Learning Robotics using Python

Second Edition

Design, simulate, program, and prototype an autonomous mobile robot using ROS, OpenCV, PCL, and Python

Packt
www.packt.com

By Lentin Joseph

Learning Robotics using Python
Second Edition

Design, simulate, program, and prototype an autonomous mobile robot using ROS, OpenCV, PCL, and Python

Lentin Joseph

Packt

BIRMINGHAM - MUMBAI

Learning Robotics using Python
Second Edition

Commissioning Editor: Gebin George
Acquisition Editor: Namrata Patil
Content Development Editor: Sharon Raj
Technical Editor: Mohit Hassija
Copy Editor: Safis Editing
Project Coordinator: Virginia Dias
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Tom Scaria
Production Coordinator: Shantanu Zagade

First published: May 2015
Second edition: June 2018

Production reference: 1250618

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78862-331-5

www.packtpub.com

To my mother, Jancy Joseph, and my father, C.G Joseph, for giving me strong support in making this project happen.

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Lentin Joseph is an author and robotics entrepreneur from India. He runs a robotics software company called Qbotics Labs in India. He has 7 years of experience in the robotics domain primarily in ROS, OpenCV, and PCL.

He has authored four books in ROS, namely, *Learning Robotics using Python*, *Mastering ROS for Robotics Programming*, *ROS Robotics Projects*, and *Robot Operating System for Absolute Beginners*.

He is currently pursuing his master's in Robotics from India and is also doing research at Robotics Institute, CMU, USA.

About the reviewer

Ruixiang Du is a PhD candidate in mechanical engineering at Worcester Polytechnic Institute. He works in the Autonomy, Control, and Estimation Laboratory with a research focus on the motion planning and control of autonomous mobile robots in cluttered and dynamic environments. He received a bachelor's degree in automation from North China Electric Power University in 2011 and a master's degree in robotics engineering from WPI in 2013. He has worked on various robotic projects with robot platforms ranging from medical robots, and unmanned aerial/ground vehicles, to humanoid robots.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Learning Robotics using Python contains nine chapters that explain how to build an autonomous mobile robot from scratch and program it using Python. The robot mentioning in this book is a service robot that can be used to serve food in home, hotels, and restaurant. From the beginning to end, the book discusses step-by-step procedures of building of this robot. The book starts with the basics concepts of robotics and then moves to the 3D modeling and simulation of the robot. After successful simulation of the robot, it discusses the hardware components required to build the robot prototype.

The software part of this robot is mainly implemented using Python programming language and software frameworks, such as Robot Operating System (ROS) and OpenCV. You can see the application of python from the designing of a robot to creating robot user interface. The Gazebo simulator is used to simulate the robot and machine vision libraries, such as OpenCV, OpenNI, and PCL, is for processing the 2D and 3D image data. Each chapter is presented with adequate theory for understanding the application part. The book is reviewed by the experts in this field and it is the result of their handwork and passion in robotics.

Who this book is for

Learning Robotics using Python is a good companion for entrepreneurs who want to explore service robotics domain, professionals who want to implement more features on their robots, researchers who want to explore more on robotics, and hobbyist or students who want to learn robotics. The book follows a step-by-step guide, which can easily be captured by anyone.

What this book covers

`Chapter 1`, *Getting Started with Robot Operating System*, explains the fundamental concepts of ROS, which are the main platform for programming robot.

`Chapter 2`, *Understanding the Basics of Differential Robots*, discusses the fundamental concepts of a differential mobile robot. The concepts are Kinematics and Inverse kinematics of differential drive. This will help you implement the differential drive controller in the software.

`Chapter 3`, *Modeling the Differential Drive Robot*, discusses the calculation of the robot design constraints and 2D/3D modeling of this mobile robot. The 2D/3D modeling is based on a set of robot requirements. After completing the design and robot modeling, the reader will get the designed parameters that can be used for creating a robot simulation.

`Chapter 4`, *Simulating a Differential Drive Robot Using ROS*, introduces a robot simulator named Gazebo and helps readers to simulate their own robot using it.

`Chapter 5`, *Designing ChefBot Hardware and Circuits*, discusses the selection of different hardware components required to build Chefbot.

`Chapter 6`, *Interfacing Actuators and Sensors to the Robot Controller*, discusses the interfacing of different actuators and sensors used in this robot with Tiva C Launchpad controller.

`Chapter 7`, *Interfacing Vision Sensors with ROS*, discusses interfacing of different vision sensors such as Kinect and Orbecc Astra that can be used in Chefbot for autonomous navigation.

`Chapter 8`, *Building ChefBot Hardware and Integration of Software*, discusses the complete construction of robot hardware and software in ROS in order to implement autonomous navigation.

`Chapter 9`, *Designing a GUI for a Robot Using Qt and Python*, discusses the development of a GUI to command the robot to move to a table in a hotel-like environment.

To get the most out of this book

The book is all about building a robot; to start with this book, you should have some hardware. The robot can be built from scratch or you can buy a differential drive configuration robot with encoder feedback. You should buy a controller board such as Texas instruments LaunchPad for embedded processing and should have at least a laptop/netbook for entire robot processing. In this book, we are using Intel NUC for robot processing, it is very compact in size and delivering high performance. For 3D vision, you should have a 3D sensor such as laser scanner, Kinect, or Orbecc Astra.

In the software section, you should have a good understanding in working with GNU/Linux commands and have good knowledge in Python too. You should install Ubuntu 16.04 LTS to work with the examples. If you have knowledge in ROS, OpenCV, OpenNI, and PCL, this will help. You have to install ROS Kinect/Melodic for these examples.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learning-Robotics-using-Python-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it from `https://www.packtpub.com/sites/default/files/downloads/LearningRoboticsusingPythonSecondEdition_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: " The first procedure is to create a world file and save it with the `.world` file extension."

A block of code is set as follows:

```
<xacro:include filename="$(find
 chefbot_description)/urdf/chefbot_gazebo.urdf.xacro"/>
 <xacro:include filename="$(find
 chefbot_description)/urdf/chefbot_properties.urdf.xacro"/>
```

Any command-line input or output is written as follows:

```
$ roslaunch chefbot_gazebo chefbot_empty_world.launch
```

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Getting Started with Robot Operating System

The main aim of this book is to teach you how to build an autonomous mobile robot from scratch. The robot will be programmed using ROS and its operations will be simulated using a simulator called Gazebo. You will also see the robot's mechanical design, circuit design, embedded programming, and high-level software programming using ROS in the upcoming chapters.

In this chapter, we will start with the basics of ROS, how to install it, how to write a basic application using ROS and Python, and the basics of Gazebo. This chapter will be the foundation of your autonomous robotics project. If you are already aware of the basics of ROS, and already have it installed on your system, you may skip this chapter. However, you can still go through this chapter later to refresh your memory as to the basics of ROS.

This chapter will cover the following topics:

- Introduction to ROS
- Installing ROS Kinetic on Ubuntu 16.04.3
- Introducing, installing, and testing Gazebo

Let's start programming robots using Python and Robot Operating System (ROS).

Technical requirements

To get the complete code that is mentioned in this chapter, you can clone the following link:

```
https://github.com/qboticslabs/learning_robotics_2nd_ed
```

Introduction to ROS

ROS is a software framework used for creating robotic applications. The main aim of the ROS framework is to provide the capabilities that you can use to create powerful robotics applications that can be reused for other robots. ROS has a collection of software tools, libraries, and collection of packages that makes robot software development easy.

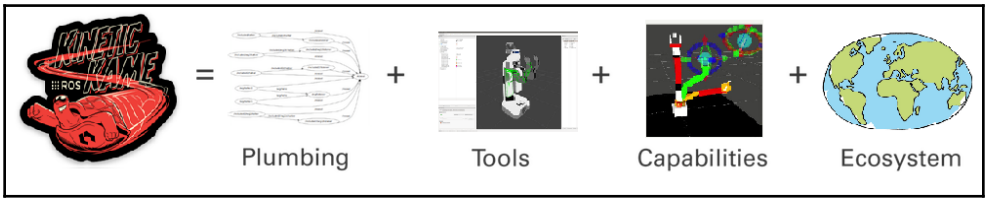
ROS is a complete open source project licensed under the BSD (`https://opensource.org/licenses/BSD-3-Clause`) license. We can use it for research and commercial applications. Even though ROS stands for Robot Operating System, it is not a real operating system. Rather, it is a meta-operating system, which provides the features of a real operating system. Here are the major features that ROS provides:

- **Message passing interface**: This is the core feature of ROS, and it enables interprocess communication. Using this message-passing capability, the ROS program can communicate with its linked systems and exchange data. We will learn more technical terms concerning the exchange of data between ROS programs/nodes in the coming sections and chapters.
- **Hardware abstraction**: ROS has a degree of abstraction that enables developers to create robot-agnostic applications. These kinds of application can be used with any robot; the developers need only worry about the underlying robot hardware.
- **Package management**: The ROS nodes are organized in packages called ROS packages. ROS packages consist of source codes, configuration files, build files, and so on. We create the package, build the package, and install the package. There is a build system in ROS that helps to build these packages. The package management in ROS makes ROS development more systematic and organized.
- **Third-party library integration:** The ROS framework is integrated with many third-party libraries, such as Open-CV, PCL, OpenNI, and so on. This helps developers to create all kinds of application in ROS.
- **Low-level device control**: When we work with robots, we may need to work with low-level devices, such as those that control I/O pins, sending data through serial ports, and so on. This can also be done using ROS.
- **Distributed computing**: The amount of computation required to process the data from robot sensors is very high. Using ROS, we can easily distribute the computation to a cluster of computing nodes. This distributes the computing power and allows you to process the data faster than you could using a single computer.

- **Code reuse**: The main goal of ROS is code reuse. Code reuse enables the growth of a good research and development community around the world. ROS executables are called nodes. These executables can be grouped into a single entity called a ROS package. A group of packages is called a meta package, and both packages and meta packages can be shared and distributed.
- **Language independence**: The ROS framework can be programmed using popular languages (such as Python, C++, and Lisp). The nodes can be written in any language and can communicate through ROS without any issues.
- **Easy testing**: ROS has a built-in unit/integration test framework called rostest to test ROS packages.
- **Scaling**: ROS can be scaled to perform complex computation in robots.
- **Free and open source**: The source code of ROS is open and it's absolutely free to use. The core part of ROS is licensed under a BSD license, and it can be reused in commercial and closed source products.

ROS is a combination of plumbing (message passing), tools, capabilities, and ecosystem. There are powerful tools in ROS to debug and visualize the robot data. There are inbuilt robot capabilities in ROS, such as robot navigation, localization, mapping, manipulation, and so on. They help to create powerful robotics applications.

The following image shows the ROS equation:



The ROS equation

Refer to `http://wiki.ros.org/ROS/Introduction` for more information on ROS.

ROS concepts

There are three main organizational levels in ROS:

- The ROS filesystem
- The ROS computation graph
- The ROS community

The ROS filesystem

The ROS filesystem mainly covers how ROS files are organized on the disk. The following are the main terms that we have to understand when working with the ROS filesystem:

- **Packages**: ROS packages are the individual unit of the ROS software framework. A ROS package may contain source code, third-party libraries, configuration files, and so on. ROS packages can be reused and shared.
- **Package manifests**: The manifests (`package.xml`) file will have all the details of the packages, including the name, description, license, and, more importantly, the dependencies of the package.
- **Message (msg) types**: Message descriptions are stored in the `msg` folder in a package. ROS messages are data structures for sending data through ROS's message-passing system. Message definitions are stored in a file with the `.msg` extension.
- **Service (srv) types**: Service descriptions are stored in the `srv` folder with the `.srv` extension. The `srv` file defines the request and response data structure for the service in ROS.

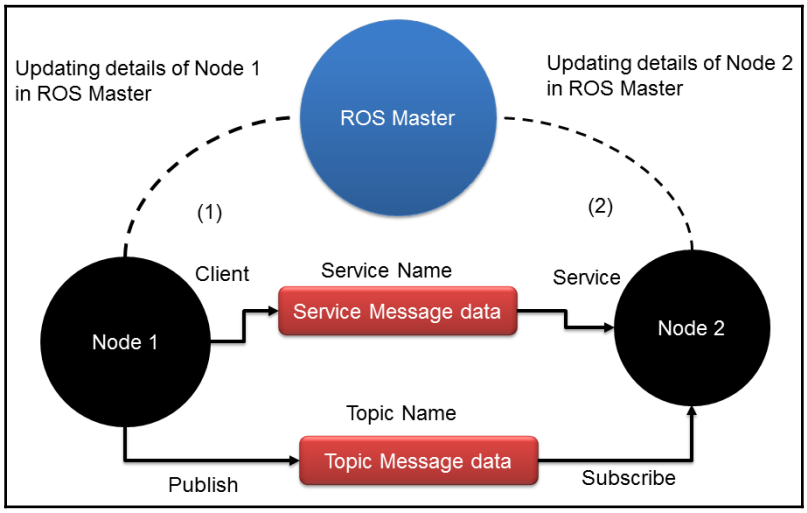
The ROS Computation Graph

The ROS Computation Graph is the peer-to-peer network of ROS systems that processes data. The basic features of ROS Computation Graph are nodes, ROS Master, the parameter server, messages, and services:

- **Nodes**: The ROS node is a process that uses ROS functionalities to process the data. A node basically computes. For example, a node can process the laser scanner data to check whether there is any collision. A ROS node is written with the help of an ROS client library (such as `roscpp` and `rospy`*), which will be discussed in the upcoming section*.

- **ROS Master**: The ROS nodes can connect to each other using a program called ROS Master. This provides the name, registration, and lookup to the rest of the computation graph. Without starting the master, the nodes will not find each other and send messages.
- **Parameter server**: The ROS parameters are static values that are stored in a global location called the parameter server. From the parameter server, all the nodes can access these values. We can even set the scope of the parameter server as private or public so that it can access one node or access all nodes.
- **ROS topics**: The ROS nodes communicate with each other using a named bus called ROS topic. The data flows through the topic in the form of messages. The sending of messages over a topic is called publishing, and receiving the data through a topic is called subscribing.
- **Messages**: A ROS message is a data type that can consist of primitive data types, such as integers, floating points, and Booleans. The ROS messages flow through the ROS topic. A topic can only send/receive one type of message at a time. We can create our own message definition and send it through the topics.
- **Services**: We have seen that the publish/subscribe model using ROS topics is a very easy way of communicating. This communication method is a one-to-many mode of communication, meaning that a topic can be subscribed to by any number of nodes. In some cases, we may also require a request/reply kind of interaction, which is usually used in distributed systems. This kind of interaction can be done using ROS services. The ROS services work in a similar way to ROS topics in that they have a message type definition. Using that message definition, we can send the service request to another node that provides the service. The result of the service will be sent as a reply. The node has to wait until the result is received from the other node.
- **Bags**: These are formats in which to save and play back the ROS topics. ROS bags are an important tool to log the sensor data and the processed data. These bags can be used later for testing our algorithm offline.

The following diagram shows how topics and services work between the nodes and the Master:



Communication between the ROS nodes and the ROS Master

In the preceding diagram, you can see two ROS nodes with the ROS Master in between them. One thing we have to remember is, before starting any nodes in ROS, you should start the ROS Master. The ROS Master acts like a mediator between nodes for exchanging information about other ROS nodes in order to establish communication. Say that Node 1 wants to publish a topic called `/xyz` with message type `abc`. It will first approach the ROS Master, and says `I am going to publish a topic called /xyz with message type abc and share its details`. When another node, say Node 2, wants to subscribe to the same topic of `/xyz` with the message type of `abc`, the Master will share the information about Node 1 and allocate a port to start communication between these two nodes directly without communicating with the ROS Master.

The ROS services works in the same way. The ROS Master is a kind of DNS server, which can share the node details when the second node requests a topic or service from the first node. The communication protocol ROS uses is TCPROS (`http://wiki.ros.org/ROS/TCPROS`), which basically uses TCP/IP sockets for the communication.

The ROS community level

The ROS community consists of ROS developers and researchers who can create and maintain packages and exchange new information related to existing packages, newly released packages, and other news related to the ROS framework. The ROS community provides the following services:

- **Distributions**: A ROS distribution has a set of packages that come with a specific version. The distribution that we are using in this book is ROS Kinetic. There are other versions available, such as ROS Lunar and Indigo, which has a specific version that we can install. It is easier to maintain the packages in each distribution. In most cases, the packages inside a distribution will be relatively stable.
- **Repositories**: The online repositories are the locations where we keep our packages. Normally, developers keep a set of similar packages called meta packages in a repository. We can also keep an individual package in a single repository. We can simply clone these repositories and build or reuse the packages.
- **The ROS wiki**: The ROS wiki is the place where almost all the documentation of ROS is available. You can learn about ROS, from its most basic concepts to the most advanced programming, using the ROS wiki (`http://wiki.ros.org`).
- **Mailing lists**: If you want to get updates regarding ROS, you can subscribe to the ROS mailing list (`http://lists.ros.org/mailman/listinfo/ros-users`). You can also get the latest ROS news from ROS Discourse (`https://discourse.ros.org`).
- **ROS answers**: This is very similar to the Stack Overflow website. You can ask questions related to ROS in this portal, and you might get support from developers across the world (`https://answers.ros.org/questions/`).

There are many other features available in ROS; you can refer to the ROS official website at www.ros.org for more information. For now, we will move on to the installation procedure of ROS.

Installing ROS on Ubuntu

As per our previous discussion, we know that ROS is a metaoperating system that is installed on a host system. ROS is completely supported on Ubuntu /Linux and in the experimental stages on Windows and OS X. Some of the latest ROS distributions are as follows:

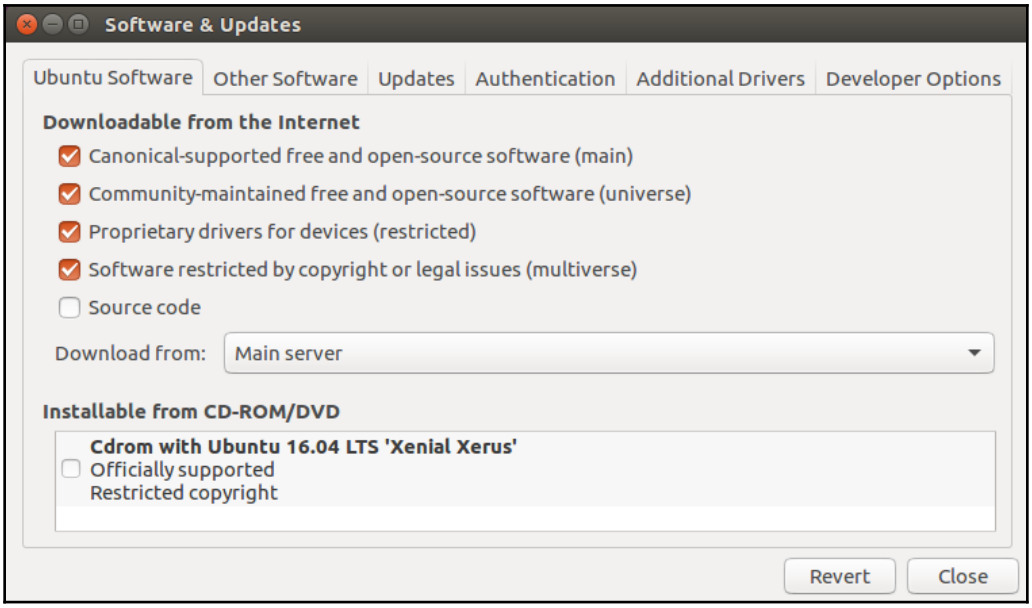
Distribution	Release date
ROS Melodic Morenia	May 23 2018
ROS Lunar Loggerhead	May 23 2017
ROS Kinetic Kame	May 23 2016
ROS Indigo Igloo	July 22 2014

We will now look at the installation procedure of the stable, long-term support (LTS) distribution of ROS called Kinetic on Ubuntu 16.04.3 LTS. ROS Kinetic Kame will be primarily targeted at Ubuntu 16.04 LTS. You can also find instructions to set up ROS in the latest LTS Melodic Morenia on Ubuntu 18.04 LTS after looking at the following instructions. If you are a Windows or OS X user, you can install Ubuntu in a VirtualBox application before installing ROS on it. The link to download VirtualBox is `https://www.virtualbox.org/wiki/Downloads`.

You can find the complete instructions for doing this at `http://wiki.ros.org/kinetic/Installation/Ubuntu`.

The steps are as follows:

1. Configure your Ubuntu repositories to allow **restricted**, **universe**, and **multiverse** downloadable files. We can configure it using Ubuntu's **Software & Update** tool. We can get this tool by simply searching on the Ubuntu Unity search menu and ticking the shown in the following screenshot:



Ubuntu's Software & Update tool

2. Set up your system to accept ROS packages from `packages.ros.org`. ROS Kinetic is supported only on Ubuntu 15.10 and 16.04. The following command will store `packages.ros.org` in Ubuntu's `apt` repository list:

```
    $ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

3. Next, we have to add apt-keys. An `apt-key` is used to manage the list of keys used by apt to authenticate the packages. Packages that have been authenticated using these keys will be considered trusted. The following command will add apt-keys for the ROS packages:

```
    sudo apt-key adv --keyserver hkp://ha.pool.sks-
keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

4. After adding the apt-keys, we have to update the Ubuntu package list. The following command will add and update the ROS packages, along with the Ubuntu packages:

```
$ sudo apt-get update
```

5. After updating the ROS packages, we can install the packages. The following command will install all the necessary packages, tools, and libraries of ROS:

```
$ sudo apt-get install ros-kinetic-desktop-full
```

6. We may need to install additional packages even after the desktop full installation. Each additional installation will be mentioned in the appropriate section. The desktop full install will take some time. After the installation of ROS, you will almost be done. The next step is to initialize `rosdep`, which enables you to easily install the system dependencies for ROS source packages:

```
$ sudo rosdep init
$ rosdep update
```

7. To access ROS's tools and commands on the current bash shell, we can add ROS environmental variables to the `.bashrc` file. This will execute at the beginning of each bash session. The following is a command to add the ROS variable to `.bashrc`:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
```

   The following command will execute the `.bashrc` script on the current shell to generate the change in the current shell:

```
source ~/.bashrc
```

8. A useful tool to install the dependency of a package is `rosinstall`. This tool has to be installed separately. It enables you to easily download many source trees for the ROS package with one command:

```
    $ sudo apt-get install python-rosinstall python-rosinstall-
generator python-wstool build-essential
```

The installation of the latest LTS Melodic is similar to the preceding instructions. You can install Melodic along with Ubuntu 18.04 LTS. You can find the complete instructions at `http://wiki.ros.org/melodic/Installation/Ubuntu`.

After the installation of ROS, we will discuss how to create a sample package in ROS. Before creating the package, we have to create a ROS workspace. The packages are created in the ROS workspace. We will use the catkin build system, which is a set of tools that is used to build packages in ROS. The catkin build system generates an executable or shared library from the source code. ROS Kinetic uses the catkin build system to build packages. Let's look at what catkin is.

Introducing catkin

Catkin is the official build system of ROS. Before catkin, ROS used the **rosbuild** system to build packages. Its replacement is catkin on the latest ROS version. Catkin combines CMake macros and Python scripts to provide the same normal workflow that CMake produces. Catkin provides better distribution of packages, better cross-compilation, and better portability than the rosbuild system. For more information, refer to wiki.ros.org/catkin.

Catkin workspace is a folder where you can modify, build, and install catkin packages.

Let's check how to create an ROS catkin workspace.

The following command will create a parent directory called `catkin_ws` and a subfolder called `src`:

```
$ mkdir -p ~/catkin_ws/src
```

Switch directory to the `src` folder using the following command. We will create our packages in the `src` folder:

```
$ cd ~/catkin_ws/src
```

Initialize the catkin workspace using the following command:

```
$ catkin_init_workspace
```

After you initialize the catkin workspace, you can simply build the package (even if there is no source file) using the following command:

```
$ cd ~/catkin_ws/
$ catkin_make
```

The `catkin_make` command is used to build packages inside the `src` directory. After building the packages, we will see a `build` and `devel` folder in `catkin_ws` The executables are stored in the `build` folder. In the `devel` folder, there are shell script files to add the workspace on the ROS environment.

Creating a ROS package

In this section, we will look at how to create a sample package that contains two Python nodes. One of the nodes is used to publish a **Hello World** string message on a topic called `/hello_pub` and the other node will subscribe to this topic.

A catkin ROS package can be created using the `catkin_create_pkg` command in ROS.

The package is created inside the `src` folder that we created during the creation of the workspace. Before creating the packages, switch to the `src` folder using the following command:

```
$ cd ~/catkin_ws/src
```

The following command will create a `hello_world` package with `std_msgs` dependencies, which contain standard message definitions. The `rospy` is the Python client library for ROS:

```
$ catkin_create_pkg hello_world std_msgs rospy
```

This is the message we get upon a successful creation:

```
Created file hello_world/package.xml
Created file hello_world/CMakeLists.txt
Created folder hello_world/src
Successfully created files in /home/lentin/catkin_ws/src/hello_world.
Please adjust the values in package.xml.
```

After the successful creation of the `hello_world` package, we need to add two Python nodes or scripts to demonstrate the subscribing and publishing of topics.

First, create a folder named `scripts` in the `hello_world` package using the following command:

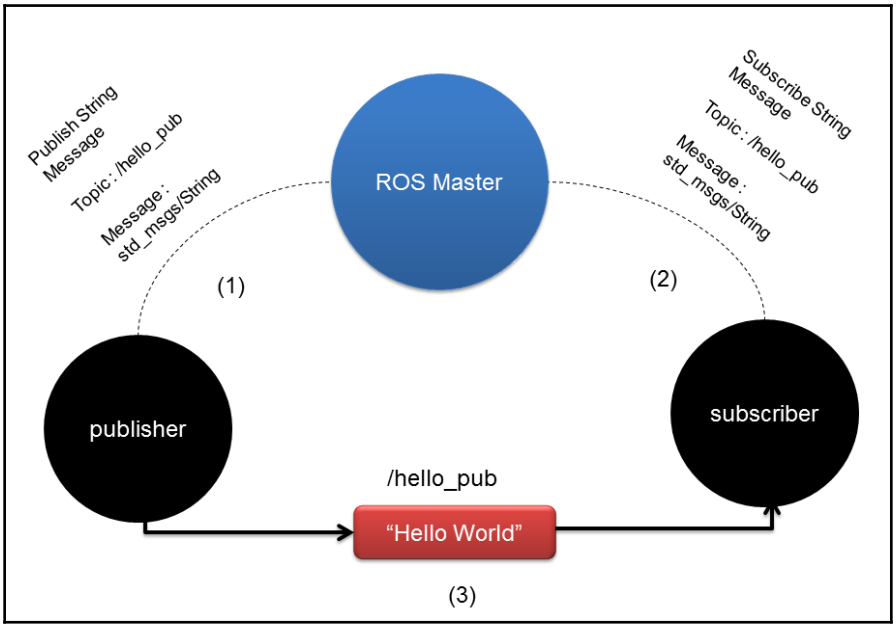
```
$ mkdir scripts
```

Switch to the `scripts` folder and create a script named `hello_world_publisher.py` and another script called `hello_world_subscriber.py` to publish and subscribe to the **hello world** message. The following section covers the code and function of these scripts or nodes:

Hello_world_publisher.py

The `hello_world_publisher.py` node basically publishes a greeting message called **hello world** to a topic called `/hello_pub`. The greeting message is published to the topic at a rate of 10 Hz.

Here is a diagram that shows how the interaction between the two ROS nodes works:



Communication between the publisher and subscriber node

The full code of this book is available at `https://github.com/qboticslabs/learning_robotics_2nd_ed`.

The step-by-step explanation of how this code works is as follows:

1. We need to import `rospy` if we are writing a ROS Python node. It contains Python APIs to interact with ROS topics, services, and so on.
2. To send the **hello world** message, we have to import a `String` message data type from the `std_msgs` package. The `std_msgs` package has the message definition for standard data types. We can import using the following lines of code:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
```

3. The following line of code creates a publisher object to a topic called `hello_pub`. The message type is `String` and the `queue_size` value is `10`. If the subscriber is not fast enough to receive the data, we can use the `queue_size` option to buffer it:

```
def talker():
    pub = rospy.Publisher('hello_pub', String, queue_size=10)
```

4. The following line of code initializes a ROS node. It will also assign a name to the node. If two nodes are running with the same node name, one will shut down. If we want to run both, use anonymous=`True` flag as shown in the following code:

```
rospy.init_node('hello_world_publisher', anonymous=True)
```

5. The following line creates a rate object called `r`. Using a `sleep()` method in the `Rate` object, we can update the loop at the desired rate. Here, we are giving the rate the value of `10`:

```
r = rospy.Rate(10) # 10hz
```

6. The following loop will check whether `rospy` constructs the `rospy.is_shutdown()` flag. Then, it executes the loop. If we click on *Ctrl* + *C*, this loop will exit.

