robot/urdf
$ gedit robot_base.urdf.xacro
```

Following the steps below.

(1) First of all, copy the following content which contains the XML and robot tag in your opened file.

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




</robot>
```

(2) Define the links. Since you're defining the robot model by parts, copy the following code into the space in step (1) content (that is, the space between the tags). The chassis link is as follows:

```
<link name="base_link">
  <visual>
     <origin xyz="0 0 0" rpy="1.5707963267949 0 3.14" />
     <geometry>
        <mesh filename="package://my_robot/meshes/robot_base.stl"/>
     </geometry>
     <material name="">
        <color rgba="0.79216 0.81961 0.93333 1" />
     </material>
  </visual>
</link>
```

The link defines the robot geometrically and helps in visualization. The preceding is the robot chassis, which we will call base_link. The tags are pretty straightforward. If you wish to learn about them, have a look at http://wiki.ros.org/urdf/XML/link.

(3) Define the wheels. Four wheels need to connect to base_link. We could reuse the same model with different names and necessary coordinate information with the help of *xacro*. So, let's create another file called robot_essentials.xacro and define standard macros so that we can reuse them:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro" name="robot_essentials" >
<xacro:macro name="robot_wheel" params="prefix">
<link name="${prefix}_wheel">
   <visual>
      <origin xyz="0 0 0" rpy="1.5707963267949 0 0" />
      <geometry>
          <mesh filename="package://my_robot/meshes/wheel.stl" />
      </geometry>
      <material name="">
          <color rgba="0.79216 0.81961 0.93333 1" />
      </material>
   </visual>
</link>
</xacro:macro>
</robot>
```

We have created a common macro for a wheel in this file. So, all you need to do now is call this macro in your actual robot file, robot_base.urdf.xacro, as shown here:

```
<xacro:robot_wheel prefix="front_left"/>
<xacro:robot_wheel prefix="front_right"/>
<xacro:robot_wheel prefix="rear_left"/>
<xacro:robot_wheel prefix="rear_right"/>
```

That's it. Can you see how quickly you have converted that many lines of code (for a link) into just one line of code for each link? If you wish to find out more about xacros, have a look at http://wiki.ros.org/xacro. Now, let's learn how to define joints.

(4) Define the joints. The wheels are connected to base_link and they rotate around their y axis on their own frame of reference. Due to this, we can make use of continuous joint types. Since they're the same for all wheels, let's define them as xacro in the robot_essentials.xacro file:

```
<xacro:macro name="wheel_joint" params="prefix origin">
  <joint name="${prefix}_wheel_joint" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link ="base_link"/>
  <child link ="${prefix}_wheel"/>
  <origin rpy ="0 0 0" xyz= "${origin}"/>
  </joint>
</xacro:macro>
```

As you can see, only the origin and the name need to change in the preceding block of code. Hence, in our robot_base.urdf.xacro file, we'll define the wheel joints as follows:

```
<xacro:wheel_joint prefix="front_left" origin="0.220 0.250 0"/>
<xacro:wheel_joint prefix="front_right" origin="0.220 -0.250 0"/>
<xacro:wheel_joint prefix="rear_left" origin="-0.220 0.250 0"/>
```

Now that you have everything in your file, let's visualize this in rviz and see whether it matches our representation. You can do that by using the following commands in a new terminal:

```
$ cd ~/project2_ws
$ source devel/setup.bash
$ cd ./src/my_robot/urdf
$ roslaunch urdf_tutorial display.launch model:=robot_base.urdf.xacro
```

Now, you should see our robot model if everything was successful.

5. Building the Robot Base for Simulation

The above steps were used to define the robot URDF model so that it could be understood by ROS. Now that we have a proper robot model that is understood by ROS, we need to add a few more tags to view the model in Gazebo.

(1) Define Collisions. To visualize the robot in Gazebo, we need to add the <collision> tags, along with the <visual> tags defined in the <link> tag, as follows:

```
<collision>
<origin
xyz="0 0 0"
rpy="1.5707963267949 0 3.14" />
<geometry>
<mesh filename="package://my_robot/meshes/robot_base.stl" />
</geometry>
</collision>
```

For the base, add them to the robot_base.urdf.xacro file since we defined base_link there.

For all the wheel links, add them to the robot_essentials.xacro file since we defined the wheel link there:

