3
Working with 3D Robot Modeling in ROS

The first phase of robot manufacturing is design and modeling. We can design and model a robot using CAD tools such as AutoCAD, SOLIDWORKS, and Blender. One of the main purposes of robot modeling is simulation.

The robotic simulation tool can check for critical flaws in the robot's design and can confirm that the robot will work before it goes to the manufacturing phase.

The virtual robot model must have all the characteristics of the real hardware. The shape of a robot may or may not look like the actual robot, but it must be abstract, which has all the physical characteristics of the actual robot.

In this chapter, we are going to discuss the designing of two robots. One is a seven **Degrees of Freedom** (**DOF**) manipulator, and the other is a differential drive robot. In the upcoming chapters, we will look at simulation, how to build the real hardware, and interfacing with ROS.

If we are planning to create the 3D model of the robot and simulate it using ROS, you need to learn about some ROS packages that help in robot designing. Creating a model for our robot in ROS is important for different reasons. For example, we can use this model to simulate and control the robot, visualize it, or use ROS tools to get information on the robotic structure and its kinematics.

ROS has a standard meta package for designing and creating robot models called `robot_model`, which consists of a set of packages, some of which are called `urdf`, `kdl_parser`, `robot_state_publisher`, and `collada_urdf`. These packages help us create the 3D robot model description with the exact characteristics of the real hardware.

In this chapter, we will cover the following topics:

- ROS packages for robot modeling
- Creating the ROS package for the robot description
- Understanding robot modeling using URDF
- Understanding robot modeling using xacro
- Converting xacro to URDF
- Creating a robot description for a seven DOF robot manipulator
- Working with the joint state publisher and robot state publisher
- Creating robot description for a differential wheeled robot

ROS packages for robot modeling

ROS provides some good packages that can be used to build 3D robot models.
In this section, we will discuss some of the important ROS packages that are commonly used to build and model a robot:

- `urdf`: The most important ROS package to model the robot is the `urdf` package. This package contains a C++ parser for the **Unified Robot Description Format** (**URDF**), which is an XML file representing a robot model. Other different components make up `urdf`:
    - `urdf_parser_plugin`: This package implements methods to fill URDF data structures
    - `urdfdom_headers`: This component provides core data structure headers to use the urdf parser
    - `collada_parser`: This package populates data structures by parsing a Collada file
    - `urdfdom`: This component populates data structures by parsing URDF files
    - `collada-dom`: This is a stand-alone component to convert Collada documents with 3D computer graphics software such as *Maya*, *Blender*, and *Soft image*

We can define a robot model, sensors, and a working environment using URDF, and we can parse it using URDF parsers. We can only describe a robot in URDF that has a tree-like structure in its links, that is, the robot will have rigid links and will be connected using joints. Flexible links can't be represented using URDF. The URDF is composed using special XML tags, and we can parse these XML tags using parser programs for further processing. We can work on URDF modeling in the upcoming sections:

- `joint_state_publisher`: This tool is very useful when designing robot models using URDF. This package contains a node called `joint_state_publisher`, which reads the robot model description, finds all joints, and publishes joint values to all nonfixed joints using GUI sliders. The user can interact with each robot joint using this tool and can visualize using RViz. While designing URDF, the user can verify the rotation and translation of each joint using this tool. We will talk more about the `joint_state_publisher` node and its usage in the upcoming section.
- `kdl_parser`: **Kinematic and Dynamics Library** (**KDL**) is an ROS package that contains parser tools to build a KDL tree from the URDF representation. The kinematic tree can be used to publish the joint states and also to forward and inverse the kinematics of the robot.
- `robot_state_publisher`: This package reads the current robot joint states and publishes the 3D poses of each robot link using the kinematics tree build from the URDF. The 3D pose of the robot is published as the `tf` (transform) ROS.
  The `tf` ROS publishes the relationship between the coordinates frames of a robot.
- `xacro`: Xacro stands for (XML Macros), and we can define how `xacro` is equal to URDF plus add-ons. It contains some add-ons to make URDF shorter and readable, and can be used for building complex robot descriptions. We can convert `xacro` to URDF at any time using ROS tools. We will learn more about `xacro` and its usage in the upcoming sections.

Understanding robot modeling using URDF

We have discussed the `urdf` package. In this section, we will look further into the `URDF XML` tags, which help to model the robot. We have to create a file and write the relationship between each link and joint in the robot and save the file with the `.urdf` extension.

URDF can represent the kinematic and dynamic description of the robot, the visual representation of the robot, and the collision model of the robot.

The following tags are the commonly used URDF tags to compose a URDF robot model:

- `link`: The `link` tag represents a single link of a robot. Using this tag, we can model a robot link and its properties. The modeling includes the size, the shape, and the color, and it can even import a 3D mesh to represent the robot link. We can also provide the dynamic properties of the link, such as the inertial matrix and the collision properties.

  The syntax is as follows:

```
<link name="<name of the link>">
<inertial>...........</inertial>
  <visual> ............</visual>
  <collision>..........</collision>
</link>
```

  The following is a representation of a single link. The **Visual** section represents the real link of the robot, and the area surrounding the real link is the **Collision** section. The **Collision** section encapsulates the real link to detect collision before hitting the real link:

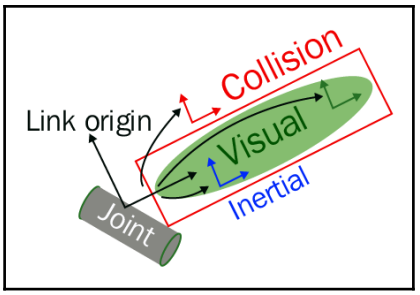

Figure 1: Visualization of a URDF link

- `joint`: The `joint` tag represents a robot joint. We can specify the kinematics and the dynamics of the joint, and set the limits of the joint movement and its velocity. The `joint` tag supports the different types of joints, such as **revolute**, **continuous**, **prismatic**, **fixed**, **floating**, and **planar**.

  The syntax is as follows:

```
<joint name="<name of the joint>">
  <parent link="link1"/>
  <child link="link2"/>
  <calibration .... />
```

```
  <dynamics damping ..../>
  <limit effort .... />
</joint>
```

A URDF joint is formed between two links; the first is called the **Parent** link, and the second is called the **Child** link. The following is an illustration of a joint and its link:

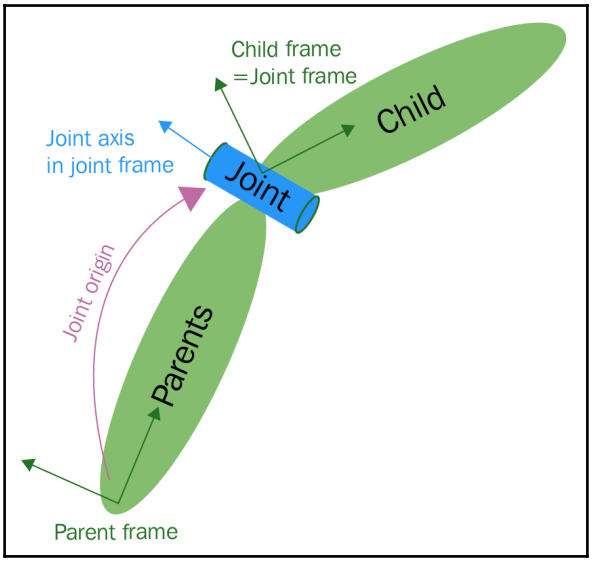

Figure 2: Visualization of a URDF joint

- `robot`: This tag encapsulates the entire robot model that can be represented using URDF. Inside the `robot` tag, we can define the name of the robot, the links, and the joints of the robot.