   Inside the loop, a **hello world** message is printed on the Terminal and published on the `hello_pub` topic with a rate of 10 Hz:

```
while not rospy.is_shutdown():
    str = "hello world %s"%rospy.get_time()
    rospy.loginfo(str)
    pub.publish(str)
    r.sleep()
```

7. The following code has Python `__main__` check and calls the `talker()` function. The code will keep on executing the talker(), and when Ctrl + C is pressed the node will get shut down:

```
if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException: pass
```

After publishing the topic, we will see how to subscribe to it. The following section covers the code needed to subscribe to the `hello_pub` topic.

Hello_world_subscriber.py

The subscriber code is as follows:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
```

The following code is a callback function that is executed when a message reaches the `hello_pub` topic. The `data` variable contains the message from the topic, and it will print using `rospy.loginfo()`:

```
def callback(data):
    rospy.loginfo(rospy.get_caller_id()+"I heard %s",data.data)
```

The following steps will start the node with a `hello_world_subscriber` name and start subscribing to the `/hello_pub` topic:

1. The data type of the message is `String`, and when a message arrives on this topic, a method called callback will be called:

```
def listener():
   rospy.init_node('hello_world_subscriber',
      anonymous=True)
   rospy.Subscriber("hello_pub", String, callback)
```

2. The following code will keep your node from exiting until the node is shut down:

```
rospy.spin()
```

3. The following is the main section of the Python code. The main section will call the `listener()` method, which will subscribe to the `/hello_pub` topic:

```
if __name__ == '__main__':
    listener()
```

4. After saving two Python nodes, you need to change the permission to executable using the `chmod` commands:

```
chmod +x hello_world_publisher.py
chmod +x hello_world_subscriber.py
```

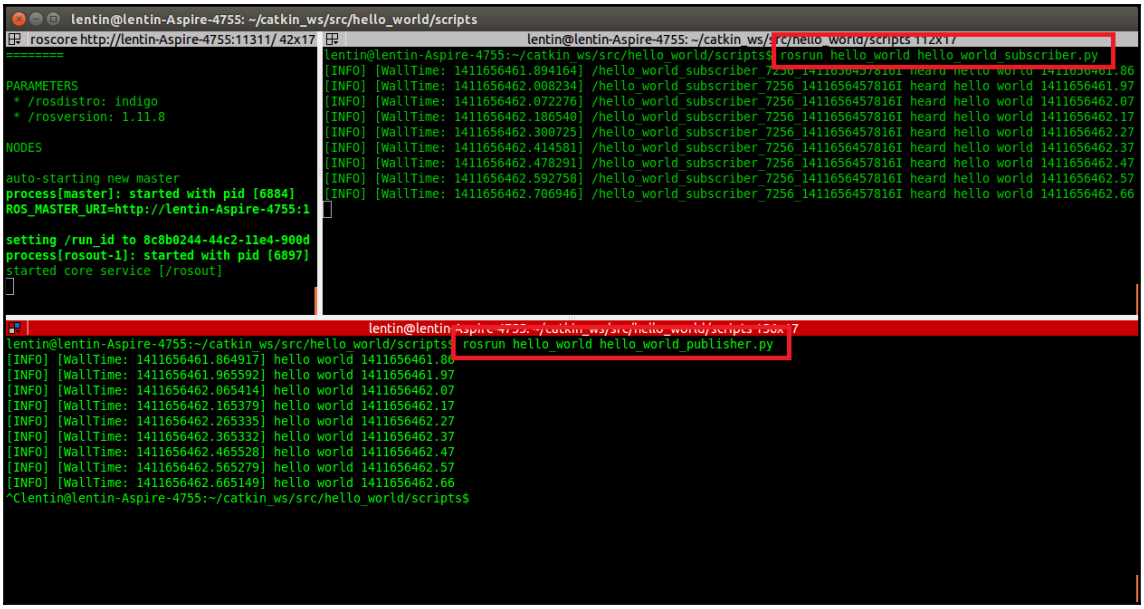
5. After changing the file permission, build the package using the `catkin_make` command:

```
cd ~/catkin_ws
catkin_make
```

6. The following command adds the current ROS workspace path in all terminals so that we can access the ROS packages inside this workspace:

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

The following is the output of the subscriber and publisher nodes:



Output of the hello world node

1. First, we need to run `roscore` before starting the nodes. The `roscore` command or ROS master is needed to communicate between nodes. So, the first command is as follows:

```
$ roscore
```

2. After executing `roscore`, run each node using the following commands:

    - The following command will run the publisher:

    ```
    $ rosrun hello_world hello_world_publisher.py
    ```

    - The following command will run the subscriber node. This node subscribes to the `hello_pub` topic, as shown in the following code:

    ```
    $ rosrun hello_world hello_world_subscriber.py
    ```

We have covered some of the basics of ROS. Now, we will see what Gazebo is and how we can work with Gazebo using ROS.

Introducing Gazebo

Gazebo is a free and open source robot simulator in which we can test our own algorithms, design robots, and test robots in different simulated environments. Gazebo can accurately and efficiently simulate complex robots in indoor and outdoor environments. Gazebo is built with a physics engine with which we can create high-quality graphics and rendering.

The features of Gazebo are as follows:

- **Dynamic simulation**: Gazebo can simulate the dynamics of a robot using physics engines such as **Open Dynamics Engine** (**ODE**). (`http://opende.sourceforge.net/`), Bullet (`http://bulletphysics.org/wordpress/`), Simbody (`https://simtk.org/home/simbody/`), and DART (`http://dartsim.github.io/`).
- **Advanced 3D graphics**: Gazebo provides high-quality rendering, lighting, shadows, and texturing using the OGRE framework (`http://www.ogre3d.org/`).
- **Sensor support**: Gazebo supports a wide range of sensors, including laser range finders, Kinect-style sensors, 2D/3D cameras, and so on. We can also use it to simulate noise to test audio sensors.
- **Plugins**: We can develop custom plugins for the robot, sensors, and environmental controls. Plugins can access Gazebo's API.
- **Robot models**: Gazebo provides models for popular robots, such as PR2, Pioneer 2 DX, iRobot Create, and TurtleBot. We can also build custom models of robots.

- **TCP/IP transport**: We can run simulations on a remote machine and a Gazebo interface through a socket-based message-passing service.
- **Cloud simulation**: We can run simulations on the cloud server using the CloudSim framework (`http://cloudsim.io/`).
- **Command-line tools**: Extensive command-line tools are used to check and log simulations.

Installing Gazebo

Gazebo can be installed as a standalone application or an integrated application along with ROS. In this chapter, we will use Gazebo along with ROS to simulation a robot's behavior and to test our written code using the ROS framework.

If you want to try the latest Gazebo simulator yourself, you can follow the steps given at `http://gazebosim.org/download`.

To work with Gazebo and ROS, we don't need to install them separately because Gazebo comes with the ROS desktop full installation.

The ROS package that integrates Gazebo with ROS is called `gazebo_ros_pkgs`. There are wrappers around the standalone Gazebo. This package provides the necessary interface to simulate a robot in Gazebo using ROS message services.

The complete `gazebo_ros_pkgs` can be installed in ROS Indigo using the following command:

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

Testing Gazebo with the ROS interface

Assuming that the ROS environment is properly set up, we can start `roscore` before starting Gazebo using the following command:

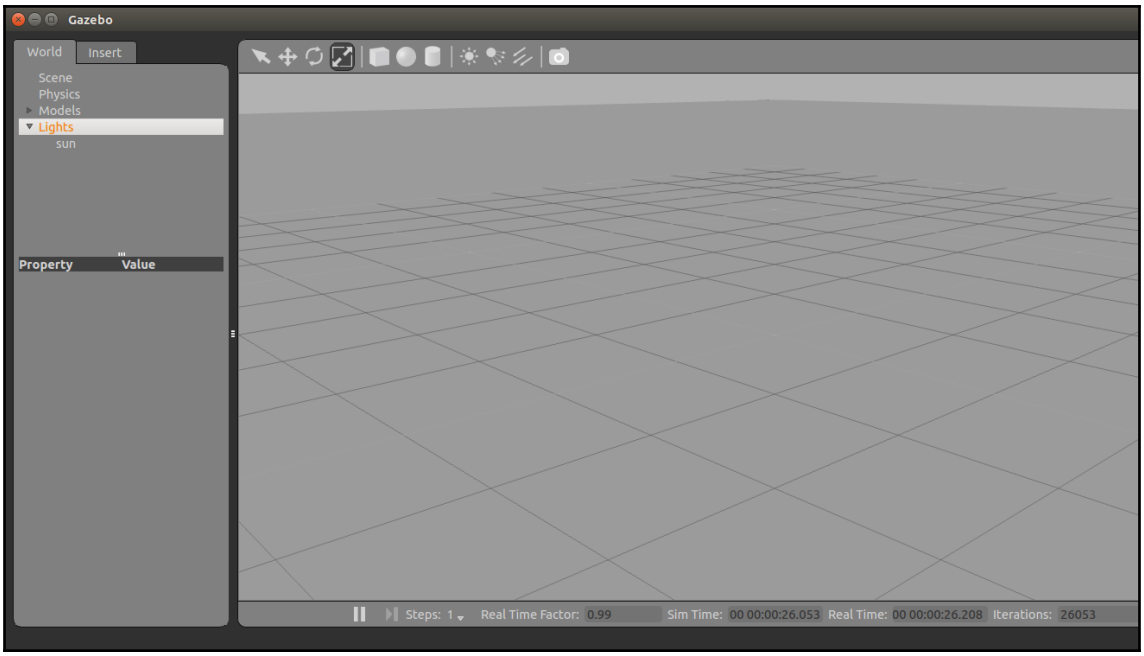
```
$ roscore
```

The following command will run Gazebo using ROS:

```
$ rosrun gazebo_ros gazebo
```

Gazebo runs two executables-the Gazebo server and the Gazebo client. The Gazebo server will execute the simulation process and the Gazebo client can be the Gazebo GUI. Using the previous command, the Gazebo client and server will run in parallel.

The Gazebo GUI is shown in the following screenshot:



The Gazebo simulator

After starting Gazebo, the following topics will be generated:

```
$ rostopic list
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
```

We can run the server and client separately using the following commands:

- Run the Gazebo server using the following command:

```
$ rosrun gazebo_ros gzserver
```

- Run the Gazebo client using the following command:

```
$ rosrun gazebo_ros gzclient
```

Summary

This chapter was an introduction to Robot Operating System. The main goal of this chapter was to give you an overview of ROS, its features, how to install it, the basic concepts of ROS, and how to program it using Python. Along with this, we have looked at a robotics simulator called Gazebo, which can work with ROS. We have seen how to install and run Gazebo. In the next chapter, we will look at the basic concepts of differential drive robots.

Questions

1. What are the important features of ROS?
2. What are the different levels of concepts in ROS?
3. What is ROS catkin build system?
4. What are ROS topics and messages?
5. What are the different concepts of the ROS Computation Graph?
6. What is the main function of the ROS Master?
7. What are the important features of Gazebo?

2
Understanding the Basics of Differential Robots

In the previous chapter, we discussed the basics of ROS, how to install it, and the basics of the Gazebo robot simulator. As we have already mentioned, we are going to create an autonomous wheeled robot from scratch. The robot that we are going to design is a differential drive robot, which involves having two wheels on opposite sides of the robot chassis, enabling the robot's direction to be adjusted by changing the speed of each of the two wheels.

It will be good to understand the basic ideas and terminology behind differential wheel robots before programming the robot. This chapter will give you an idea of how to analyze the robot mathematically and how to solve the robot's kinematics equation. The kinematics equation helps you to predict the robot's position from its sensor data.

In this chapter, we will cover the following topics:

- Mathematical modeling of differential drive robots
- Forward kinematics of differential drive robots
- Inverse kinematics of differential drive robots

Mathematical modeling of the robot

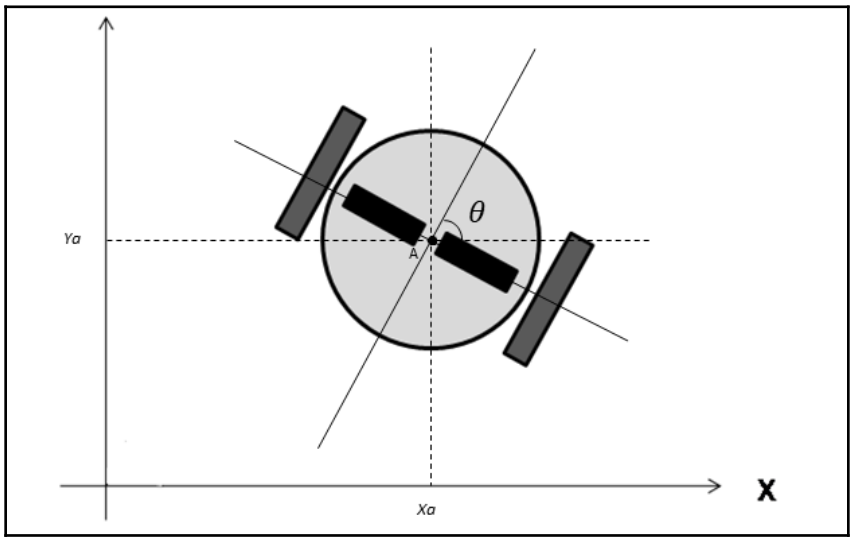
An important part of a mobile robot is its steering system. This will help the robot to navigate its environment. One of the simplest and most cost-effective steering systems is the differential drive system. A differential drive robot consists of two main wheels mounted on a common axis that are controlled by separate motors. A differential drive system/steering system is a nonholonomic system, which means that it has constraints for the changing the robot's pose.

A car is an example of a nonholonomic system, as it cannot change its position without changing its pose. Let's look at how this type of robot works and how we can model the robot in terms of its mathematics.

Introduction to the differential drive system and robot kinematics

Robot kinematics is the study of the mathematics of motion without considering the forces that affect the motion. It mainly deals with the geometric relationships that govern the system. **Robot dynamics** is the study of motion in robots in which all the forces involved in the robots' movement are modeled.

A mobile robot or vehicle has six **degrees of freedom** (**DOFs**) expressed by the pose (x, y, z, roll, pitch, and yaw). These DOFs consist of the position (x, y, z) and attitude (roll, pitch, and yaw). **Roll** refers to sidewise rotation, **pitch** refers to forward and backward rotation, and **yaw** (called the heading or orientation) refers to the direction in which the robot moves in the x-y plane. The differential drive robot moves from x to y in the horizontal plane, so the 2D pose contains mainly x, y, and θ, where θ is the heading of the robot that points in the robot's forward direction. This information is sufficient to describe a differential robot pose:



The pose of the robot in x, y, and θ in the global coordinate system

In a differential drive robot, the motion can be controlled by adjusting the velocity of two independently controlled motors on the left-hand and right-hand side, named V-left and V-right, respectively. The following image shows a couple of popular differential drive robots available on the market:

Robot Roomba (https://en.wikipedia.org/wiki/IRobot)

The Roomba series of autonomous vacuum cleaners is a popular differential robot from iRobot.

Pioneer 3-DX (http://robots.ros.org/pioneer-3-dx/)

The Pioneer 3-DX is a popular differential drive research platform from Omron Adept Mobile Robots.

Forward kinematics of a differential robot

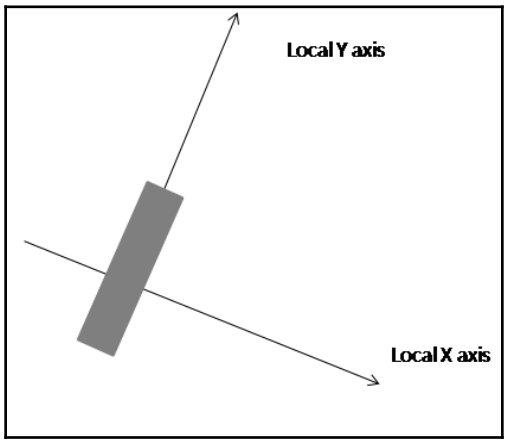
The forward kinematics equations for a robot with a differential drive system are used to solve the following problem:

If a robot is standing in a position (x, y, θ) at time t, determine the pose (x', y', θ') at $t + \delta t$ given the control parameters *V-left* and *V-right*.

This technique can be calculated by the robot to follow a particular trajectory.

Explanations of the forward kinematics equation

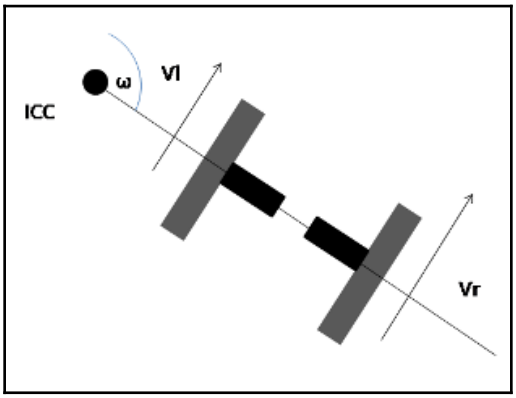
We can start by formulating a solution for forward kinematics. The following figure is an illustration of one of the wheels of the robot:



A single wheel of the robot rotating along the local y-axis

The motion around the **y-axis** is known as the roll; everything else is called the slip. Let's assume that no slip occurs in this case. When the wheel completes one full rotation, it covers a distance of $2\pi r$, where r is the radius of the wheel. We will assume that the movement is two-dimensional. This means that the surface is flat and even.

When the robot is about to perform a turning motion, the robot must rotate around a point that lies along its common left and right wheel axis. The point that the robot rotates around is known as the **ICC-the instantaneous center of curvature**. The ICC is located outside the robot. The following diagram shows the wheel configuration of the differential drive robot in relation to its ICC:



Wheel configuration for a robot with a differential drive

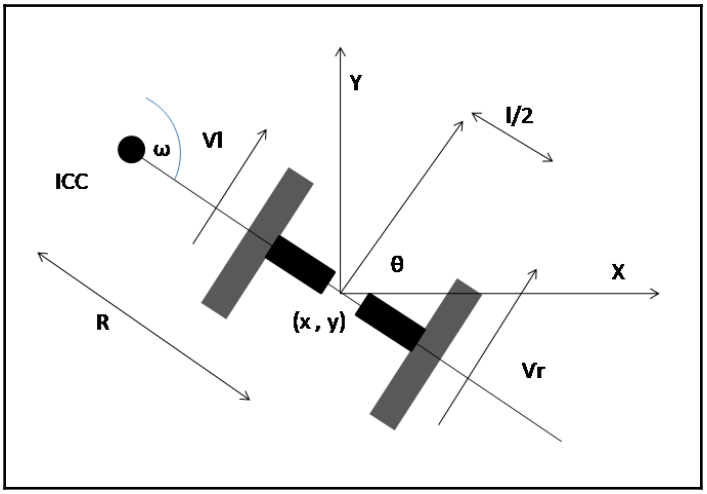
The central concept for the derivation of the kinematic equation is the ω angular velocity of the robot. Each wheel on the robot rotates around the ICC along the circumference of a circle with a wheel radius of r.

The speed of the wheel is $v = 2 \pi r / T$, where T is the time taken to complete one full turn around the ICC. The ω angular velocity is defined as $2 \pi / T$, and typically has the unit of radians (or degrees) per second. Combining the equations for v and w yields $\omega = 2 \pi / T$, and we can conclude the following:

$$v = r \omega \qquad (1)$$

Equation of linear velocity

A detailed model of the differential drive system is shown in the following diagram:



Detailed diagram of the differential drive system

If we apply the previous equation to both wheels, the result will be the same-that is, ω:

$$\omega(R+l/2)=Vr \quad (2)$$
$$\omega(R-l/2)=Vl \quad (3)$$

Differential drive wheel equation

Here, R is the distance between the ICC and the midpoint of the wheel axis and l is the length of the wheel axis. After solving ω and R, we get the following result:

$$R=l/2(Vl+Vr)/(Vr-Vl) \quad (4)$$
$$\omega=(Vr-Vl)/l \quad (5)$$

Equation to find the distance from the ICC to the center of the robot and the angular velocity of the robot

The previous equation is useful for solving the forward kinematics problem. Suppose the robot moves with an angular velocity of ω for δt seconds. This will result in the robot's orientation or heading changed to the following:

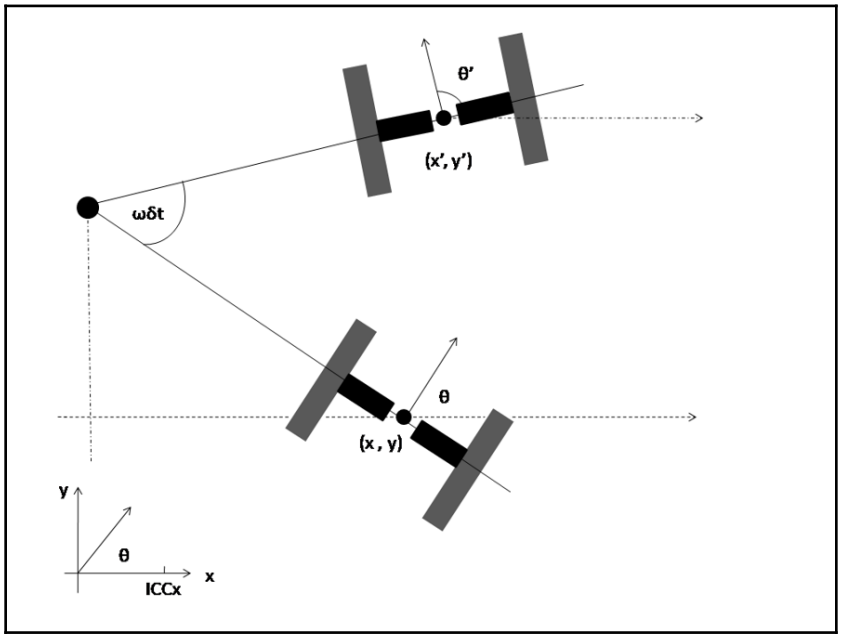
$$\theta' = \omega \delta t + \theta \quad (6)$$

Equation to find the change in heading

Here, the center of the ICC rotation is given by basic trigonometry as the following:

$$ICC = \left[ICC_x, ICC_y \right] = \left[x - R\,sin\theta, y + R\,cos\theta \right] \quad (7)$$

Equation to find the ICC



Rotating the robot ωδt degrees around the ICC

Given a starting position (x, y), the new position (x', y') can be computed using the 2D rotation matrix. The rotation around the ICC with the angular velocity ω for δt seconds yields the following position at the time $t + \delta t$:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos(\omega\delta t) & -sin(\omega\delta t) \\ sin(\omega\delta t) & cos(\omega\delta t) \end{pmatrix} \begin{pmatrix} x - ICC_x \\ y - ICC_y \end{pmatrix} + \begin{pmatrix} ICC_x \\ ICC_y \end{pmatrix} \quad (8)$$

Equation to calculate the new position of the robot

The new pose (x', y', and θ') can be computed from equation (6) and (8), given ω, δt, and R.

ω can be computed from equation (5); Vr and Vl are often more difficult to measure accurately. Instead of measuring the velocity, the rotation of each wheel can be measured using sensors called **wheel encoders**. The data from the wheel encoders is the robot's **odometry** values. These sensors are mounted on the wheel axes and deliver binary signals for each degree that the wheel rotates (each degree may be in the order of 0.1 mm). We will look at the detailed workings of the wheel encoders in *Chapter 6, Interfacing Actuators and Sensors to the Robot Controller*. These signals are fed to a counter so that $v\delta t$ *is* the distance traveled from the time t to $t + \delta t$. We can write the following:

$n * step = v\delta t$

From this, we can calculate v:

$$v = n * step / \delta t \quad (9)$$

Equation calculating the linear velocity from the encoder data

If we insert equation (9) in equations (3) and (4), we get the following result:

$$R = l/2(Vl + Vr)/(Vr - Vl) = l/2(nl + nr)/(nr - nl) \quad (10)$$

$$w\delta t = (Vr - Vl)\delta t/l = (nr - nl) * step/l \quad (11)$$

Equation to calculate R from the encoder values

Here, ***nl*** and ***nr*** are the encoder counts of the left and right wheels. ***Vl*** and ***Vr*** are the speeds of the left and right wheels respectively. Thus, the robot stands in pose (x, y, θ) and moves ***nl*** and ***nr*** counts during a time frame of δt; the new pose (x', y', θ') is given by calculating the following:

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} \cos(\omega\delta t) & -\sin(\omega\delta t) & 0 \\ \sin(\omega\delta t) & \cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x\text{-}ICC_x \\ y\text{-}ICC_y \\ \theta \end{pmatrix} + \begin{pmatrix} ICC_x \\ ICC_y \\ \omega\delta t \end{pmatrix} \quad (12)$$

Equation to calculate the robot's position from the encoder values

where,

$$R = l/2(nl+nr)/(nr-nl) \quad (13)$$

$$\omega\delta t = (nr-nl)*step/l \quad (14)$$

$$ICC = [x - R\sin\theta, y + R\cos\theta] \quad (15)$$

Equation to calculate the ICC and other parameters from the encoder values

The derived kinematic equation depends mainly on the design and geometry of the robot. Different designs can lead to different equations.

Inverse kinematics

The forward kinematics equation provides an updated pose at a given wheel speed. We can now think about the inverse problem.

Stand in pose (x, y, θ) at time t and determine the *V-left* and *V-right* control parameters so that the pose at time $t + \delta t$ is (x', y', θ').

In differential drive systems, this problem may not always have a solution because this kind of robot can't be moved to any pose by simply setting the wheel velocity. It's because of the nonholonomic robots' constraints.

In nonholonomic robots, there are some ways to increase the constrained mobility if we allow a sequence of different (*V-left*, *V-right*) movements. If we insert the values from equations (12) and (15), we can identify some special movements that we can program:

- If *V-right* = *V-left* => $nr = nl$ => $R = \infty$, $\omega\delta T = 0$ =>, this means that the robot moves in a straight line and θ remains the same
- If *V-right* = *-V-left* => $nr = -nl$ => $R=0$, $\omega\delta t = 2nl * step / l$ and $ICC = \left[ICC_x, ICC_y\right] = \left[x, y\right]$ => $x' = x$, $y' = y$, $\theta' = \theta + \omega\delta t$ =>, this means the robot rotates in position around ICC-that is, any θ is reachable, while (x, y) remains unchanged

Combining these operations, the following steps can be used to reach any target pose from the starting pose:

1. Rotate until the robot's orientation coincides with the line leading from the starting position to the target position, *V-right* = *-V-left* = *V-rot*.
2. Drive straight until the robot's position coincides with the target position, *V-right* = *V-left* = *V-ahead*.
3. Rotate until the robot's orientation coincides with the target orientation, *V-right* = *-V-left* = *V-rot*. Here, *V-rot* and *V-ahead* can be chosen arbitrarily.

We will see how we can implement the kinematics equation of the robot using ROS in the upcoming chapters.

Summary

This chapter was about the fundamental concepts of differential drive robots, and looked at how you can derive the kinematics equations of such robots. At the start of the chapter, we saw the basics of differential drive robots, and then we discussed the forward kinematics equations that are used in these robots. These equations were explained using diagrams. After looking at forward kinematics equations, we looked at the inverse kinematics equations for differential drive robots. We also looked at the basics of inverse kinematics equations.

In the next chapter, we will see how we can create a simulation of the autonomous mobile robot using ROS and Gazebo.

Questions

1. What are holonomic and nonholonomic configurations?
2. What are robot kinematics and dynamics?
3. What is the ICC of a differential drive robot?
4. What is the forward kinematic equation in a differential robot?
5. What is the inverse kinematic equation in a differential robot?

Further information

Refer to `http://www8.cs.umu.se/~thomash/reports/KinematicsEquationsForDifferentialDriveAndArticulatedSteeringUMINF-11.19.pdf` for more information on kinematic equations.

3
Modeling the Differential Drive Robot

In this chapter, we will look at how to model the differential drive robot and create the URDF model of this robot in ROS. The main use case of the robot that we are going to design in this chapter is to serve food and drinks in hotels and restaurants. The robot is named *Chefbot*. We will cover the complete modeling of this robot in this chapter.

We will look at the CAD design of various mechanical components used in this robot and how to assemble them. We will look at the 2D and 3D CAD design of this robot and will discuss how to create the URDF model of the robot.

The actual robot model deployed in hotels may be big in size, but here we intend to build a miniature version for testing our software. If you are interested in building a robot from scratch, this chapter is for you. If you are not interested in building the robot, you can choose some robotic platforms, such as Turtlebot, which are already available on the market, to work with this book.

To build the robot hardware, first we need to get the requirements of the robot. After getting the requirements, we can design it and draw the model in 2D CAD tools to manufacture the robot parts. The 3D modeling of the robot will give us more idea about the looks of the robot. After the 3D modeling, we can convert the design into a URDF model that can be used along with ROS.

The following topics will be covered in the chapter:

- Designing robot parameters from the given specification
- Designing robot body parts in 2D using LibreCAD
- Designing a 3D robot model using Blender and Python
- Creating a URDF model for Chefbot
- Visualizing the Chefbot model in Rviz

Technical requirements

To test the application and code in this chapter, you need an Ubuntu 16.04 LTS PC/laptop with ROS Kinetic installed

Requirements of a service robot

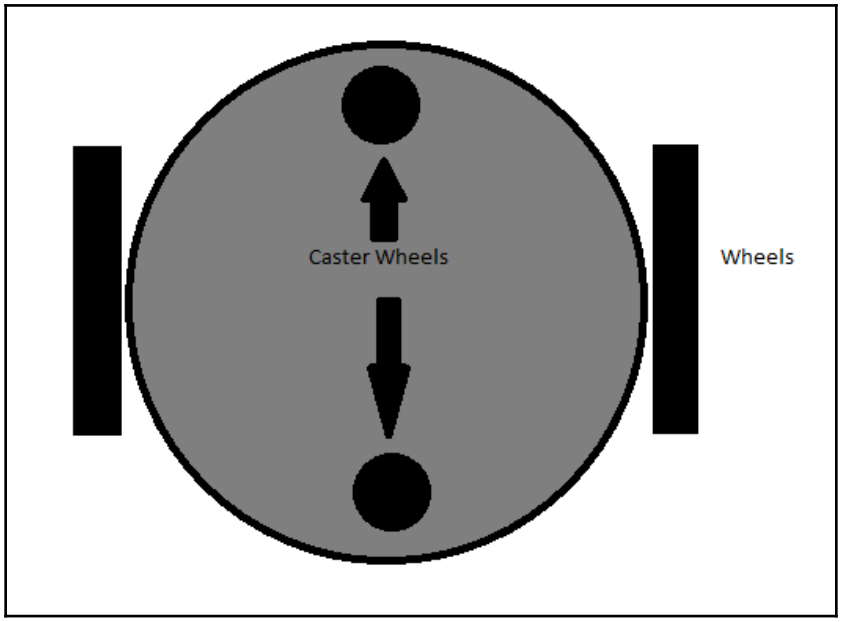
Before designing any robotic system, the first procedure is to identify the requirements of the system. The following are the set of robot design requirements to be met by this robot. This includes hardware and software requirements:

- The robot should have a provision to carry food
- The robot should carry a maximum payload of 2 kg
- The robot should move at a speed between 0.25 m/s and 0.35 m/s
- The ground clearance of the robot should be greater than 3 cm
- The robot has to work for 2 hours continuously
- The robot should be able to move and supply food to any table, avoiding obstacles
- The robot height can be between 80 cm and 100 cm.
- The robot should be of low cost (less than 500 USD)

Now we have the design requirements, such as payload, speed, ground clearance, robot height, cost of the robot, and the capabilities to be implemented in the robot, we can design a robot body and select components that are matching the aforementioned requirements. Let's discuss the robot mechanism we can use to match these requirements.

Robot drive mechanism

One of the cost-effective solutions for mobile robot navigation is the differential drive system. It's one of the simplest drive mechanisms for a mobile robot and is mainly indented for indoor navigation. The **differential drive robot** consists of two wheels mounted on a common axis controlled by two separate motors. There are two supporting wheels called caster wheels. This ensures stability and weight distribution of the robot. The following diagram shows a typical differential drive system:



Differential drive system

The next step is to select the mechanical components of this robot drive system, that is, mainly motors, wheels, and robot chassis. Based on the requirements, we will first discuss how to select the motor.

Selection of motors and wheels

Motors are selected after a look at the specifications. Some of the important parameters for motor selection are torque and RPM. We can compute these values from the given requirements.

Calculation of RPM of motors

The range of speed required for this robot is from 0.25 to 0.35m/s. We can take the maximum speed of this robot as 0.35 m/s for the design. Take the diameter of the wheel as 9 cm, because according to the requirement, the ground clearance should be greater than 3 cm and we will fix the robot body in same level as the motor shaft. In that case, we will get more ground clearance.

Using the following equation, we can calculate the RPM of the motors:

*RPM = ((60 * Speed /(3.14 * Diameter of Wheel)*

*RPM = (60 * 0.35)/(3.14 * 0.09) = 21 / 0.2826 = 74 RPM*

You can also take a look at `http://www.robotshop.com/blog/en/vehicle-speed-rpm-and-wheel-diameter-finder-9786` for the computation.

The calculated RPM with a 9 cm diameter wheel and 0.35 m/s speed is 74 RPM. We can consider 80 RPM as the standard value.

Calculation of motor torque

Let's calculate the torque required to move the robot:

1. Number of wheels = Four wheels including two caster wheels.
2. Number of motors = Two.
3. Let's assume the coefficient of friction is 0.6 and radius of the wheel is 4.5 cm.
4. Take the total weight of robot = weight of robot + payload = (W = mg) = (~100 N + ~20 N) W= ~ 150 N, whereas total mass = 12 Kg.
5. The weight acting on the four wheels can be written as *2 * N1 + 2 * N2 = W*; that is, *N1* is the weight acting on each caster wheel and *N2* on motor wheels.
6. Assume that the robot is stationary. The maximum torque is required when the robot starts moving. It should also overcome friction.
7. We can write the frictional force as robot torque = 0 until the robot moves. If we get the robot torque in this condition, we get the maximum torque as follows:
    - $\mu * N * r - T = 0$, where μ is the coefficient of friction, *N* is the average weight acting on each wheel, *r* is the radius of wheels, and *T* is the torque.
    - *N = W/2* (in the robot, actuation is only for two wheels, so we are taking W/2 for computing the maximum torque).
    - Therefore, we get: *0.6 * (120/2) * 0.045 - T = 0*
    - Hence, **T = 1.62 N-m or 16.51 Kg-cm**

The design summary

After the design, we calculate the following values and rounding to standard motor specifications that are available in the market:

- Motor RPM = 80 (rounding to standard value)
- Motor torque = 18 kg-cm
- Wheel diameter = 9 cm

The robot chassis design

After computing the robot's motors and wheels parameters, we can design the robot chassis or robot body. As required, the robot chassis should have a provision to hold food, it should be able to withstand up to 5 kg payload, the ground clearance of the robot should be greater than 3 cm, and it should be low in cost. Apart from this, the robot should have a provision to place electronics components, such as a **personal computer** (**PC**), sensors, and a battery.

One of the easiest designs to satisfy these requirements is a multi-layered architecture such as Turtlebot 2 (http://www.turtlebot.com/). It has three layers in the chassis. The robot platform called **Kobuki** (http://kobuki.yujinrobot.com/about2/) is the primary drive mechanism of this platform. The Roomba platform has motors and sensors inbuilt, so there is no need to worry about designing the robot drive system. The following image shows the **TurtleBot 2** robot chassis design:

TurtleBot 2 robot (http://robots.ros.org/turtlebot/)

We will design a robot similar to TurtleBot 2 with our own moving platform and components. Our design also has a three-layer architecture. Let's identify all the tools that we need before we start designing.

Before we start designing the robot chassis, we need **computer-aided design** (**CAD**) tools. The popular tools available for CAD are:

- SolidWorks (`http://www.solidworks.com/default.html`)
- AutoCAD (`http://www.autodesk.com/products/autocad/overview`)
- Maya (`http://www.autodesk.com/products/maya/overview`)
- Inventor (`http://www.autodesk.com/products/inventor/overview`)
- SketchUp (`http://www.sketchup.com/`)
- Blender (`http://www.blender.org/download/`)
- LibreCAD (`http://librecad.org/cms/home.html`)

The chassis design can be designed in any software you are comfortable with. Here, we will demonstrate the 2D model in **LibreCAD** and 3D model in **Blender**. One of the highlights of these applications is that they are free and available for all OS platforms. We will use a 3D mesh viewing tool called **MeshLab** to view and check the 3D model design and use Ubuntu as the main operating system. Also, we can see the installation procedures of these applications in Ubuntu 16.04 to start the designing process. We will provide tutorial links to install applications in other platforms too.

Installing LibreCAD, Blender, and MeshLab

LibreCAD is a free, open source 2D CAD application for Windows, OS X, and Linux. **Blender** is a free, open source 3D computer graphics software used to create 3D models, animation, and video games. It comes with a GPL license, allowing users to share, modify, and distribute the application. **MeshLab** is an open source, portable, and extensible system to process and edit unstructured 3D triangular meshes.

The following are the links to install LibreCAD in Windows, Linux, and OS X:

- Visit `http://librecad.org/cms/home.html` to download LibreCAD
- Visit `http://librecad.org/cms/home/from-source/linux.html` to build LibreCAD from source

- Visit `http://librecad.org/cms/home/installation/linux.html` to install LibreCAD in Debian/Ubuntu
- Visit `http://librecad.org/cms/home/installation/rpm-packages.html` to install LibreCAD in Fedora
- Visit `http://librecad.org/cms/home/installation/osx.html` to install LibreCAD in OS X
- Visit `http://librecad.org/cms/home/installation/windows.html` to install LibreCAD in Windows

You can find the documentation on LibreCAD at the following link: `http://wiki.librecad.org/index.php/Main_Page`.

Installing LibreCAD

The installation procedure for all operating systems is provided. If you are an Ubuntu user, you can simply install it from the Ubuntu Software Center as well.

Here are the commands to install LibreCAD if you are using Ubuntu:

```
$ sudo add-apt-repository ppa:librecad-dev/librecad-stable
$ sudo apt-get update
$ sudo apt-get install librecad
```

Installing Blender

Visit the following download page to install Blender for your OS platform: `http://www.blender.org/download/`.You can find the latest version of Blender here. Also, you can find the latest documentation on Blender at `http://wiki.blender.org/`.

If you are using Ubuntu/Linux, you can simply install Blender via the Ubuntu Software Center or use following command:

```
$ sudo apt-get install blender
```

Installing MeshLab

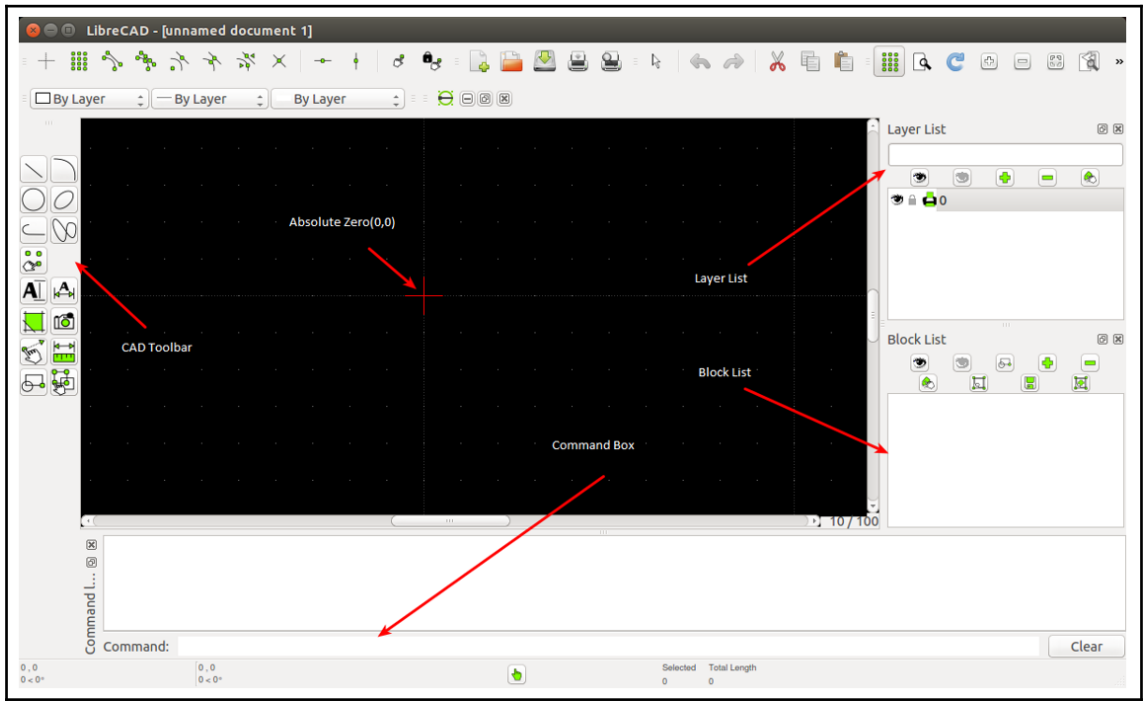
MeshLab is available for all OS platforms. The following link will provide you with the download links of prebuilt binaries and the source code of MeshLab: `http://meshlab.sourceforge.net/`

If you are an Ubuntu user, you can install **MeshLab** from an apt package manager using the following command:

```
$sudo apt-get install meshlab
```

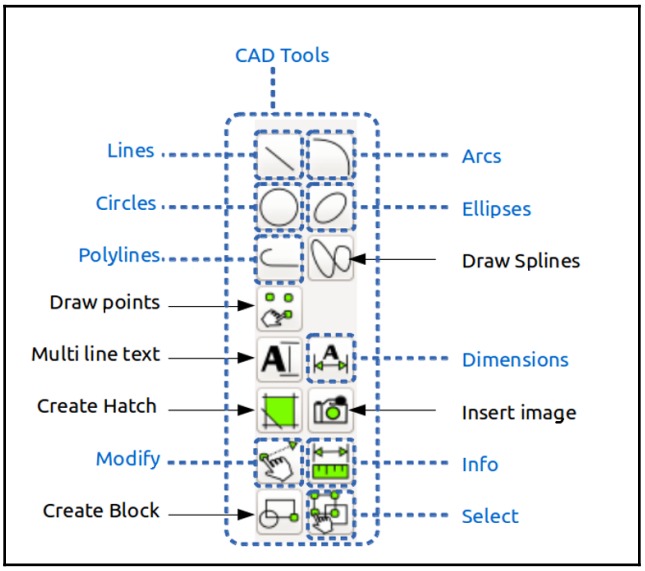
Creating 2D CAD drawing of a robot using LibreCAD

We will take a look at the basic interface of LibreCAD. The following screenshot shows the interface of LibreCAD:



LibreCAD tool

A CAD toolbar has the necessary components to draw a model. The following diagram shows the detailed overview of the CAD toolbar:



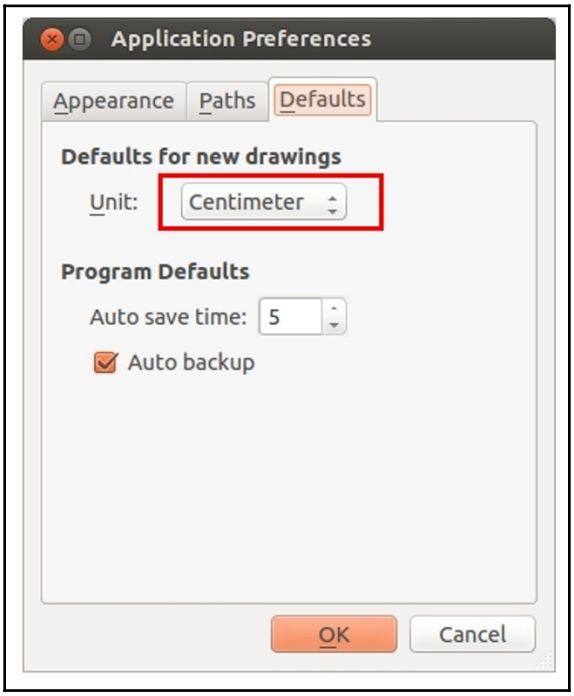
`http://wiki.librecad.org/`

A detailed description of LibreCAD tools is available at the following link:
`http://wiki.librecad.org/index.php/LibreCAD_users_Manual`

Here is a short explanation of each tool:

- **Command Box**: This is used to draw figures by only using commands. We can draw diagrams without touching any toolbar. A detail explanation about the usage of the Command Box can be found at:
- **Layer List**: This will have layers used in the current drawing. A basic concept in computer-aided drafting is the use of layers to organize a drawing. A detailed explanation of layers can be found at:
  `http://wiki.librecad.org/index.php/Layers`.
- **Blocks**: This is a group of entities and can be inserted in the same drawing more than once with different attributes at different locations, different scales, and rotation angles. A detailed explanation of Blocks can be found at the following link: `http://wiki.librecad.org/index.php/Blocks`.
- **Absolute Zero**: This is the origin of the drawing (0,0).

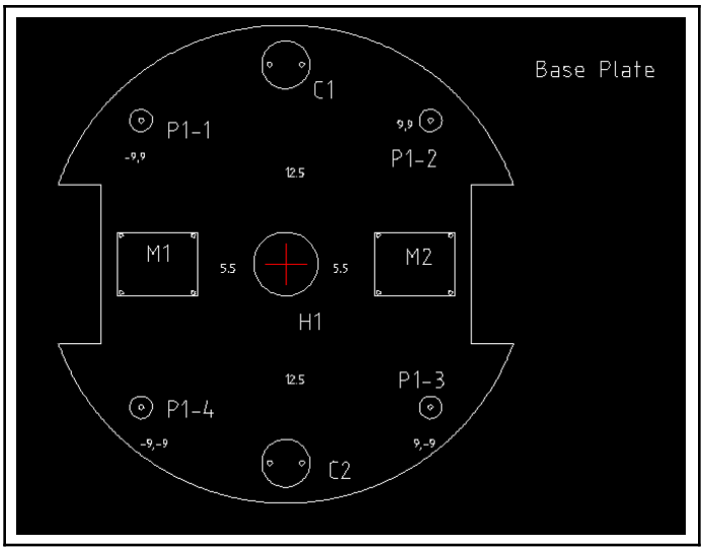
Now, start sketching by setting the unit of drawing. Set the drawing unit to centimeters. Open LibreCAD, and navigate to **Edit** | **Application Preference**. Set **Unit** as **Centimeter**, as shown in the following screenshot:



Let's start with the base plate design of the robot. The base plate has provisions to connect motors, place a battery, and a control board.

The base plate designs

The following diagram shows the robot's base plate. This plate provides provisions for two motors for the differential drive and each caster wheel on the front and back of the base plate. Motors are mentioned as **M1** and **M2** in the diagram and caster wheels are represented as **C1** and **C2**. It also holds four poles to connect to the next plates. Poles are represented as **P1-1**, **P1-2**, **P1-3**, and **P1-4**. The screws are indicated as **S** and we will use the same screws here. There is a hole at the center to bring the wires from the motor to the top of the plate. The plate is cut on the left-hand side and the right-hand side so that the wheels can be attached to the motor. The distance from the center to the caster wheels is mentioned as **12.5** cm and the distance from the center to motors is mentioned as **5.5** cm. The center of poles is at **9** cm in length and **9** cm in height from the center. The holes of all the plates follow the same dimensions:



Design of base plate

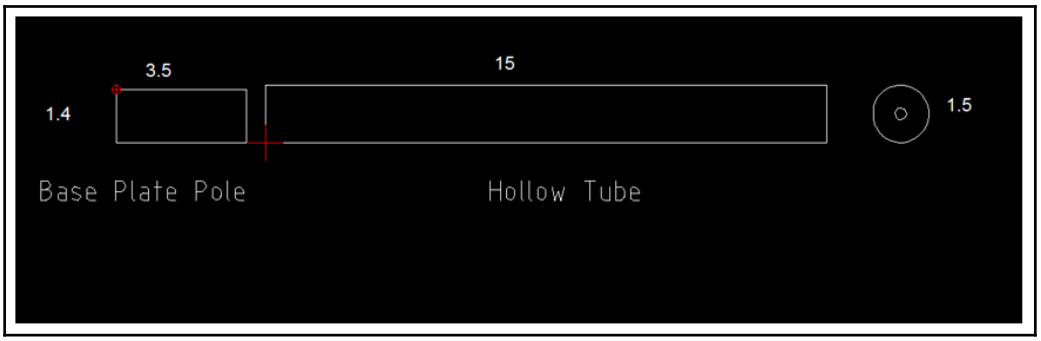
The dimensions are not marked in the diagram; instead, they are listed in the following table:

Parts	**Dimension(cm) (Length x Height) (radius)**
M1 and **M2**	5 x 4
C1 and **C2**	Radius = 1.5
S (Screw) (not shown in diagram)	0.15
P1-1,**P1-2**,**P1-3**,**P1-4**	Outer radius 0.7, Height 3.5cm
Left and right wheel sections	2.5 x 10
Base plate	Radius = 15

We will discuss the motor dimensions and clamp dimensions in more detail later.

Base plate pole design

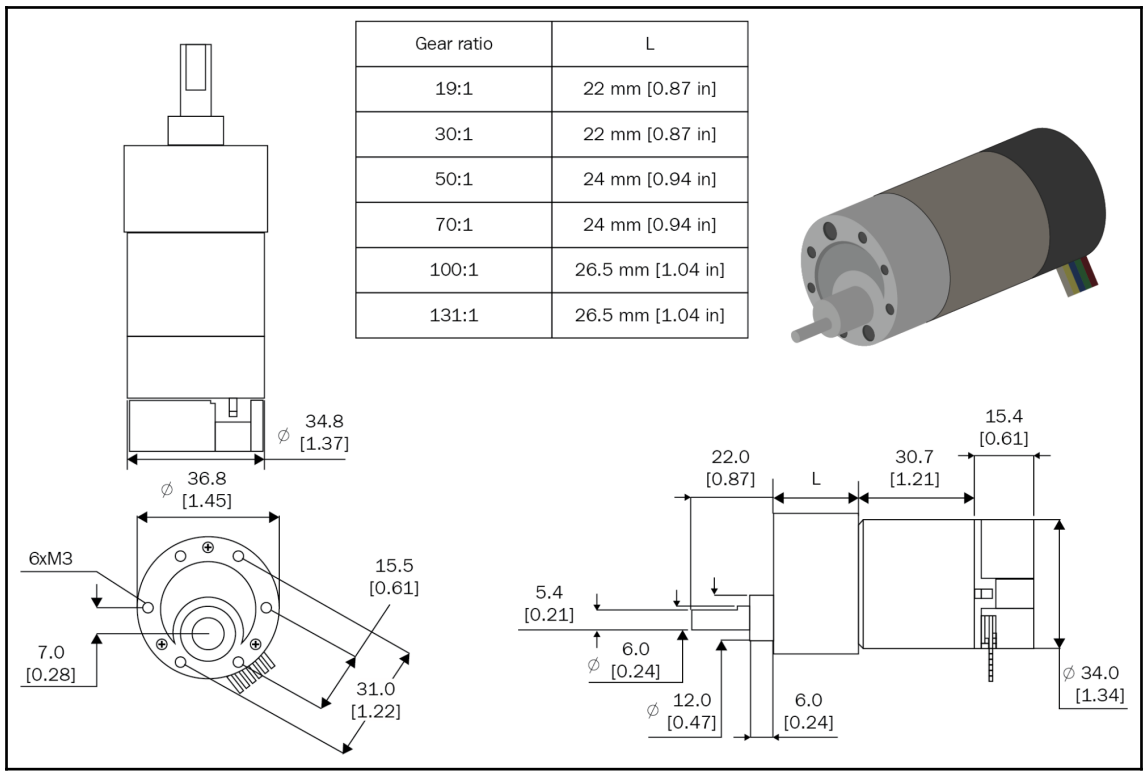
The base plate has four poles to extend to the next layer. The poles are **3.5** cm in length with a radius of **0.7** cm. We can extend to the next plate by attaching hollow tubes to the poles. At the top of the hollow tube, we will insert a hard plastic to make a screw hole. This hole will be useful for extending to the top layer. The base plate pole and hollow tubes on each pole are shown in the following diagram. The hollow tube has a radius of **0.75** cm and length of **15** cm:



Design of hollow tube 15 cm

Wheel, motor, and motor clamp design

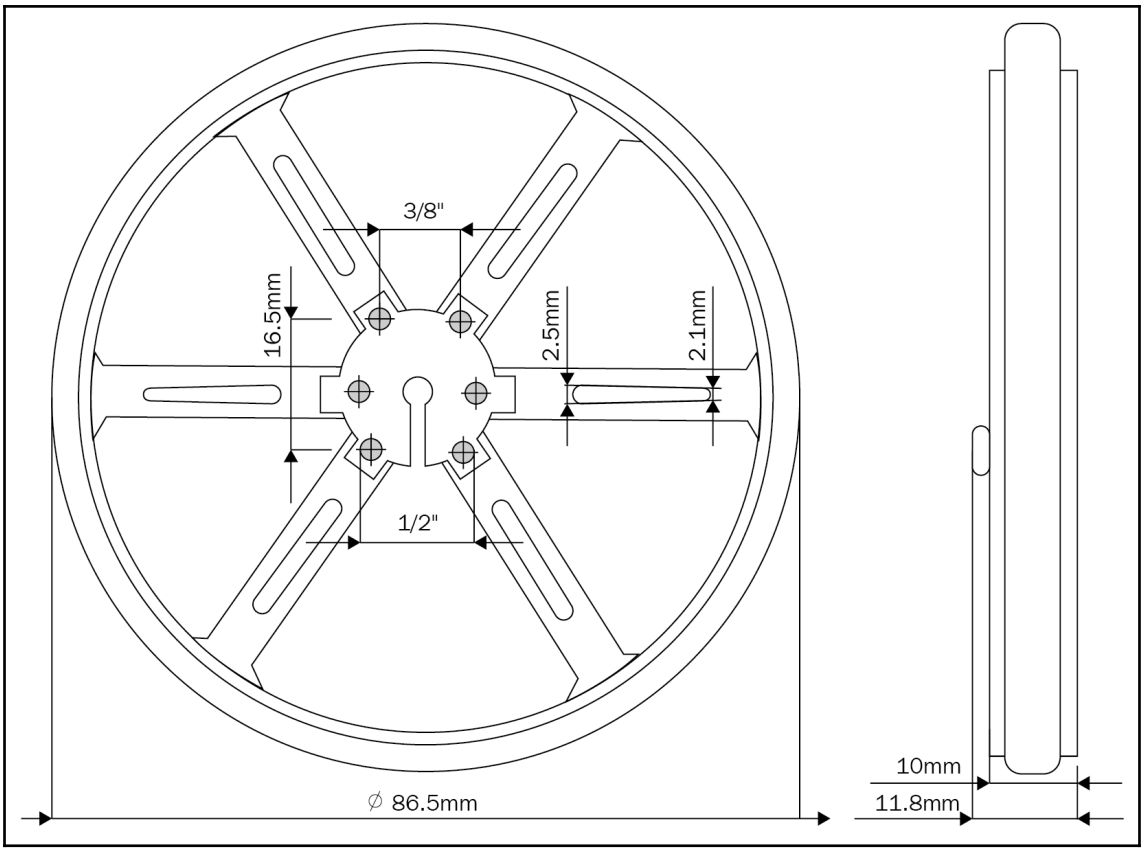
We have to decide the diameter of the wheel and compute motor requirements. Here, we are giving a typical motor and wheel that we can use if the design is successful:



Motor design of robot

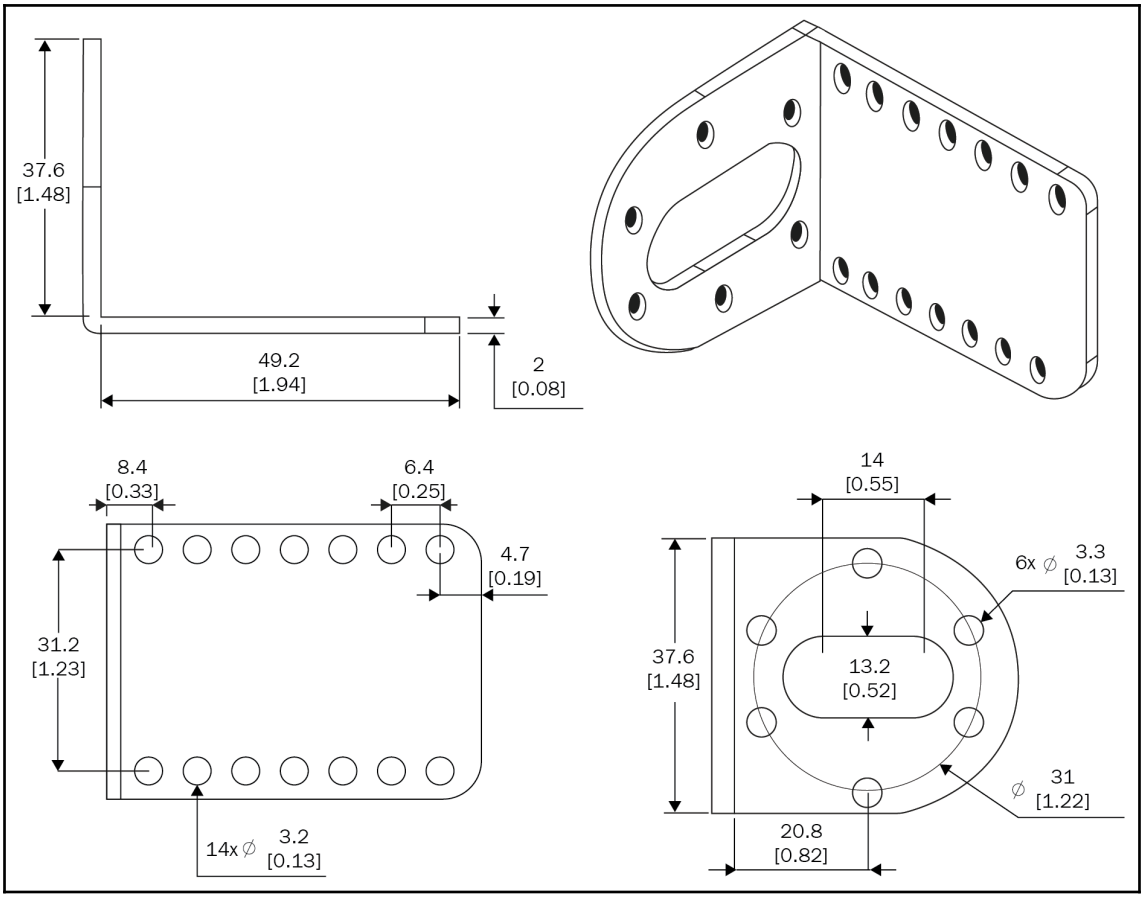
The motor design can vary according to the motor selection; if necessary, this motor can be taken as the design and can change after simulation. The **L** value in the motor diagram can vary according to the speed and torque of the motors. This is the gear assembly of the motor.

The following diagram shows a typical wheel that we can use with a diameter of **90** cm. The wheel with a diameter of **86.5** mm will become **90** mm after placing the grip:



Wheel design of robot

The motors need to be mounted on the base plate. To mount, we need a clamp that can be screwed onto the plate and also connect the motor to the clamp. The following diagram shows a typical clamp that we can use for this purpose. It's an L-shaped clamp with which we can mount the motor on one side and fit another side to the plate:



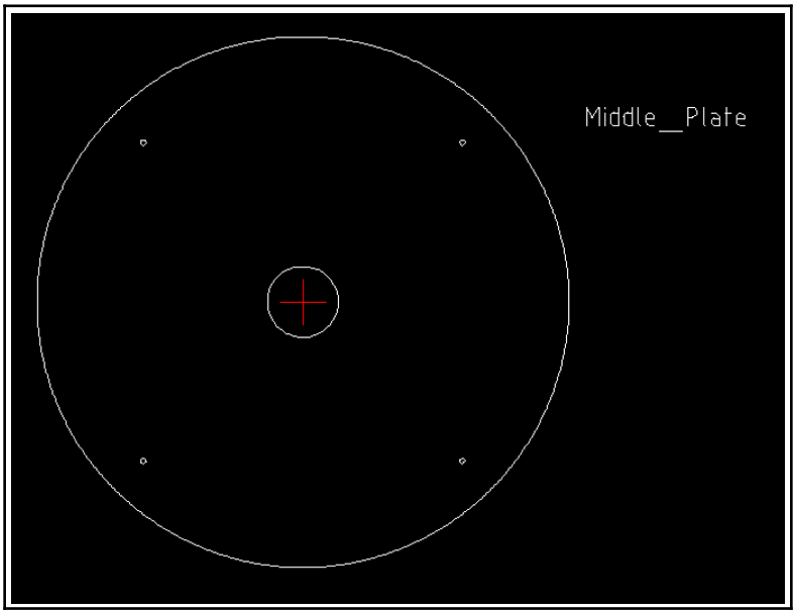
Typical clamp design of robot

Caster wheel design

Caster wheels need not have a special design; we can use any caster wheel that can touch the ground similar to the wheels. The following link has a collection of caster wheels that we can use for this design: `http://www.pololu.com/category/45/pololu-ball-casters`.

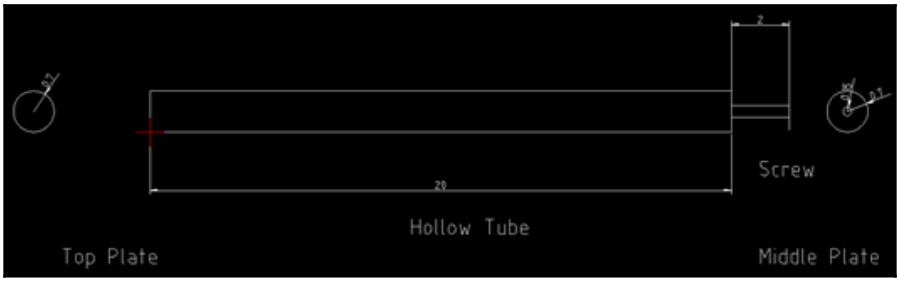
Middle plate design

The dimension of this plate is same as the base plate and the screw size is also similar:



Design of middle plate of robot

The middle plate can be held above the hollow tubes from the base plate. This arrangement is connected using another hollow tube that extends from the middle plate. The tube from the middle plate will have a screw at the bottom to fix the tube from the base plate and the middle plate and a hollow end to connect the top plate. The top and side view of the tube extending from the middle plate is shown in the following diagram:

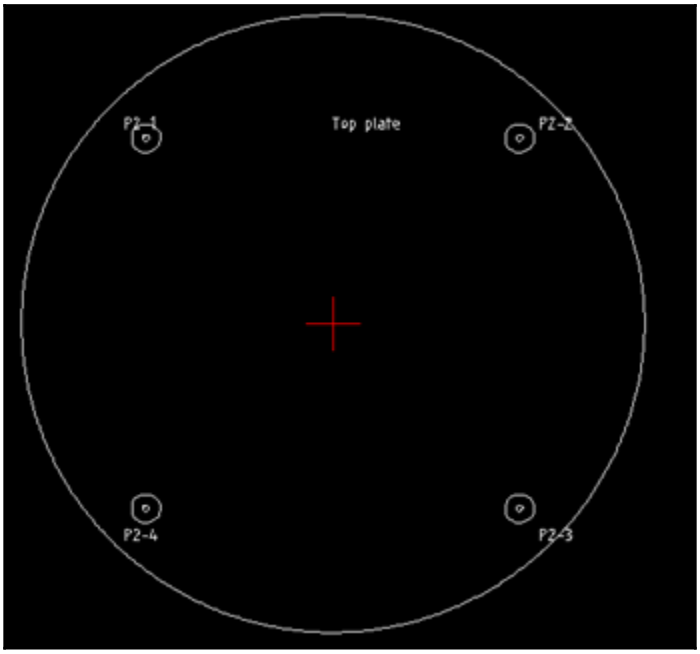


Design of hollow tube 20 cm

This tube will connect the middle plate to the base plate and at the same time provide a connect the top plate.

Top plate design

The top plate is similar to the other plates; it has four small poles of 3 cm, similar to the base plate. The poles can be placed on the hollow tubes from the middle plate. The four poles are connected to the plate itself:



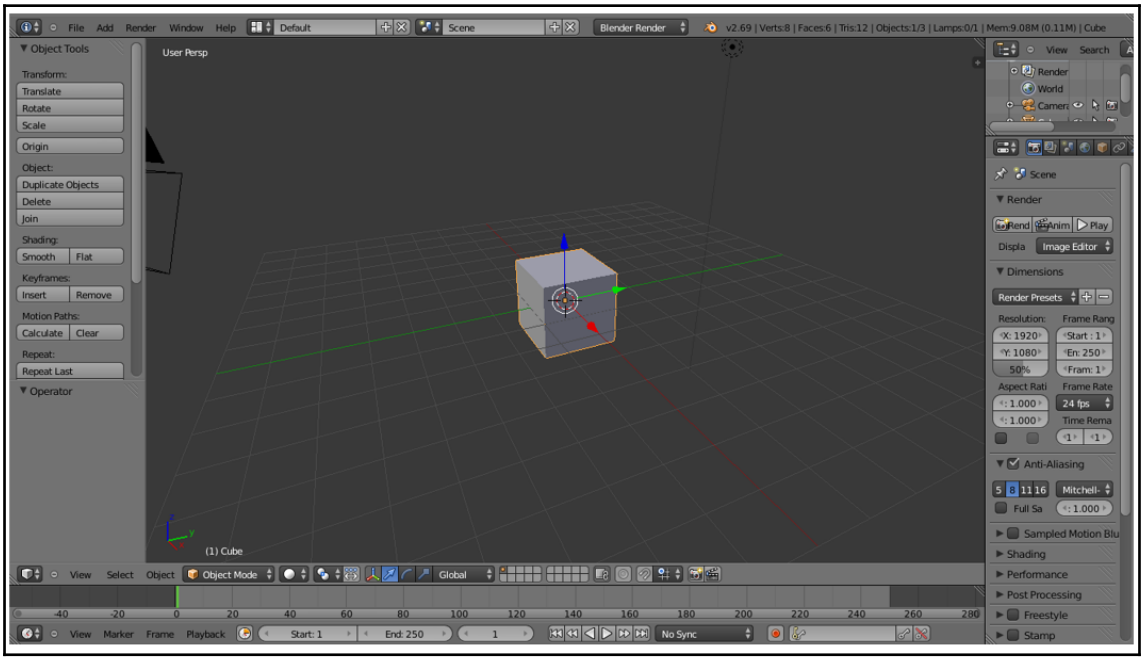
Design of top plate

After the top plate design, the robot chassis design is almost finished. Let's look at the 3D model building of this robot using Blender. The 3D model is built for simulation purposes and the 2D design we build is mainly for manufacturing purposes.

Working with a 3D model of the robot using Blender

In this section, we will design the 3D model of the robot. The 3D model is mainly used for simulation purposes. The modeling will be done using Blender. The version must be greater than 2.6 because we only tested the tutorials on these versions.

The following screenshot shows the Blender workspace and tools that can be used to work with 3D models:



Blender 3D CAD tools

The main reason why we are using Blender here is that we can model the robot using Python scripts. Blender has an inbuilt Python interpreter and a Python script editor for coding purposes. We will not discuss the user interface of Blender here. You can find a good tutorial of Blender on its website. Refer to the following link to learn about Blender's user interface:

`http://www.blender.org/support/tutorials/.`

Let's start coding in Blender using Python.

Python scripting in Blender

Blender is mainly written in C, C++, and Python. Users can write their own Python script and access all the functionalities of Blender. If you are an expert in Blender Python APIs, you can model the entire robot using a Python script instead of manual modeling.

Blender uses Python 3.x. Blender. The Python APIs are generally stable, but some areas are still added and improved. Refer to `http://www.blender.org/documentation/blender_python_api_2_69_7/` for the documentation on the Blender Python APIs.

Let's give a quick overview of the Blender Python APIs that we will use in our robot model script.

Introduction to Blender Python APIs

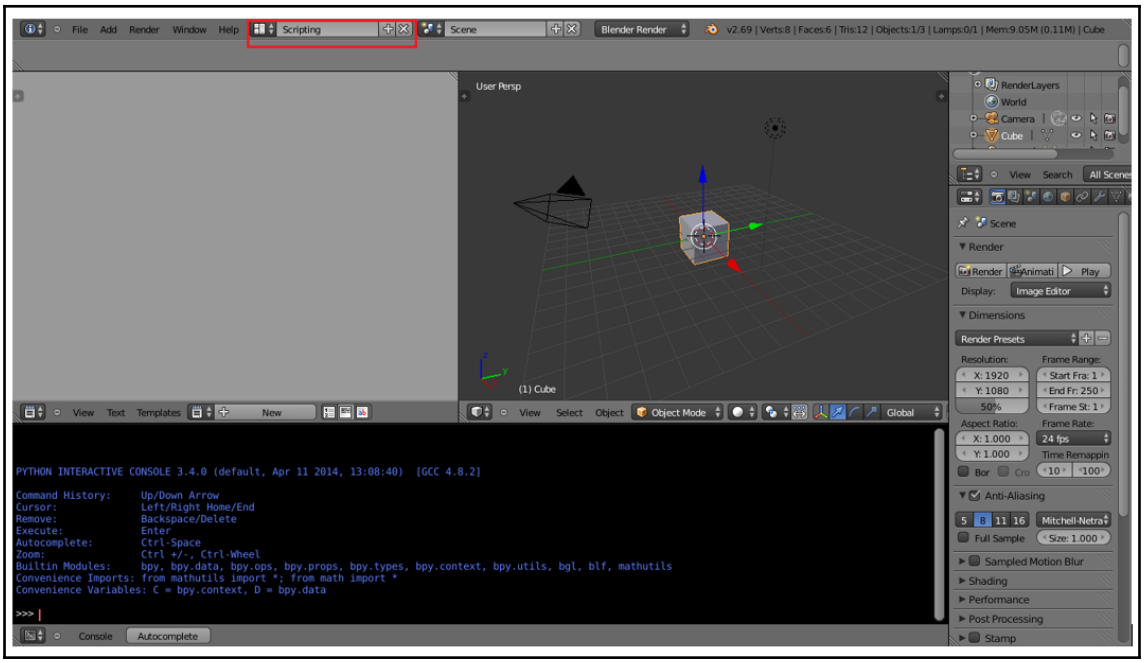
Python APIs in Blender can do most of the functionalities of Blender. The main jobs that can be done by the APIs are as follows:

- Edit any data inside Blender, such as scenes, meshes, particles, and so on
- Modify user preferences, key maps, and themes
- Create new Blender tools
- Draw the 3D view using OpenGL commands from Python

Blender provides the **bpy** module to the Python interpreter. This module can be imported in a script and gives access to Blender data, classes, and functions; scripts that deal with Blender data will need to import this module. The main Python modules we will use in bpy are:

- **Context access**: This provides access to Blender user interface functions from the (`bpy.context`) script
- **Data access**: This provides access to the Blender internal data (`bpy.data`)
- **Operators**: This provides Python access to calling operators, which includes operators written in C, Python, or Macros (`bpy.ops`)

For switching to scripting in Blender, we need to change the screen layout of Blender. The following screenshot shows the option that helps you to switch to the **Scripting** layout:



Blender Scripting option

After selecting the **Scripting** tab, we can see a text editor and Python console window in Blender. In the text editor, we can code using Blender APIs and also try Python commands via the Python console. Click on the **New** button to create a new Python script and name it `robot.py`. Now, we can design the 3D model of the robot using only Python scripts. The upcoming section has the complete script to design our robot model. We can discuss the code before running it. Hopefully, you have read the Python APIs of Blender from their site. The code in the upcoming section is split into six Python functions to draw three robot plates, draw motors and wheels, draw four support tubes, and export into the **STereoLithography** (**STL**) 3D file format for simulation.

Python script of the robot model

The following is the Python script of the robot model that we will design:

1. Before starting the Python script in Blender, we must import the `bpy` module. The `bpy` module contains all the functionalities of Blender and it can only be accessed from inside the Blender application:

```
import bpy
```

2. The following function will draw the base plate of the robot. This function will draw a cylinder with a radius of 5 cm and cut a portion from the opposite sides so that motors can be connected using the `Boolean` modifier inside Blender:

```
#This function will draw base plate
def Draw_Base_Plate():
```

3. The following two commands will create two cubes with a radius of 0.05 meters on either side of the base plate. The purpose of these cubes is to create a modifier that subtracts the cubes from the base plate. So in effect, we will get a base plate with two cuts. After cutting the two sides, we will delete the cubes:

```
bpy.ops.mesh.primitive_cube_add(radius=0.05,
location=(0.175,0,0.09))bpy.ops.mesh.primitive_cube_add(radius=0.05
,
       location=(-0.175,0,0.09))

    ####################################################
    ####################################################

    #Adding base plate
bpy.ops.mesh.primitive_cylinder_add(radius=0.15,
       depth=0.005, location=(0,0,0.09))

    #Adding boolean difference modifier from first cube

bpy.ops.object.modifier_add(type='BOOLEAN')
bpy.context.object.modifiers["Boolean"].operation =
       'DIFFERENCE'bpy.context.object.modifiers["Boolean"].object =
 bpy.data.objects["Cube"]
bpy.ops.object.modifier_apply(modifier="Boolean")

    ####################################################
    ####################################################

    #Adding boolean difference modifier from second cube
```

```
bpy.ops.object.modifier_add(type='BOOLEAN')
bpy.context.object.modifiers["Boolean"].operation =
       'DIFFERENCE'bpy.context.object.modifiers["Boolean"].object =
 bpy.data.objects["Cube.001"]
bpy.ops.object.modifier_apply(modifier="Boolean")

    ######################################################
     ######################################################

    #Deselect cylinder and delete cubes
bpy.ops.object.select_pattern(pattern="Cube")
bpy.ops.object.select_pattern(pattern="Cube.001")
bpy.data.objects['Cylinder'].select = False
bpy.ops.object.delete(use_global=False)
```

4. The following function will draw motors and wheels attached to the base plate:

```
#This function will draw motors and wheels
def Draw_Motors_Wheels():
```

5. The following commands will draw a cylinder with a radius of 0.045 and 0.01 meters in depth for the wheels. After creating the wheels, it will be rotated and translated into the cut portion of the base plate:

```
    #Create first Wheel

bpy.ops.mesh.primitive_cylinder_add(radius=0.045,
       depth=0.01, location=(0,0,0.07))
    #Rotate
bpy.context.object.rotation_euler[1] = 1.5708
    #Transalation
bpy.context.object.location[0] = 0.135

    #Create second wheel
bpy.ops.mesh.primitive_cylinder_add(radius=0.045,
       depth=0.01, location=(0,0,0.07))
    #Rotate
bpy.context.object.rotation_euler[1] = 1.5708
    #Transalation
bpy.context.object.location[0] = -0.135
```

6. The following code will add two dummy motors to the base plate. The dimensions of the motors are mentioned in the 2D design. The motor is basically a cylinder and it will be rotated and placed in the base plate:

```
    #Adding motors

bpy.ops.mesh.primitive_cylinder_add(radius=0.018,
 depth=0.06, location=(0.075,0,0.075))
bpy.context.object.rotation_euler[1] = 1.5708

bpy.ops.mesh.primitive_cylinder_add(radius=0.018,
 depth=0.06, location=(-0.075,0,0.075))
bpy.context.object.rotation_euler[1] = 1.5708
```

7. The following code will add a shaft to the motors, similar to the motor model. The shaft is also a cylinder and it will be rotated and inserted into the motor model:

```
    #Adding motor shaft
bpy.ops.mesh.primitive_cylinder_add(radius=0.006,
 depth=0.04, location=(0.12,0,0.075))
bpy.context.object.rotation_euler[1] = 1.5708

bpy.ops.mesh.primitive_cylinder_add(radius=0.006,
 depth=0.04, location=(-0.12,0,0.075))
bpy.context.object.rotation_euler[1] = 1.5708

    ######################################################
 ######################################################
```

8. The following code will add two caster wheels on the base plate. Currently, we are adding a cylinder as a wheel. In the simulation, we can assign it as a wheel:

```
    #Adding Caster Wheel

bpy.ops.mesh.primitive_cylinder_add(radius=0.015,
       depth=0.05,
location=(0,0.125,0.065))bpy.ops.mesh.primitive_cylinder_add(radius
=0.015,
       depth=0.05, location=(0,-0.125,0.065))
```

9. The following code will add a dummy Kinect sensor:

```
    #Adding Kinect

bpy.ops.mesh.primitive_cube_add(radius=0.04,
       location=(0,0,0.26))
```

10. This function will draw the middle plate of the robot:

```
#Draw middle plate
def Draw_Middle_Plate():
bpy.ops.mesh.primitive_cylinder_add(radius=0.15,
       depth=0.005, location=(0,0,0.22))

#Adding top plate
def Draw_Top_Plate():
bpy.ops.mesh.primitive_cylinder_add(radius=0.15,
       depth=0.005, location=(0,0,0.37))
```

11. This function will draw all the four supporting hollow tubes for all the three plates:

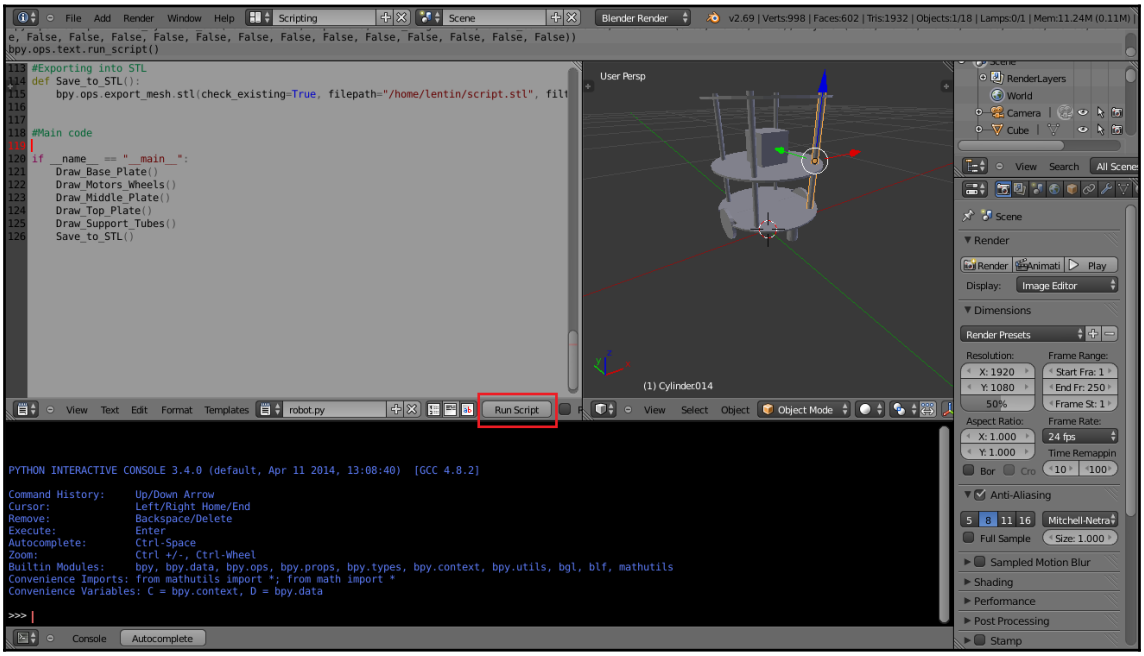
```
#Adding support tubes
def Draw_Support_Tubes():
##########################################################################
##########################

    #Cylinders
bpy.ops.mesh.primitive_cylinder_add(radius=0.007,
       depth=0.30,
location=(0.09,0.09,0.23))bpy.ops.mesh.primitive_cylinder_add(radiu
s=0.007,
       depth=0.30,
location=(-0.09,0.09,0.23))bpy.ops.mesh.primitive_cylinder_add(radi
us=0.007,
       depth=0.30,
location=(-0.09,-0.09,0.23))bpy.ops.mesh.primitive_cylinder_add(rad
ius=0.007,
       depth=0.30, location=(0.09,-0.09,0.23))
```

12. This function will export the designed robot to STL. We have to change the STL filepath before executing the script:

```
#Exporting into STL
def Save_to_STL():
bpy.ops.object.select_all(action='SELECT')
#    bpy.ops.mesh.select_all(action='TOGGLE')
```

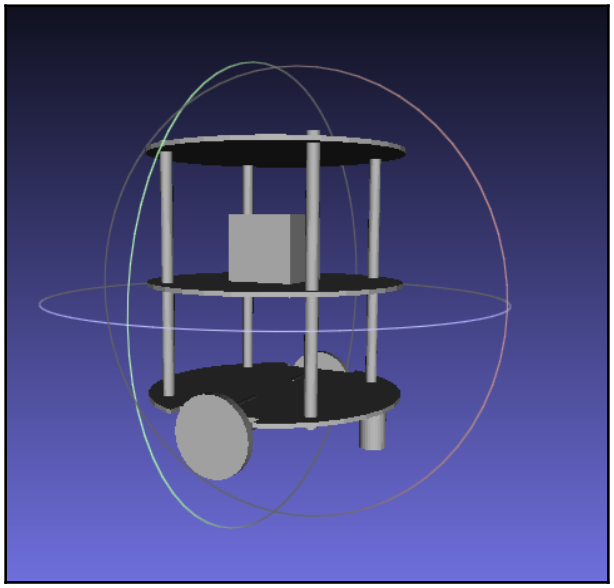
```
bpy.ops.export_mesh.stl(check_existing=True,
 filepath="/home/lentin/Desktop/exported.stl",
 filter_glob="*.stl", ascii=False,
 use_mesh_modifiers=True, axis_forward='Y',
 axis_up='Z', global_scale=1.0)


#Main code

if __name__ == "__main__":
Draw_Base_Plate()
Draw_Motors_Wheels()
Draw_Middle_Plate()
Draw_Top_Plate()
Draw_Support_Tubes()
Save_to_STL()
```

13. After entering the code in the text editor, execute the script by pressing the **Run Script** button, as shown in the following screenshot. The output 3D model will be shown on the 3D view of Blender. Also, if we check the desktop, we can see the `exported.stl` file for the simulation purpose:



Running Python script in Blender

14. The `exported.stl` file can be opened with MeshLab and the following is a screenshot of MeshLab:



3D model of Chefbot in MeshLab

Creating a URDF model of the robot

The **robot model** in ROS contains packages to model the various aspects of the robot, which is specified in the XML Robot Description Format. The core package of this stack is URDF, which parses URDF files and constructs an object model of the robot.

Unified Robot Description Format (**URDF**) is an XML specification to describe the model of a robot. We can represent the following features of the robot using URDF:

- The kinematic and dynamic description of the robot
- The visual representation of the robot
- The collision model of the robot

The description of the robot consists of a set of **links** (parts), elements, and a set of **joint** elements, which connect these links together. A typical robot description is shown in the following code:

```
<robot name="chefbot">
<link> ... </link>
<link> ... </link>
<link> ... </link>

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

It would be good if you refer to the following links for more information on URDF:
`http://wiki.ros.org/urdf`
`http://wiki.ros.org/urdf/Tutorials`

Xacro (XML Macros) is an XML macro language. With xacro, we can create shorter and more readable XML files. We can use xacro along with URDF to simplify the URDF file. If we add xacro to URDF, we have to call the additional parser program to convert xacro to URDF.

The following link will give you further details about xacro:
`http://wiki.ros.org/xacro`

robot_state_publisher allows you to publish the state of the robot to **tf** (`http://wiki.ros.org/tf`). This node read the URDF parameter called **robot_description** and reads the joint angles of the robot from a topic called **joint_states** as input and publishes the 3D poses of the robot links using the kinematic tree model of the robot. The package can be used as a library and as an ROS node. This package has been well tested and the code is stable.

- **World files**: These represent the environment of Gazebo, which has to be loaded along with the robot model. *empty.world* and *playground.world* are some examples of Gazebo world files. *empty.world* contains just an empty space. In *playground.world*, there will be some static objects in the environment. We can create our own `*.world` file using Gazebo. We will cover Gazebo world files further in the next chapter.

- **CMakeList.txt and package.xml**: These files are created during the creation of the package. The `CmakeList.txt` file helps to build the ROS C++ nodes or libraries within a package and the `package.xml` file holds the list of all the dependencies of this package.

Creating a Chefbot description ROS package

The `chefbot_description` package contains the URDF model of our robot. Before creating this package by yourself, you can go through the downloaded packages of Chefbot from `chapter3_codes`. It will help you to speed up the process.

Let's check how to create the `chefbot_description` package. The following procedure will guide you in creating this package:

1. First, we need to switch to the `chefbot` folder in the `src` folder:

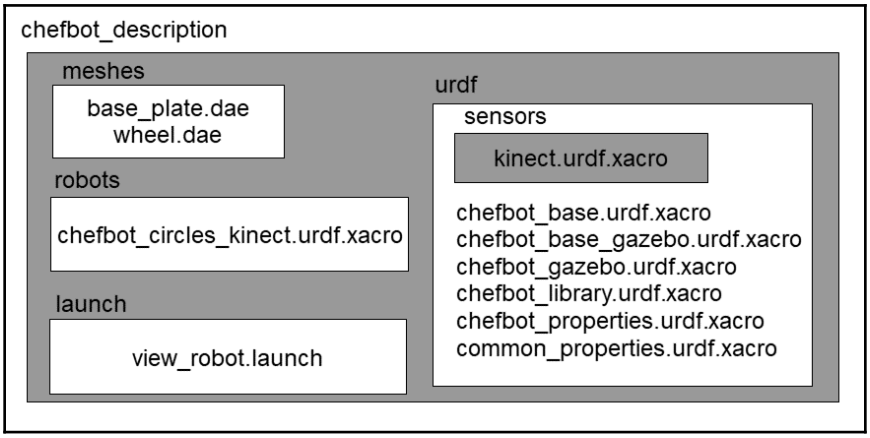
```
$ cd ~/catkin_ws/src/
```

2. The following command will create the robot description package along with dependencies, such as `urdf` and `xacro`. This will create the `chefbot_description` package in the `catkin_ws/src` folder:

```
$ catkin_create_pkgchefbot_descriptioncatkinxacro
```

3. Copy all the folders from the downloaded `chefbot_description` package to the new package folder. The `meshes` folder holds the 3D parts of the robot and the `urdf` folder contains the URDF files that have the kinematics and dynamics model of the robot. The robot model is split into several xacro files, which enables easier debugging and better readability.

Let's look at the functionality of each file inside this package. You can check each of the files inside `chefbot_description`. The following diagram shows the files inside this package:



Chefbot description package

The functionalities of each file in the package are as follows:

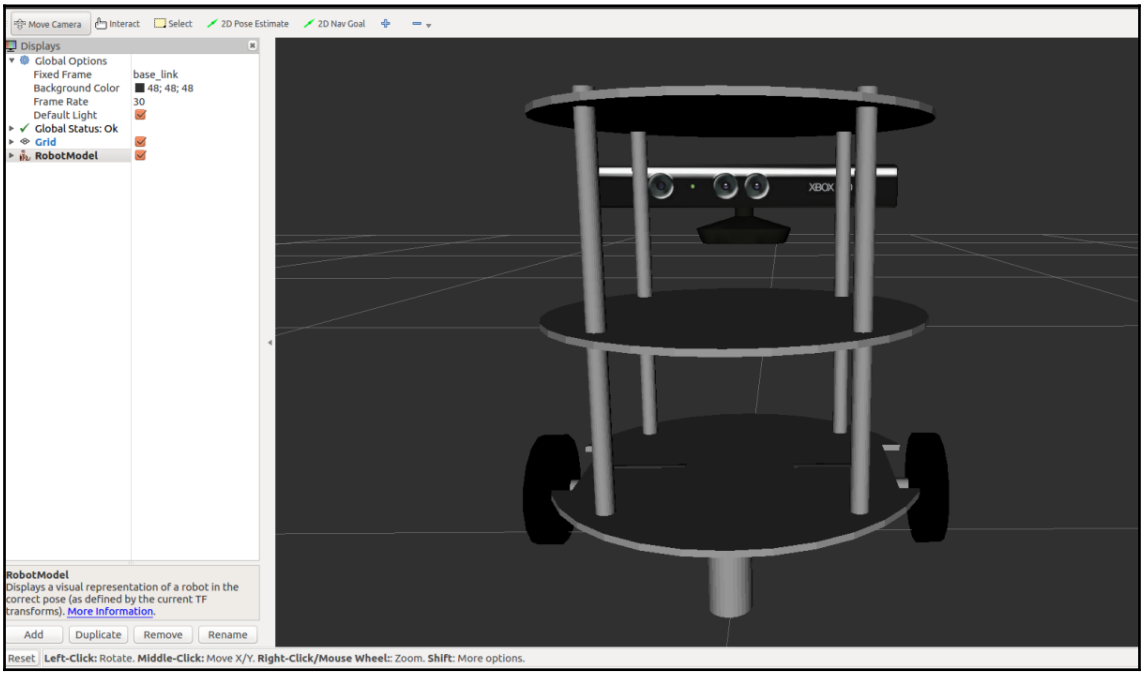
- `urdf/chefbot.xacro`: This is the main xacro file that has kinematic and dynamic parameters of the robot.
- `urdf/common_properties.xacro`: This xacro file consists of some properties and its values used inside the robot model. For example, different color definitions of robot links and some constants.
- `gazebo/chefbot.gazebo.xacro`: This file consists of simulation parameters of the robot. It mainly has Gazebo parameters and plugins for performing simulations. These parameters will only be active when we start the simulation using this model.
- `launch/upload_model.launch`: This launch file has a node that basically parses the robot xacro file and uploads the parsed data to a ROS parameter called `robot_description`. The `robot_description` parameter is then used in Rviz for visualization and used in Gazebo for simulation. If our xacro model is wrong, then this launch file will throw an error.
- `launch/view_model.launch`: This launch file will upload the robot URDF model and view the model in Rviz.
- `launch/view_navigation.launch`: The will show the URDF model and navigation-related display types in Rviz.

- `launch/view_robot_gazebo.launch`: This will launch the URDF model in Gazebo and start all Gazebo plugins.
- `meshes/`: This folder contains the necessary meshes for the robot model.
- You can build the workspace using the *catkin_make* command.

After building the packages, we can launch the Chefbot model in Rviz using the following command:

```
$ roslaunch chefbot_descriptionview_robot.launch
```

The robot model in Rviz is shown in the following screenshot:



Chefbot URDF model in Rviz

Here is the `view_robot.launch` file that visualizes the robot in Rviz:

```
<launch>

<!-- This launch file will parse the URDF model and create
robot_description parameter  - ->

<include file="$(find chefbot_description)/launch/upload_model.launch" />

<!-Publish TF from joint states -- >

<node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" />


<!-Start slider GUI for controlling the robot joints -- >
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" args="_use_gui:=True" />


<!-Start Rviz with a specific configuration -- >

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find
chefbot_description)/rviz/robot.rviz" />

</launch>
```

Here is the definition of `upload_model. launch`. The `xacro` command is to parse the `chefbot.xacro` file and store to `robot_description`:

```
<launch>

<!-- Robot description -->
<param name="robot_description" command="$(find xacro)/xacro --inorder
'$(find chefbot_description)/urdf/chefbot.xacro'" />

</launch>
```

We can have a look at the `udf/chefbot.xacro`, which is the main URDF model file. We can see how the links and joints are defined inside the xacro file.

The following code snippet shows the header of the robot xacro model. It has an XML version, robot name, and it includes some other xacro files, such as `common_properties.xacro` and `chefbot.gazebo.xacro`. After that, we can see some camera properties that are defined in the header:

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

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


<xacro:include filename="$(find
chefbot_description)/urdf/common_properties.xacro" />

<xacro:include filename="$(find
chefbot_description)/gazebo/chefbot.gazebo.xacro" />

<xacro:property name="astra_cam_py" value="-0.0125"/>
<xacro:property name="astra_depth_rel_rgb_py" value="0.0250" />
<xacro:property name="astra_cam_rel_rgb_py"   value="-0.0125" />
<xacro:property name="astra_dae_display_scale"   value="0.8" />
```

The following code snippet shows the definition of links and joints in the model:

```
<link name="base_footprint"/>

<joint name="base_joint" type="fixed">
<origin xyz="0 0 0.0102" rpy="0 0 0" />
<parent link="base_footprint"/>
<child link="base_link" />
</joint>
<link name="base_link">
<visual>
<geometry>
<!-- new mesh -->
<mesh filename="package://chefbot_description/meshes/base_plate.dae" />
<material name="white"/>
</geometry>


   <origin xyz="0.001 0 -0.034" rpy="0 0 ${M_PI/2}"/>
</visual>
<collision>
<geometry>
<cylinder length="0.10938" radius="0.178"/>
</geometry>
<origin xyz="0.0 0 0.05949" rpy="0 0 0"/>
</collision>
```

```
<inertial>
<!-- COM experimentally determined -->
<origin xyz="0.01 0 0"/>
<mass value="2.4"/><!-- 2.4/2.6 kg for small/big battery pack -->