```
<collision>
  <origin
  xyz="0 0 0"
  rpy="1.5707963267949 0 0" />
  <geometry>
  <mesh filename="package://my_robot/meshes/wheel.stl" />
  </geometry>
</collision>
```

Since Gazebo is a physics simulator, we define the physical properties using the tags. We can acquire the mass and inertial properties from this third-party software. The inertial properties that are acquired from this software are added inside the tag, along with suitable tags, as indicated here:

- For the base, the following is added to robot_base.urdf.xacro:

```
<inertial>
  <origin xyz="0.0030946 4.78250032638821E-11 0.053305" rpy="0 0 0" />
  <mass value="47.873" />
  <inertia
      ixx="0.774276574699151"
      ixy="-1.03781944357671E-10"
      ixz="0.00763014265820928"
      iyy="1.64933255189991"
      iyz="1.09578155845563E-12"
      izz="2.1239326987473" />
</inertial>
```

- For all the wheels, the following is added to robot_essentials.xacro, for later reuse:

```
<inertial>
  <origin
  xyz="-4.1867E-18 0.0068085 -1.65658661799998E-18"
  rpy="0 0 0" />
  <mass value="2.6578" />
  <inertial>
  ixx="0.00856502765719703"
  ixy="1.5074118157338E-19"
  ixz="-4.78150098725052E-19"
  iyy="0.013670640432096"
  iyz="-2.68136447099727E-19"
  izz="0.00856502765719703" />
</inertial>
```

Now that the Gazebo properties have been created, let's create the mechanisms.

(2) Define actuators. Now, we need to define the actuator information for our robot wheels in the robot_base_essentials.xacro file:

```
<xacro:macro name="base_transmission" params="prefix ">
 <transmission name="${prefix}_wheel_trans" type="SimpleTransmission">
  <type>transmission_interface/SimpleTransmission</type>
  <actuator name="${prefix}_wheel_motor">
    <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
  <joint name="${prefix}_wheel_joint">
    <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
  </joint>
 </transmission>
</xacro:macro>
```

Let's call them in our robot file robot_base.urdf.xacro as macros:

```
<xacro:base_transmission prefix="front_left"/>
<xacro:base_transmission prefix="front_right"/>
<xacro:base_transmission prefix="rear_left"/>
<xacro:base_transmission prefix="rear_right"/>
```

Now that the mechanisms have been called, let's call the controllers that use them and make our robot dynamic.

(3) Define ros_controllers. Finally, we need to port the plugins that are needed to establish communication between Gazebo and ROS. For these, we need to create another file called gazebo_essentials_base.xacro that will contain the tags. In the created file, add the following gazebo_ros_control plugin:

```
<gazebo>
 <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
 <robotNamespace>/</robotNamespace>
 <controlPeriod>0.001</controlPeriod>
 <legacyModeNS>false</legacyModeNS>
 </plugin>
</gazebo>
```

The robot's differential drive plugin is as follows:

```
<gazebo>
        <plugin name="diff_drive_controller" filename="libgazebo_ros_diff_drive.so">
          <legacyMode>false</legacyMode>
          <alwaysOn>true</alwaysOn>
          <updateRate>1000.0</updateRate>
```

```
            <leftJoint>front_left_wheel_joint, rear_left_wheel_joint</leftJoint>
            <rightJoint>front_right_wheel_joint, rear_right_wheel_joint</rightJoint>
            <wheelSeparation>0.5</wheelSeparation>
            <wheelDiameter>0.2</wheelDiameter>
            <wheelTorque>10</wheelTorque>
            <publishTf>1</publishTf>
            <odometryFrame>map</odometryFrame>
            <commandTopic>cmd_vel</commandTopic>
            <odometryTopic>odom</odometryTopic>
            <robotBaseFrame>base_link</robotBaseFrame>
            <wheelAcceleration>2.8</wheelAcceleration>
            <publishWheelJointState>true</publishWheelJointState>
            <publishWheelTF>false</publishWheelTF>
            <odometrySource>world</odometrySource>
            <rosDebugLevel>Debug</rosDebugLevel>
        </plugin>
</gazebo>
```

The friction property for the wheels as macros is as follows:

```
<xacro:macro name="wheel_friction" params="prefix ">

<gazebo reference="${prefix}_wheel">
  <mu1 value="1.0"/>
  <mu2 value="1.0"/>
  <kp value="10000000.0" />
  <kd value="1.0" />
  <fdir1 value="1 0 0"/>
</gazebo>

</xacro:macro>
```

The call macro in the robot file robot_base.urdf.xacro is as follows:

```
<xacro:wheel_friction prefix="front_left"/>
<xacro:wheel_friction prefix="front_right"/>
<xacro:wheel_friction prefix="rear_left"/>
<xacro:wheel_friction prefix="rear_right"/>
```

Now that we have defined the macros for our robot, along with the Gazebo plugins, let's add them to our robot file robot_base.urdf.xacro. This can be easily done by just adding the following two lines in the robot file inside the <robot> macro tag:

```
<xacro:include filename="$(find my_robot)/urdf/robot_base_essentials.xacro" />
<xacro:include filename="$(find my_robot)/urdf/gazebo_essentials_base.xacro" />
```

Now that the URDFs are complete, let's configure the controllers. Let's create the following config file to define the controllers we are using. For this, let's go to our workspace, go inside the config folder we created, and create a controller config file, as shown here:

```
$ cd ~/project2_ws/src/my_robot/config
$ gedit control.yaml
```

Copy the following code into the file:

```
robot_base_joint_publisher:
  type: "joint_state_controller/JointStateController"
  publish_rate: 50

robot_base_velocity_controller:
  type: "diff_drive_controller/DiffDriveController"
  left_wheel: ['front_left_wheel_joint', 'rear_left_wheel_joint']
  right_wheel: ['front_right_wheel_joint', 'rear_right_wheel_joint']
  publish_rate: 50
  pose_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03]
  twist_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03]
  cmd_vel_timeout: 0.5
  wheel_separation : 0.445208
  wheel_radius : 0.0625
  base_frame_id: base_link
  odom_frame_id: odom
  enable_odom_tf: true
  estimate_velocity_from_position: false
  wheel_separation_multiplier: 1.0
  wheel_radius_multiplier    : 1.0
  linear:
    x:
      has_velocity_limits    : true
      max_velocity           : 2.0
      has_acceleration_limits: true
      max_acceleration       : 3.0
  angular:
    z:
      has_velocity_limits    : true
      max_velocity           : 3.0
      has_acceleration_limits: true
```

```
        max_acceleration            : 6.0

/gazebo_ros_control:
      pid_gains:
        front_left_wheel_joint: {p: 20.0, i: 0.01, d: 0.0, i_clamp: 0.0}
        front_right_wheel_joint: {p: 20.0, i: 0.01, d: 0.0, i_clamp: 0.0}
        rear_left_wheel_joint: {p: 20.0, i: 0.01, d: 0.0, i_clamp: 0.0}
        rear_right_wheel_joint: {p: 20.0, i: 0.01, d: 0.0, i_clamp: 0.0}
```

Congratulations! You've completed the robot base model. It is now the time to simulate it in Gazebo.

6. Testing the Robot Base

Now that we have the complete model for our robot base, let's put it into action and see how our base moves. Follow these steps:
(1). Let's create a launch file that will spawn the robot and its controllers. Now, go into the launch folder of our my_robot package and create the following launch file:

```
$ cd ~/project2_ws/src/my_robot/launch
$ gedit base_gazebo_control_xacro.launch
```

(2). Now, copy the following code into it and save the file:

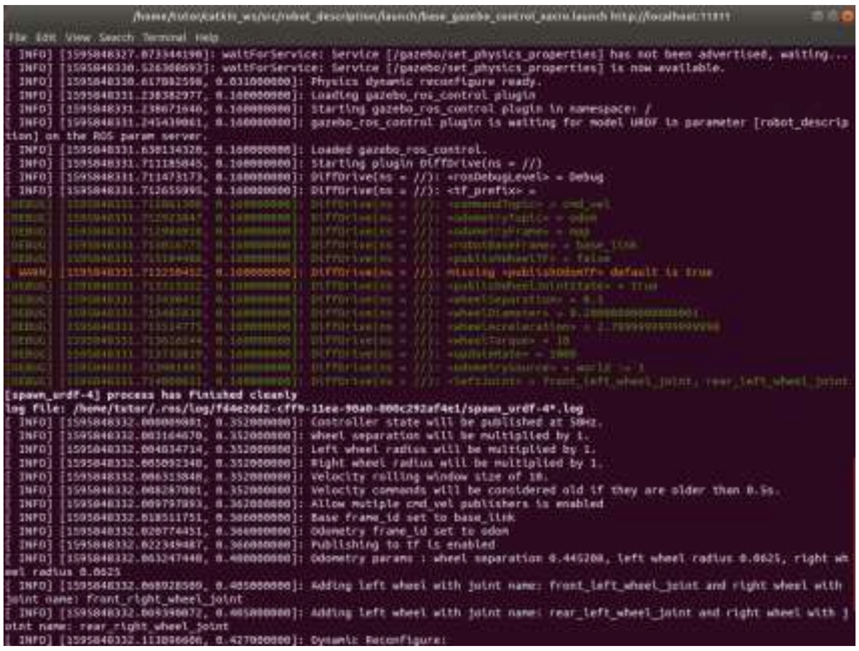
```
<?xml version="1.0"?>
<launch>
   <param name="my_robot" command="$(find xacro)/xacro --inorder $(find
my_robot)/urdf/robot_base.urdf.xacro" />
   <include file="$(find gazebo_ros)/launch/empty_world.launch"/>
   <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param
my_robot -urdf -model robot_base" />
   <rosparam command="load" file="$(find my_robot)/config/control.yaml" />
   <node name="base_controller_spawner" pkg="controller_manager" type="spawner"
args="robot_base_joint_publisher robot_base_velocity_controller"/>

</launch>
```

(3). Now, you can visualize your robot by running the following command:

```
$ source ~/project2_ws/devel/setup.bash
$ roslaunch my_robot base_gazebo_control_xacro.launch
```

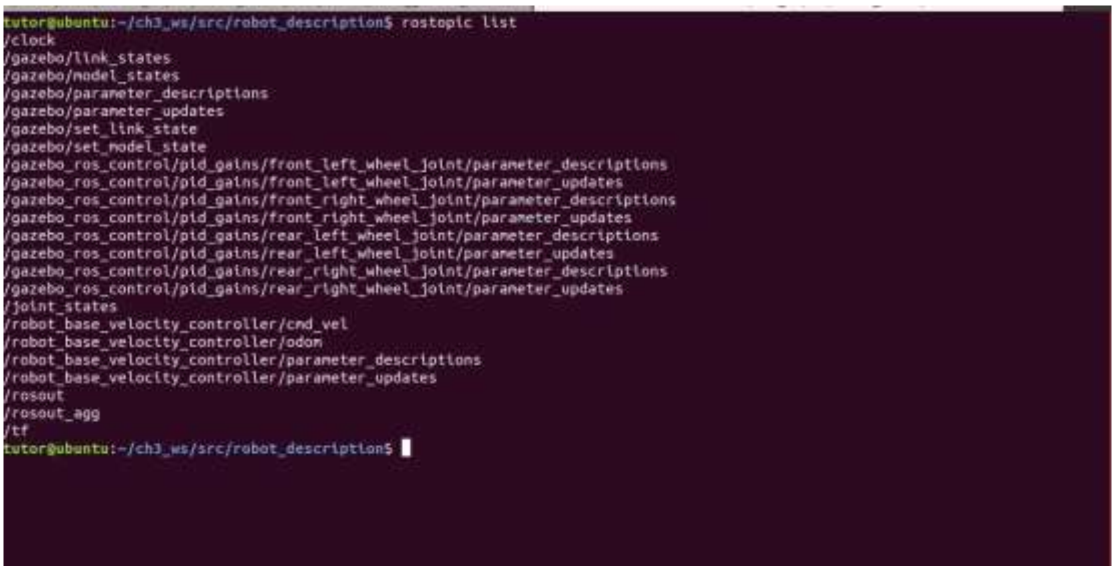
Once launched, you should see something like below, without errors.



