

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 ~/chapter3_ws/src/robot_description/config/  
$ gedit control.yaml
```

Now, copy the code that's available in this book's GitHub repository, https://github.com/PacktPublishing/ROS-Robotics-Projects-SecondEdition/blob/master/chapter_3_ws/src/robot_description/config/control.yaml, into the file. Now that we have completed our robot_base model, let's test it in Gazebo.

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 and create the following launch file:

```
$ cd ~/chapter3_ws/src/robot_description/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="robot_description" command="$(find xacro)/xacro --  
inorder $(find robot_description)/urdf/robot_base.urdf.xacro" />  
  <include file="$(find gazebo_ros)/launch/empty_world.launch"/>  
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"  
args="-param robot_description -urdf -model robot_base" />  
  <rosparam command="load" file="$(find  
robot_description)/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:

```
$ cd ~/chapter3_ws
$ source devel/setup.bash
$ roslaunch robot_description base_gazebo_control_xacro.launch
```

Once the Gazebo environment launches, you should see something like the following without any error prompt:

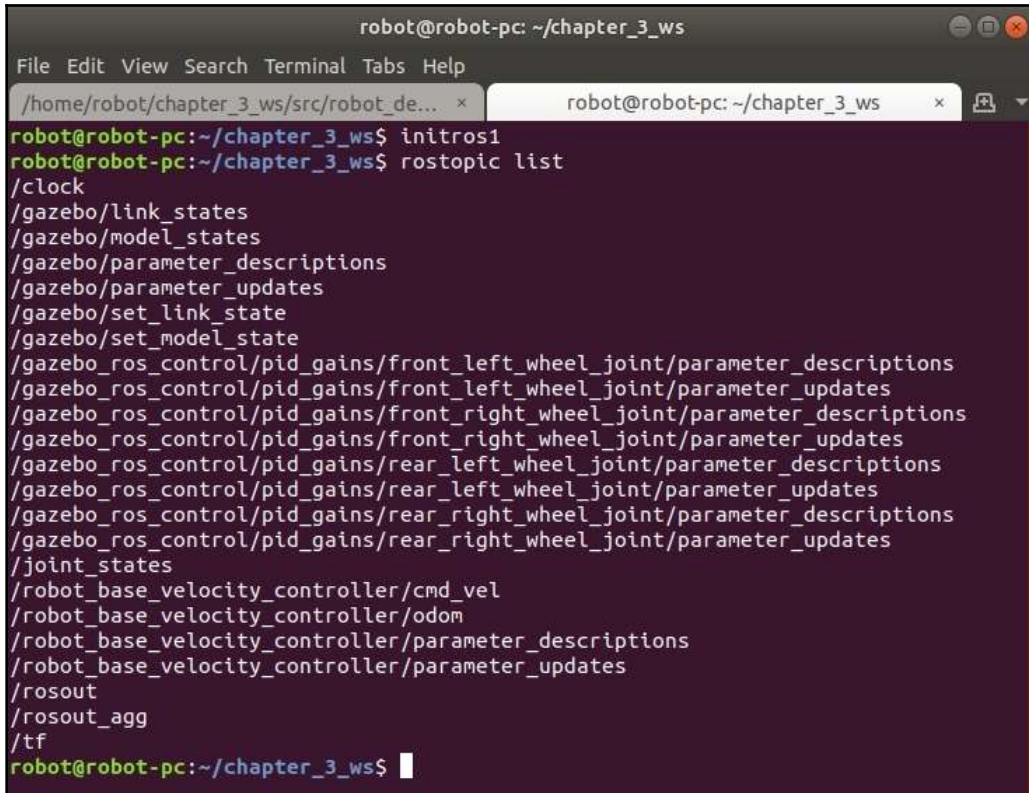
```
/home/robot/chapter_3_ws/src/robot_description/launch/base_gazebo_control_xacro.launch http://localhost:11311
File Edit View Search Terminal Help
process[spawn_urdf-4]: started with pid [23271]
process[base_controller_spawner-5]: started with pid [23272]
[ INFO] [1557664801.100103700]: Finished loading Gazebo ROS API Plugin.
[ INFO] [1557664801.103041871]: waitforService: Service [/gazebo/set_physics_properties] has not been advertised, waiting...
[ INFO] [1557664801.239502410]: Finished loading Gazebo ROS API Plugin.
[ INFO] [1557664801.242057079]: waitforService: Service [/gazebo_gui/set_physics_properties] has not been advertised, waiting...
[ INFO] [1557664801.615825535, 0.073000000]: waitforService: Service [/gazebo/set_physics_properties] is now available.
[ INFO] [1557664801.666036358, 0.069000000]: Physics dynamic reconfigure ready.
[ INFO] [1557664802.009308761, 0.314000000]: Loading gazebo_ros_control plugin.
[ INFO] [1557664802.009494434, 0.314000000]: Starting gazebo_ros_control plugin in namespace: /