  The syntax is as follows:

```
<robot name="<name of the robot>"
  <link>  ..... </link>
  <link> ..... </link>

  <joint> ..... </joint>
  <joint> ........</joint>
</robot>
```

A robot model consists of connected links and joints. Here is a visualization of the robot model:

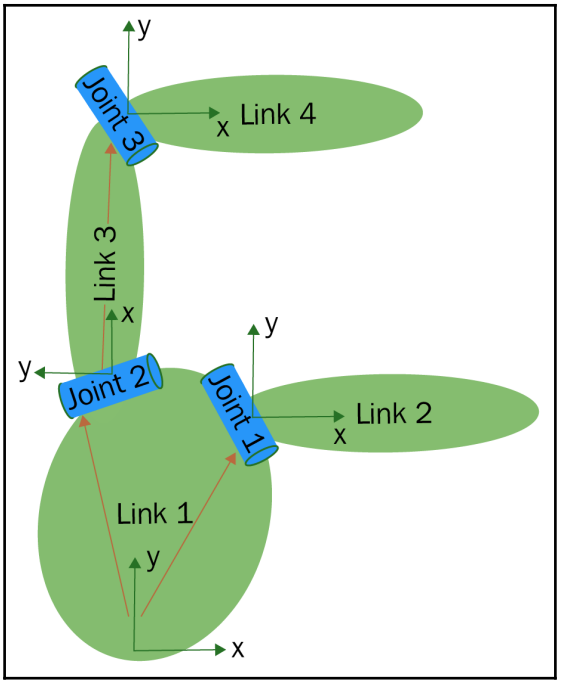

Figure 3: Visualization of a robot model having joints and links

- `gazebo`: This tag is used when we include the simulation parameters of the Gazebo simulator inside the URDF. We can use this tag to include `gazebo` plugins, `gazebo` material properties, and so on. The following shows an example using `gazebo` tags:

```
<gazebo reference="link_1">
   <material>Gazebo/Black</material>
</gazebo>
```

You can find more URDF tags at `http://wiki.ros.org/urdf/XML`.

Creating the ROS package for the robot description

Before creating the URDF file for the robot, let's create an ROS package in the `catkin` workspace so that the robot model keeps using the following command:

```
$ catkin_create_pkg mastering_ros_robot_description_pkg roscpp tf
 geometry_msgs urdf rviz xacro
```

The package mainly depends on the `urdf` and `xacro` packages. If these packages have not been installed on to your system, you can install them using the package manager:

```
$sudo apt-get install ros-kinetic-urdf
$sudo apt-get install ros-kinetic-xacro
```

We can create the `urdf` file of the robot inside this package and create launch files to display the created `urdf` in RViz. The full package is available on 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/mastering_ros_robot_description_pkg.git
```

Before creating the `urdf` file for this robot, let's create three folders called `urdf`, `meshes`, and `launch` inside the package folder. The `urdf` folder can be used to keep the `urdf` and `xacro` files that we are going to create. The `meshes` folder keeps the meshes that we need to include in the `urdf` file, and the `launch` folder keeps the ROS launch files.

Creating our first URDF model

After learning about URDF and its important tags, we can start some basic modeling using URDF. The first robot mechanism that we are going to design is a pan-and-tilt mechanism, as shown in the following figure.

There are three links and two joints in this mechanism. The base link is static, and all the other links are mounted on it. The first joint can pan on its axis, and the second link is mounted on the first link, and it can tilt on its axis. The two joints in this system are of a revolute type:

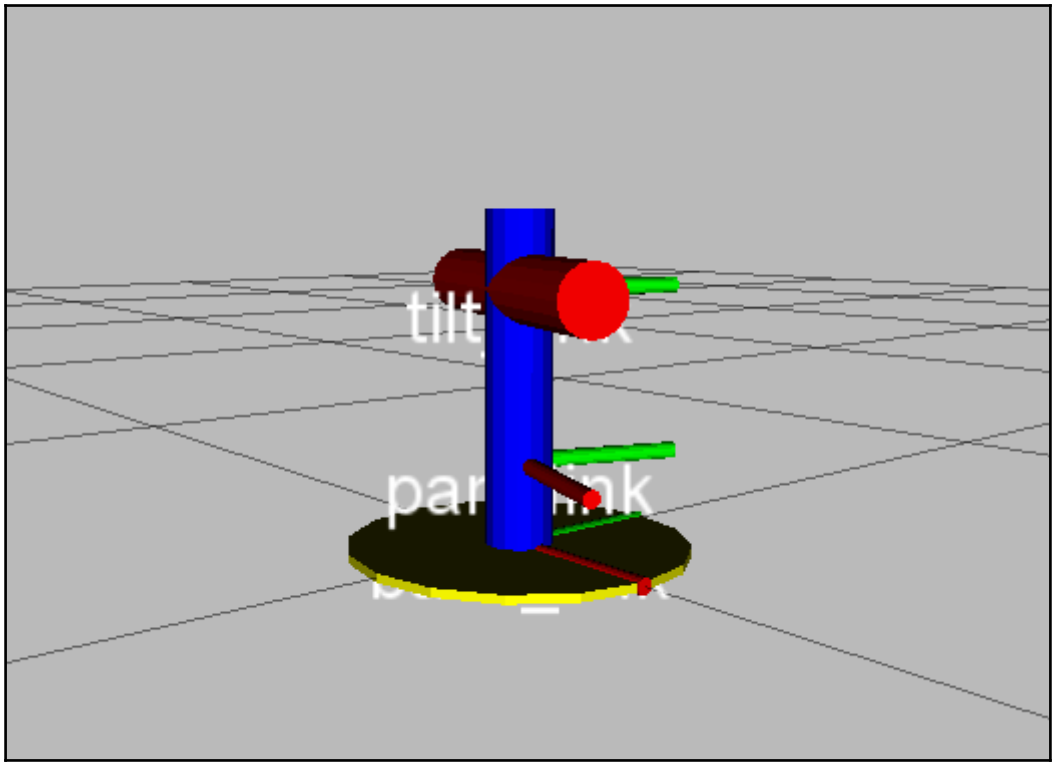

Figure 4: Visualization of a pan-and-tilt mechanism in RViz

Let's see the URDF code of this mechanism. Navigate to the `mastering_ros_robot_description_pkg/urdf` directory and open `pan_tilt.urdf`:

```
<?xml version="1.0"?>
<robot name="pan_tilt">
  <link name="base_link">
    <visual>
      <geometry>
      <cylinder length="0.01" radius="0.2"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
```

