4
Simulating Robots Using ROS and Gazebo

After designing the 3D model of a robot, the next phase is its simulation. Robot simulation will give you an idea about the working of robots in a virtual environment.

We are going to use the Gazebo (`http://www.gazebosim.org/`) simulator to simulate the seven DOF arms and the mobile robot.

Gazebo is a multi-robot simulator for complex indoor and outdoor robotic simulation. We can simulate complex robots, robot sensors, and a variety of 3D objects. Gazebo already has simulation models of popular robots, sensors, and a variety of 3D objects in their repository (`https://bitbucket.org/osrf/gazebo_models/`). We can directly use these models without having to create new ones.

Gazebo has a good interface in ROS, which exposes the whole control of Gazebo in ROS. We can install Gazebo without ROS, and we should install the ROS-Gazebo interface to communicate from ROS to Gazebo.

In this chapter, we will discuss the simulation of seven DOF arms and differential wheeled robots. We will discuss ROS controllers that help to control the robot's joints in Gazebo.

We will cover the following topics in this chapter:

- Understanding robotic simulation and Gazebo
- Simulation a model of a robotic arm for Gazebo
- Simulating the robotic arm with an *rgb-d* sensor
- Moving robot joints using ROS controllers in Gazebo
- Simulating a differential wheeled robot in Gazebo
- Teleoperating a mobile robot in Gazebo

Simulating the robotic arm using Gazebo and ROS

In the previous chapter, we designed a seven-DOF arm. In this section, we will simulate the robot in Gazebo using ROS.

Before starting with Gazebo and ROS, we should install the following packages to work with Gazebo and ROS:

```
$ sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-msgs
ros-kinetic-gazebo-plugins ros-kinetic-gazebo-ros-control
```

The default version installed from kinetic ROS packages is Gazebo 7.0. The use of each package is as follows:

- `gazebo_ros_pkgs`: This contains wrappers and tools for interfacing ROS with Gazebo
- `gazebo-msgs`: This contains messages and service data structures for interfacing with Gazebo from ROS
- `gazebo-plugins`: This contains Gazebo plugins for sensors, actuators, and so on.
- `gazebo-ros-control`: This contains standard controllers to communicate between ROS and Gazebo

After installation, check whether Gazebo is properly installed using the following commands:

```
$ roscore & rosrun gazebo_ros gazebo
```

These commands will open the Gazebo GUI. If we have the Gazebo simulator, we can proceed to develop the simulation model of the seven-DOF arm for Gazebo.

Creating the robotic arm simulation model for Gazebo

We can create the simulation model for a robotic arm by updating the existing robot description by adding simulation parameters.

We can create the package needed to simulate the robotic arm using the following command:

```
$ catkin_create_pkg seven_dof_arm_gazebo gazebo_msgs gazebo_plugins
gazebo_ros gazebo_ros_control mastering_ros_robot_description_pkg
```

Alternatively, the full package is available in the following Git repository; you can clone the repository for a reference to implement this package, or you can get the package from the book's source code:

```
$ git clone  https://github.com/jocacace/seven_dof_arm_gazebo.git
```

You can see the complete simulation model of the robot in the `seven_dof_arm.xacro` file, placed in the `mastering_ros_robot_description_pkg/urdf/` folder.

The file is filled with URDF tags, which are necessary for the simulation. We will define the sections of collision, inertial, transmission, joints, links, and Gazebo.

To launch the existing simulation model, we can use the `seven_dof_arm_gazebo` package, which has a launch file called `seven_dof_arm_world.launch`. The file definition is as follows:

```
<launch>

  <!-- these are the arguments you can pass this launch file, for example
paused:=true -->
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <!-- We resume the logic in empty_world.launch -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>

  <!-- Load the URDF into the ROS Parameter Server -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder
'$(find mastering_ros_robot_description_pkg)/urdf/seven_dof_arm.xacro'" />

  <!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen"
  args="-urdf -model seven_dof_arm -param robot_description"/>
</launch>
```

Launch the following command and check what you get:

```
    $ roslaunch seven_dof_arm_gazebo seven_dof_arm_world.launch
```

You can see the robotic arm in Gazebo, as shown in the following figure; if you get this output, without any errors, you are done:

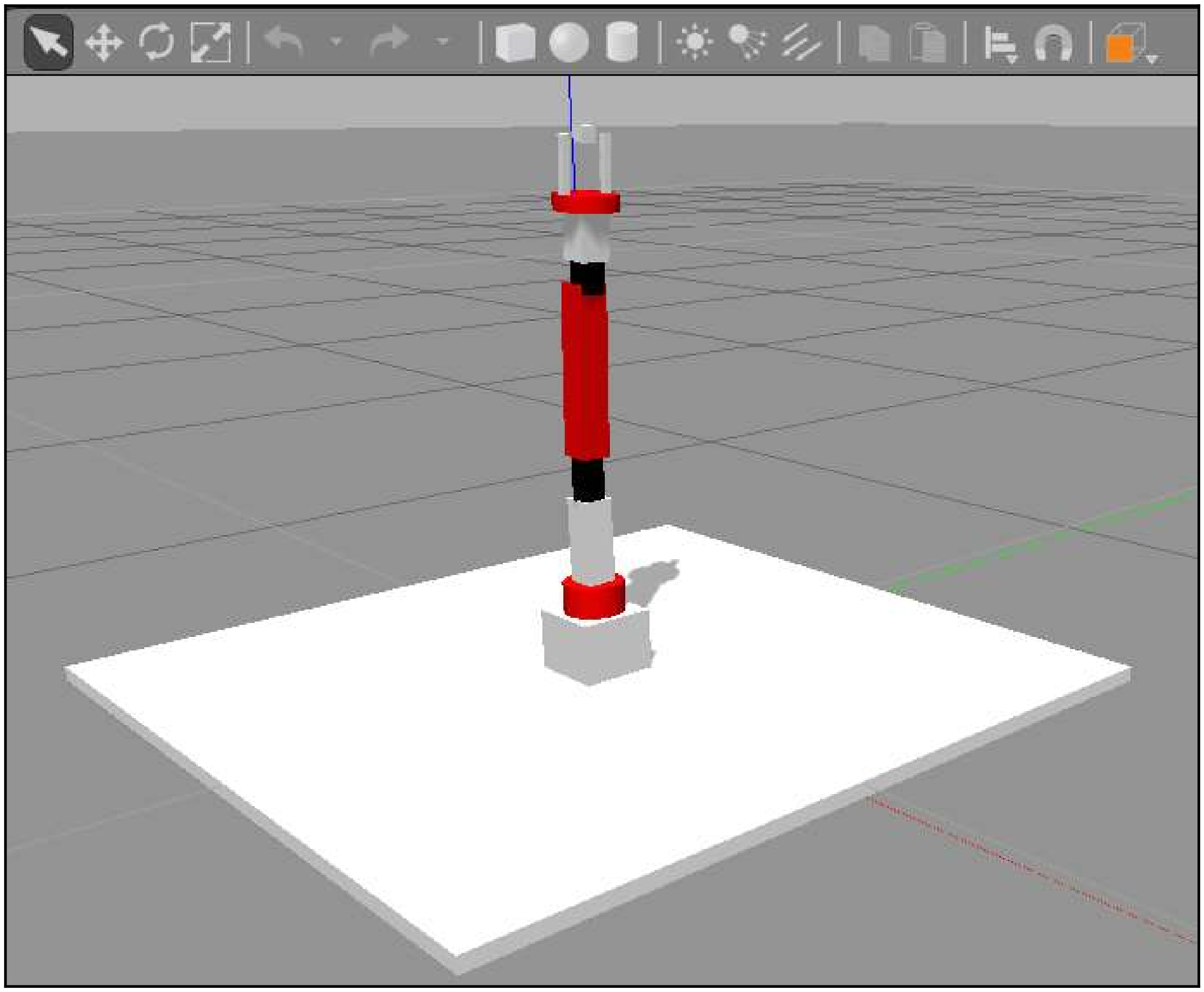


Figure 1: Simulation of seven-DOF arm in Gazebo

Let's discuss the robot simulation model files in detail.