[ INFO] [1557664802.010807312, 0.314000000]: gazebo_ros_control plugin is waiting for model URDF in parameter [robot_description] on the ROS p
aram server.
[ INFO] [1557664802.215271155, 0.314000000]: Loaded gazebo_ros_control.
[spawn_urdf-4] process has finished cleanly
log file: /home/robot/.ros/log/0dccc156c-74b3-11e9-8061-38b1dbec8a43/spawn_urdf-4*.log
[ INFO] [1557664802.251202646, 0.314000000]: Starting plugin DiffDrive(ns = /)
[ INFO] [1557664802.251419943, 0.314000000]: DiffDrive(ns = /): <rosDebugLevel> = Debug
[ INFO] [1557664802.252370583, 0.314000000]: DiffDrive(ns = /): <tf_prefix> =
[DEBUG] [1557664802.252506285, 0.314000000]: DiffDrive(ns = /): <commandTopic> = cmd_vel
[DEBUG] [1557664802.252539237, 0.314000000]: DiffDrive(ns = /): <odometryTopic> = odom
[DEBUG] [1557664802.252554858, 0.314000000]: DiffDrive(ns = /): <odometryFrame> = map
[DEBUG] [1557664802.252611353, 0.314000000]: DiffDrive(ns = /): <robotBaseFrame> = base_link
[DEBUG] [1557664802.252722067, 0.314000000]: DiffDrive(ns = /): <publishWheelTF> = false
[WARN] [1557664802.252766247, 0.314000000]: DiffDrive(ns = /): missing <publishOdontF> default is true
[DEBUG] [1557664802.252795882, 0.314000000]: DiffDrive(ns = /): <publishWheelJointState> = true
[DEBUG] [1557664802.252900507, 0.314000000]: DiffDrive(ns = /): <wheelSeparation> = 0.5
[DEBUG] [1557664802.252940285, 0.314000000]: DiffDrive(ns = /): <wheelDiameter> = 0.20000000000000001
[DEBUG] [1557664802.252969589, 0.314000000]: DiffDrive(ns = /): <wheelAcceleration> = 2.7999999999999998
[DEBUG] [1557664802.252998123, 0.314000000]: DiffDrive(ns = /): <wheelTorque> = 10
[DEBUG] [1557664802.253028667, 0.314000000]: DiffDrive(ns = /): <updateRate> = 1000
[DEBUG] [1557664802.253199561, 0.314000000]: DiffDrive(ns = /): <odometrySource> = world := 1
[DEBUG] [1557664802.253282733, 0.314000000]: DiffDrive(ns = /): <leftJoint> = front_left_wheel_joint, rear_left_wheel_joint
[ INFO] [1557664802.553809450, 0.600000000]: Controller state will be published at 50Hz.
[ INFO] [1557664802.556058217, 0.600000000]: Wheel separation will be multiplied by 1.
[ INFO] [1557664802.557208962, 0.601000000]: Left wheel radius will be multiplied by 1.
[ INFO] [1557664802.557322897, 0.601000000]: Right wheel radius will be multiplied by 1.
[ INFO] [1557664802.558400608, 0.602000000]: Velocity rolling window size of 10.
[ INFO] [1557664802.561558254, 0.605000000]: Velocity commands will be considered old if they are older than 0.5s.
```