```
/home/tutor/catkin_ws/src/robot_description/launch/base_gazebo_control_xacro.launch http://localhost:11311
[ INFO] [1595848327.073344190]: waitForService: Service [/gazebo/set_physics_properties] has not been advertised, waiting...
[ INFO] [1595848330.526300693]: waitForService: Service [/gazebo/set_physics_properties] is now available.
[ INFO] [1595848330.617082598, 0.031000000]: Physics dynamic reconfigure ready.
[ INFO] [1595848331.238382977, 0.160000000]: Loading gazebo_ros_control plugin
[ INFO] [1595848331.238671646, 0.160000000]: Starting gazebo_ros_control plugin in namespace: /
[ INFO] [1595848331.245430061, 0.160000000]: gazebo_ros_control plugin is waiting for model URDF in parameter [robot_descrip
tion] on the ROS param server.
[ INFO] [1595848331.630134328, 0.160000000]: Loaded gazebo_ros_control.
[ INFO] [1595848331.711185845, 0.160000000]: Starting plugin DiffDrive(ns = //)
[ INFO] [1595848331.711473173, 0.160000000]: DiffDrive(ns = //): <rosDebugLevel> = Debug
[ INFO] [1595848331.712655995, 0.160000000]: DiffDrive(ns = //): <tf_prefix> =
[DEBUG] [illegible]: DiffDrive(ns = //): <commandTopic> = cmd_vel
[DEBUG] [illegible]: DiffDrive(ns = //): <odometryTopic> = odom
[DEBUG] [illegible]: DiffDrive(ns = //): <odometryFrame> = map
[DEBUG] [illegible]: DiffDrive(ns = //): <robotBaseFrame> = base_link
[DEBUG] [illegible]: DiffDrive(ns = //): <publishWheelTF> = false
[ WARN] [1595848331.713250452, 0.160000000]: DiffDrive(ns = //): missing <publishOdomTF> default is true
[DEBUG] [illegible]: DiffDrive(ns = //): <publishWheelJointState> = true
[DEBUG] [illegible]: DiffDrive(ns = //): <wheelSeparation> = 0.1
[DEBUG] [illegible]: DiffDrive(ns = //): <wheelDiameter> = 0.100000000000000001
[DEBUG] [illegible]: DiffDrive(ns = //): <wheelAcceleration> = 2.79999999999999998
[DEBUG] [illegible]: DiffDrive(ns = //): <wheelTorque> = 10
[DEBUG] [illegible]: DiffDrive(ns = //): <updateRate> = 1000
[DEBUG] [illegible]: DiffDrive(ns = //): <odometrySource> = world := 1
[DEBUG] [illegible]: DiffDrive(ns = //): <leftJoint> = front_left_wheel_joint; rear_left_wheel_joint
[spawn_urdf-4] process has finished cleanly
log file: /home/tutor/.ros/log/fd4e26d2-cff9-11ea-90a0-000c292af4e1/spawn_urdf-4*.log
[ INFO] [1595848332.000000001, 0.352000000]: Controller state will be published at 50Hz.
[ INFO] [1595848332.003164670, 0.352000000]: Wheel separation will be multiplied by 1.
[ INFO] [1595848332.004034714, 0.352000000]: Left wheel radius will be multiplied by 1.
[ INFO] [1595848332.005092340, 0.352000000]: Right wheel radius will be multiplied by 1.
[ INFO] [1595848332.006313848, 0.352000000]: Velocity rolling window size of 10.
[ INFO] [1595848332.008287001, 0.352000000]: Velocity commands will be considered old if they are older than 0.5s.
[ INFO] [1595848332.009797893, 0.362000000]: Allow mutiple cmd_vel publishers is enabled
[ INFO] [1595848332.018511751, 0.366000000]: Base frame_id set to base_link
[ INFO] [1595848332.020774451, 0.366000000]: Odometry frame_id set to odom
[ INFO] [1595848332.022349487, 0.366000000]: Publishing to tf is enabled
[ INFO] [1595848332.063247448, 0.400000000]: Odometry params : wheel separation 0.445208, left wheel radius 0.0625, right wh
eel radius 0.0625
[ INFO] [1595848332.068928509, 0.405000000]: Adding left wheel with joint name: front_left_wheel_joint and right wheel with
joint name: front_right_wheel_joint
[ INFO] [1595848332.069390072, 0.405000000]: Adding left wheel with joint name: rear_left_wheel_joint and right wheel with j
oint name: rear_right_wheel_joint
[ INFO] [1595848332.113086606, 0.427000000]: Dynamic Reconfigure:
```

Figure 4: Launch outcomes

(4). You can view the necessary ROS topics by opening another Terminal and running rostopic list:



```
tutor@ubuntu:~/ch3_ws/src/robot_description$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_ros_control/pid_gains/front_left_wheel_joint/parameter_descriptions
/gazebo_ros_control/pid_gains/front_left_wheel_joint/parameter_updates
/gazebo_ros_control/pid_gains/front_right_wheel_joint/parameter_descriptions
/gazebo_ros_control/pid_gains/front_right_wheel_joint/parameter_updates
/gazebo_ros_control/pid_gains/rear_left_wheel_joint/parameter_descriptions
/gazebo_ros_control/pid_gains/rear_left_wheel_joint/parameter_updates
/gazebo_ros_control/pid_gains/rear_right_wheel_joint/parameter_descriptions
/gazebo_ros_control/pid_gains/rear_right_wheel_joint/parameter_updates
/joint_states
/robot_base_velocity_controller/cmd_vel
/robot_base_velocity_controller/odom
/robot_base_velocity_controller/parameter_descriptions
/robot_base_velocity_controller/parameter_updates
/rosout
/rosout_agg
/tf
tutor@ubuntu:~/ch3_ws/src/robot_description$ 
```

Figure 5: rostopic list outcomes

The Gazebo view of robot should be as follows.

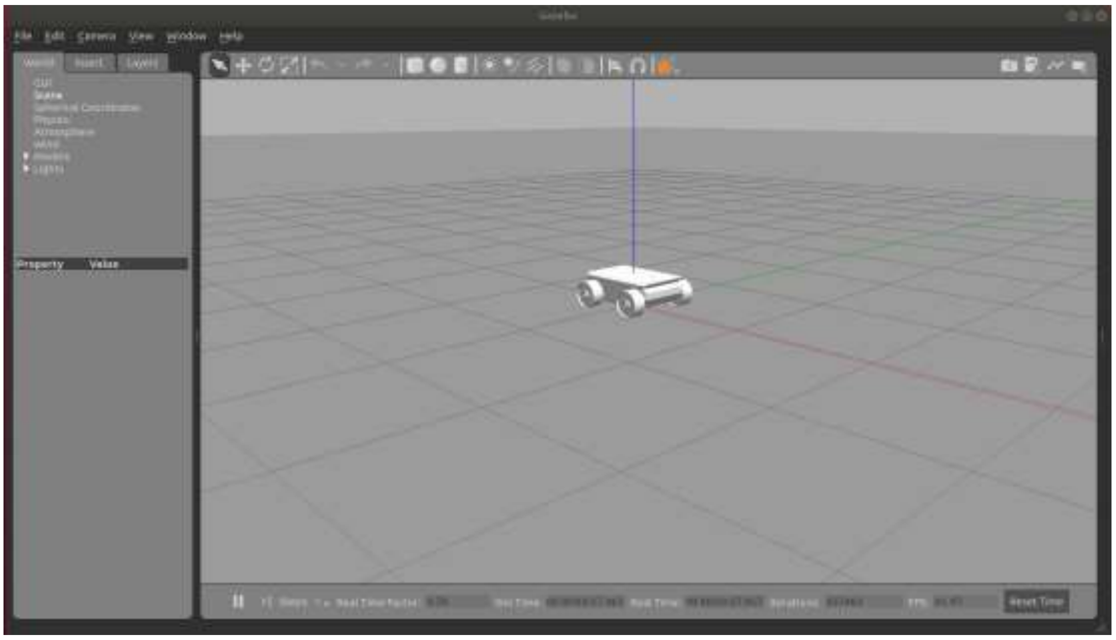


Figure 6: The Gazebo simulation

(5). Try using the rqt_robot_steering node and move the robot, as follows:

```
$ rosrun rqt_robot_steering rqt_robot_steering
```

In the window, mention our topic, /robot_base_controller/cmd_vel. Now, move the sliders slowly and see how the robot base moves around. The robot base is all DONE!

7. Building the Robot Arm

Now that we built a robot base in URDF and visualized it in Gazebo, let's get to building the robot arm. The robot arm will be built in a similar way to using URDF and there are no differences in terms of its approach. There may be only a few changes that need to be made to the robot base URDF. We will not describe this in detail, and simply show the file names and commands for the simulation. If you are suggested to review the code after the session if you are interested.

(1). Robot Arm Specification

- Type: 5 DOF (short for degrees of freedom) robot arm
- Payload: Within 3-5 kgs

(2). Software Parameters

If we consider the arm a black box, the arm gives out a pose based on the commands each actuator receives. The commands may be in the form of position, force/effort, or velocity commands. A simple representation is shown here:

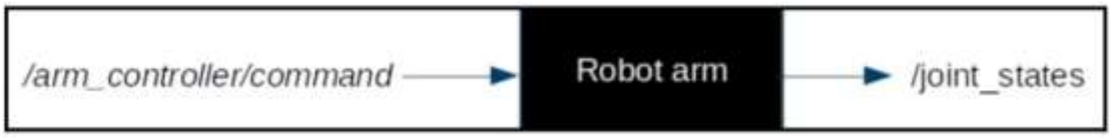


Figure 7: The robot arm as a black box

ROS message: The /arm_controller/command topic, which is used to command or control the robot arm, is of the trajectory_msgs/JointTrajectory message format.

ROS controllers: joint_trajectory_controller is used for executing joint space trajectories on a list of given joints. The trajectories to the controller are sent using the action interface, control_msgs/FollowJointTrajectoryAction, in the follow_joint_trajectory namespace of the controller.

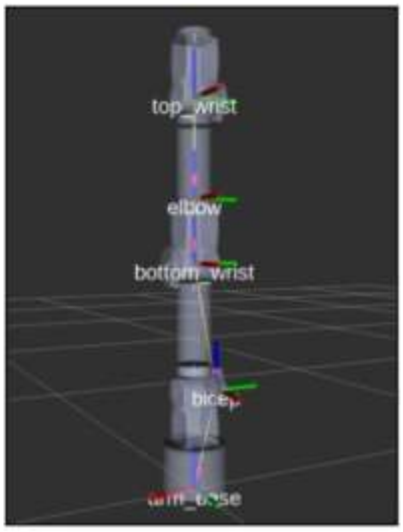


Figure 8: Representing the arm in 3D with links

(3). Copy robot_arm.urdf.xacro from the reference package you've downloaded, to the same urdf directory of your package my_robot. You can analyse the file by yourself.

```
$ roscd my_robot
$ cp ../robot_description/urdf/robot_arm.urdf.xacro ./urdf
$ cp ../robot_description/urdf/robot_essentials.xacro ./urdf
```

You can analyse the file, focusing on the components of
Links: robot_arm.urdf.xacro
Joints: robot_essentials.xacro, robot_arm.urdf.xacro
Collision: robot_arm.urdf.xacro
Actuators: robot_arm_essentials.xacro, robot_arm.urdf.xacro
Controllers: gazebo_essentials_arm.xacro, arm_control.yaml, robot_arm.urdf.xacro

Now you can visualize your arm in rviz:

```
$ roscd my_robot/urdf
$ roslaunch urdf_tutorial display.launch model:=robot_arm.urdf.xacro
```

Add the robot model and, in Global options, set Fixed Frame to arm_base. Now, you should see our robot model if everything was successful.

(4). Test the Robot Arm

```
$ roscd my_robot
$ cp ../robot_description/urdf/robot_arm_essentials.xacro ./urdf
$ cp ../robot_description/urdf/gazebo_essentials_arm.xacro ./urdf
$ cp ../robot_description/config/arm_control.yaml ./config
$ cp ../robot_description/launch/arm_gazebo_control_xacro.launch ./launch
```

Now, you could visualize your robot by running the following command:

```
$ roslaunch my_robot arm_gazebo_control_xacro.launch
```

You can evaluate the project by using, e.g. rostopic list, etc.
You can also publish in command line a command to control the arm.

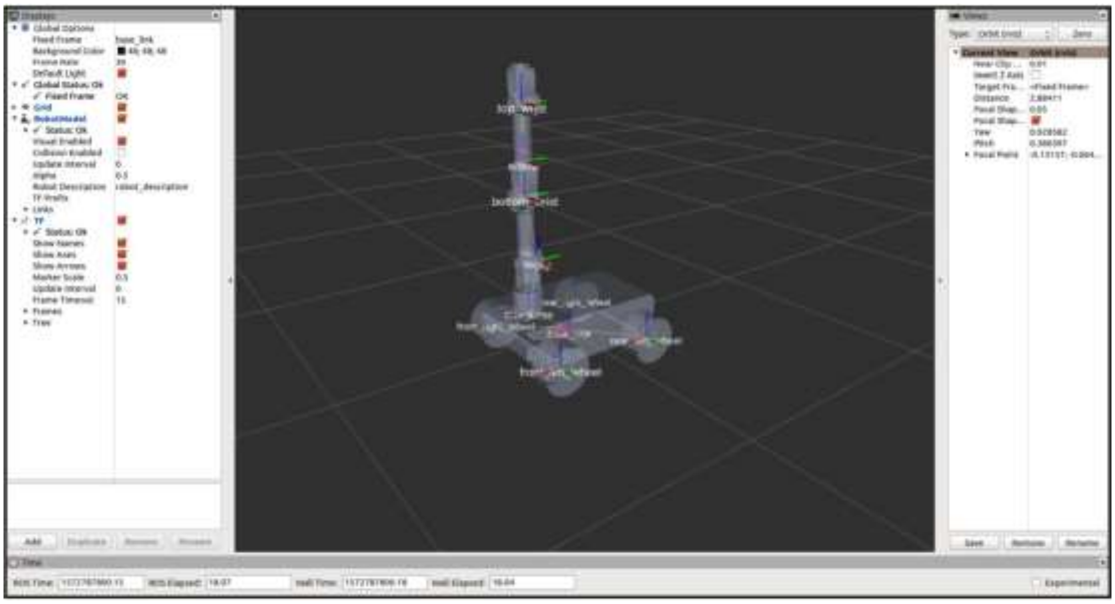
```
$  rostopic pub /arm_controller/command trajectory_msgs/JointTrajectory
'{joint_names: ["arm_base_joint", "shoulder_joint", "bottom_wrist_joint", "elbow_joint",
"top_wrist_joint"], points: [{positions: [-0.1, 0.210116830848170721,
0.022747275919015486, 0.0024182584123728645, 0.00012406874824844039],
time_from_start: [1.0,0.0]}]}'
```

It seems a bit complex. Do not worry, we will learn how to move the arm in a more convenient way.

8. Putting Things Together

We now combine arm and base together to form a mobile manipulator. We still copy files from our sources, while we need to analyse them.

```
$ roscd my_robot
$ cp ../robot_description/urdf/mobile_manipulator.urdf.xacro ./urdf
$ roslaunch urdf_tutorial display.launch model:=./urdf/mobile_manipulator.urdf.xacro
```



```
Displays
Global Options
  Fixed Frame            base_link
  Background Color       48; 48; 48
  Frame Rate             30
  Default Light          ✓
Global Status: Ok
  Fixed Frame            OK
Grid                     ✓
RobotModel               ✓
  Status: Ok
  Visual Enabled         ✓
  Collision Enabled
  Update Interval        0
  Alpha                  0.5
  Robot Description      robot_description
  TF Prefix
  Links
TF                       ✓
  Status: Ok
  Show Names             ✓
  Show Axes              ✓
  Show Arrows            ✓
  Marker Scale           0.5
  Update Interval        0
  Frame Timeout          15
  Frames
  Tree
```

```
Views
Type: Orbit (rviz)    Zero
Current View     Orbit (rviz)
  Near Clip ...  0.01
  Invert Z Axis
  Target Fra...  <Fixed Frame>
  Distance       2.88411
  Focal Shap...  0.05
  Focal Shap...  ✓
  Yaw            0.508582
  Pitch          0.388397
  Focal Point    (0.13131; -0.004...
```

Figure 9: The rviz model view of the mobile manipulator

Copy the launch file to simulate the robot in Gazebo:

```
$ cp ../robot_description/launch/mobile_manipulator_gazebo_control_xacro.launch ./launch
```

Now you can visualize your robot by running the following command：

```
$ roslaunch my_robot mobile_manipulator_gazebo_xacro.launch
```

You can now try to move the base and arm, as shown in their respective sections. We will learn how to control the manipulator using MoveIt! package in the next half.