Adding colors and textures to the Gazebo robot model

We can see in the simulated robot that each link has different colors and textures. The following tags inside the xacro file provide textures and colors to robot links:

```
<gazebo reference="bottom_link">
  <material>Gazebo/White</material>
</gazebo>
<gazebo reference="base_link">
  <material>Gazebo/White</material>
</gazebo>
<gazebo reference="shoulder_pan_link">
  <material>Gazebo/Red</material>
</gazebo>
```

Adding transmission tags to actuate the model

To actuate the robot using ROS controllers, we should define the `<transmission>` element to link actuators to joints. Here is the macro defined for transmission:

```
   <xacro:macro name="transmission_block" params="joint_name">
    <transmission name="tran1">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="${joint_name}">
<hardwareInterface>hardware_interface/PositionJointInterface</hardwareInter
face>
      </joint>
      <actuator name="motor1">
       <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>
   </xacro:macro>
```

Here, `<joint name = "">` is the joint in which we link the actuators. The `<type>` element is the type of transmission. Currently, `transmission_interface/SimpleTransmission` is only supported. The `<hardwareInterface>` element is the type of hardware interface to load (position, velocity, or effort interfaces). In the proposed example, a position control hardware interface has been used. The hardware interface is loaded by the `gazebo_ros_control` plugin; we look at this plugin in the next section.

Adding the gazebo_ros_control plugin

After adding the transmission tags, we should add the `gazebo_ros_control` plugin in the simulation model to parse the transmission tags and assign appropriate hardware interfaces and the control manager. The following code adds the `gazebo_ros_control` plugin to the xacro file:

```
<!-- ros_control plugin -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/seven_dof_arm</robotNamespace>
  </plugin>
</gazebo>
```

Here, the `<plugin>` element specifies the plugin name to be loaded, which is `libgazebo_ros_control.so`. The `<robotNamespace>` element can be given as the name of the robot; if we are not specifying the name, it will automatically load the name of the robot from the URDF. We can also specify the controller update rate (`<controlPeriod>`), the location of `robot_description` (URDF) on the parameter server (`<robotParam>`), and the type of robot hardware interface (`<robotSimType>`). The default hardware interfaces are `JointStateInterface`, `EffortJointInterface`, and `VelocityJointInterface`.

Adding a 3D vision sensor to Gazebo

In Gazebo, we can simulate the robot movement and its physics; we can also simulate sensors too.

To build a sensor in Gazebo, we must model the behavior of that sensor in Gazebo. There are some prebuilt sensor models in Gazebo that can be used directly in our code without writing a new model.

Here, we are adding a 3D vision sensor (commonly known as an *rgb-d* sensor) called the Asus Xtion Pro model in Gazebo. The sensor model is already implemented in the `gazebo_ros_pkgs/gazebo_plugins` ROS package, which we have already installed in our ROS system.

Each model in Gazebo is implemented as a Gazebo-ROS plugin, which can be loaded by inserting it into the URDF file.

Here is how we include a Gazebo definition and a physical robot model of Xtion Pro in the `seven_dof_arm_with_rgbd.xacro` robot xacro file:

```
<xacro:include filename="$(find
mastering_ros_robot_description_pkg)/urdf/sensors/xtion_pro_live.urdf.xacro
"/>
```

Inside `xtion_pro_live.urdf.xacro`, we can see the following lines:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="$(find
mastering_ros_robot_description_pkg)/urdf/sensors/xtion_pro_live.gazebo.xac
ro"/>
...................
  <xacro:macro name="xtion_pro_live" params="name parent *origin
*optical_origin">
...................
    <link name="${name}_link">
       ......................
  <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
          <mesh
filename="package://mastering_ros_robot_description_pkg/meshes/sensors/xtio
n_pro_live/xtion_pro_live.dae"/>
        </geometry>
        <material name="DarkGrey"/>
    </visual>
    </link>

</robot>
```

Here, we can see it includes another file called `xtion_pro_live.gazebo.xacro`, which consists of the complete Gazebo definition of Xtion Pro.

We can also see a macro definition named `xtion_pro_live`, which contains the complete model definition of Xtion Pro, including links and joints:

```
<mesh
filename="package://mastering_ros_robot_description_pkg/meshes/sensors/xtio
n_pro_live/xtion_pro_live.dae"/>
```

In the macro definition, we are importing a mesh file of the Asus Xtion Pro, which will be shown as the camera link in Gazebo.

In the `mastering_ros_robot_description_pkg/urdf/sensors/xtion_pro_live.gazebo.xacro` file, we can set the Gazebo-ROS plugin of Xtion Pro. Here, we will define the plugin as macro with RGB and depth camera support. Here is the plugin definition:

```
          <plugin name="${name}_frame_controller"
filename="libgazebo_ros_openni_kinect.so">
          <alwaysOn>true</alwaysOn>
          <updateRate>6.0</updateRate>
          <cameraName>${name}</cameraName>
          <imageTopicName>rgb/image_raw</imageTopicName>

        </plugin>
```

The plugin filename of Xtion Pro is `libgazebo_ros_openni_kinect.so`, and we can define the plugin parameters, such as the camera name, image topics, and so on.

Simulating the robotic arm with Xtion Pro

Now that we have learned about the camera plugin definition in Gazebo, we can launch the complete simulation using the following command:

```
$ roslaunch seven_dof_arm_gazebo seven_dof_arm_with_rgbd_world.launch
```

We can see the robot model with a sensor on the top of the arm, as shown here:

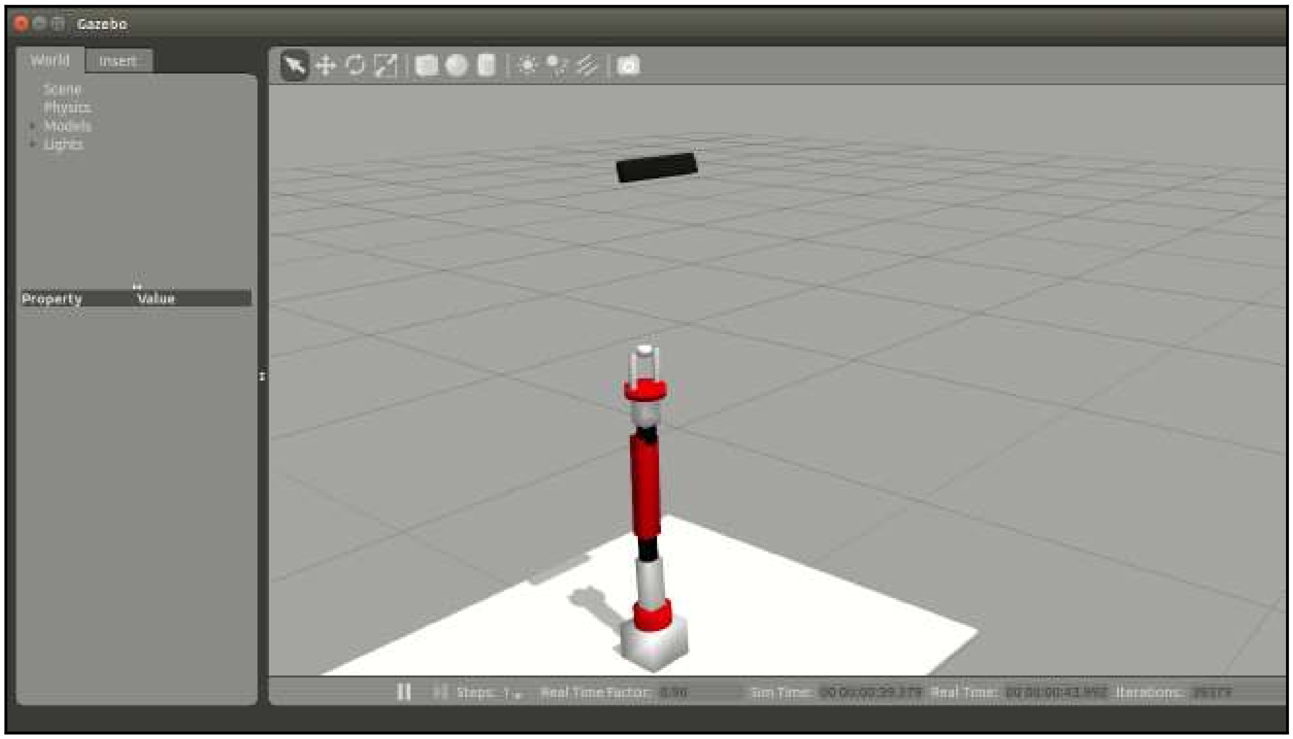
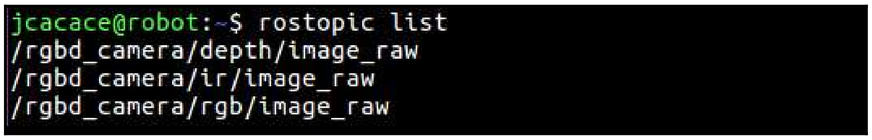


Figure 2: Simulation of seven-DOF arm with Asus Xtion Pro in Gazebo

We can now work with the simulated *rgb-d* sensor as if it were directly plugged into our computer. So we can check whether it provides the correct image output.

Visualizing the 3D sensor data

After launching the simulation using the preceding command, we can check topics generated by the sensor plugin:



```
jcacace@robot:~$ rostopic list
/rgbd_camera/depth/image_raw
/rgbd_camera/ir/image_raw
/rgbd_camera/rgb/image_raw
```

Figure 3: *rgb-d* image topics generated by Gazebo

Let's view the image data of a 3D vision sensor using the following tool called `image_view`:

- View the RGB raw image:

```
$ rosrun image_view image_view image:=/rgbd_camera/rgb/image_raw
```

- View the IR raw image:

```
$ rosrun image_view image_view image:=/rgbd_camera/ir/image_raw
```

- View the depth image:

```
$ rosrun image_view image_view image:=/rgbd_camera/depth/image_raw
```

Here is the screenshot with all these images:

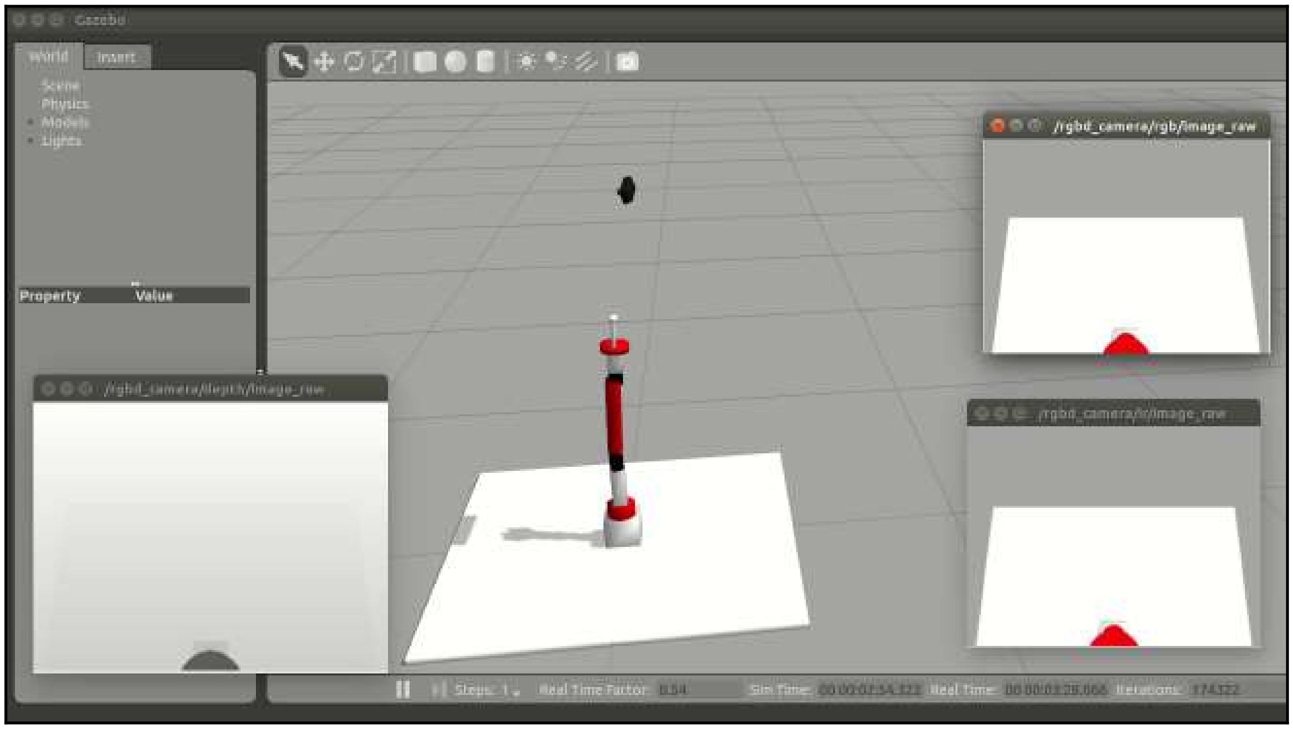


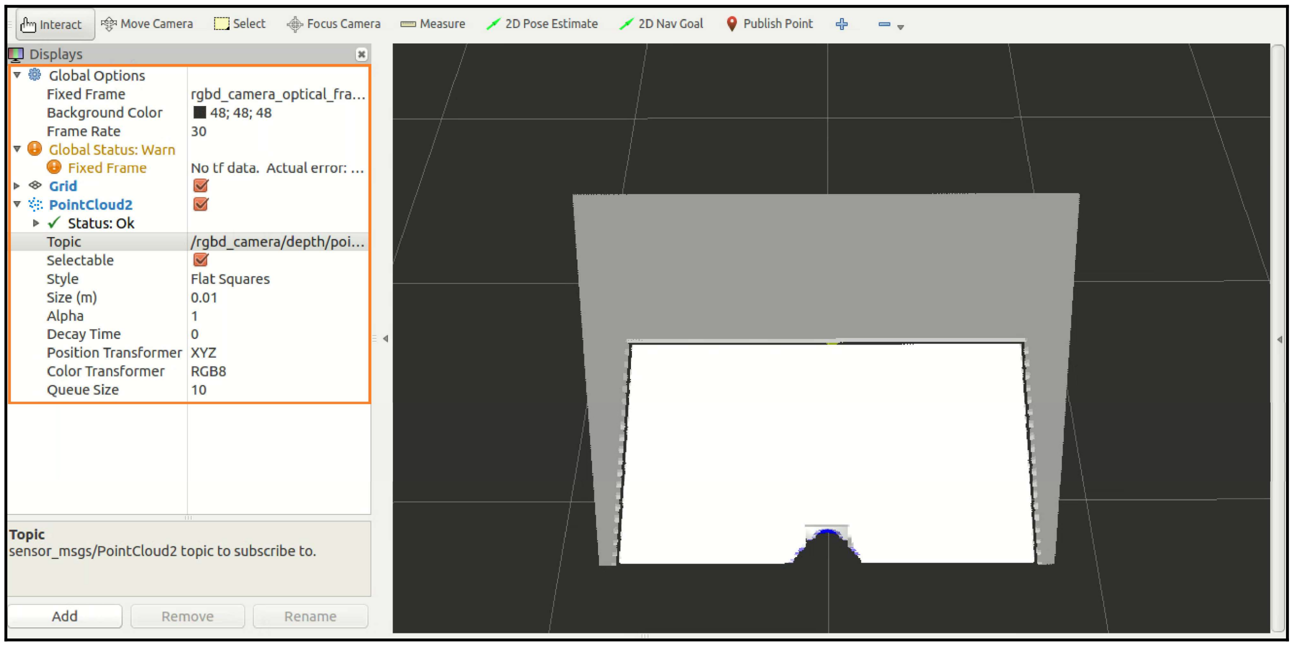
Figure 4: Viewing images of the *rgb-d* sensor in Gazebo

We can also view the point cloud data of this sensor in RViz.

Launch RViz using the following command:

```
$ rosrun rviz rviz -f /rgbd_camera_optical_frame
```

Add a **PointCloud2** display type and set the **Topic** as `/rgbd_camera/depth/points`. We will get a point cloud view as follows:



```
Global Options
    Fixed Frame           rgbd_camera_optical_fra...
    Background Color      ■ 48; 48; 48
    Frame Rate            30
Global Status: Warn
    Fixed Frame           No tf data.  Actual error: ...
Grid                      ✓
PointCloud2               ✓
    Status: Ok
    Topic                 /rgbd_camera/depth/poi...
    Selectable            ✓
    Style                 Flat Squares
    Size (m)              0.01
    Alpha                 1
    Decay Time            0
    Position Transformer  XYZ
    Color Transformer     RGB8
    Queue Size            10