Successful Gazebo launch

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

```
$ initros1
$ rostopic list
```

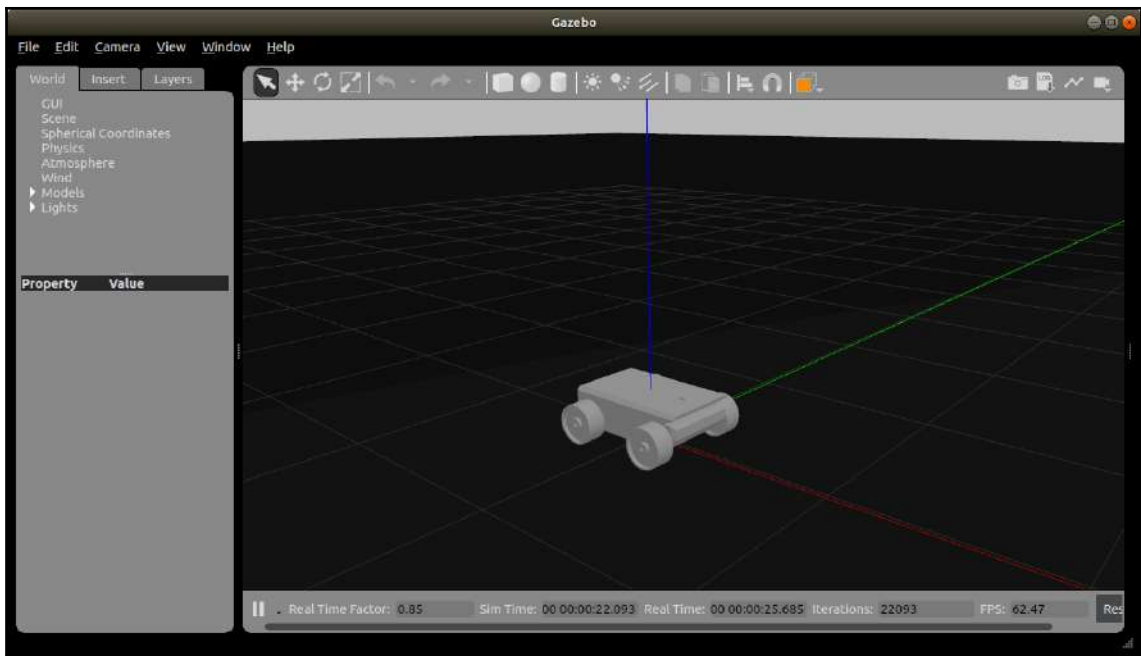
The following screenshot shows the list of ROS topics:



```
robot@robot-pc: ~/chapter_3_ws
File Edit View Search Terminal Tabs Help
/home/robot/chapter_3_ws/src/robot_de... x robot@robot-pc: ~/chapter_3_ws x
robot@robot-pc:~/chapter_3_ws$ initros1
robot@robot-pc:~/chapter_3_ws$ 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
robot@robot-pc:~/chapter_3_ws$
```

List of ROS topics

The Gazebo view of the robot should be as follows:



Robot base in Gazebo

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

```
$ rosrund 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.



Have a look at the robot URDF file in this book's GitHub repository: https://github.com/PacktPublishing/ROS-Robotics-Projects-SecondEdition/blob/master/chapter_3_ws/src/robot_description/urdf/robot_base.urdf.xacro before testing the robot base.

We have completed our robot base. Now, let's learn how to build the robot arm.

Getting started 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. Let's see how the robot arm is built, step by step.

Robot arm prerequisites

To build a robot arm, we need the following:

- A good set of linkages that move independently with good physical strength
- A good set of actuators that could help us withstand enough payload
- Drive controls

In case you plan to build a real robot arm, you might need similar embedded architecture and electronic controls, real-time communication between the actuators, a power management system, and maybe a good end effector, based on your application requirements. As you know, there's more to consider and this is not in the scope of this book. However, the aim is to emulate a robot arm in ROS so that it works the same way in reality as well. Now, let's look at the robot arm specifications.

Robot arm specifications

Here, we want to build a mobile manipulator. Our robot arm doesn't need to carry payloads of tons. In fact, in reality, such a robot would require heavy machinery drives and mechanical power transmission systems and so it might not be that easy to mount on a movable platform.

Let's be practical and consider common parameters that some of the industrial cobots follow in the market:

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

Now, let's look at the robot arm kinematics.

Robot arm kinematics

Robot arm kinematics is slightly different compared to robot base kinematics. You would need to move five different actuators to five different positions to move the robot arm's tooltip to a required position. The mathematical modeling follows the **Denavit-Hartenberg (DH)** method of computing kinematics. Explaining the DH method is out of our scope, so we shall look at the kinematics equation directly.

The arm kinematics equation is defined by a 4×4 homogeneous transformation matrix that connects all of the five links with respect to the robot base coordinate system, as shown here:

$$T = \begin{bmatrix} C_1 C_{234} C_5 + S_1 S_5 & -C_1 C_{234} S_5 + S_1 C_5 & -C_1 S_{234} & C_1 (-d_5 S_{234} + a_3 C_{23} + a_2 C_2) \\ C_1 C_{234} C_5 - S_1 S_5 & -S_1 C_{234} S_5 - C_1 C_5 & -S_1 S_{234} & S_1 (-d_5 S_{234} + a_3 C_{23} + a_2 C_2) \\ -S_{234} C_5 & S_{234} S_5 & -C_{234} & d_1 - a_2 S_2 - a_3 S_{23} - d_5 C_{234} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here, we have the following:

$$C_{ijk} = \cos(q_i + q_j + q_k), S_{ijk} = \sin(q_i + q_j + q_k),$$

This represents trigonometric equations; q_i is the angle between the normal to rotation axis (usually represented as x_i) and the rotation axis (usually represented as z_i), and is the same for q_j and q_k , respectively. d_i is the distance from the rotation axis (z_i) to the origin ($i-1$) system of axis, and a_i is the shortest distance between two consecutive rotation axes.

From the preceding homogeneous transform, the first 3×3 element indicates the rotation of the gripper or tool, the bottom row defines the scale factor, and the tool's pose is given by the remaining elements:

$$\begin{aligned} Tool_{pose} = & \begin{bmatrix} C_1 (-d_5 S_{234} + a_3 C_{23} + a_2 C_2) \\ S_1 (-d_5 S_{234} + a_3 C_{23} + a_2 C_2) \\ d_1 - a_2 s_2 - a_3 s_{23} - d_5 C_{234} \end{bmatrix} \end{aligned}$$

Now, let's look at the software parameters.

Software parameters

So, 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:



The robot arm as a black box

Let's look into their message representations.

The ROS message format

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

This message structure can be found at http://docs.ros.org/melodic/api/trajectory_msgs/html/msg/JointTrajectory.html.

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.

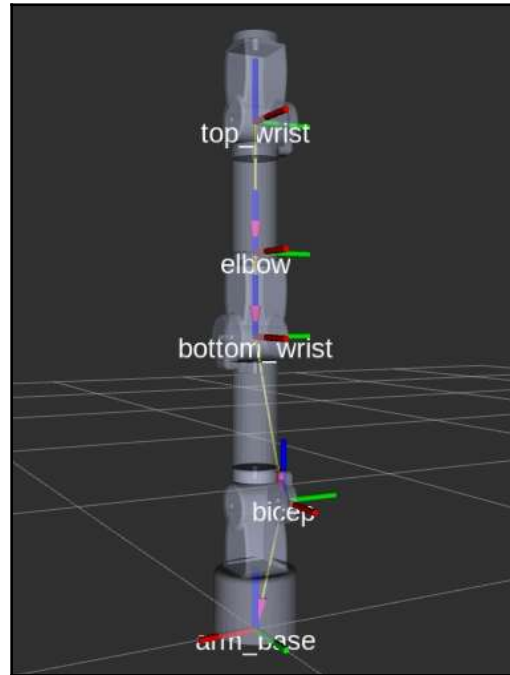


More information about this controller is available at the following website: http://wiki.ros.org/joint_trajectory_controller.

Now, let's learn how to model the robot arm.

Modeling the robot arm

This is what our robot arm will look like once it's been built in ROS:



Representing the robot arm in 2D with link representation

Since all of the necessary explanation has already been provided in the *Modeling the robot base* section, and since this section takes the same approach as well, let's get into modeling the robot arm step by step.

Initializing the workspace

To initialize the workspace, we will make use of the same workspace we created for this chapter (`chapter_3_ws`). Download the meshes from the link to the meshes folder you created in the *Building the robot base* section. Now, go to the `urdf` folder and create a file called `robot_arm.urdf.xacro` using the following commands:

```
$ cd ~/chapter3_ws/src/robot_description/urdf/  
$ gedit robot_arm.urdf.xacro
```


Initialize the XML version tag and the `<robot>` tag as follows and begin copying the given XML code into it step by step:

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

</robot>
```

Now that we have initialized the workspace, let's move on to the next step, which is defining the links.

Defining the links

Now, since you're defining the robot model by parts, copy the following code into the `<robot>` tags (that is, the space between the `<robot>` tags). Note that we'll define all of the five links directly since they contain coordinate information. Let's look at one of them in detail, the arm base:

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

You could find the other links such as `bicep`, `bottom_wrist`, `elbow`, and `top_wrist`, that have been defined in this file in this book's GitHub repository: https://github.com/PacktPublishing/ROS-Robotics-Projects-SecondEdition/blob/master/chapter_3_ws/src/robot_description/urdf/robot_arm.urdf.xacro.

You could try their more macros for `<visual>` and `<collision>` tags with respective parameter definitions, but this sometimes might lead to confusion for beginners. Hence, to keep it simple, we're only defining minimal macros in this book. If you feel you have mastered macros, you are welcome to test your skills by adding more macros.

Defining the joints

The joints we will define for the robot arm are revolute, the joint can move between limits. Since all of the joints in this robot arm move between limits and are common for all, we define them in the same `robot_essentials.xacro` file we created while building the robot base:

```
<xacro:macro name="arm_joint" params="prefix origin">

  <joint name="${prefix}_joint" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="arm_base"/>
    <child link="${prefix}_joint"/>
    <origin rpy="0 0 0" xyz="${origin}"/>
  </joint>

</xacro:macro>
```

In the `robot_arm.urdf.xacro` file, we define them as follows:

```
<xacro:arm_joint prefix="shoulder" parent="arm_base" child="bicep"
originxyz="-0.05166 0.0 0.20271" originrpy="0 0 1.5708"/>
<xacro:arm_joint prefix="bottom_wrist" parent="bicep" child="bottom_wrist"
originxyz="0.0 -0.05194 0.269" originrpy="0 0 0"/>
<xacro:arm_joint prefix="elbow" parent="bottom_wrist" child="elbow"
originxyz="0.0 0 0.13522" originrpy="0 0 0"/>
<xacro:arm_joint prefix="top_wrist" parent="elbow" child="top_wrist"
originxyz="0.0 0 0.20994" originrpy="0 0 0"/>
```

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

```
$ initros1
$ cd ~/chapter3_ws/
$ source devel/setup.bash
$ roscd robot_description/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. You could add the `tf` display and see whether the representation matches the *Representing the robot arm in 2D with link representation* screenshot. You could move the arm links using the sliders that were launched as a result of setting the `gui` argument to `true`.

Simulating the robot arm

The first three steps were initially used to define the arm URDF model that 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. We will start by defining collisions.

Defining collisions

To visualize the robot in Gazebo, we need to add `<collision>` tags, along with the `<visual>` tags we defined in the `<link>` tag, like we did previously. Hence, just specify the necessary visual and inertial tags on the links:

- For the `arm_base` link, add the following:

```
<collision>
  <origin
    xyz="0 0 0"
    rpy="0 0 0" />
  <geometry>
    <mesh
      filename="package://robot_description/meshes/arm_base.stl" />
    </geometry>
  </collision>