<inertia ixx="0.019995" ixy="0.0" ixz="0.0"
iyy="0.019995" iyz="0.0"
izz="0.03675" />
</inertial>
</link>
```

In this code, we can see the definition of two links called `base_footprint` and `base_link`. The `base_footprint` link is a dummy link, meaning it has any properties; it is just for showing the origin of the robot. The `base_link` is the origin of the robot and it has visual and collision properties. We can also see that the link is visualized as a mesh file. We can also see the inertial parameters of the link in the definition. The joint is the combination of two link. We can define a joint in URDF by mentioning two links and the type of the joint. There are different types of joints available in URDF, such as fixed, revolute, continuous, and prismatic. In this snippet, we are creating a fixed joint because there is no movement between these frames.

This chapter has been all about the basics of the Chefbot URDF. We will learn more about Chefbot simulation and give an explanation of the parameters in the next chapter.

Summary

In this chapter, we discussed the modeling of the Chefbot robot. The modeling involves 2D and 3D designing of the robot hardware and ends up in as URDF model, which can be used in ROS. The chapter started with the various requirements to be satisfied by the robot and we have seen how to calculate various design parameters. After calculating the design parameters, we started to design the 2D sketches of the robot hardware. The designing was done using LibreCAD, a free CAD tool. After that, we worked on the 3D model in Blender using Python scripting. We have created the mesh model from Blender and created the URDF model of the robot. After the creation of the URDF model, we looked at how to visualize the robot in Rviz.

In the next chapter, we will discuss how to simulate this robot and perform mapping and localization.

Questions

1. What is robot modeling and what are its uses?
2. What is the aim of a 2D robot model?
3. What is the aim of a 3D robot model?
4. What is the advantage of Python scripting over manual modeling?
5. What is a URDF file and what are its uses?

Further reading

To learn more about URDF and Xacro and Gazebo you can refer book: *Mastering ROS for Robotics Programming - Second Edition* (`https://www.packtpub.com/hardware-and-creative/mastering-ros-robotics-programming-second-edition`)

4
Simulating a Differential Drive Robot Using ROS

In the previous chapter, we looked at how to model Chefbot. In this chapter, we are going to learn how to simulate the robot using the Gazebo simulator in ROS. We will learn how to create a simulation model of Chefbot, and we will create a hotel-like environment in Gazebo to test our application, which is programmed to automatically deliver food to customers. We will look at a detailed explanation of each of the steps to test out our application. The following are the important topics that we are going to cover in this chapter:

- Getting started with the Gazebo simulator
- Working with the TurtleBot 2 simulation
- Creating a simulation of Chefbot
- URDF tags and plugins for simulations
- Getting started with simultaneous localization and mapping
- Implementing SLAM in a Gazebo environment
- Creating a map using SLAM
- Getting started with adaptive Monte Carlo localization
- Implementing AMCL in a Gazebo environment
- Autonomous navigation of Chefbot in a hotel using Gazebo

Technical requirements

To test the application and codes in this chapter, you need an Ubuntu 16.04 LTS PC/laptop with ROS Kinetic installed.

Getting started with the Gazebo simulator

In the first chapter, we looked at the basic concepts of the Gazebo simulator and its installation procedures. In this chapter, we will learn more about the usage of Gazebo and how to simulate a differential drive robot in the Gazebo simulator. The first step is to understand the GUI interfaces and its various controls. As we have discussed in the first chapter, Gazebo has two main sections. The first is the Gazebo server and the second is the Gazebo client. The simulation is done on the Gazebo server, which acts as a backend. The GUI is the frontend, which acts as the Gazebo client. We will also look at Rviz (ROS Visualizer), which is a GUI tool in ROS that is used to visualize different kinds of robot sensor data from robot hardware or a simulator, such as Gazebo.

We can use Gazebo as an independent simulator to simulate the robot, or we can use interfaces with ROS and Python that can be used to program robots in the Gazebo simulator. If we are using Gazebo as an independent simulator, the default option to simulate the robot is by writing C++-based plugins (`http://gazebosim.org/tutorials/?tut=plugins_hello_world`). We can write C++ plugins for simulating a robot's behavior, creating new sensors, creating a new world, and so on. By default, the modeling of robots and environments in Gazebo is done using the SDF (`http://sdformat.org/`) file. If we are using an ROS interface for Gazebo, we have to create a URDF file that contains all the parameters of the robot and has Gazebo-specific tags to mention the simulation properties of the robot. When we start the simulation using URDF, it will convert to an SDF file using some tools and display the robot in Gazebo. The ROS interface of Gazebo is called gazebo-ros-pkgs. It is a set of wrappers and plugins that have the ability to model a sensor, robot controller, and other simulations in Gazebo and communicate over ROS topics. In this chapter, we will be mainly focusing on the ROS-Gazebo interface for simulating Chefbot. The advantage of the ROS-Gazebo interface is that we can program the robot by making use of the ROS framework. We can program the robot using popular programming languages such as C++ and Python using ROS.

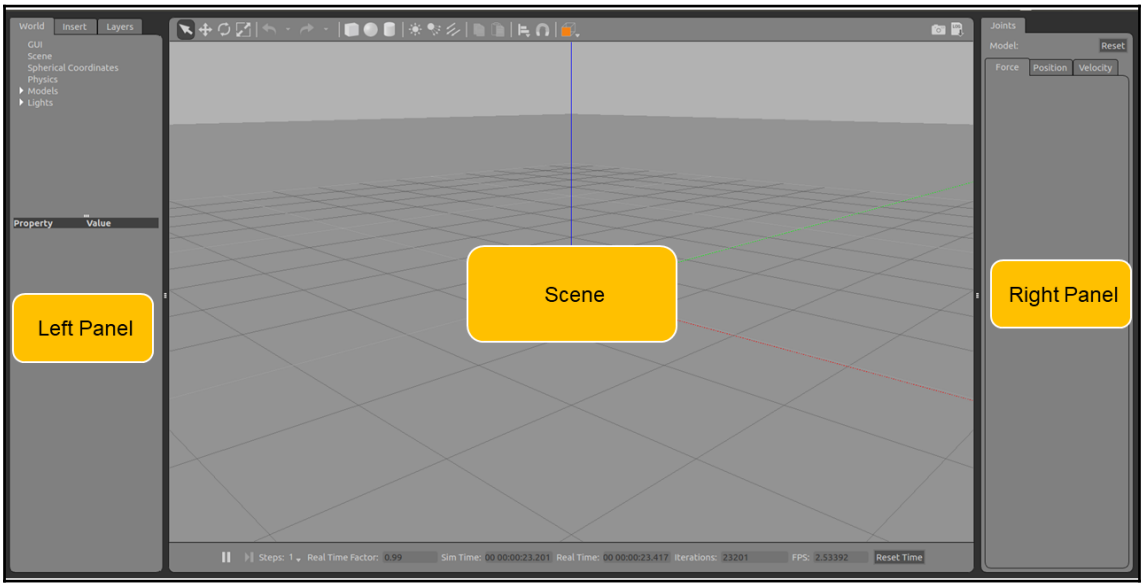
If you are not interested in using ROS and want to program the robot using Python, you should check out an interface called pygazebo (`https://github.com/jpieper/pygazebo`). It is a Python binding of Gazebo. In the next section, we will see the GUI of Gazebo, along with some of its important controls.

The Gazebo's graphical user interface

We can start Gazebo in several ways. You have already seen this in *Chapter 1, Getting Started with Robot Operating System*. In this chapter, we are using the following command to start an empty world, meaning that there is no robot and no environment:

```
$ roslaunch gazebo_ros empty_world.launch
```

The preceding command will start the Gazebo server and client and load an empty world into Gazebo. Here is the view of the empty world in Gazebo:



Gazebo user interface

The Gazebo user interface can be divided into three sections: **Scene**, **Left Panel**, and the **Right Panel**.

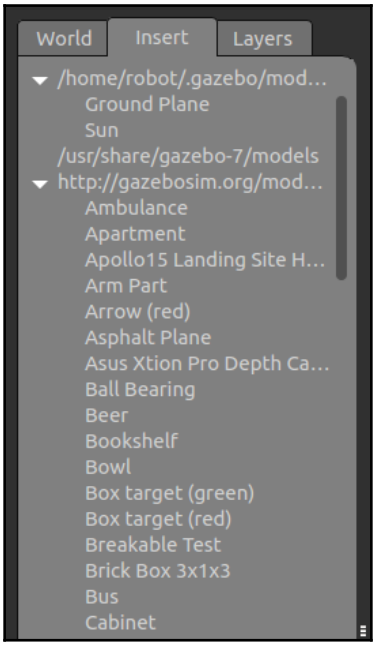
The Scene

The Scene is the place where the simulation of the robot takes place. We can add various objects to the scene, and we can interact with the robot in the scene using the mouse and keyboard.

The Left Panel

You can see the left Panel when we launch Gazebo. There are three main tabs in the Left Panel:

- **World**: The **World** tab contains a list of models in the current Gazebo Scene. Here, we can modify model parameters, such as the pose, and can also change the camera's pose.
- **Insert**: The **Insert** tab allows you to add a new simulation model to the Scene. The models are available in the local system, as well as the remote server. The `/home/<user_name>/.gazebo/model` folder will keep the local model files and models in the remote server in `http://gazebosim.org/models`, as shown in the following screenshot:



The Insert tab in the left panel of Gazebo

You can see both the local files and remote files in the **Insert** tab that is shown in the preceding screenshot.

When you start Gazebo for the first time, or when you start a world that has models from the remote server, you may see a black screen on Gazebo or a warning on the terminal. This is because the model in the remote server is being downloaded and Gazebo has to wait a while. The waiting time can vary according to the speed of your internet connection. Once the model is downloaded, it will be kept in the local model folder, so there will not be any delay the next time.

- **Layers**: Most of the time, we will not use this tab. This tab is for organizing the different visualizations available in the simulation. We can hide/unhide the models in the simulation by toggling each layer. Most of the time in the simulation, this tab will be empty.

Right Panel

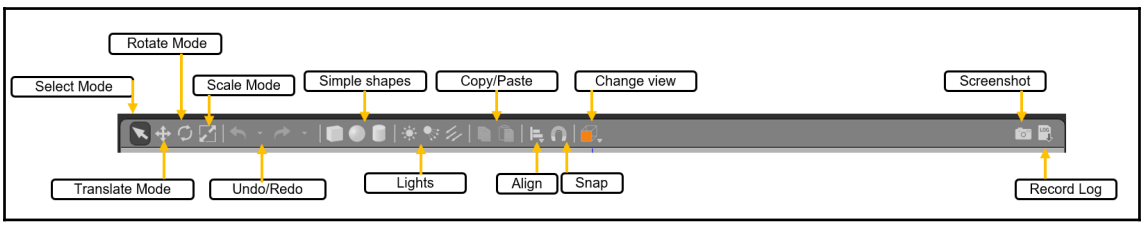
The Right panel is hidden by default. We have to drag it in order to view it. This panel enables us to interact with the mobile parts of the model. We can see the joints of the model if we select the model in the scene.

Gazebo toolbars

The Gazebo has two toolbars. One is above the Scene and one is below it.

Upper toolbar

he upper toolbar is very useful for interacting with the Gazebo Scene. This toolbar is mainly for manipulating the Gazebo Scene. It has functions to select the model, scale it, translate and rotate it, and add new shapes to the scene:



Upper toolbar of Gazebo

The following list shows you detailed descriptions of each option:

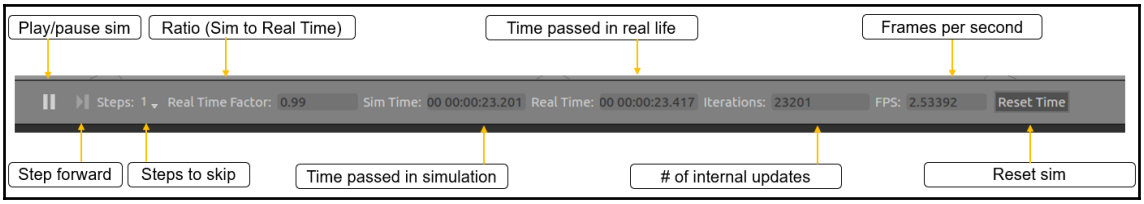
- **Select Mode**: If we are in Select Mode, we can select the models in the Scene and set their properties, as well as navigate inside the Scene.
- **Translate Mode**: In Translate Mode, we can select a model and translate it model by clicking the Left button.
- **Rotate Mode**: In Rotate Mode, we can select the model and change its orientation.
- **Scale Mode**: In Scale Mode, we can select the model and scale it.
- **Undo/Redo**: This enables us to undo or redo actions in the Scene.
- **Simple Shapes**: With this option, we can insert primitive shapes into the scene, such as a cylinder, cube, or sphere.
- **Lights**: The Lights option enables us to add different kinds of light sources into the Scene.
- **Copy/Paste**: The Copy and Paste options enable us to copy and paste different models and parts of the Scene.
- **Align**: This enables us to align models to one another.
- **Snap**: This snaps one model and moves it inside the Scene.
- **Change view**: This changes the view of the Scene. It mainly uses the perspective view and orthogonal view.
- **Screenshot:** This takes a screenshot of the current Scene.
- **Record Log:** This saves Gazebo's logs.

Bottom toolbar

The bottom toolbar mainly gives us an idea about the simulation. It displays the Simulation Time, which refers to the time that is passing within the simulator. The simulation can sped up or slowed down. This depends on the computation required for the current simulation.

The Real Time display refers to the actual time passing in real life when the simulator is running. The **real time factor** (**RTF**) is the ratio between simulation time and the speed of real time. If the RTF is one, it means that the simulation is happening at a rate identical to the speed of time in reality.

The state of the world in Gazebo can change with each iteration. Each iteration can make changes in Gazebo for a fixed amount of time. That fixed time is called the step size. By default, the step size is 1 millisecond. The step size and iteration are shown in the tool bar, as shown in the following screenshot:



Lower toolbar of Gazebo

We can pause the simulation and see each step using the **Step** button.

You can get more information about the Gazebo GUI from `http://gazebosim.org/tutorials?cat=guided_b&amp;tut=guided_b2`.

Before going to the next section, you can play with Gazebo and learn more about how it works.

Working with a TurtleBot 2 simulation

After working with Gazebo, now it's time to run a simulation on it and work with some robots. One of the popular robots available for education and research is TurtleBot. The TurtleBot software was developed within the ROS framework, and there is a good simulation of its operations available in Gazebo. The popular versions of TurtleBot are TurtleBot 2 and 3. We will learn about TurtleBot 2 in this section because our development of Chefbot was inspired by its design.

Installing TurtleBot 2 simulation packages in Ubuntu 16.04 is straightforward. You can use the following command to install TurtleBot 2 simulation packages for Gazebo:

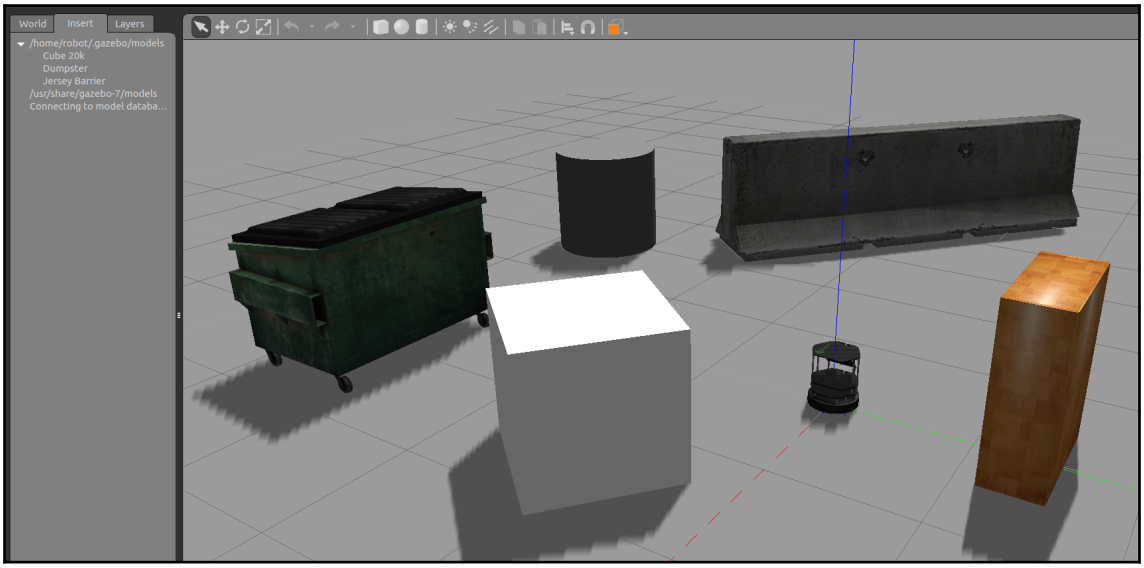
```
$ sudo apt-get install ros-kinetic-turtlebot-gazebo
```

After installing the packages, we can start running the simulation. There are several launch files inside the turtlebot-gazebo packages that have different world files. A Gazebo world file (`*.world`) is an SDF file consisting of the properties of the models in the environment. When the world file changes, Gazebo will load with a different environment.

The following command will start a world that has a certain set of components:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

The simulation will take some time to load, and when it loads, you will see the following models in the Gazebo Scene:



TurtleBot 2 simulation in Gazebo

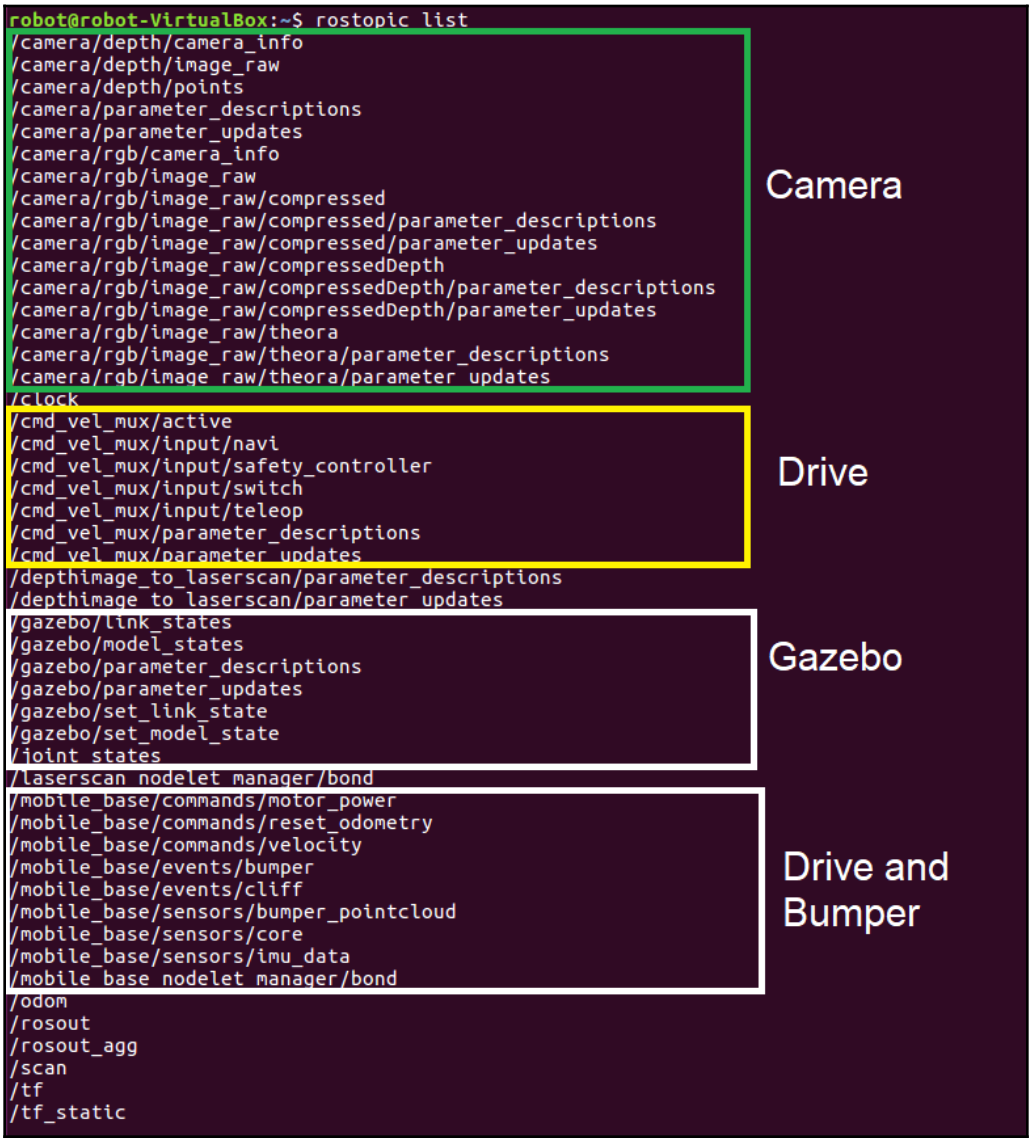
When we load the simulation in Gazebo, it will also load the necessary plugins to interact with ROS. TurtleBot 2 has the following important components:

- A mobile base with a differential drive
- A depth sensor for creating a map
- A bumper switch to detect collision

When the simulation loads, it will load the ROS-Gazebo plugins to simulate a differential drive mobile base, depth sensor (Kinect or Astra), and plugins for bumper switches. So, after loading the simulation, if we enter a `$ rostopic list` command in the terminal, a selection of topics will appear as shown in the following screenshot.

As we saw earlier, we can see the topics from the differential drive plugin, depth sensor, and bumper switches. In addition to these, we can see the topics from the ROS-Gazebo plugins that mainly contain the current state of the robot and other models in the simulation.

The Kinect/Astra sensors can give an RGB image and depth image. The differential drive plugin can send the odometry data of the robot in the `/odom` (`nav_msgs/Odometry`) topic and can publish the robot's transformation in the `/tf` (`tf2_msgs/TFMessage`) topics, as shown in the following screenshot:



ROS topics from the TurtleBot 2 simulation

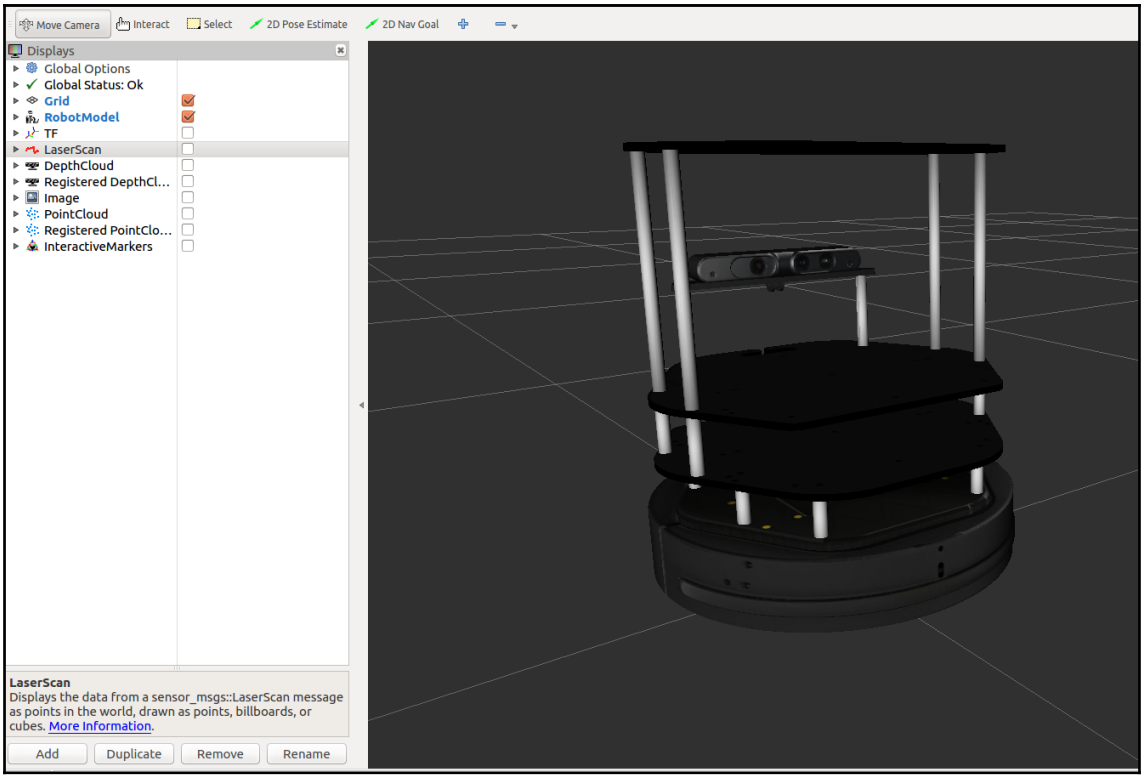
We can visualize the robot model and sensor data in Rviz. There is a TurtleBot package dedicated for visualization. You can install the following package to visualize the robot data:

```
$ sudo apt-get install ros-kinetic-turtlebot-rviz-launchers
```

After installing this package, we can use the following launch file to visualize the robot and its sensor data:

```
$ roslaunch turtlebot-rviz-launchers view_robot.launch
```

We will get the following Rviz window with the robot model displayed in it. We can then enable the sensor displays to visualize this particular data, as shown in the following screenshot:



TurtleBot 2 visualization in Rviz

In the next section, we will learn how to move this robot.

Moving the robot

The differential drive plugin of the robot is capable of receiving ROS Twist messages (`geometry_msgs/Twist`), which consist of the current linear and angular velocities of the robot. The teleoperation of the robot means moving the robot manually using a joy stick or keyboard by using ROS Twist messages. We will now look at how to move the Turtlebot 2 robot using keyboard teleoperation.

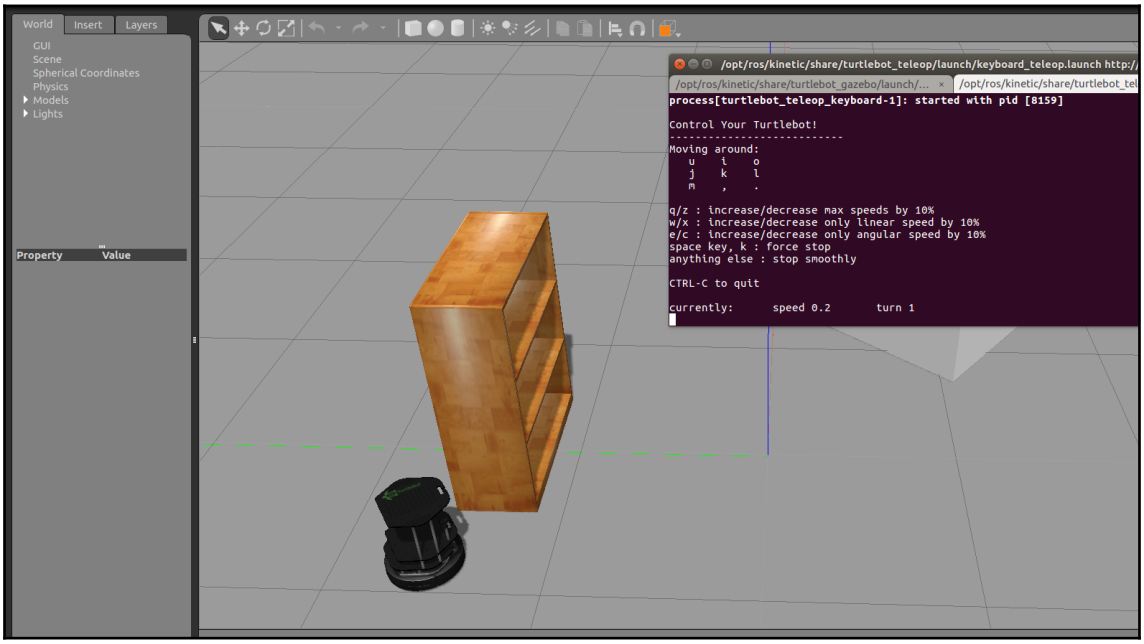
We have to install a package to teleoperate the TurtleBot 2 robot. The following command will install the TurtleBot teleoperation package:

```
$ sudo apt-get install ros-kinetic-turtlebot-teleop
```

To start teleoperation, we have to start the Gazebo simulator first, and then start the teleoperation node using the following command:

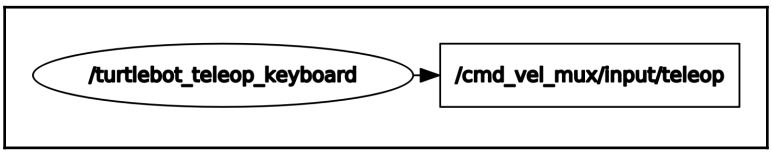
```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

In the terminal, we can see the key combination for moving the robot. You can move it using those keys, and you will see the robot moving in Gazebo and Rviz, as shown in the following screenshot:



TurtleBot 2 keyboard teleoperation

When we press the buttons on the keyboard, it will send a Twist message to the differential drive controller, and the controller will move the robot in the simulation. The teleop node sends a topic called `/cmd_vel_mux/input/teleop` (`geometry_msgs/Twist`), which is shown in the following diagram:



The TurtleBot keyboard teleoperation node

Creating a simulation of Chefbot

We have seen how the turtlebot simulation works. In this section, we will be looking at how to create our own robot simulation using Gazebo.