Topic
sensor_msgs/PointCloud2 topic to subscribe to.
```

Figure 5: Viewing point cloud data from an *rgb-d* sesor in RViz

Moving robot joints using ROS controllers in Gazebo

In this section, we are going to discuss how to move each joint of the robot in Gazebo.

To move each joint, we need to assign an ROS controller. In particular, for each joint we need to attach a controller that is compatible with the hardware interface mentioned inside the `transmission` tags.

An ROS controller mainly consists of a feedback mechanism that can receive a set point and control the output using the feedback from the actuators.

The ROS controller interacts with the hardware using the hardware interface. The main function of the hardware interface is to act as a mediator between ROS controllers and the real or simulated hardware, allocating the resources to control it considering the data generated by the ROS controller.

In this robot, we have defined the position controllers, velocity controllers, effort controllers, and so on. The ROS controllers are provided by a set of packages called `ros_control`.

For a proper understanding of how to configure ROS controllers for the arm, we should understand its concepts. We will discuss more on the `ros_control` packages, different types of ROS controllers, and how an ROS controller interacts with the Gazebo simulation.

Understanding the ros_control packages

The `ros_control` packages have the implementation of robot controllers, controller managers, hardware interfaces, different transmission interfaces, and control toolboxes. The `ros_controls` packages are composed of the following individual packages:

- `control_toolbox`: This package contains common modules (PID and Sine) that can be used by all controllers
- `controller_interface`: This package contains the `interface` base class for controllers
- `controller_manager`: This package provides the infrastructure to `load`, `unload`, `start`, and `stop` controllers
- `controller_manager_msgs`: This package provides the message and service definition for the controller manager
- `hardware_interface`: This contains the base class for the hardware interfaces
- `transmission_interface`: This package contains the interface classes for the `transmission` interface (differential, four bar linkage, joint state, position, and velocity)

Different types of ROS controllers and hardware interfaces

Let's see the list of ROS packages that contain the standard ROS controllers:

- `joint_position_controller`: This is a simple implementation of the joint position controller
- `joint_state_controller`: This is a controller to publish joint states
- `joint_effort_controller`: This is an implementation of the joint effort (force) controller

The following are some of the commonly used hardware interfaces in ROS:

- `Joint Command Interfaces`: This will send the commands to the hardware
- `Effort Joint Interface`: This will send the `effort` command
- `Velocity Joint Interface`: This will send the `velocity` command
- `Position Joint Interface`: This will send the `position` command
- `Joint State Interfaces`: This will retrieve the joint states from the actuators encoder

How the ROS controller interacts with Gazebo

Let's see how an ROS controller interacts with Gazebo. The following figure shows the interconnection of the ROS controller, robot hardware interface, and simulator/real hardware:

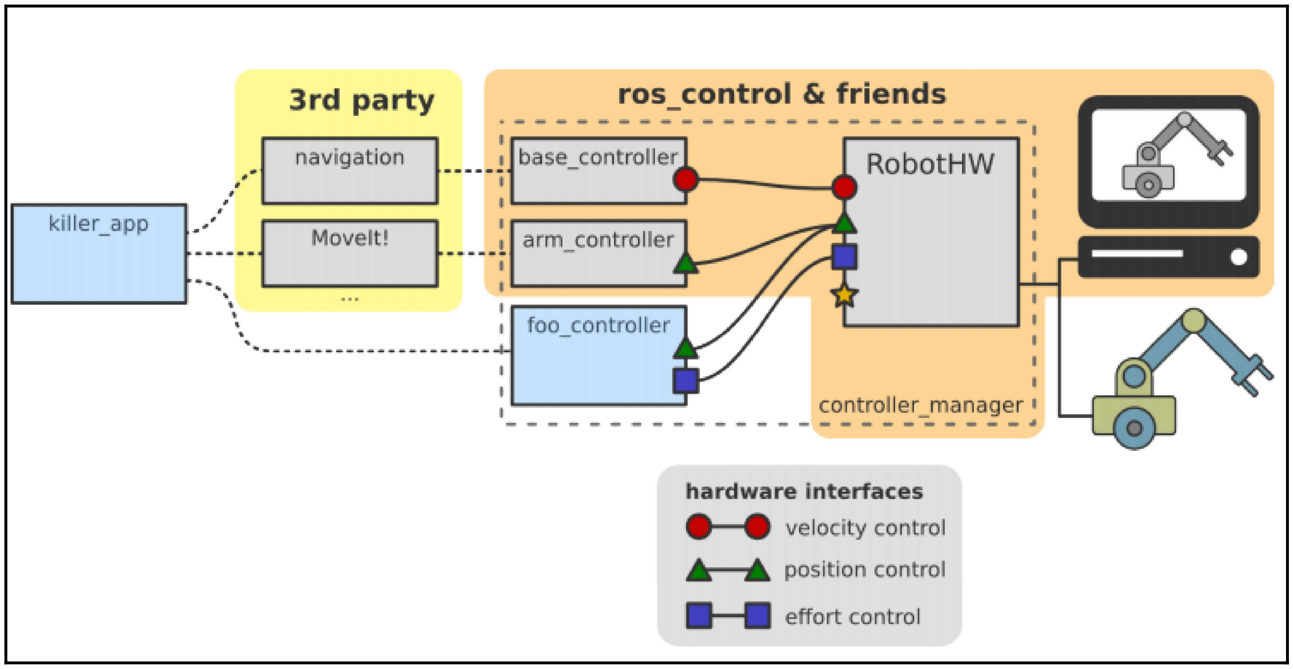


Figure 6: Interacting ROS controllers with Gazebo

We can see the third-party tools, the `navigation` and `MoveIt` packages. These packages can give the goal (set point) to the mobile robot controllers and robotic arm controllers. These controllers can send the position, velocity, or effort to the robot hardware interface.

The hardware interface allocates each resource to the controllers and sends values to each resource. The communications between the robot controllers and robot hardware interfaces are shown in the following diagram:

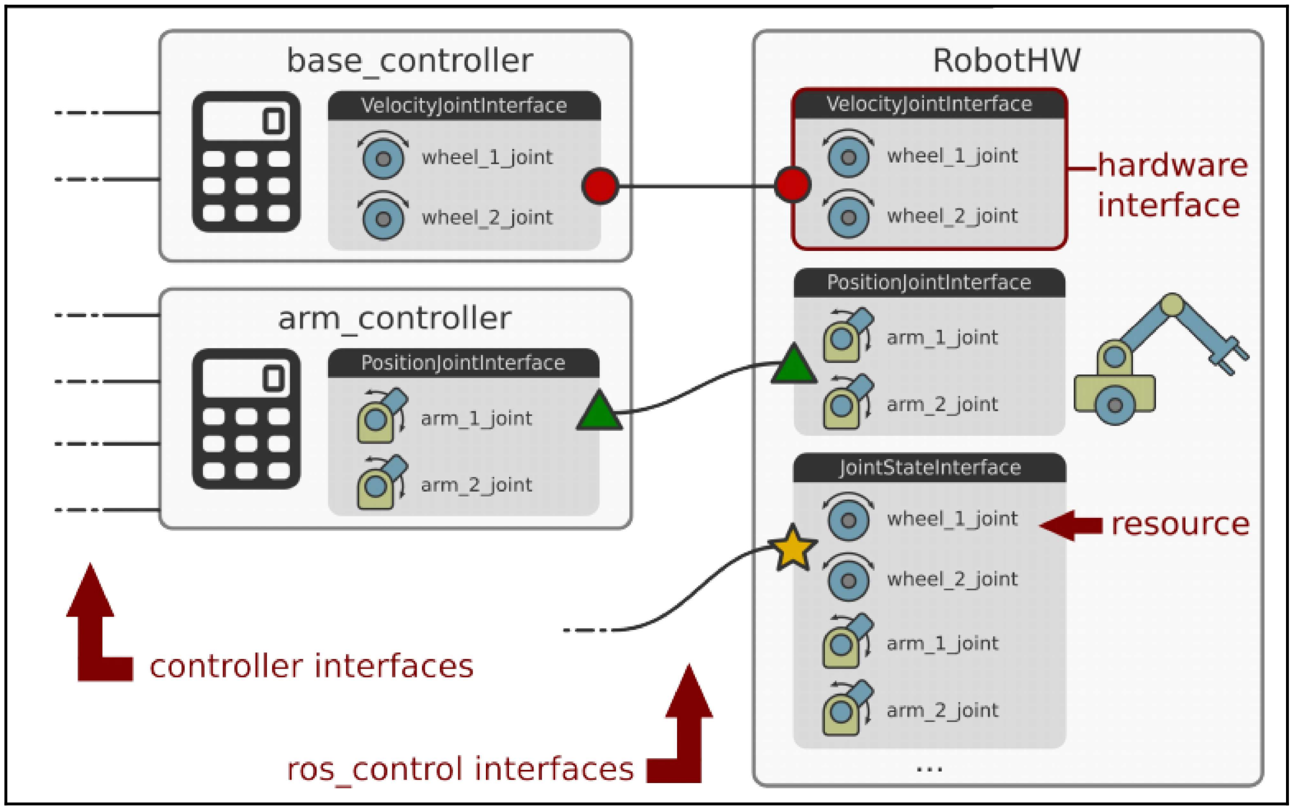


Figure 7: Illustration of ROS controllers and hardware interfaces.

The hardware interface is decoupled from actual hardware and simulation. The values from the hardware interface can be fed to Gazebo for simulation or to the actual hardware itself.

The hardware interface is a software representation of the robot and its abstract hardware. The resource of the hardware interfaces are actuators, joints, and sensors. Some resources are read-only, such as joint states, IMU, force-torque sensors, and so on, and some are read and write compatible, such as position, velocity, and effort joints.

Interfacing joint state controllers and joint position controllers to the arm

Interfacing robot controllers to each joint is a simple task. The first task is to write a configuration file for two controllers.

The joint state controllers will publish the joint states of the arm and the joint position controllers can receive a goal position for each joint and can move each joint.

We will find the configuration file for the controller at `seven_dof_arm_gazebo_control.yaml` in the `seven_dof_arm_gazebo/config` folder.

Here is the configuration file definition:

```
seven_dof_arm:
  # Publish all joint states -----------------------------------
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50
  # Position Controllers ---------------------------------------
  joint1_position_controller:
    type: position_controllers/JointPositionController
    joint: shoulder_pan_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint2_position_controller:
    type: position_controllers/JointPositionController
    joint: shoulder_pitch_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint3_position_controller:
    type: position_controllers/JointPositionController
    joint: elbow_roll_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint4_position_controller:
    type: position_controllers/JointPositionController
    joint: elbow_pitch_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint5_position_controller:
    type: position_controllers/JointPositionController
    joint: wrist_roll_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint6_position_controller:
    type: position_controllers/JointPositionController
    joint: wrist_pitch_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint7_position_controller:
    type: position_controllers/JointPositionController
    joint: gripper_roll_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
```

We can see that all the controllers are inside the namespace `seven_dof_arm`, and the first line represents the joint state controllers, which will publish the joint state of the robot at the rate of 50 Hz.

The remaining controllers are joint position controllers, which are assigned to the first seven joints, and they also define the PID gains.

Launching the ROS controllers with Gazebo

If the controller configuration is ready, we can build a launch file that starts all the controllers along with the Gazebo simulation. Navigate to the `seven_dof_arm_gazebo/launch` directory and open the `seven_dof_arm_gazebo_control.launch` file:

```
<launch>
  <!-- Launch Gazebo  -->
  <include file="$(find
seven_dof_arm_gazebo)/launch/seven_dof_arm_world.launch" />


  <!-- Load joint controller configurations from YAML file to parameter
server -->
  <rosparam file="$(find
seven_dof_arm_gazebo)/config/seven_dof_arm_gazebo_control.yaml"
command="load"/>


  <!-- load the controllers -->
  <node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false"
  output="screen" ns="/seven_dof_arm" args="joint_state_controller
            joint1_position_controller
            joint2_position_controller
            joint3_position_controller
            joint4_position_controller
            joint5_position_controller
            joint6_position_controller
            joint7_position_controller"/>


  <!-- convert joint states to TF transforms for rviz, etc -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
```

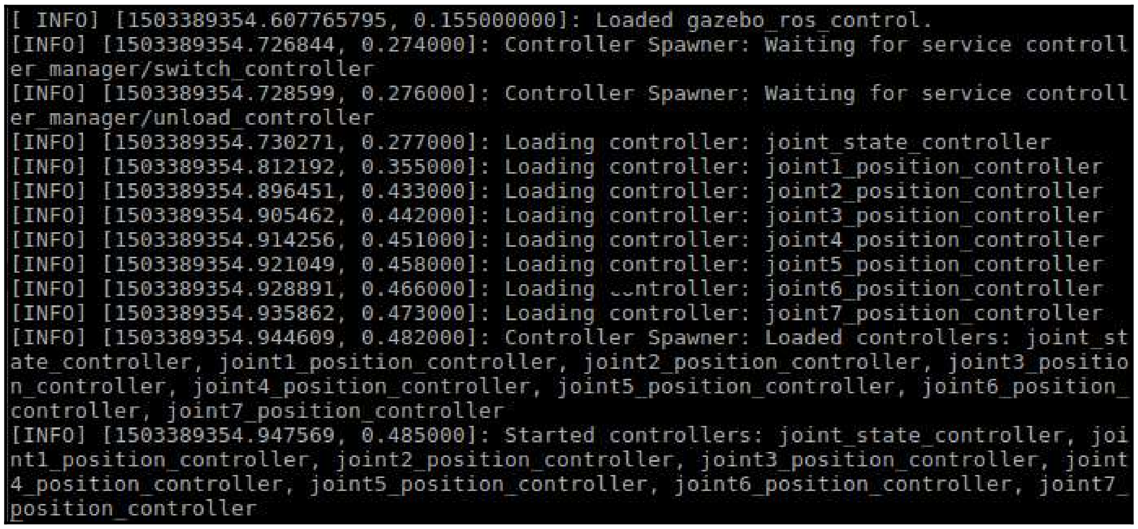
```
type="robot_state_publisher"
  respawn="false" output="screen">
    <remap from="/joint_states" to="/seven_dof_arm/joint_states" />
  </node>

</launch>
```

The launch files start the Gazebo simulation of the arm, load the controller configuration, load the joint state controller and joint position controllers, and, finally, run the robot state publisher, which publishes the joint states and TF.

Let's check the controller topics generated after running this launch file:

```
$ roslaunch seven_dof_arm_gazebo seven_dof_arm_gazebo_control.launch
```

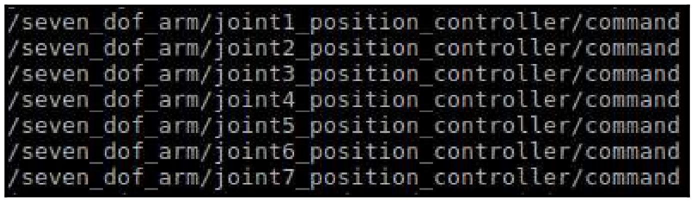
If the command is successful, we can see these messages in the Terminal:



```
[ INFO] [1503389354.607765795, 0.155000000]: Loaded gazebo_ros_control.
[INFO] [1503389354.726844, 0.274000]: Controller Spawner: Waiting for service controll
er_manager/switch_controller
[INFO] [1503389354.728599, 0.276000]: Controller Spawner: Waiting for service controll
er_manager/unload_controller
[INFO] [1503389354.730271, 0.277000]: Loading controller: joint_state_controller
[INFO] [1503389354.812192, 0.355000]: Loading controller: joint1_position_controller
[INFO] [1503389354.896451, 0.433000]: Loading controller: joint2_position_controller
[INFO] [1503389354.905462, 0.442000]: Loading controller: joint3_position_controller
[INFO] [1503389354.914256, 0.451000]: Loading controller: joint4_position_controller
[INFO] [1503389354.921049, 0.458000]: Loading controller: joint5_position_controller
[INFO] [1503389354.928891, 0.466000]: Loading controller: joint6_position_controller
[INFO] [1503389354.935862, 0.473000]: Loading controller: joint7_position_controller
[INFO] [1503389354.944609, 0.482000]: Controller Spawner: Loaded controllers: joint_st
ate_controller, joint1_position_controller, joint2_position_controller, joint3_positio
n_controller, joint4_position_controller, joint5_position_controller, joint6_position_
controller, joint7_position_controller
[INFO] [1503389354.947569, 0.485000]: Started controllers: joint_state_controller, joi
nt1_position_controller, joint2_position_controller, joint3_position_controller, joint
4_position_controller, joint5_position_controller, joint6_position_controller, joint7_
position_controller
```

Figure 8: Terminal messages while loading the ROS controllers of seven-DOF arm

Here are the topics generated from the controllers when we run this launch file:



```
/seven_dof_arm/joint1_position_controller/command
/seven_dof_arm/joint2_position_controller/command
/seven_dof_arm/joint3_position_controller/command
/seven_dof_arm/joint4_position_controller/command
/seven_dof_arm/joint5_position_controller/command
/seven_dof_arm/joint6_position_controller/command
/seven_dof_arm/joint7_position_controller/command
```

Figure 9: Position controller command topics generated by the ROS-controllers

Moving the robot joints

After finishing the preceding topics, we can start commanding positions to each joint.

To move a robot joint in Gazebo, we should publish a desired joint value with a message type `std_msgs/Float64` to the joint position controller command topics.

Here is an example of moving the fourth joint to 1.0 radians:

```
$ rostopic pub /seven_dof_arm/joint4_position_controller/command
std_msgs/Float64 1.0
```

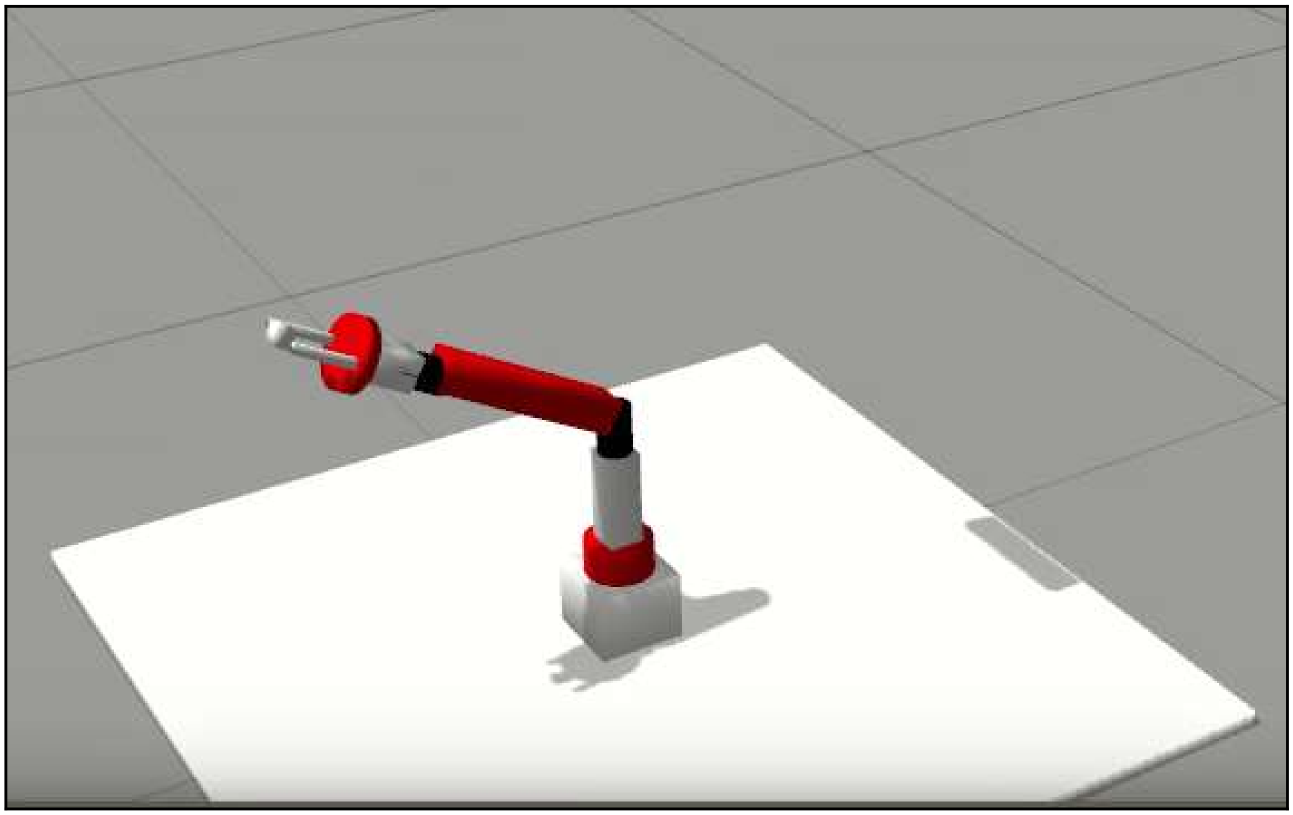


Figure 10: Moving a joint of the arm in Gazebo

We can also view the joint states of the robot by using the following command:

```
$ rostopic echo /seven_dof_arm/joint_states
```

Simulating a differential wheeled robot in Gazebo

We have seen the simulation of the robotic arm. In this section, we can set up the simulation for the differential wheeled robot that we designed in the previous chapter.

You will get the `diff_wheeled_robot.xacro` mobile robot description from the `mastering_ros_robot_description_pkg/urdf` folder.

Let's create a launch file to spawn the simulation model in Gazebo. As we did for the robotic arm, we can create a `ROS` package to launch a Gazebo simulation using the same dependencies of the `seven_dof_arm_gazebo` package, clone the entire package from the following Git repository, or get the package from the book's source code:

```
$ git clone  https://github.com/jocacace/diff_wheeled_robot_gazebo.git
```

Navigate to the `diff_wheeled_robot_gazebo/launch` directory and take the `diff_wheeled_gazebo.launch` file. Here is the definition of this launch:

```
<launch>
  <!-- these are the arguments you can pass this launch file, for example
paused:=true -->
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <!-- We resume the logic in empty_world.launch -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>

  <!-- urdf xml robot description loaded on the Parameter Server-->
  <param name="robot_description" command="$(find xacro)/xacro --inorder
'$(find
mastering_ros_robot_description_pkg)/urdf/diff_wheeled_robot.xacro'" />

  <!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
```

```
respawn="false" output="screen"
  args="-urdf -model diff_wheeled_robot -param robot_description"/>

</launch>
```

To launch this file, we can use the following command:

```
$ roslaunch diff_wheeled_robot_gazebo diff_wheeled_gazebo.launch
```

You will see the following robot model in Gazebo. If you get this model, you have successfully finished the first phase of the simulation:

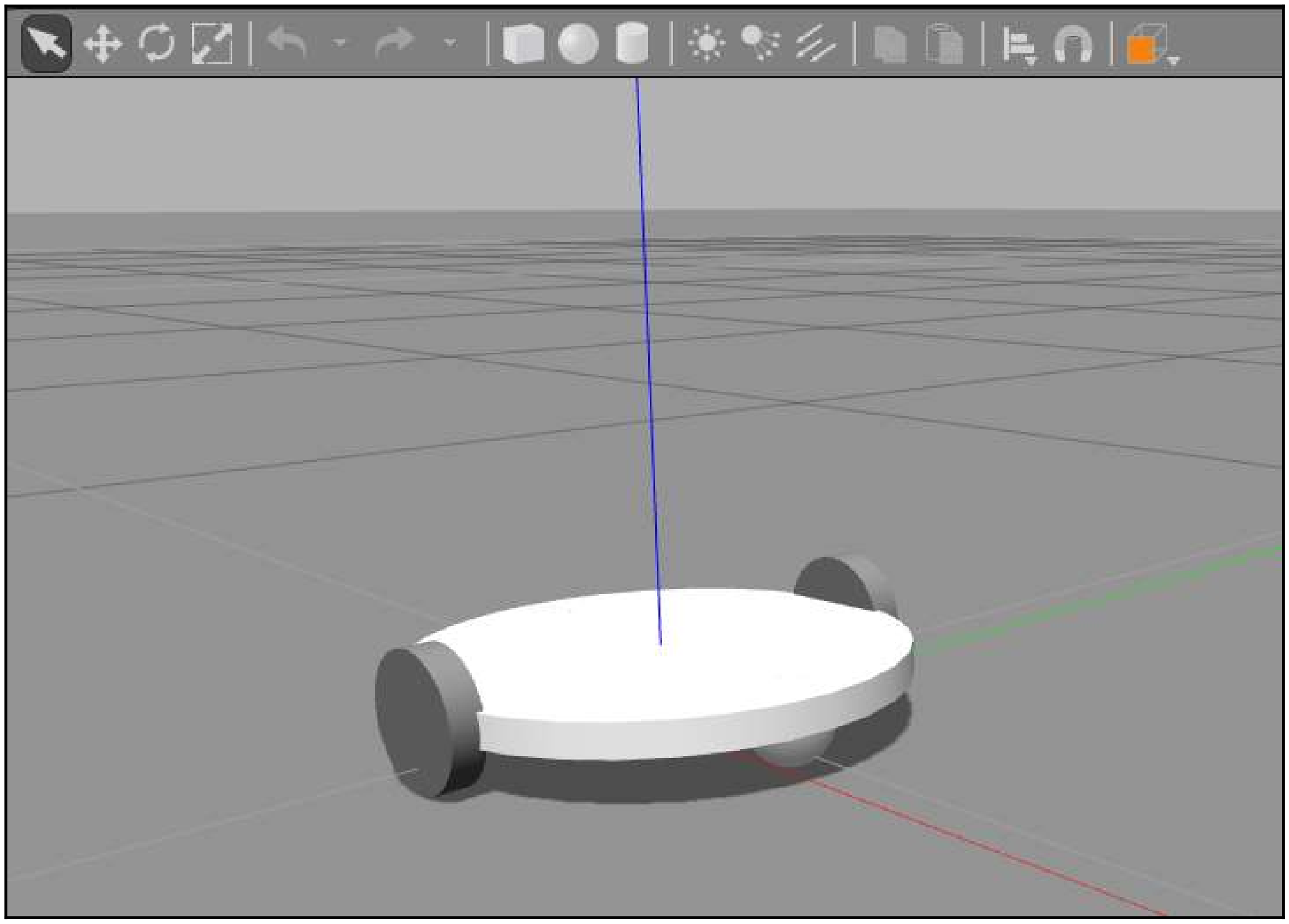


Figure 11: Differential wheeled robot in Gazebo

After successful simulation, let's add the laser scanner to the robot.

Adding the laser scanner to Gazebo

We add the laser scanner on the top of Gazebo to perform high-end operations, such as autonomous navigation or map creation using this robot. Here, we should add the following extra code section to `diff_wheeled_robot.xacro` to add the laser scanner to the robot:

```
    <link name="hokuyo_link">
      <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
          <box size="${hokuyo_size} ${hokuyo_size} ${hokuyo_size}"/>
        </geometry>
        <material name="Blue" />
      </visual>
    </link>
    <joint name="hokuyo_joint" type="fixed">
      <origin xyz="${base_radius - hokuyo_size/2} 0
  ${base_height+hokuyo_size/4}" rpy="0 0 0" />
      <parent link="base_link"/>
      <child link="hokuyo_link" />
    </joint>
    <gazebo reference="hokuyo_link">
      <material>Gazebo/Blue</material>
      <turnGravityOff>false</turnGravityOff>
      <sensor type="ray" name="head_hokuyo_sensor">
        <pose>${hokuyo_size/2} 0 0 0 0 0</pose>
        <visualize>false</visualize>
        <update_rate>40</update_rate>
        <ray>
          <scan>
            <horizontal>
              <samples>720</samples>
              <resolution>1</resolution>
              <min_angle>-1.570796</min_angle>
              <max_angle>1.570796</max_angle>
            </horizontal>
          </scan>
          <range>
            <min>0.10</min>
            <max>10.0</max>
            <resolution>0.001</resolution>
          </range>
        </ray>
        <plugin name="gazebo_ros_head_hokuyo_controller"
  filename="libgazebo_ros_laser.so">
          <topicName>/scan</topicName>
          <frameName>hokuyo_link</frameName>
        </plugin>
      </sensor>
    </gazebo>
```

In this section, we use the Gazebo ROS plugin file called `libgazebo_ros_laser.so` to simulate the laser scanner. The complete code can be found in the `diff_wheeled_robot_with_laser.xacro` description file in the `mastering_ros_robot_description_pkg/urdf/` directory.

We can view the laser scanner data by adding some objects in the simulation environment. Here, we add some cylinders around the robot and can see the corresponding laser view in the next section of the figure:

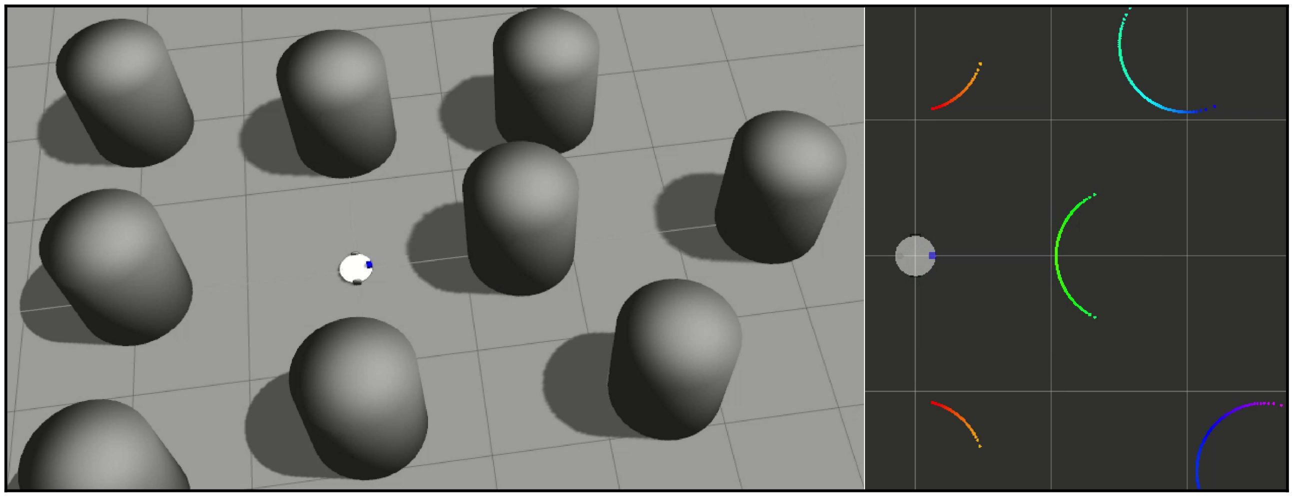


Figure 12: Differential drive robot in random object in Gazebo

The laser scanner plugin publishes laser data (`sensor_msgs/LaserScan`) into the `/scan` topic.

Moving the mobile robot in Gazebo

The robot we are working with is a differential robot with two wheels and two caster wheels. The complete characteristics of the robot should model as the Gazebo-ROS plugin for the simulation. Luckily, the plugin for a basic differential drive is already implemented.

To move the robot in Gazebo, we should add a Gazebo-ROS plugin file called `libgazebo_ros_diff_drive.so` to get the differential drive behavior in this robot.

Here is the complete code snippet of the definition of this plugin and its parameters:

```
    <!-- Differential drive controller  -->
    <gazebo>
      <plugin name="differential_drive_controller" 
  filename="libgazebo_ros_diff_drive.so">

        <rosDebugLevel>Debug</rosDebugLevel>
        <publishWheelTF>false</publishWheelTF>
        <robotNamespace>/</robotNamespace>
        <publishTf>1</publishTf>
        <publishWheelJointState>false</publishWheelJointState>
        <alwaysOn>true</alwaysOn>
        <updateRate>100.0</updateRate>

        <leftJoint>front_left_wheel_joint</leftJoint>
        <rightJoint>front_right_wheel_joint</rightJoint>

        <wheelSeparation>${2*base_radius}</wheelSeparation>
        <wheelDiameter>${2*wheel_radius}</wheelDiameter>
        <broadcastTF>1</broadcastTF>
        <wheelTorque>30</wheelTorque>
        <wheelAcceleration>1.8</wheelAcceleration>
        <commandTopic>cmd_vel</commandTopic>
        <odometryFrame>odom</odometryFrame>
        <odometryTopic>odom</odometryTopic>
        <robotBaseFrame>base_footprint</robotBaseFrame>


      </plugin>
    </gazebo>
```

We can provide the parameters such as the wheel joints of the robot (joints should be of a continuous type), wheel separation, wheel diameters, odometry topic, and so on, in this plugin.

An important parameter that we need to move the robot is:

```
<commandTopic>cmd_vel</commandTopic>
```

This parameter is the command velocity topic to the plugin, which is basically a `Twist` message in ROS (`sensor_msgs/Twist`). We can publish the `Twist` message into the `/cmd_vel` topic, and we can see the robot start to move from its position.

Adding joint state publishers in the launch file

After adding the differential drive plugin, we need to join state publishers to the existing launch file, or we can build a new one. You can see the new final launch file, `diff_wheeled_gazebo_full.launc`, in `diff_wheeled_robot_gazebo/launch`.

The launch file contains joint state publishers, which help to visualize in RViz. Here are the extra lines added in this launch file for the joint state publishing:

```
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
  <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen" >
    <param name="publish_frequency" type="double" value="50.0" />
  </node>
```

Adding the ROS teleop node

The ROS teleop node publishes the ROS `Twist` command by taking keyboard inputs. From this node, we can generate both linear and angular velocity, and there is already a standard teleop node implementation available; we can simply reuse the node.

The teleop is implemented in the `diff_wheeled_robot_control` package. The script folder contains the `diff_wheeled_robot_key` node, which is the teleop node. As per usual, you can get this package from the code provided with the book or download it from the following link:

```
$ git clone  https://github.com/jocacace/diff_wheeled_robot_control.git
```

To successfully compile and use this package, you may need to install the `joy_node` package:

```
$ sudo apt-get install ros-kinetic-joy
```

Here is the launch file called `keyboard_teleop.launch` to start the teleop node:

```
 <launch>
  <!-- differential_teleop_key already has its own built in velocity
smoother -->
  <node pkg="diff_wheeled_robot_control" type="diff_wheeled_robot_key"
name="diff_wheeled_robot_key"  output="screen">

    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="/cmd_vel"/>
  </node>
</launch>
```

Let's start moving the robot.

Launch Gazebo with complete simulation settings, using the following command:

```
$ roslaunch diff_wheeled_robot_gazebo diff_wheeled_gazebo_full.launch
```

Start the teleop node:

```
$ roslaunch diff_wheeled_robot_control keyboard_teleop.launch
```

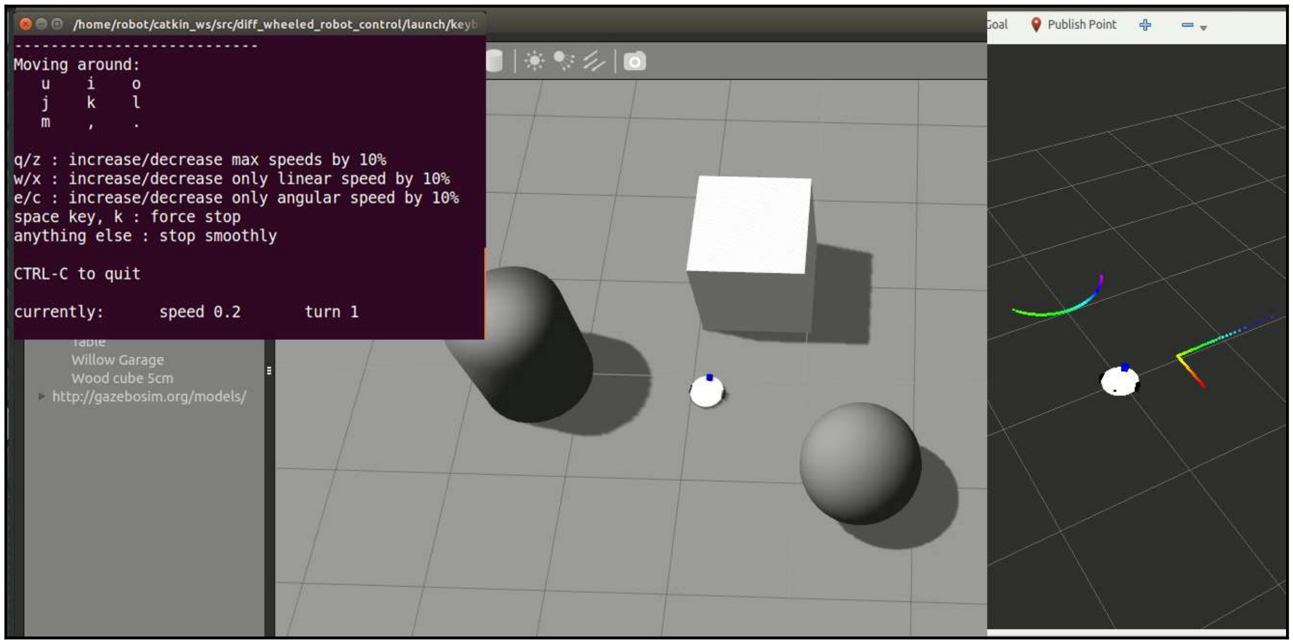
Start RViz to visualize the robot state and laser data:

```
$ rosrun rviz rviz
```

Add `Fixed Frame : /odom`, add `Laser Scan`, and the topic as `/scan` to view the laser scan data, and add the `Robot model` to view the robot model.

In the teleop terminal, we can use some keys (U, I, O, J, K, L, M, ",", ".") for direction adjustment and other keys (q, z, w, x, e, c, k, space key) for speed adjustments. Here is the screenshot showing the robot moving in Gazebo using the teleop and its visualization in RViz.

We can add primitive shapes from the Gazebo toolbar to the robot environment or we can add objects from the online library, which is on the left-side panel:


/home/robot/catkin_ws/src/diff_wheeled_robot_control/launch/keyb
Goal
Publish Point
Table
Willow Garage
Wood cube 5cm
http://gazebosim.org/models/

```
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit

currently:      speed 0.2       turn 1
```

Figure 13: Moving differential drive robot in Gazebo using teleoperation

The robot will only move when we press the appropriate key inside the teleop node terminal. If this terminal is not active, pressing the key will not move the robot. If everything works well, we can explore the area using the robot and visualizing the laser data in RViz.

Questions

- Why do we perform robotic simulation?
- How can we add sensors into a Gazebo simulation?
- What are the different types of ROS controllers and hardware interfaces?
- How can we move the mobile robot in a Gazebo simulation?

Summary

After designing the robot, the next phase is its simulation. There are a lot of uses in simulation. We can validate a robot design, and at the same time, we can work with a robot without having its real hardware. There are some situations when we need to work without having a robot hardware. Simulators are useful in all these situations.

In this chapter, we were trying to simulate two robots, one was a robotic arm with seven-DOF and the other was a differential wheeled mobile robot. We started with the robotic arm, and discussed the additional Gazebo tags needed to launch the robot in Gazebo. We discussed how to add a 3D vision sensor to the simulation. Later, we created a launch file to start Gazebo with a robotic arm and discussed how to add controllers to each joint. We added the controllers and worked with each joint.

Like the robotic arm, we created the URDF for the Gazebo simulation and added the necessary Gazebo-ROS plugin for the laser scanner and differential drive mechanism. After completing the simulation model, we launched the simulation using a custom launch file. Finally, we looked at how to move the robot using the teleop node.

We can learn more about the robotic arm and mobile robots, which are supported by ROS, from the following link: `http://wiki.ros.org/Robots`.

In the next chapter, we will see how to simulate robots using another famous robotics simulation: *V-REP*.

5
Simulating Robots Using ROS and V-REP

Having learned how to simulate robots with Gazebo, in this chapter, we will discuss how to use another powerful and famous simulation software: V-REP (Virtual Robot Experimentation Platform, `http://www.coppeliarobotics.com`).

V-REP is a multi-platform robotic simulator developed by Coppelia Robotics. It offers many simulation models of popular industrial and mobile robots ready to be used, and different functionalities that can be easily integrated and combined through a dedicated API. In addition, V-REP can operate with ROS using a communication interface that allows us to control the simulation scene and the robots via topics and services. Like Gazebo, V-REP can be used as a standalone software, while an external plugin must be installed to work with ROS.

In this chapter, we will learn how to set up the V-REP simulator and install the ROS communication bridge, discussing some initial codes to understand how it works. We will show how to interact with V-REP using services and topics and how to import and interface a new robot model using the URDF file. Finally, we will discuss how to interact with popular mobile robots imported from the V-REP model database, enriching it with additional sensors.

We will cover the following topics in this chapter:

- Setting up V-REP with ROS
- Understanding `vrep_plugin`
- Interacting with V-REP using ROS
- Importong a robot model using an URDF file
- Implementing a ROS interface to simulate a robotic arm in V-REP
- Controlling a mobile robot using V-REP
- Adding additional sensors to simulated robots

Setting up V-REP with ROS

Before starting to work with V-REP, we need to install it in our system and compile the ROS packages needed to establish the communication bridge between ROS and the simulation scene. V-REP is a cross-platform software, available for different operating systems, such as Windows, macOS, and Linux. It is developed by *Coppelia Robotics GmbH* and is distributed with both free educational and commercial licenses. Download the last version of the V-REP simulator from the *Coppelia Robotics* download page: `http://www.coppeliarobotics.com/downloads.html`, choosing the Linux version of the V-REP PRO EDU software.

In this chapter, we will refer to the V-REP version `v3.4.0`. You can download this version, if already available on the website, using the following command in any desired directory:

```
    $ wget
http://coppeliarobotics.com/files/V-REP_PRO_EDU_V3_4_0_Linux.tar.gz
```

After completing the download, extract the archive:

```
    $ tar -zxvf V-REP_PRO_EDU_V3_4_0_Linux.tar.gz
```

To easily access V-REP resources, it is convenient to set the `VREP_ROOT` environmental variable that points to the V-REP main folder:

```
    $ echo "export VREP_ROOT=/path/to/v_rep/folder >> ~/.bashrc"
```

V-REP offers the following modes to control simulated robots from external applications:

- **Remote API**: The V-REP remote API is composed of several functions that can be called from external applications developed in C/C++, Python, Lua, or Matlab. The remote API interacts with V-REP over the network, using socket communication. You can integrate the remote API in your C++ or Python nodes, in order to connect ROS with the simulation scene. The list of all remote APIs available in V-REP can be found on the *Coppelia Robotics* website: `http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctions.htm`. To use the remote API, you must implement both client and server sides:
    - **V-REP Client:** The client side resides in the external application. It can be implemented in a ROS node or in a standard program written in one of the supported programming languages.
    - **V-REP Server:** This side is implemented in V-REP scripts and allows the simulator to receive external data to interact with the simulation scene.
- **RosPlugin**: The V-REP RosPlugin implements a high-level abstraction that directly connects the simulated object scene with the ROS communication system. Using this plugin, you can automatically apply subscribed messages and publish topics from scene objects to get information or control simulated robots.
- **RosInterface**: Introduced in the latest versions of V-REP, this interface will substitute the RosPlugin in the future versions. Differently from the RosPlugin, this module duplicates the C++ API functions to allow ROS and V-REP communication.

In this book, we will discuss how to interact with V-REP using the RosPlugin. The first thing to do is to compile the ROS communication bridge. We must add two packages to our ROS workspace: `vrep_common` and `vrep_plugin`. As usual, you can clone the entire package from the following GitHub repository or obtain the package from the book's source code:

```
$ git clone https://github.com/jocacace/vrep_common.git
$ git clone https://github.com/jocacace/vrep_plugin.git
```

Compile the packages with the `catkin_make` command. If everything goes right, the compilation will create the `vrep_plugin` shared library: `libv_repExtRos.so`. This file is located in the `devel/lib/`directory in the ROS workspace. To enable V-REP to use this library, we need to copy it into the main `vrep` folder:

```
$ cp devel/lib/libv_repExtRos.so $VREP_ROOT
```