<inertial>
  <origin
    xyz="7.7128E-09 -0.063005 -3.01969999961422E-08"
    rpy="0 0 0" />
  <mass
    value="1.6004" />
  <inertia
    ixx="0.00552196561445819"
    ixy="7.9550614501301E-10"
    ixz="-1.34378458924839E-09"
    iyy="0.00352397447953875"
    iyz="-1.10071809773382E-08"
    izz="0.00553739792746489" />
</inertial>
```

- For the other links, please have a look at the code for `robot_arm.urdf.xacro` at https://github.com/PacktPublishing/ROS-Robotics-Projects-SecondEdition/blob/master/chapter_3_ws/src/robot_description/urdf/robot_arm.urdf.xacro.

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

Defining actuators

Now, we can define the actuator information for all of the links. The actuator macro is defined in the `robot_arm_essentials.xacro` file, as shown here:

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

  <transmission name="${prefix}_trans" type="SimpleTransmission">
    <type>transmission_interface/SimpleTransmission</type>
    <actuator name="${prefix}_motor">
<hardwareInterface>hardware_interface/PositionJointInterface</hardwareInter
face>
    <mechanicalReduction>1</mechanicalReduction>
    </actuator>
    <joint name="${prefix}_joint">
<hardwareInterface>hardware_interface/PositionJointInterface</hardwareInter
face>
    </joint>
  </transmission>

</xacro:macro>
```

Call them in the robot arm file as follows:

```
<xacro:arm_transmission prefix="arm_base"/>
<xacro:arm_transmission prefix="shoulder"/>
<xacro:arm_transmission prefix="bottom_wrist"/>
<xacro:arm_transmission prefix="elbow"/>
<xacro:arm_transmission prefix="top_wrist"/>
```



You can find the `robot_arm_essentials.xacro` file in this book's GitHub repository: https://github.com/PacktPublishing/ROS-Robotics-Projects-SecondEdition/blob/master/chapter_3_ws/src/robot_description/urdf/robot_arm_essentials.xacro.

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

Defining ROS_CONTROLLERS

Finally, we can port the plugins that are needed to establish communication between Gazebo and ROS and more. We add one more controller, `joint_state_publisher` into the already created `gazebo_essentials_arm.xacro` file:

```
<Gazebo>
  <plugin name="joint_state_publisher"
    filename="libgazebo_ros_joint_state_publisher.so">
    <jointName>arm_base_joint, shoulder_joint, bottom_wrist_joint,
    elbow_joint, bottom_wrist_joint</jointName>
  </plugin>
</Gazebo>
```

The `joint_state_publisher` controller (http://wiki.ros.org/joint_state_publisher) is used to publish the robot arm link's state information in space.



You can find the `gazebo_essentials_arm.xacro` file at our repository: https://github.com/PacktPublishing/ROS-Robotics-Projects-SecondEdition/blob/master/chapter_3_ws/src/robot_description/urdf/gazebo_essentials_arm.xacro.

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

```
<xacro:include filename="$(find
robot_description)/urdf/robot_arm_essentials.xacro" />
  <xacro:include filename="$(find
robot_description)/urdf/gazebo_essentials_arm.xacro" />
```

Let's create an `arm_control.yaml` file and define the arm controller's configurations:

```
$ cd ~/chapter3_ws/src/robot_description/config/
$ gedit arm_control.yaml'
```

Now, copy the following code into the file:

```
arm_controller:
  type: position_controllers/JointTrajectoryController
  joints:
    - arm_base_joint
    - shoulder_joint
    - bottom_wrist_joint
    - elbow_joint
    - top_wrist_joint
  constraints:
```

```

    goal_time: 0.6
    stopped_velocity_tolerance: 0.05
    hip: {trajectory: 0.1, goal: 0.1}
    shoulder: {trajectory: 0.1, goal: 0.1}
    elbow: {trajectory: 0.1, goal: 0.1}
    wrist: {trajectory: 0.1, goal: 0.1}
    stop_trajectory_duration: 0.5
    state_publish_rate: 25
    action_monitor_rate: 10
/gazebo_ros_control:
  pid_gains:
    arm_base_joint: {p: 100.0, i: 0.0, d: 0.0}
    shoulder_joint: {p: 100.0, i: 0.0, d: 0.0}
    bottom_wrist_joint: {p: 100.0, i: 0.0, d: 0.0}
    elbow_joint: {p: 100.0, i: 0.0, d: 0.0}
    top_wrist_joint: {p: 100.0, i: 0.0, d: 0.0}

```

Now that we have completed our `robot_base` model, let's test it in Gazebo.