```
      <material name="yellow">
        <color rgba="1 1 0 1"/>
      </material>
    </visual>
  </link>
  <joint name="pan_joint" type="revolute">
    <parent link="base_link"/>
    <child link="pan_link"/>
    <origin xyz="0 0 0.1"/>
    <axis xyz="0 0 1" />
  </joint>
  <link name="pan_link">
    <visual>
      <geometry>
      <cylinder length="0.4" radius="0.04"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0.09"/>
      <material name="red">
        <color rgba="0 0 1 1"/>
      </material>
    </visual>
  </link>
  <joint name="tilt_joint" type="revolute">
    <parent link="pan_link"/>
    <child link="tilt_link"/>
    <origin xyz="0 0 0.2"/>
    <axis xyz="0 1 0" />
  </joint>
  <link name="tilt_link">
    <visual>
      <geometry>
  <cylinder length="0.4" radius="0.04"/>
      </geometry>
      <origin rpy="0 1.5 0" xyz="0 0 0"/>
      <material name="green">
        <color rgba="1 0 0 1"/>
      </material>
    </visual>
  </link>
</robot>
```

Explaining the URDF file

When we check the code, we can add a `<robot>` tag at the top of the description:

```
<?xml version="1.0"?>
<robot name="pan_tilt">
```

The `<robot>` tag defines the name of the robot that we are going to create. Here, we named the robot `pan_tilt`.

If we check the sections after the `<robot>` tag definition, we can see link and joint definitions of the pan-and-tilt mechanism:

```
<link name="base_link">
  <visual>
    <geometry>
    <cylinder length="0.01" radius="0.2"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <material name="yellow">
      <color rgba="1 1 0 1"/>
    </material>
  </visual>
</link>
```

The preceding code snippet is the `base_link` definition of the pan-and-tilt mechanism. The `<visual>` tag describes the visual appearance of the link, which is shown on the robot simulation. We can define the link geometry (`cylinder`, `box`, `sphere`, or `mesh`) and the material (`color` and `texture`) of the link using this tag:

```
<joint name="pan_joint" type="revolute">
  <parent link="base_link"/>
  <child link="pan_link"/>
  <origin xyz="0 0 0.1"/>
  <axis xyz="0 0 1" />
</joint>
```

In the preceding code snippet, we define, a joint with a unique name and its joint type. The joint type we used here is `revolute`, and the parent and child links are `base_link` and `pan_link`, respectively. The joint origin is also specified inside this tag.

Save the preceding URDF code as `pan_tilt.urdf` and check whether the `urdf` contains errors using the following command:

```
$ check_urdf pan_tilt.urdf
```

The `check_urdf` command will parse the `urdf` tag and show an error, if there are any. If everything is OK, it will output the following:

```
robot name is: pan_tilt
---------- Successfully Parsed XML ---------------
  root Link: base_link has 1 child(ren)
    child(1):  pan_link
      child(1):  tilt_link
```

If we want to view the structure of the robot links and joints graphically, we can use a command tool called `urdf_to_graphiz`:

```
$ urdf_to_graphiz pan_tilt.urdf
```

This command will generate two files: `pan_tilt.gv` and `pan_tilt.pdf`. We can view the structure of this robot using this command:

```
$ evince pan_tilt.pdf
```

We will get the following output:

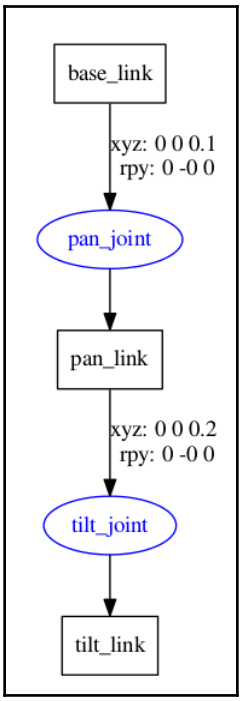

Figure 5: Graph of joint and links in the pan-and-tilt mechanism

Visualizing the 3D robot model in RViz

After designing the URDF, we can view it on RViz. We can create a `view_demo.launch` launch file and put the following code into the `launch` folder. Navigate to the `mastering_ros_robot_description_pkg/launch` directory for the code:

```
<launch>
  <arg name="model" />
  <param name="robot_description" textfile="$(find
mastering_ros_robot_description_pkg)/urdf/pan_tilt.urdf" />
  <param name="use_gui" value="true"/>
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mastering_ros_robot_description_pkg)/urdf.rviz" required="true" />
</launch>
```

We can launch the model using the following command:

```
$ roslaunch mastering_ros_robot_description_pkg view_demo.launch
```

If everything works fine, we will get a pan-and-tilt mechanism in RViz, as shown here:

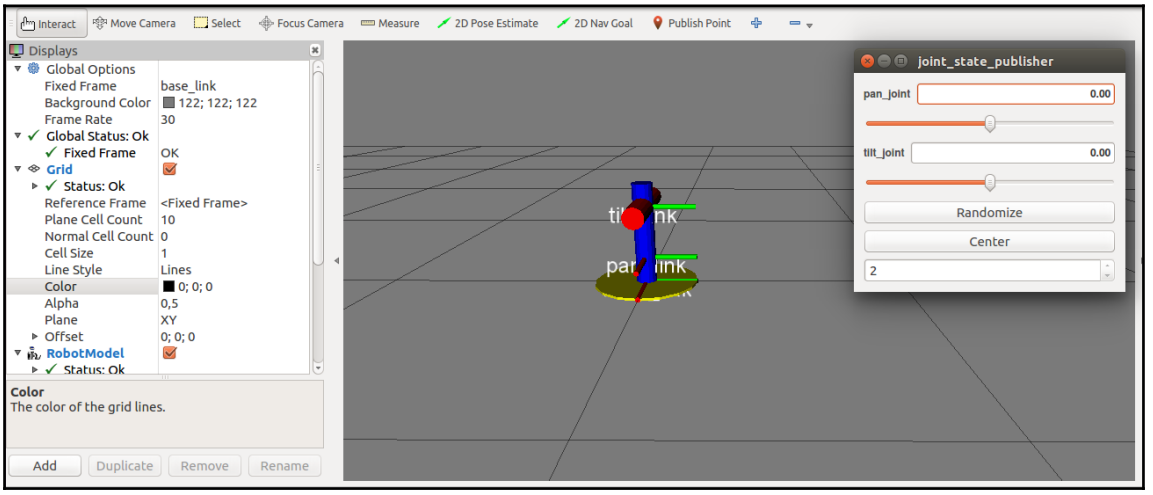

Figure 6: Joint level of pan-and-tilt mechanism

Interacting with pan-and-tilt joints

We can see that an extra GUI came along with RViz; it contains sliders to control the pan joints and the tilt joints. This GUI is called the Joint State Publisher node, from the `joint_state_publisher` package:

```
    <node name="joint_state_publisher" pkg="joint_state_publisher"
  type="joint_state_publisher" />
```

We can include this node in the `launch` file, using this statement. The limits of pan-and-tilt should be mentioned inside the `joint` tag:

```
    <joint name="pan_joint" type="revolute">
      <parent link="base_link"/>
      <child link="pan_link"/>
      <origin xyz="0 0 0.1"/>
      <axis xyz="0 0 1" />
      <limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
      <dynamics damping="50" friction="1"/>
    </joint>
```

`<limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>` defines the limits of effort, the velocity, and the angle limits. The effort is the maximum force supported by this joint; `lower` and `upper` indicate the lower and upper limit of the joint in the radian for the revolute type joint, and meters for prismatic joints. The velocity is the maximum joint velocity:

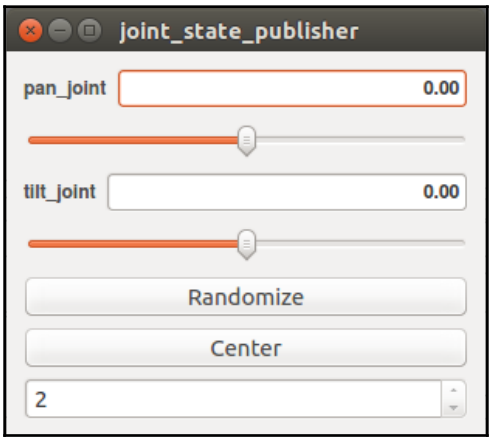

Figure 7: Joint level of pan-and-tilt mechanism

The preceding screenshot shows the GUI of `joint_state_publisher`, with sliders and current joint values shown in the box.

Adding physical and collision properties to a URDF model

Before simulating a robot in a robot simulator, such as Gazebo or V-REP, we need to define the robot link's physical properties, such as geometry, color, mass, and inertia, as well as the collision properties of the link.

We will only get good simulation results if we define all these properties inside the robot model. URDF provides tags to include all these parameters and code snippets of `base_link` contained in these properties, as given here:

```
<link>
......
<collision>
      <geometry>
      <cylinder length="0.03" radius="0.2"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>

    <inertial>
    <mass value="1"/>
    <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
    </inertial>
...........
</link>
```

Here, we define the collision geometry as cylinder and the mass as 1 kg, and we also set the inertial matrix of the link.

The `collision` and `inertia` parameters are required in each link, otherwise Gazebo will not load the robot model properly.

Understanding robot modeling using xacro

The flexibility of URDF reduces when we work with complex robot models. Some of the main features that URDF is missing are simplicity, reusability, modularity, and programmability.

If someone wants to reuse a URDF block 10 times in his robot description, he can copy and paste the block 10 times. If there is an option to use this code block and make multiple copies with different settings, it will be very useful while creating the robot description.

The URDF is a single file and we can't include other URDF files inside it. This reduces the modular nature of the code. All code should be in a single file, which reduces the code's simplicity.

Also, if there is some programmability, such as adding variables, constants, mathematical expressions, and conditional statements, in the description language, it will be more user-friendly.

The robot modeling using xacro meets all of these conditions. Some of the main features of xacro are as follows:

- **Simplify URDF**: The xacro is the cleaned-up version of URDF. It creates macros inside the robot description and reuses the macros. This can reduce the code length. Also, it can include macros from other files and make the code simpler, more readable, and more modular.
- **Programmability**: The xacro language supports a simple programming statement in its description. There are variables, constants, mathematical expressions, conditional statements, and so on that make the description more intelligent and efficient.

We can say that xacro is an updated version of URDF, and we can convert the xacro definition to URDF whenever it is necessary, using some ROS tools.

We can talk about the same description of pan-and-tilt using xacro. Navigate to `mastering_ros_robot_description_pkg/urdf`, and the file name is `pan_tilt.xacro`. Instead of `.urdf`, we need to use the `.xacro` extension for xacro files. Here is the explanation of the xacro code:

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

These lines specify a namespace that is needed in all xacro files for parsing the xacro file. After specifying the namespace, we need to add the name of the xacro file.

Using properties

Using xacro, we can declare constants or properties that are the named values inside the xacro file, which can be used anywhere in the code. The main use of these constant definitions is, instead of giving hardcoded values on links and joints, we can keep constants, and it will be easier to change these values rather than finding the hardcoded values and replacing them.

An example of using properties is given here. We declare the base link and the pan link's length and radius. So, it will be easy to change the dimension here rather than changing the values in each one:

```
<xacro:property name="base_link_length" value="0.01" />
<xacro:property name="base_link_radius" value="0.2" />
<xacro:property name="pan_link_length" value="0.4" />
<xacro:property name="pan_link_radius" value="0.04" />
```

We can use the value of the variable by replacing the hardcoded value with the following definition:

```
 <cylinder length="${pan_link_length}"
radius="${pan_link_radius}"/>
```

Here, the old value, `"0.4"`, is replaced with `"{pan_link_length}"`, and `"0.04"` is replaced with `"{pan_link_radius}"`.

Using the math expression

We can build mathematical expressions inside `${}` using basic operations such as `+`, `-`, `*` , `/` , unary minus, and parenthesis. Exponentiation and modulus are not supported yet. The following is a simple math expression used inside the code:

```
 <cylinder length="${pan_link_length}"
radius="${pan_link_radius+0.02}"/>
```

Using macros

One of the main features of xacro is that it supports macros. We can use xacro to reduce the length of complex definitions. Here is a `xacro` definition we used in our code for inertial:

```
<xacro:macro name="inertial_matrix" params="mass">
  <inertial>
       <mass value="${mass}" />
          <inertia ixx="0.5" ixy="0.0" ixz="0.0"
          iyy="0.5" iyz="0.0" izz="0.5" />
   </inertial>
</xacro:macro>
```

Here, the macro is named `inertial_matrix`, and its parameter is mass. The mass parameter can be used inside the inertial definition using `${mass}`. We can replace each inertial code with a single line, as given here:

```
<xacro:inertial_matrix mass="1"/>
```

The xacro definition improved the code readability and reduced the number of lines compared to `urdf`. Next, we will look at how to convert xacro to a URDF file.

Converting xacro to URDF

After designing the xacro file, we can use the following command to convert it to a URDF file:

```
$ rosrun xacro xacro pan_tilt.xacro --inorder > pan_tilt_generated.urdf
```

The `--inorder` option has been recently introduced in ROS to increase the power of the conversion tool. It allows us to process the document in read order, adding more features than there were in older ROS versions.

We can use the following line in the ROS launch file for converting xacro to URDF and use it as a `robot_description` parameter:

```
 <param name="robot_description" command="$(find xacro)/xacro --inorder
$(find mastering_ros_robot_description_pkg)/urdf/pan_tilt.xacro"
/>
```

We can view the xacro of pan-and-tilt by making a launch file, and it can be launched using the following command:

```
    $ roslaunch mastering_ros_robot_description_pkg
view_pan_tilt_xacro.launch
```

Creating the robot description for a seven DOF robot manipulator

Now, we can create some complex robots using URDF and xacro. The first robot we are going to deal with is a seven DOF robotic arm, which is a serial link manipulator with multiple serial links. The seven DOF arm is kinematically redundant, which means it has more joints and DOF than required to achieve its goal position and orientation. The advantage of redundant manipulators is that we can have more joint configuration for a desired goal position and orientation. It will improve the flexibility and versatility of the robot movement and can implement effective collision-free motion in a robotic workspace.

Let's start creating the seven DOF arm; the final output model of the robot arm is shown here (the various joints and links in the robot are also marked on the image):

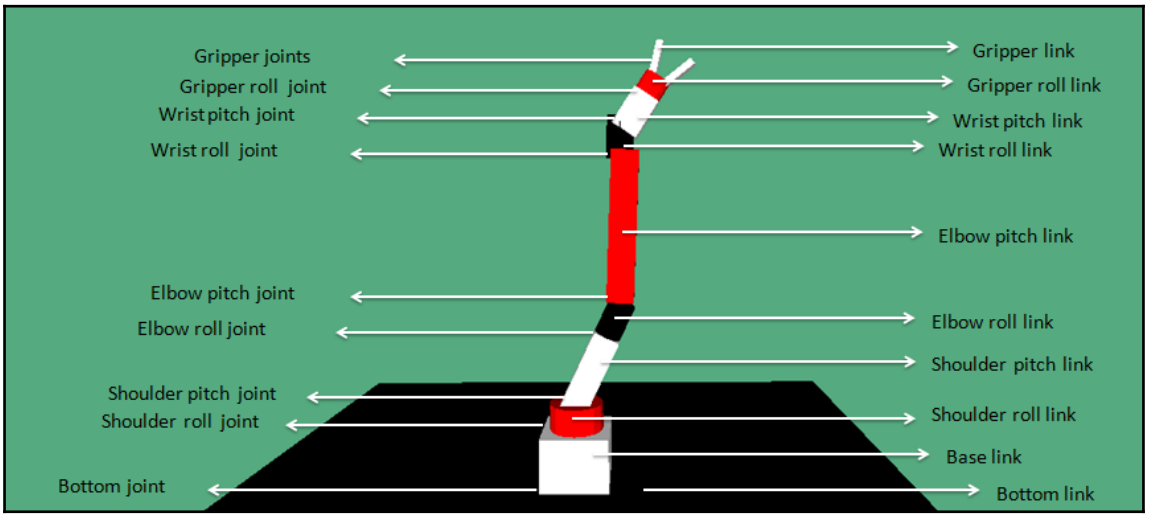

Figure 8: Joints and Links of seven DOF arm robot

The preceding robot is described using xacro. We can take the actual description file from the cloned repository. We can navigate to the `urdf` folder inside the cloned package and open the `seven_dof_arm.xacro` file. We will copy and paste the description to the current package and discuss the major section of this robot description.

Arm specification

Here is the robot arm specification of this seven DOF arm:

- Degrees of freedom: 7
- Length of the arm: 50 cm
- Reach of the arm: 35 cm
- Number of links: 12
- Number of joints: 11

Type of joints

Here is the list of joints containing the joint name and its type of robot:

Joint number	Joint name	Joint type	Angle limits (in degrees)
1	`bottom_joint`	Fixed	--
2	`shoulder_pan_joint`	Revolute	-150 to 114
3	`shoulder_pitch_joint`	Revolute	-67 to 109
4	`elbow_roll_joint`	Revolute	-150 to 41
5	`elbow_pitch_joint`	Revolute	-92 to 110
6	`wrist_roll_joint`	Revolute	-150 to 150
7	`wrist_pitch_joint`	Revolute	92 to 113
8	`gripper_roll_joint`	Revolute	-150 to 150
9	`finger_joint1`	Prismatic	0 to 3 cm
10	`finger_joint2`	Prismatic	0 to 3 cm

We design the xacro of the arm using the preceding specifications; next up is the explanation of the xacro arm file.

Explaining the xacro model of the seven DOF arm

We will define ten links and nine joints on this robot, and two links and two joints in the robot gripper.

Let's start by looking at the xacro definition:

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

Because we are writing a xacro file, we should mention the xacro namespace to parse the file.

Using constants

We use constants inside this xacro to make robot descriptions shorter and more readable. Here, we define the degree to the radian conversion factor, PI value, length, height, and width of each of the links:

```
  <property name="deg_to_rad" value="0.01745329251994329577"/>
  <property name="M_PI" value="3.14159"/>
  <property name="elbow_pitch_len" value="0.22" />
  <property name="elbow_pitch_width" value="0.04" />
  <property name="elbow_pitch_height" value="0.04" />
```

Using macros

We define macros in this code to avoid repeatability and to make the code shorter. Here are the macros we have used in this code:

```
   <xacro:macro name="inertial_matrix" params="mass">
      <inertial>
        <mass value="${mass}" />
        <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="0.5" iyz="0.0"
izz="1.0" />
      </inertial>
   </xacro:macro>
```

This is the definition of the `inertial matrix` macro, in which we can use `mass` as its parameter:

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

In the section of the code, we can see the definition using the `transmission` tag.

The `transmission` tag relates a joint to an actuator. It defines the type of transmission that we are using in a particular joint, as well as the type of motor and its parameters. It also defines the type of hardware interface we use when we interface with the ROS controllers.

Including other xacro files

We can extend the capabilities of the robot xacro by including the xacro definition of sensors using the `xacro:include` tag. The following code snippet shows how to include a sensor definition in the robot xacro:

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

Here, we include a xacro definition of a sensor called **Asus Xtion pro**, and this will be expanded when the xacro file is parsed.

Using `"$(find mastering_ros_robot_description_pkg)/urdf/sensors/xtion_pro_live.urdf.xacro"`, we can access the xacro definition of the sensor, where `find` is to locate the current `mastering_ros_robot_description_pkg` package.

We will talk more about vision-processing in `Chapter 10`, *Building and Interfacing Differential Drive Mobile Robot Hardware in ROS*.

Using meshes in the link

We can insert a primitive shape in to a link, or we can insert a mesh file using the `mesh` tag. The following example shows how to insert a mesh into the vision sensor:

```
<visual>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <geometry>
    <mesh filename=
"package://mastering_ros_robot_description_pkg/meshes/sensors/xtion_pro_liv
e/xtion_pro_live.dae"/>
  </geometry>
<material name="DarkGrey"/>
</visual>
```

Working with the robot gripper

The gripper of the robot is designed for the picking and placing of blocks; the gripper is in the simple linkage category. There are two joints for the gripper, and each joint is prismatic. Here is the `joint` definition of one gripper joint:

```
 <joint name="finger_joint1" type="prismatic">
  <parent link="gripper_roll_link"/>
  <child link="gripper_finger_link1"/>
  <origin xyz="0.0 0 0" />
  <axis xyz="0 1 0" />
    <limit effort="100" lower="0" upper="0.03" velocity="1.0"/>
    <safety_controller k_position="20"
                       k_velocity="20"
                       soft_lower_limit="${-0.15 }"
                       soft_upper_limit="${ 0.0 }"/>
  <dynamics damping="50" friction="1"/>
</joint>
```

Here, the first gripper joint is formed by `gripper_roll_link` and `gripper_finger_link1`, and the second joint is formed by `gripper_roll_link` and `gripper_finger_link2`.

The following graph shows how the gripper joints are connected in `gripper_roll_link`:

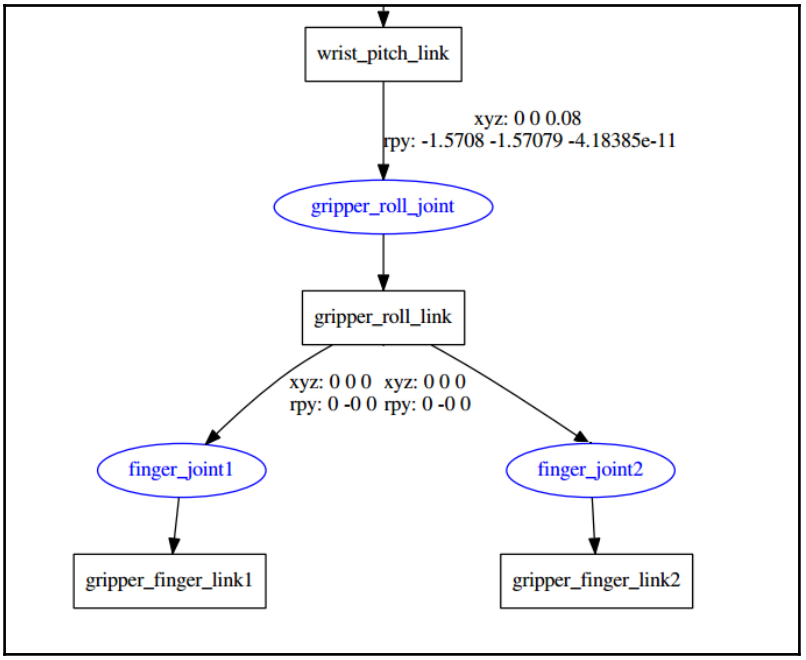

Figure 9: Graph of the end effector section of the seven DOF arm robot

Viewing the seven DOF arm in RViz

After talking about the robot model, we can view the designed xacro file in **RViz** and control each joint using the `joint state publisher` node and publish the robot state using `Robot State Publisher`.

The preceding task can be performed using a launch file called `view_arm.launch`, which is inside the `launch` folder of this package:

```
<launch>
  <arg name="model" />

  <!-- Parsing xacro and loading robot_description parameter -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder
$(find mastering_ros_robot_description_pkg)/urdf/ seven_dof_arm.xacro " />

  <!-- Setting gui parameter to true for display joint slider, for getting
```

```
joint control -->
  <param name="use_gui" value="true"/>

  <!-- Starting Joint state publisher node which will publish the joint
values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />

  <!-- Starting robot state publish which will publish current robot joint
states using tf -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />

  <!-- Launch visualization in rviz -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mastering_ros_robot_description_pkg)/urdf.rviz" required="true" />
</launch>
```

Create the following launch file inside the `launch` folder, and build the package using the `catkin_make` command. Launch the `urdf` using the following command:

```
$ roslaunch mastering_ros_robot_description_pkg view_arm.launch
```

The robot will be displayed on RViz, with the joint state publisher GUI:

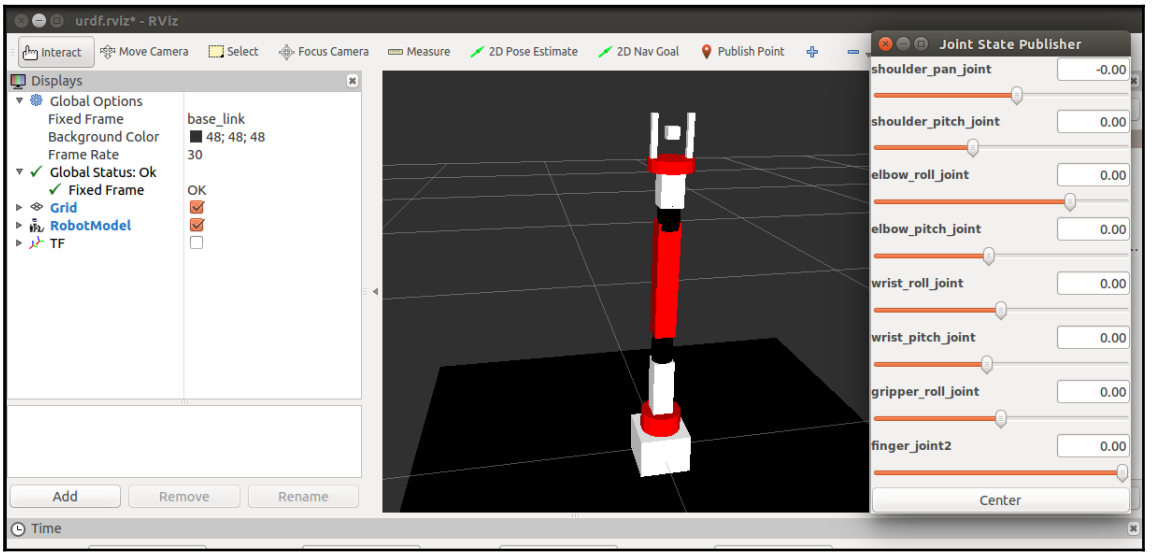

Figure 10: Seven DOF arm in RViz with joint_state_publisher

We can interact with the joint slider and move the joints of the robot. Next, we will talk about what `joint state publisher` can do.

Understanding joint state publisher

Joint state publisher is one of the ROS packages that is commonly used to interact with each joint of the robot. The package contains the `joint_state_publisher` node, which will find the nonfixed joints from the URDF model and publish the joint state values of each joint in the `sensor_msgs/JointState` message format.

In the preceding launch file, `view_arm.launch`, we started the `joint_state_publisher` node and set a parameter called `use_gui` to `true`, as follows:

```
  <param name="use_gui" value="true"/>

  <!-- Starting Joint state publisher node which will publish the joint
values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
```

If we set `use_gui` to `true`, the `joint_state_publisher` node displays a slider-based control window to control each joint. The lower and upper value of a joint will be taken from the lower and upper values associated with the `limit` tag used inside the `joint` tag. The preceding screenshot shows the robot model in RViz, along with a user interface to change the position of robot joints, which states with the `use_gui` parameter set to `true`.

We can find more on the `joint state publisher` package at `http://wiki.ros.org/joint_state_publisher`.

Understanding robot state publisher

The `robot state publisher` package helps to publish the state of the robot to `tf`. This package subscribes to joint states of the robot and publishes the 3D pose of each link using the kinematic representation from the URDF model. We can implement the `robot state publisher` node using the following line inside the launch file:

```
<!-- Starting robot state publish which will publish tf -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
```

In the preceding launch file, `view_arm.launch`, we started this node to publish the `tf` of the arm. We can visualize the transformation of the robot by clicking the `tf` option on RViz, shown as follows:

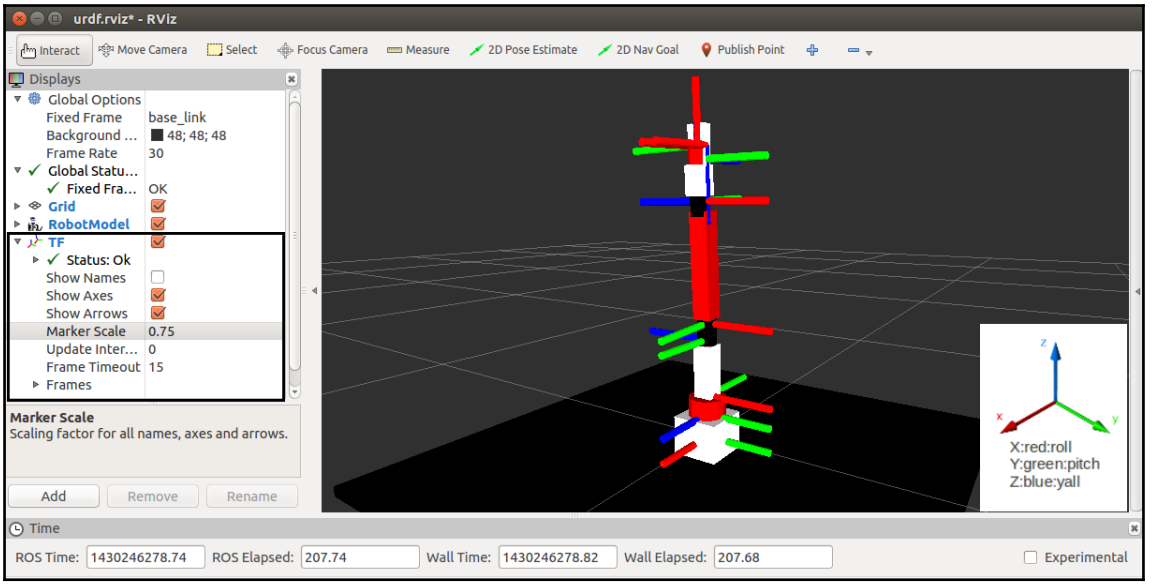

Figure 11: TF view of seven DOF arm in RViz

The `joint_state_publisher` and `robot_state_publisher` packages are installed along with the ROS desktop's installations.

After creating the robot description of the seven DOF arm, we can talk about how to make a mobile robot with differential wheeled mechanisms.

Creating a robot model for the differential drive mobile robot

A differential wheeled robot will have two wheels connected on opposite sides of the robot chassis, which is supported by one or two caster wheels. The wheels will control the speed of the robot by adjusting individual velocity. If the two motors are running at the same speed, the wheels will move forward or backward. If one wheel is running slower than the other, the robot will turn to the side of the lower speed. If we want to turn the robot to the left side, we reduce the velocity of the left wheel, and vice versa.

There are two supporting wheels, called caster wheels, that will support the robot and rotate freely based on the movement of the main wheels.

The URDF model of this robot is present in the cloned ROS package. The final robot model is shown here:

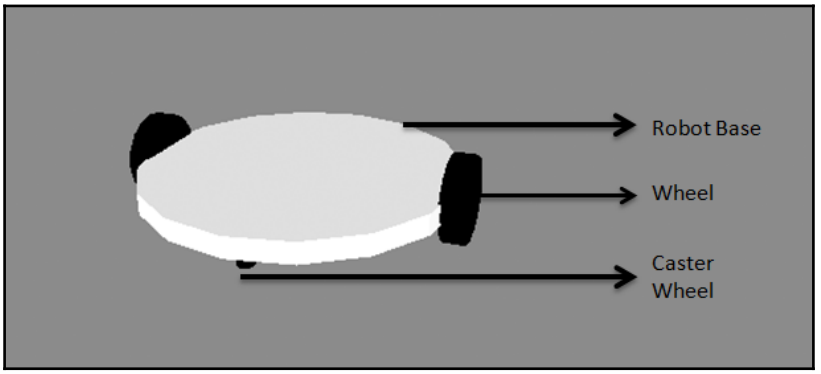

Figure 12: Differential drive mobile robot

The preceding robot has five joints and links. The two main joints connect the wheels to the robot, while the others are fixed joints connecting the caster wheels and the base footprint to the body of the robot.

The preceding robot has five joints and five links. The two main joints are two-wheel joints, and the other three joints are two fixed joints by caster wheels, and one fixed joint by base foot print to the base link of the robot. Here is the connection graph of this robot:

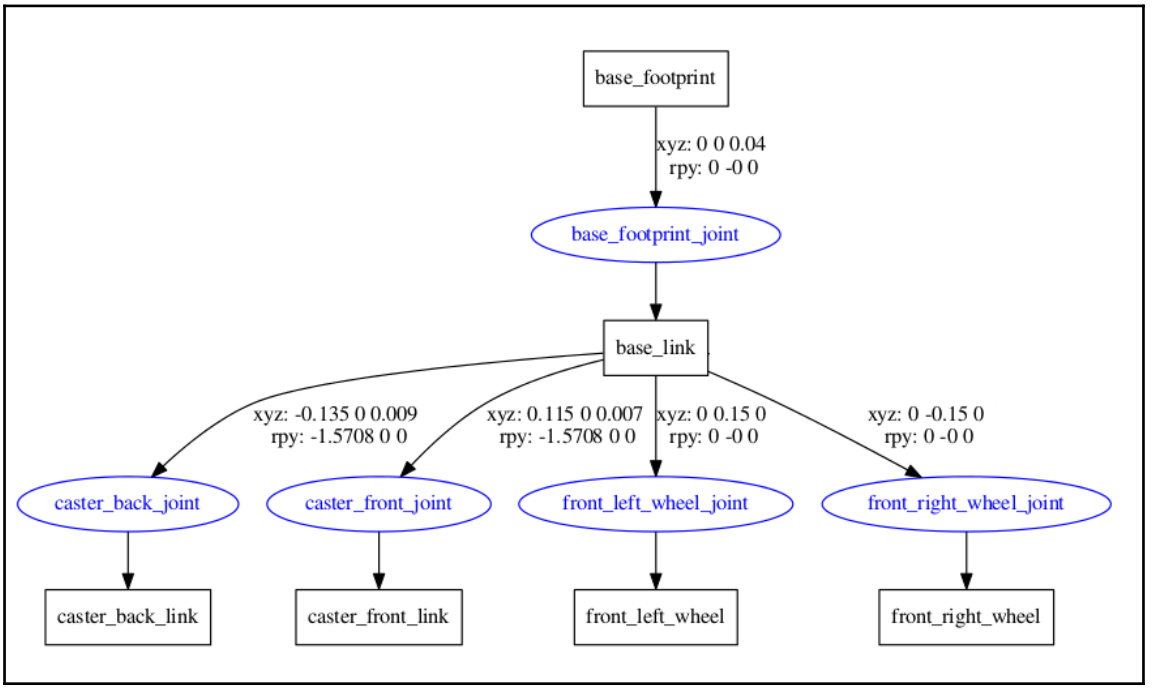

Figure 13: Graphical representation of links and joints for the differential drive mobile robot

We can go through the important section of code in the URDF file. The URDF file, called `diff_wheeled_robot.xacro`, is placed inside the `urdf` folder of the cloned ROS package.

The first section of the URDF file is given here. The robot is named `differential_wheeled_robot`, and it also includes a URDF file, called `wheel.urdf.xacro`. This xacro file contains the definition of the wheel and its transmission; if we use this xacro file, then we can avoid writing two definitions for the two wheels. We use this xacro definition because the two wheels are identical in shape and size:

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

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

The definition of a wheel inside `wheel.urdf.xacro` is given here. We can mention whether the wheel has to be placed to the left, right, front, or back. Using this macro, we can create a maximum of four wheels but, for now, we require only two:

```
<xacro:macro name="wheel" params="fb lr parent translateX translateY
flipY"> <!--fb : front, back ; lr: left, right -->
    <link name="${fb}_${lr}_wheel">
```

We also mention the Gazebo parameters required for simulation. Mentioned here are the Gazebo parameters associated with a wheel. We can mention the frictional co-efficient and the stiffness co-efficient using the `gazeboreference` tag:

```
    <gazebo reference="${fb}_${lr}_wheel">
      <mu1 value="1.0"/>
      <mu2 value="1.0"/>
      <kp  value="10000000.0" />
      <kd  value="1.0" />
      <fdir1 value="1 0 0"/>
      <material>Gazebo/Grey</material>
      <turnGravityOff>false</turnGravityOff>
    </gazebo>
```

The joints that we define for a wheel are continuous joints because there is no limit in the `wheel` joint. The `parent link` here is the robot base, and the `child link` is each wheel:

```
    <joint name="${fb}_${lr}_wheel_joint" type="continuous">
      <parent link="${parent}"/>
      <child link="${fb}_${lr}_wheel"/>
      <origin xyz="${translateX *
```

We also need to mention the `transmission` tag of each wheel; the macro of the wheel is as follows:

```
    <!-- Transmission is important to link the joints and the controller -
->
    <transmission name="${fb}_${lr}_wheel_joint_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="${fb}_${lr}_wheel_joint" />
      <actuator name="${fb}_${lr}_wheel_joint_motor">
        <hardwareInterface>EffortJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>
  </xacro:macro>
</robot>
```

In `diff_wheeled_robot.xacro`, we can use the following lines to use the macros defined inside `wheel.urdf.xacro`:

```
  <wheel fb="front" lr="right" parent="base_link" translateX="0"
translateY="-0.5" flipY="-1"/>
  <wheel fb="front" lr="left" parent="base_link" translateX="0"
translateY="0.5" flipY="-1"/>
```

Using the preceding lines, we define the wheels on the left and right of the robot base. The robot base is cylindrical, as shown in the preceding figure. The inertia calculating macro is given here. This xacro snippet will use the mass, radius, and height of the cylinder to calculate inertia using this equation:

```
  <!-- Macro for calculating inertia of cylinder -->
  <macro name="cylinder_inertia" params="m r h">
    <inertia  ixx="${m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0"
              iyy="${m*(3*r*r+h*h)/12}" iyz = "0"
              izz="${m*r*r/2}" />
  </macro>
```

The launch file definition for displaying this root model in RViz is given here. The launch file is named `view_mobile_robot.launch`:

```
<launch>
  <arg name="model" />
  <!-- Parsing xacro and setting robot_description parameter -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder
$(find mastering_ros_robot_description_pkg)/urdf/diff_wheeled_robot.xacro"
/>
  <!-- Setting gui parameter to true for display joint slider -->
  <param name="use_gui" value="true"/>
```

```
  <!-- Starting Joint state publisher node which will publish the joint
values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
  <!-- Starting robot state publish which will publish tf -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
  <!-- Launch visualization in rviz -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mastering_ros_robot_description_pkg)/urdf.rviz" required="true" />
</launch>
```

The only difference between the arm URDF file is the change in the name; the other sections are the same.

We can view the mobile robot using the following command:

```
$ roslaunch mastering_ros_robot_description_pkg view_mobile_robot.launch
```

The screenshot of the robot in RViz is as follows:

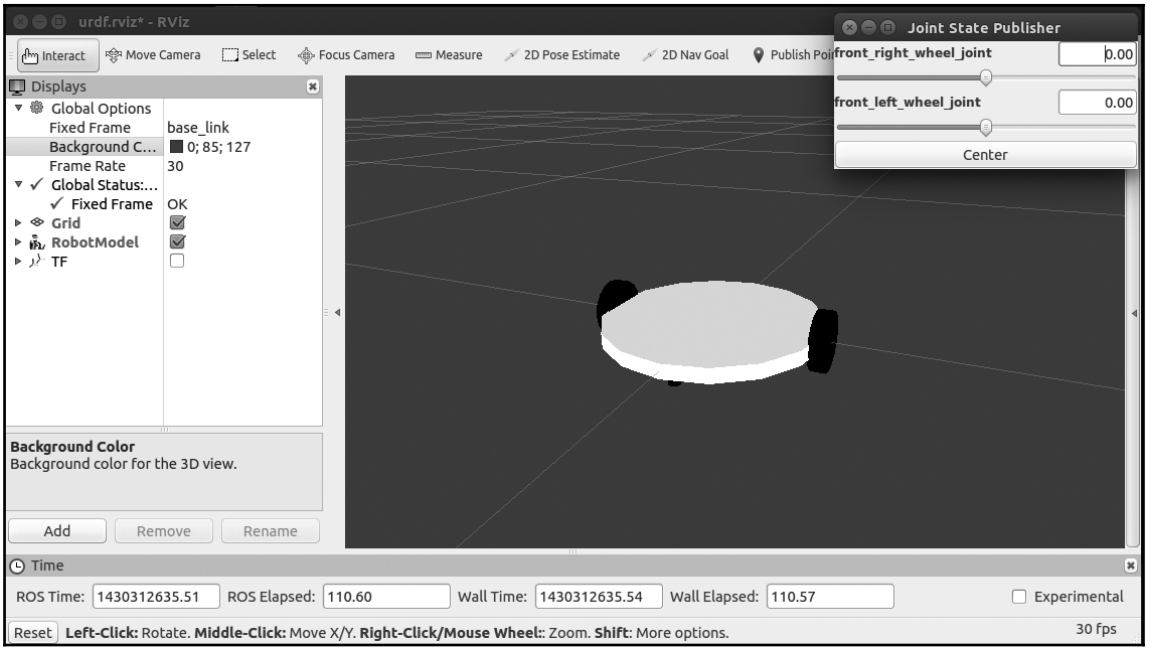

Figure 14: Visualizing mobile robot in RViz with joint state publisher

Questions

- What are the packages used for robot modeling in ROS?
- What are the important URDF tags used for robot modeling?
- What are the reasons for using xacro over URDF?
- What is the function of the joint state publisher and robot state publisher packages?
- What is the function of the transmission tag in URDF?

Summary

In this chapter, we mainly looked at the importance of robot modeling and how we can model a robot in ROS. We talked more about the `robot_model` meta package and the packages inside `robot_model`, such as `urdf`, `xacro`, and `joint_state_publisher`. We discussed URDF, xacro, and the main URDF tags that we are going to use. We also created a sample model in URDF and xacro and discussed the difference between the two. After that, we created a complex robotic manipulator with seven DOF and looked at the usage of the `joint state publisher` and `robot state publisher` packages. Towards the end of the chapter, we reviewed the designing procedure of a differential drive mobile robot using xacro. In the next chapter, we will look at the simulation of these robots using Gazebo.