Testing the robot arm

Have a look at the robot URDF file in this book's GitHub repository, https://github.com/PacktPublishing/ROS-Robotics-Projects-SecondEdition/blob/master/chapter_3_ws/src/robot_description/urdf/robot_arm.urdf.xacro, into testing the robot base.

Now that we have the complete model of our robot arm, let's get it into action and see how our arm moves. Follow the steps:

1. Create a launch file that would spawn the robot arm and its controllers. Now, go into the `launch` folder and create the following launch file:

```

$ cd ~/chapter3_ws/src/robot_description/launch
$ gedit arm_gazebo_control_xacro.launch

```

Now, copy the following code into it:

```

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

<param name="robot_description" command="$(find xacro)/xacro --
inorder $(find robot_description)/urdf/robot_arm.urdf.xacro" />
  <include file="$(find gazebo_ros)/launch/empty_world.launch"/>
    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
args="-param robot_description -urdf -model robot_arm" />
    <rosparam command="load" file="$(find
robot_description)/config/arm_control.yaml" />

```

```

    <node name="arm_controller_spawner" pkg="controller_manager"
    type="controller_manager" args="spawn arm_controller"
    respawn="false" output="screen"/>
    <rosparam command="load" file="$(find
    robot_description)/config/joint_state_controller.yaml" />
    <node name="joint_state_controller_spawner"
    pkg="controller_manager" type="controller_manager" args="spawn
    joint_state_controller" respawn="false" output="screen"/>
    <node name="robot_state_publisher" pkg="robot_state_publisher"
    type="robot_state_publisher" respawn="false" output="screen"/>

</launch>

```

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

```

$ initros1
$ roslaunch robot_description arm_gazebo_control_xacro.launch

```

You could see the necessary ROS topics by opening another Terminal and running `rostopic list`:

```

$ initros1
$ rostopic list

```

3. Try to move the arm using the following command:

```

$ 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]}]}'

```

You will learn how to move the arm without using publishing values in the upcoming chapters. We'll learn how to set up the mobile manipulator in the next section.

Putting things together

You have now created a robot base and robot arm successfully and have simulated them in Gazebo. Now we're just one step away from getting our mobile manipulator model.

Modeling the mobile manipulator

The use of `xacro` will help us connect both with ease. Before creating the final URDF, let's understand something that we saw in the *Successful Gazebo launch* screenshot. The arm needs to connect to the base, and that's our goal. Hence, all you need to do now is connect the robot arm's `arm_base` link to the robot base's `base_link`. Create a file called `mobile_manipulator.urdf.xacro` and copy the following code into it:

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

<robot xmlns:xacro="http://ros.org/wiki/xacro" name="robot_base" >

  <xacro:include filename="$(find
robot_description)/urdf/robot_base.urdf.xacro" />
  <xacro:include filename="$(find
robot_description)/urdf/robot_arm.urdf.xacro" />

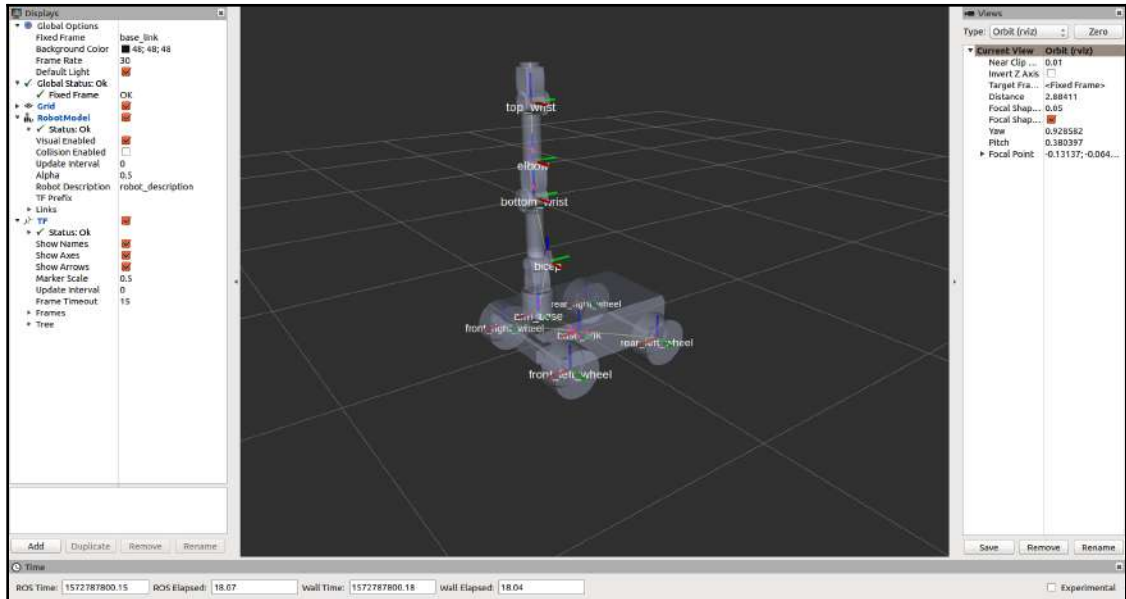
  <xacro:arm_joint prefix="arm_base_link" parent="base_link" child="arm_base"
originxyz="0.0 0.0 0.1" originrpy="0 0 0"/>

</robot>
```

You could view the model the same way you viewed in `rviz` for the robot base and robot arm using the following command:

```
$ cd ~/chapter_3_ws/
$ source devel/setup.bash
$ roslaunch urdf_tutorial display.launch model:=mobile_manipulator.urdf
```


The rviz display will look as follows:



The rviz model view of the mobile manipulator

Now, let's simulate the model.

Simulating and testing the mobile manipulator

Let's create the following launch file:

```
$ cd ~/chapter3_ws/src/robot_description/launch
$ gedit mobile_manipulator_gazebo_control_xacro.launch
```

Now, copy the following launch file:

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

<param name="robot_description" command="$(find xacro)/xacro --inorder
$(find robot_description)/urdf/mobile_manipulator.urdf" />
<include file="$(find gazebo_ros)/launch/empty_world.launch"/>
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param
robot_description -urdf -model mobile_manipulator" />
<rosparam command="load" file="$(find
```

```

robot_description)/config/arm_control.yaml" />
  <node name="arm_controller_spawner" pkg="controller_manager"
type="controller_manager" args="spawn arm_controller" respawn="false"
output="screen"/>
  <rosparam command="load" file="$(find
robot_description)/config/joint_state_controller.yaml" />
  <node name="joint_state_controller_spawner" pkg="controller_manager"
type="controller_manager" args="spawn joint_state_controller"
respawn="false" output="screen"/>
  <rosparam command="load" file="$(find
robot_description)/config/control.yaml" />
  <node name="base_controller_spawner" pkg="controller_manager"
type="spawner" args="robot_base_joint_publisher
robot_base_velocity_controller"/>
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" respawn="false" output="screen"/>

</launch>

```

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

```

$ initros1
$ cd ~/chapter_3_ws
$ source devel/setup.bash
$ roslaunch robot_description mobile_manipulator_gazebo_xacro.launch

```

You can now try moving the base and arm, as shown in their respective sections. Some might be happy with the way the robot has been imported and is moving, but some might not be. Obviously, they're shaking, they're jerky, and they don't intend to look great in motion. This is simply because the controllers haven't been tuned properly.

Summary

In this chapter, we understood how a robot can be defined, modeled, and simulated in ROS with the help of Gazebo. We defined the robot's physical characteristics through plugins that were available in Gazebo. We began with creating a robot base, created a five DOF robot arm, and finally, put them together to create a mobile manipulator. We simulated the robot in Gazebo to understand how a robot application could be defined in ROS. This chapter helped us understand the use of robots in industries.

With this understanding, let's move on to the next chapter, where we will learn how the robot can handle complex tasks. Then, we will learn how to create a robot application in ROS and simulate it in Gazebo.