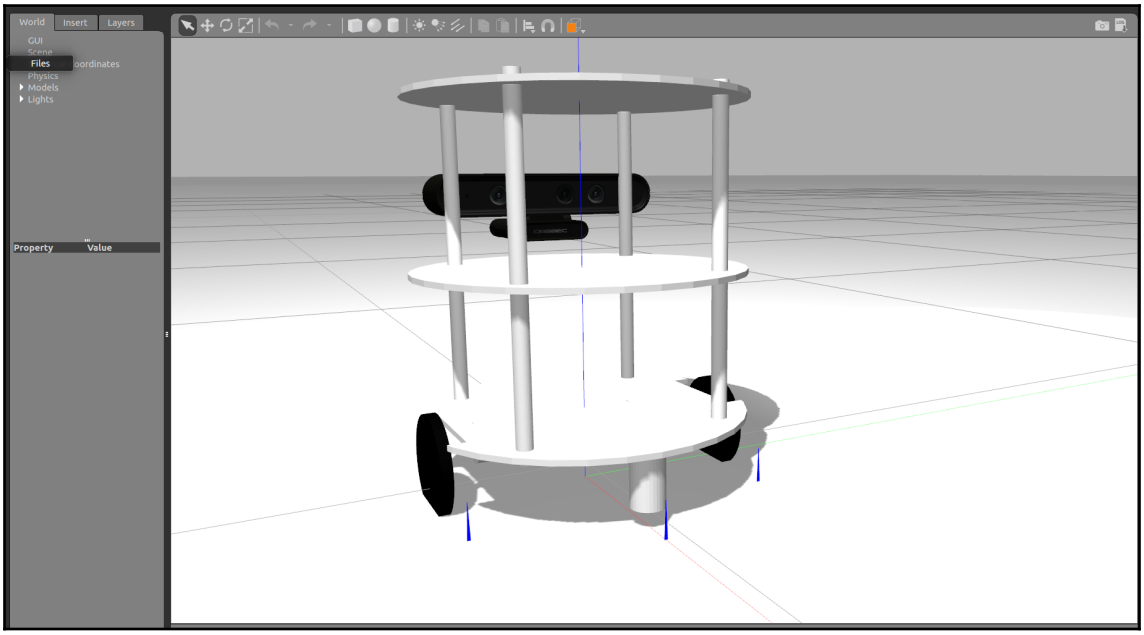
Before we start discussing this subject, you should copy the `chefbot_gazebo` package to your catkin workspace and enter `catkin_make` to build the package. Make sure you have two packages in your workspace, one called `chefbot_description` and the other called `chefbot_gazebo`. The `chefbot_gazebo` package contains a simulation-related launch file and parameters, and `chefbot_description` has the robot's URDF model, along with its simulation parameters, and the launch file that is used to view the robot in Rviz and Gazebo.

Let's begin creating our Chefbot model in Gazebo so that you can familiarize yourself with the procedure. After that, we will dig deep into the xacro file and can look at the simulation parameters.

The following launch file will show the robot model in Gazebo with an empty world and start all the Gazebo plugins for the robot:

```
$ roslaunch chefbot_description view_robot_gazebo.launch
```

The following figure shows a screenshot of the Chefbot in Gazebo:



The Chefbot in Gazebo

Let's see how we can add a URDF robot model in Gazebo. You can find the definition of the URDF robot model at `chefbot_description/launch/view_robot_gazebo.launch`.

The first section of the code calls the `upload_model.launch` file for creating the `robot_description` parameter. If it is successful, then it will start an empty world in Gazebo:

```
<launch>
  <include file="$(find chefbot_description)/launch/upload_model.launch" />

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="recording" value="false"/>
    <arg name="debug" value="false"/>
  </include>
```

So how does the robot model in the `robot_description` parameter show in Gazebo? The following code snippet in the launch file does that job:

```
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param
robot_description -urdf -z 0.1 -model chefbot" />
```

The node called `spawn_model` inside the `gazebo_ros` package will read the `robot_description` and spawn the model in Gazebo. The `-z 0.1` argument indicates the height of the model to be placed in Gazebo. If the height is 0.1, the model will be spawned at a height of 0.1. If gravity is enabled, then the model will fall to the ground. We can change this parameter according to our requirement. The `-model` argument is the name of the robot model in Gazebo. This node will parse all the Gazebo parameters from the `robot_description` and start the simulation in Gazebo.

After spawning the model, we can publish the robot transformation (tf) using the following lines of code:

```
  <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher">
    <param name="publish_frequency" type="double" value="30.0" />
  </node>
```

We are publishing the ROS tf at 30 Hz.

Depth image to laser scan conversion

The depth sensor on the robot provides the 3D coordinates of the environment. To achieve autonomous navigation, we can use this data to create a 3D map. There are different techniques for creating a map of the environment. One of the algorithms that we are using for this robot is called gmapping (`http://wiki.ros.org/gmapping`). The gmapping algorithm mainly use a laser scan for creating the map, but in our case, we get an entire 3D point cloud from the sensor. We can convert the 3D depth data from a laser scan by taking a slice of the depth data. The following nodelet (`http://wiki.ros.org/nodelet`) in this launch file is able to receive the depth data and convert it to laser scan data:

```
  <node pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager"
args="manager"/>
  <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
        args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet
laserscan_nodelet_manager">
    <param name="scan_height" value="10"/>
    <param name="output_frame_id" value="/camera_depth_frame"/>
    <param name="range_min" value="0.45"/>
```

```
    <remap from="image" to="/camera/depth/image_raw"/>
    <remap from="scan" to="/scan"/>
  </node>
</launch>
```

The nodelet is a special kind of ROS node that has a property called zero copy transport, meaning that it doesn't take network bandwidth to subscribe to a topic. This will make the conversion from the depth image (`sensor_msgs/Image`) to the laser scan (`sensor_msgs/LaserScan`) faster and more efficient. One of the other properties of the nodelet is that it can be dynamically loaded as plugins. We can set various properties of this nodelet, such as the `range_min`, name of the image topic, and the output laser topic.

URDF tags and plugins for Gazebo simulation

We have seen the simulated robot in Gazebo. Now, we will look in more detail at the simulation-related tags in URDF and the various plugins we have included in the URDF model.

Most of the Gazebo-specific tags are in the `chefbot_description/gazebo/chefbot.gazebo.xacro` file. Also, some of the tags in `chefbot_description/urdf/chefbot.xacro` are used in the simulation. Defining the `<collision>` and `<inertial>` tags in chefbot.xacro is very important for our simulation. The `<collision>` tag in URDF defines a boundary around the robot link, which is mainly used to detect the collision of that particular link, whereas the `<inertial>` tag encompasses the mass of the link and the moment of inertia. Here is an example of the `<inertial>` tag definition:

```
<inertial>
  <mass value="0.564" />
  <origin xyz="0 0 0" />
  <inertia ixx="0.003881243" ixy="0.0" ixz="0.0"
           iyy="0.000498940" iyz="0.0"
           izz="0.003879257" />
</inertial>
```

These parameters are part of the robot's dynamics, so in the simulation these values will have an effect on the robot model. Also, in the simulation, it will process all the links and joints, as well as its properties.

Next, we will look at the tags inside the `gazebo/chefbot.gazebo.xacro` file. The important Gazebo-specific tag we are using is `<gazebo>`, which is used to define the simulation properties of an element in the robot. We can either define a property that is applicable to all the links or one that is specific to a link. Here is a code snippet inside the xacro file that defines the coefficient of the friction of a link:

```
<gazebo reference="chefbot_wheel_left_link">
  <mu1>1.0</mu1>
  <mu2>1.0</mu2>
  <kp>1000000.0</kp>
  <kd>100.0</kd>
  <minDepth>0.001</minDepth>
  <maxVel>1.0</maxVel>

</gazebo>
```

The `reference` property is used to specify a link in the robot. So, the preceding properties will only be applicable to the `chefbot_wheel_left_link`.

The following code snippet shows you how to set the color of a robot link. We can create custom colors, define the custom colors, or use the default colors in Gazebo. You can see that for the `base_link`, we are using the `Gazebo/White` color from Gazebo's default property:

```
<material name="blue">
    <color rgba="0 0 0.8 1"/>
</material>

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

Refer to `http://gazebosim.org/tutorials/?tut=ros_urdf` to see all the tags that are used in the simulation.

That covers the main tags of the simulation. Now we will look at the Gazebo-ROS plugins that we have used in this simulation.

Cliff sensor plugin

The cliff sensor is a set of IR sensors that detect cliffs, which helps to avoid steps and prevents the robot from falling. This is one of the sensors in the mobile base of Turtlebot 2, called Kobuki (`http://kobuki.yujinrobot.com/`). We're using this plugin in the Turtlebot 2 simulation.

We can set the parameters of the sensors, such as the minimum and maximum angle of the IR beams, the resolution, and the number of samples per second. We can also limit the detection range of the sensor. There are three cliff sensors in our simulation model, as shown in the following code:

```
<gazebo reference="cliff_sensor_front_link">
  <sensor type="ray" name="cliff_sensor_front">
    <always_on>true</always_on>
    <update_rate>50</update_rate>
    <visualize>true</visualize>
    <ray>
      <scan>
        <horizontal>
          <samples>50</samples>
          <resolution>1.0</resolution>
          <min_angle>-0.0436</min_angle>  <!-- -2.5 degree -->
          <max_angle>0.0436</max_angle> <!-- 2.5 degree -->
        </horizontal>

      </scan>
      <range>
        <min>0.01</min>
        <max>0.15</max>
        <resolution>1.0</resolution>
      </range>
    </ray>
  </sensor>
</gazebo>
```

Contact sensor plugin

Here is the code snippet for the contact sensor on our robot. If the base of the robot collides with any objects, this plugin will trigger. It is commonly attached to the `base_link` of the robot, so whenever the bumper hits any object, this sensor will be triggered:

```
<gazebo reference="base_link">
  <mu1>0.3</mu1>
  <mu2>0.3</mu2>
  <sensor type="contact" name="bumpers">
    <always_on>1</always_on>
    <update_rate>50.0</update_rate>
    <visualize>true</visualize>
    <contact>
      <collision>base_footprint_collision_base_link</collision>
    </contact>
  </sensor>
</gazebo>
```

Gyroscope plugin

The gyroscope plugin is used to measure the angular velocity of the robot. Using the angular velocity, we can compute the orientation of the robot. The orientation of the robot is used in the robot drive controller for computing the robot's pose, as shown in the following code:

```
<gazebo reference="gyro_link">
 <sensor type="imu" name="imu">
   <always_on>true</always_on>
   <update_rate>50</update_rate>
   <visualize>false</visualize>
   <imu>
     <noise>
       <type>gaussian</type>
        <rate>
          <mean>0.0</mean>
          <stddev>${0.0014*0.0014}</stddev> <!-- 0.25 x 0.25 (deg/s) -
->
          <bias_mean>0.0</bias_mean>
          <bias_stddev>0.0</bias_stddev>
        </rate>
            <accel> <!-- not used in the plugin and real robot, hence
using tutorial values -->
                    <mean>0.0</mean>
                    <stddev>1.7e-2</stddev>
```

```
                        <bias_mean>0.1</bias_mean>
                        <bias_stddev>0.001</bias_stddev>
                    </accel>
            </noise>
           </imu>
                  </sensor>
      </gazebo>
```

Differential drive plugin

The differential drive plugin is the most important plugin of the simulation. This plugin simulates the differential drive behavior in the robot. It will move the robot model when it receives the command velocity (the linear and angular velocity) in the form of ROS Twist messages (`geometry_msgs/Twist`). This plugin also computes the odometry of the robot, which gives the local position of the robot, as shown in the following code:

```
    <gazebo>
         <plugin name="kobuki_controller" filename="libgazebo_ros_kobuki.so">
           <publish_tf>1</publish_tf>

           <left_wheel_joint_name>wheel_left_joint</left_wheel_joint_name>
           <right_wheel_joint_name>wheel_right_joint</right_wheel_joint_name>
           <wheel_separation>.30</wheel_separation>
           <wheel_diameter>0.09</wheel_diameter>
           <torque>18.0</torque>
           <velocity_command_timeout>0.6</velocity_command_timeout>
           <cliff_detection_threshold>0.04</cliff_detection_threshold>
           <cliff_sensor_left_name>cliff_sensor_left</cliff_sensor_left_name>
<cliff_sensor_center_name>cliff_sensor_front</cliff_sensor_center_name>
<cliff_sensor_right_name>cliff_sensor_right</cliff_sensor_right_name>
           <cliff_detection_threshold>0.04</cliff_detection_threshold>
           <bumper_name>bumpers</bumper_name>

            <imu_name>imu</imu_name>
         </plugin>
       </gazebo>
```

To compute the robot's odometry, we have to provide the robot's parameters, such as the distance between the wheels, wheel diameter, and the torque of the motors. According to our design, the wheel separation is 30 cm, the wheel diameter is 9 cm, and the torque is 18 N. If we want to publish the transformation of the robot, we can set the `publish_tf` as 1. Each tag inside the plugin is the parameter of the corresponding plugin. As you can see, it takes all the inputs from the contact sensor, imu, and cliff sensor.

The `libgazebo_ros_kobuki`.so plugin is installed along with Turtlebot 2 simulation packages. We are using the same plugin in our robot. We have to make sure that, the Turtlebot 2 simulation is installed on your system, prior to running this simulation.

Depth camera plugin

The depth camera plugin simulates the characteristics of a depth camera, such as Kinect or Astra. The plugin name is `libgazebo_ros_openni_kinect.so`, and it helps us to simulate different kinds of depth sensors that have different characteristics. The plugin is shown in the following code:

```
      <plugin name="kinect_camera_controller"
filename="libgazebo_ros_openni_kinect.so">
          <cameraName>camera</cameraName>
          <alwaysOn>true</alwaysOn>
          <updateRate>10</updateRate>
          <imageTopicName>rgb/image_raw</imageTopicName>
          <depthImageTopicName>depth/image_raw</depthImageTopicName>
          <pointCloudTopicName>depth/points</pointCloudTopicName>
          <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
<depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopic
Name>
          <frameName>camera_depth_optical_frame</frameName>
          <baseline>0.1</baseline>
          <distortion_k1>0.0</distortion_k1>
          <distortion_k2>0.0</distortion_k2>
          <distortion_k3>0.0</distortion_k3>
          <distortion_t1>0.0</distortion_t1>
          <distortion_t2>0.0</distortion_t2>
          <pointCloudCutoff>0.4</pointCloudCutoff>
        </plugin>
```

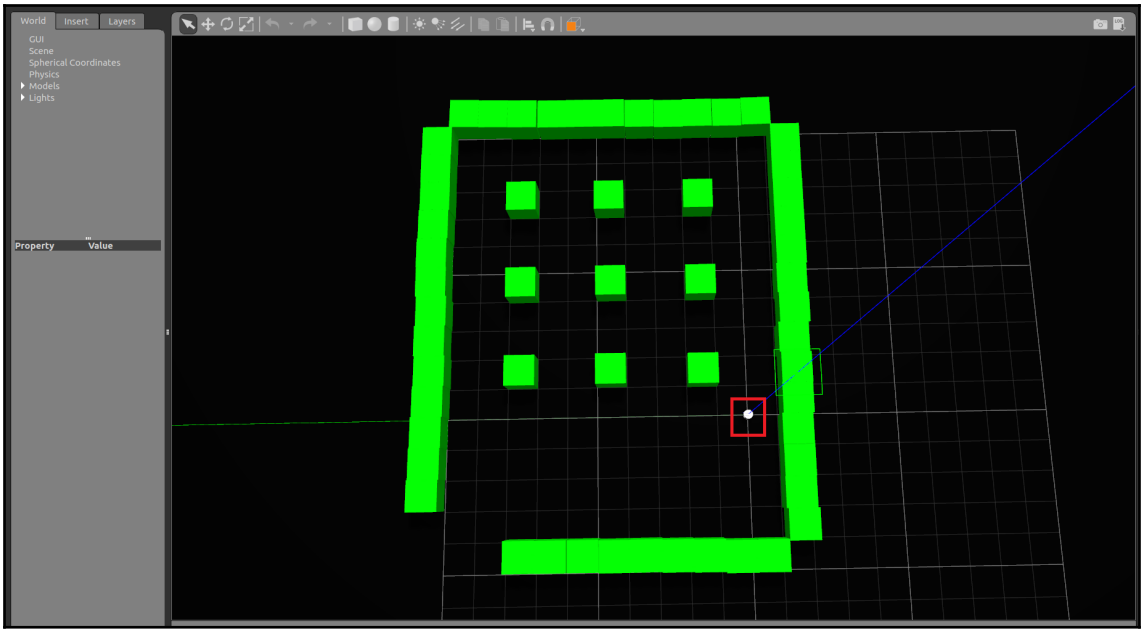
The plugin's publishers, the RGB image, depth image, and the point cloud data. We can set the camera matrix in the plugin, as well as customize other parameters.

You can refer to `http://gazebosim.org/tutorials?tut=ros_depth_cameraamp;cat=connect_ros` to learn more about the depth camera plugin in Gazebo.

Visualizing the robot sensor data

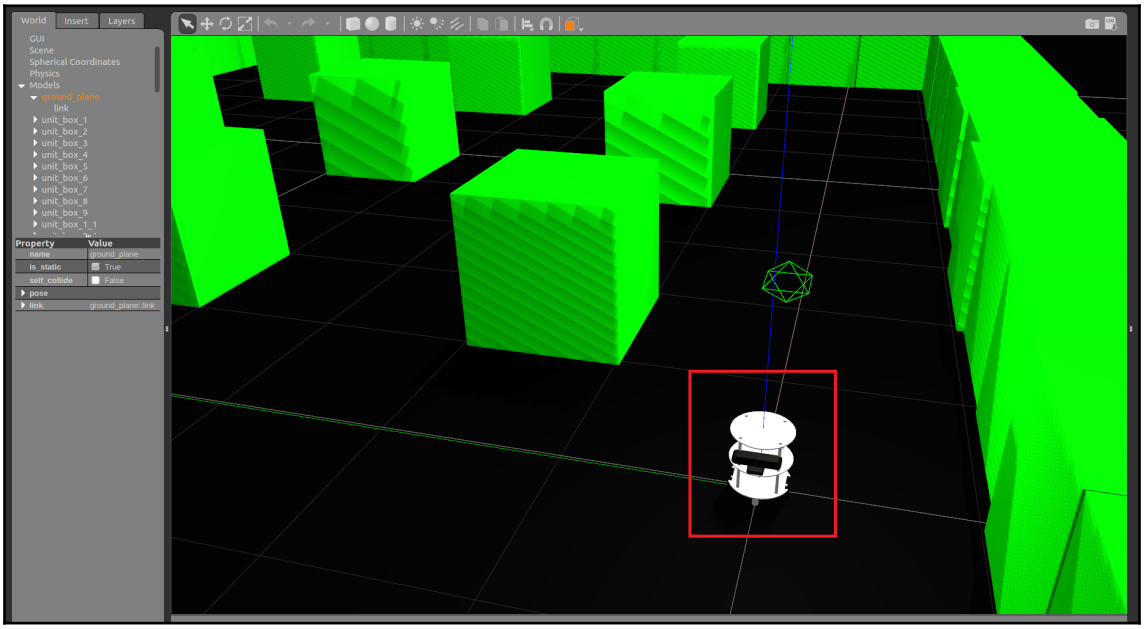
In this section, we learn how to visualize the sensor data from the simulated robot. In the `chefbot_gazebo` package, there are launch files to start the robot in an empty world or in a hotel-like environment. The custom environment can be built using Gazebo itself. Just create the environment using primitive meshes and save as a `*. world` file, which can be the input of the `gazebo_ros` node in the launch file. For starting the hotel environment in Gazebo, you can use the following command:

```
$ roslaunch chefbot_gazebo chefbot_hotel_world.launch
```



The Chefbot in Gazebo in the hotel environment

The nine cubes inside the space represent nine tables. The robot can navigate to any of the tables to deliver food. We will learn how to do this, but before that, we will learn how to visualize the different kinds of sensor data from the robot model.

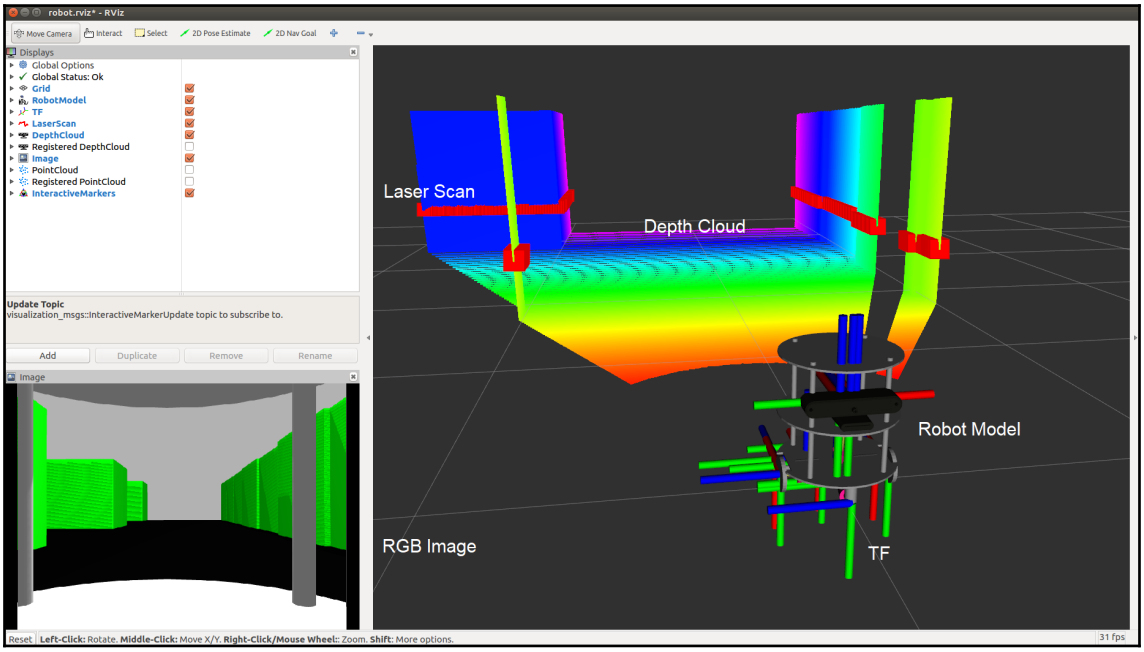


The Chefbot in Gazebo in the hotel environment

The following command will launch the Rviz, which displays the sensor data from the robot:

```
$ roslaunch chefbot_description view_robot.launch
```

This generates a visualization of the sensor data, as shown in the following screenshot:



The sensor visualization of Chefbot in Rviz

We can enable the Rviz display types to view different kinds of sensor data. In the preceding figure, you can see the depth cloud, laser scan, TF, robot model, and RGB camera images.

Getting started with Simultaneous Localization and Mapping

One of the requirements of the Chefbot was that it should be able to navigate the environment autonomously and deliver food. To achieve this requirement, we have to use several algorithms, such as SLAM (Simultaneous Localization and Mapping) and AMCL (Adaptive Monte Carlo Localization). There are different approaches to solving the autonomous navigation problem. In this book, we are mainly sticking with these algorithms. The SLAM algorithms are used for mapping an environment at the same time as localizing the robot on the same map. It's seems like a chicken-and-egg problem, but now there are different algorithms to solve it. The AMCL algorithm is used to localize the robot in an existing map. The algorithm that we use in this book is called Gmapping (`http://www.openslam.org/gmapping.html`), which implements Fast SLAM 2.0 (`http://robots.stanford.edu/papers/Montemerlo03a.html`). The standard gmapping library is wrapped in an ROS package called ROS Gmapping (`http://wiki.ros.org/gmapping`), which can be used in our application.

The idea of the SLAM node is that as we move the robot around the environment, it will create a map of the environment using the laser scan data and the odometry data.

Refer to the ROS Gmapping wiki page at `http://wiki.ros.org/gmapping` for more details.

Implementing SLAM in the Gazebo environment

In this section, we will learn how to implement SLAM and apply it to the simulation that we built. You can check the code at `chefbot_gazebo/launch/gmapping_demo.launch` and `launch/includes/ gmapping.launch.xml`. Basically, we are using a node from the gmapping package and configuring it with the proper parameters. The `gmapping.launch.xml` code fragment has the complete definition of this node. The following is the code snippet of this node:

```
<launch>
 <arg name="scan_topic" default="scan" />

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
output="screen">
    <param name="base_frame" value="base_footprint"/>
    <param name="odom_frame" value="odom"/>
```

```
<param name="map_update_interval" value="5.0"/>
<param name="maxUrange" value="6.0"/>
<param name="maxRange" value="8.0"/>
```

The name of the node that we are using is `slam_gmapping` and the package is `gmapping`. We have to provide a few parameters to this node, which can be found in the Gmapping wiki page.

Creating a map using SLAM

In this section, we will learn how to create a map of our environment using SLAM. First, however, there are several commands that we have to use to start mapping. You should execute each command in each Linux terminal.

First, we have to start our simulation using the following command:

```
$ roslaunch chefbot_gazebo chefbot_hotel_world.launch
```

Next, we have to start the keyboard teleoperation node in a new terminal. This will help us move the robot manually using the keyboard:

```
$ roslaunch chefbot_gazebo keyboard_teleop.launch
```

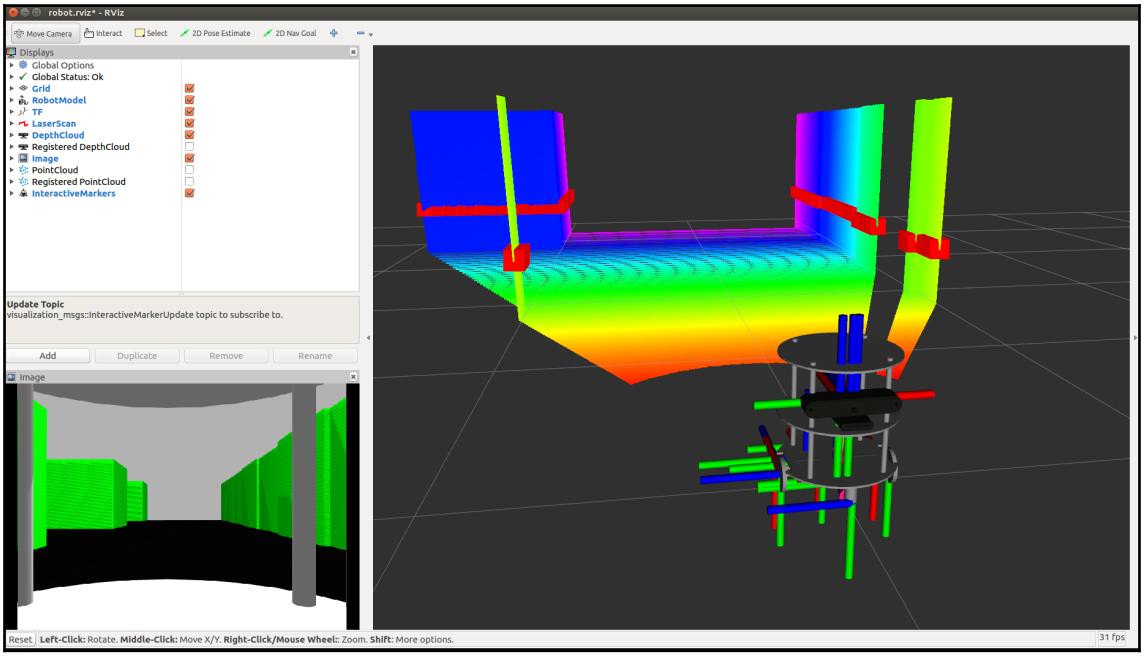
The next command starts the SLAM in a new terminal:

```
$ roslaunch chefbot_gazebo gmapping_demo.launch
```

Now the mapping will begin. To visualize the mapping process, we can start Rviz with the help of **Navigation** settings:

```
$ roslaunch chefbot_description view_navigation.launch
```

Now we can see the map created in Rviz, as shown in the following screenshot:



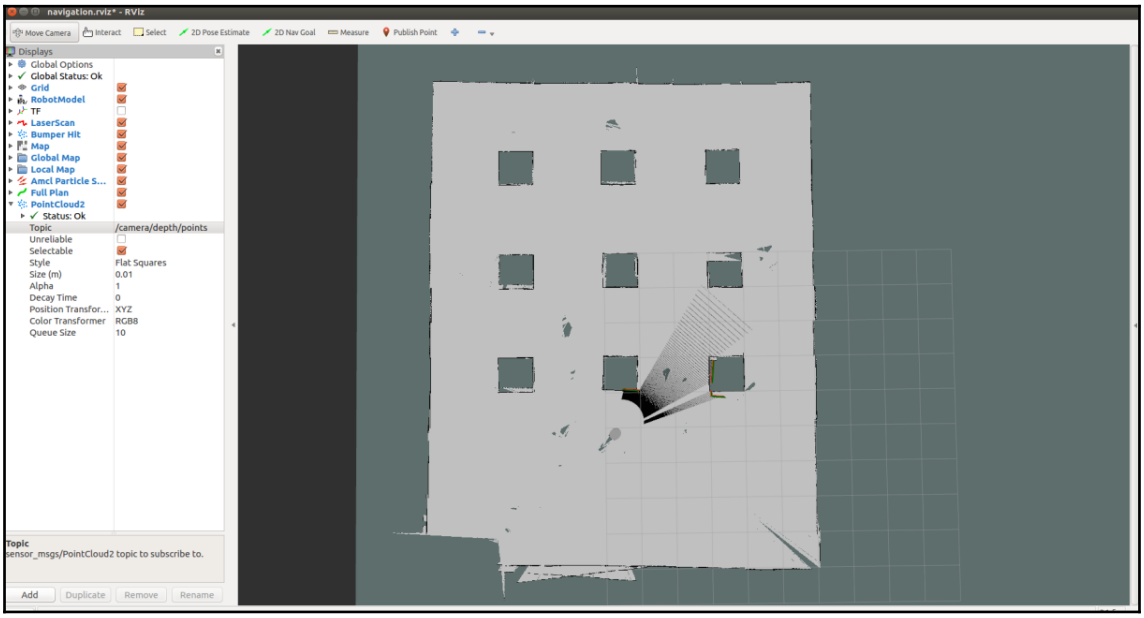
Creating a map in Rviz using Gmapping.

Now we can use the teleop node to move the robot, and you can see that a map is being created in Rviz. To create a good map of the environment, you have to move the robot slowly, and often you have to rotate the robot. When we move the robot in the environment and build the map, you can save the current map using the following command:

```
$ rosrun map_server map_saver -f ~/Desktop/hotel
```

The map will be saved as `*.pgm` and `*.yaml`, where the `pgm` file is the map and the `yaml` file is the configuration of the map. You can see the saved map in your desktop.

After moving the robot around the environment, you may get a complete map, such as the one shown in the following screenshot:



Final map using Gmapping.

The map can be saved at any time, but make sure that the robot covers the entire area of the environment and has mapped all of its space, as shown in the preceding screenshot. Once we are sure that the map is completely built, enter the `map_saver` command again and close the terminals. If you aren't able to map the environment, you can check the existing map from `chefbot_gazebo/maps/hotel`.

Getting started with Adaptive Monte Carlo Localization

We have successfully built the map of the environment. Now we have to navigate autonomously from the current robot position to target position. The first step before starting autonomous navigation is localizing the robot in the current map. The algorithm we are using to localize on the map is called AMCL. The AMCL uses a particle filter to track the robot's position with respect to the map. We are using an ROS package to implement AMCL in our robot (`http://wiki.ros.org/amcl`). Similar to Gmapping, there are a lot of parameters to configure for the `amcl` node, which is inside the `amcl` package. You can find all the parameters of amcl on the ROS wiki page itself.

So how we can start AMCL for our robot? There is a launch file for doing that, which is placed in `chefbot_gazebo/amcl_demo.launch` and `chefbot_gazebo/includes/amcl.launch.xml`.

We can see the definition of `amcl_demo.launch`. The following code shows the definition of this launch file:

```
<launch>
  <!-- Map server -->
  <arg name="map_file" default="$(find chefbot_gazebo)/maps/hotel.yaml"/>

  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />
```

The first node in this launch file starts `map_server` from the `map_server` package. The `map_server` node loads a static map that we have already saved and publishes it into a topic called `map` (`nav_msgs/OccupancyGrid`). We can mention the map file as an argument of the `amcl_demo.launch` file, and if there is a map file, the `map_server` node will load that; otherwise it will load the default map, which is located in the `chefbot_gazeob/maps/hotel.yaml` file.

After loading the map, we start the `amcl` node and move the base node. The AMCL node helps to localize the robot on the current `map` and the `move_base` node inside the ROS navigation stack, which helps in navigating the robot from the start to the target position. We will learn more about the `move_base` node in the upcoming chapters. The `move_base` node also needs to be configured with parameters. The parameter files are kept inside the `chefbot_gazebo/param` folder, as shown in the following code:

```
  <!-- Localization -->
  <arg name="initial_pose_x" default="0.0"/>
```

```
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find chefbot_gazebo)/launch/includes/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <!-- Move base -->
  <include file="$(find
chefbot_gazebo)/launch/includes/move_base.launch.xml"/>
</launch>
```

You can refer more about ROS navigation stack from following link
`http://wiki.ros.org/navigation/Tutorials/RobotSetup`

Implementing AMCL in the Gazebo environment

In this section, we will learn how to implement AMCL in our Chefbot. We will use the following procedures to incorporate AMCL within the simulation. Each command should be executed in each terminal.

The first command starts the Gazebo simulator:

```
$ roslaunch chefbot_gazebo chefbot_hotel_world.launch
```

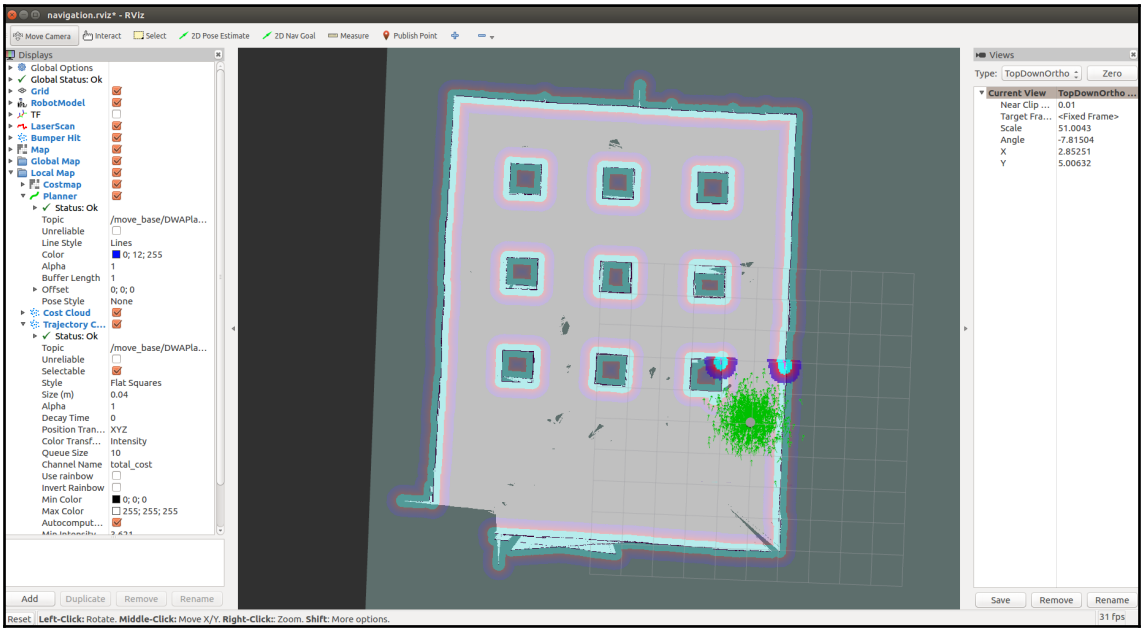
Now we can start the AMCL launch file, with or without the map file as an argument. If you want to use a custom map that you have built, then use the following command:

```
$ roslaunch chefbot_gazebo amcl_demo.launch
map_file:=/home/<your_user_name>/Desktop/hotel
```

If you want to use the default map, you can use the following command:

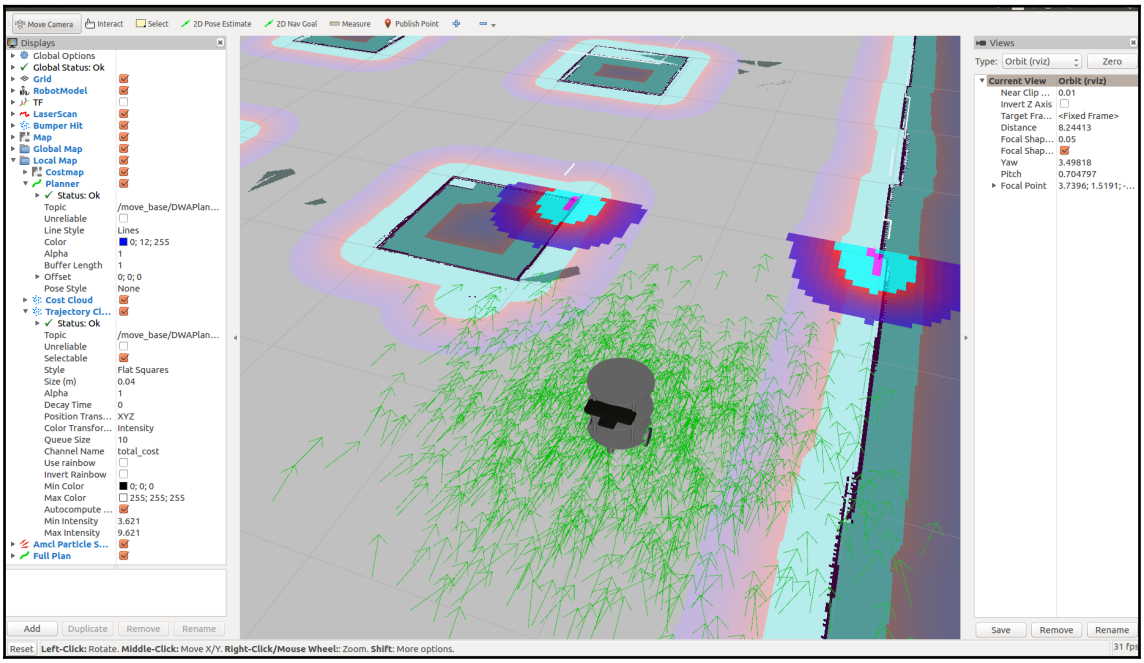
```
$ roslaunch chefbot_gazebo amcl_demo.launch
```

After starting AMCL, we can start Rviz to visualize the map and robot. We will see a view in Rviz as shown in the following screenshot. You can see a map and a robot surrounded by green particles. The green particles are called `amcl` particles. They indicate the uncertainty of the robot's position. If there are more particles around the robot, then this means that the uncertainty of the robot's position is higher. When it starts moving, the particle count will reduce and its position will be more certain. If the robot isn't able to localize the position of the map, we can use the *2D Pose Estimate* button in Rviz (on the toolbar) to manually set the initial position of the robot on the map. You can see the button in the following screenshot:



Starting AMCL on the hotel map.

If you zoom into the robot's position in Rviz, you can see the particles, as shown in the preceding screenshot. We can also see the obstacles around the robot in different colors:

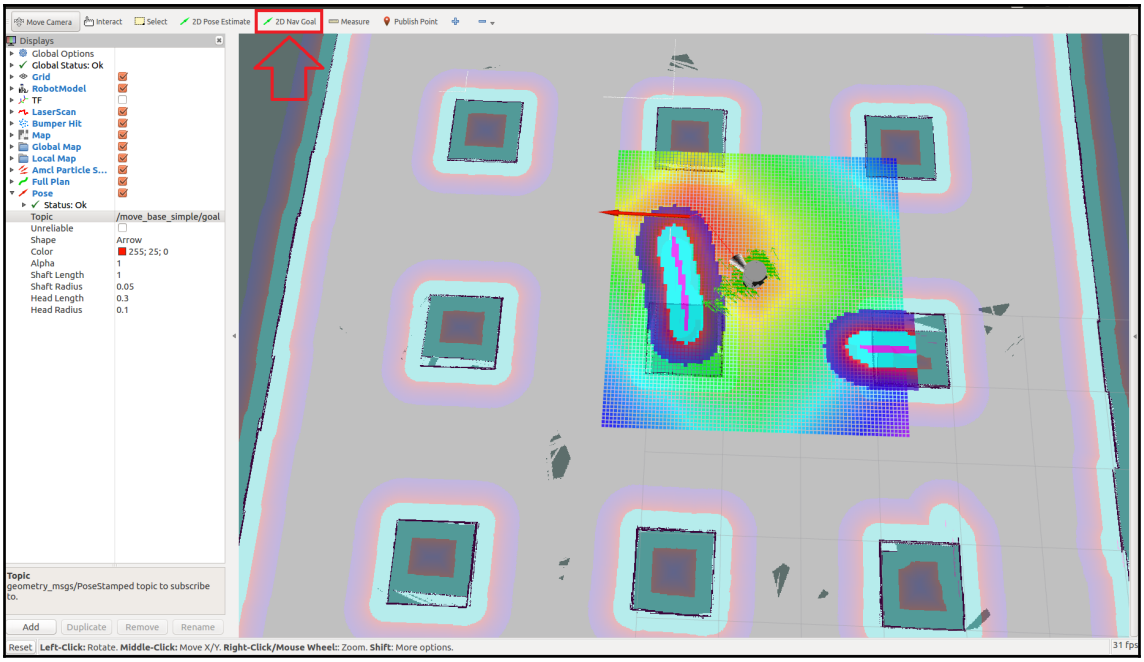


AMCL cloud around the robot.

In the next section, we will learn how to program the Chefbot to autonomously navigate this map. You don't need to close the current terminals; we can navigate the robot autonomously in the Rviz itself.

Autonomous navigation of Chefbot in the hotel using Gazebo

To start the robot's autonomous navigation, we just need to command the target robot position on the map. There is a button in Rviz called **2D Nav Goal**. We can click that button and click on a point on the map. You can now see an arrow indicating the position of the robot. When you give the target position in the map, you can see that the robot is planning a path from its current position to the target position. It will slowly move from its current position to the target position, avoiding all obstacles. The following screenshot shows the path planning and navigation of the robot to the target position. The color grid around the robot shows the local cost map of the robot, as well as the local planner path and the obstacles around the robot:



Autonomous navigation of the robot.

In this way, if we command a position inside the map that is nearer to a table, the robot can go to that table and serve the food and then return back to its home position. Instead of commanding it from Rviz, we can write an ROS node to do the same. This will be explained in the last few chapters of this book.

Summary

In this chapter, we learned how to simulate our own robot, called Chefbot. We looked at the design of the Chefbot in the previous chapter. We started the chapter by learning about the Gazebo simulator and its different features and capabilities. After that, we looked at how the ROS framework and Gazebo simulator are used to perform a robot simulation. We installed the TurtleBot 2 package and tested the Turtlebot 2 simulation in Gazebo. After that, we created the Chefbot simulation and used Gmapping, AMCL, and autonomous navigation in a hotel environment. We learned that the accuracy of the simulation depends on the map, and that the robot will work better in a simulation if the generated map is perfect.

In the next chapter, we will learn how to design the robot's hardware and electronic circuit.

Questions

1. How we can model a sensor in Gazebo?
2. How is ROS interfaced with Gazebo?
3. What are the important URDF tags for simulation?
4. What is Gmapping, and how we can implement it in ROS?
5. What is the function of the `move_base` node in ROS?
6. What is AMCL, and how we can implement it in ROS?

Further reading

To learn more about URDF, Xacro, and Gazebo, you can refer to the book *Mastering ROS for Robotics Programming - Second Edition* (`https://www.packtpub.com/hardware-and-creative/mastering-ros-robotics-programming-second-edition`).

5
Designing ChefBot Hardware and Circuits

In this chapter, we will discuss the design and workings of ChefBot hardware and look at a selection of its hardware components. In the previous chapter, we designed and simulated the basic robot framework in a hotel environment using Gazebo and ROS, and tested a few variables, such as the robot body mass, motor torque, wheel diameter, and more. We also tested the autonomous navigation capability of ChefBot in a hotel environment.

To achieve this using hardware, we need to select all the hardware components and figure out how to connect all these components. We know that the main functionality of this robot is navigation: this robot will have the ability to navigate from a start position to an end position without any collision with its surroundings. We will discuss the different sensors and hardware components required to achieve this goal. We will look at a block diagram representation of these components and its explanation, and also discuss the main functions and physical operations of the robot. Finally, we need to select the components required to build the robot. We will also familiarize ourselves with the online stores where we can purchase these components.

If you have a TurtleBot, you may skip this chapter because this chapter is only for those who need to create their robot's hardware. Let's look at the specifications that we have to meet in the design of the hardware. The robot hardware mainly includes the robot chassis, sensors, actuators, controller boards, and PC.

The following topics will be covered in this chapter:

- Block diagram and description of the Chefbot robot
- Robot component selection and description
- The workings of Chefbot's hardware

Technical requirements

The components required to build the robot are described in this chapter. You have to purchase these components or similar components in order to build the ChefBot.

Specifications of the ChefBot's hardware

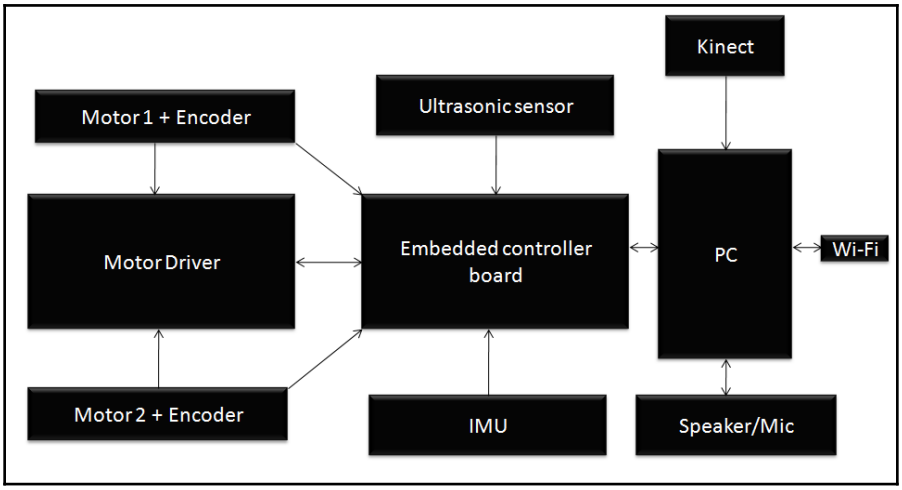
In this section, we will be discussing some of the important specifications that we mentioned in Chapter 3, *Modeling the Differential-Drive Robot*. The final robot prototype will meet the following specifications:

- **Simple and cost-effective robot chassis design**: The robot chassis design should be simple and cost effective compared to existing robots.
- **Autonomous navigation functionality**: The robot should autonomously navigate and it should contain the necessary sensors for doing this.
- **Long battery life**: The robot should have a long battery life in order to work continuously. The length of time that it can work should be greater than one hour.
- **Obstacle avoidance**: The robot should be able to avoid static and dynamic objects in its surroundings.

The robot hardware design should meet these specifications. Let's look at one of the possible ways of interconnecting the components in this robot. In the next section, we will look at a block diagram of the robot and use it to examine its workings.

Block diagram of the robot

The robot's movement is controlled by two **direct current** (**DC**) gear motors using an encoder. The two motors are driven using a motor driver. The motor driver is interfaced with an embedded controller board, which will send commands to the motor driver to control the motor's movements. The encoder of the motor is interfaced with the controller board in order to count the number of rotations of the motor shaft. This data is used to compute the odometry data of the robot. There are ultrasonic sensors that are interfaced with the controller board in order to sense the obstacles and measure the distance from the obstacles. There is an IMU sensor to improve odometry calculation. The embedded controller board is interfaced with a PC, which does all the high-end processing in the robot. Vision and sound sensors are interfaced with the PC and Wi-Fi is attached for remote operations. Each component of the robot is explained in the following diagram:



Robot hardware block diagram

Motor and encoder

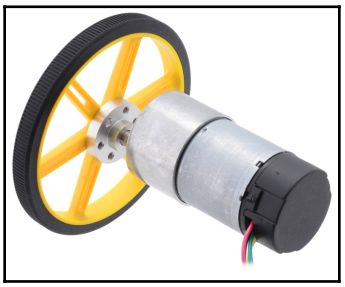
The robot that we are going to design is a differential-drive robot with two wheels, so we will require two motors for its locomotion. Each motor consists of quadrature encoders (`http://www.creative-robotics.com/quadrature-intro`) so that we can get motor rotation feedback data.

The quadrature encoder will send data regarding of the rotation of the motor as square pulses; we can decode the pulses to get the number of the encoder's ticks, which can be used for feedback. If we know the wheel's diameter and the number of ticks of the motor, we can compute the displacement and the angle of the robot that moved. This computation is very useful for us in our attempts to navigate the robot.

Selecting motors, encoders, and wheels for the robot

From the simulation, we got an idea of the robot's parameters. While experimenting with the simulation's parameters, we mentioned that the motor torque needed to drive the robot is 18 N, but the calculated torque is slightly more than this; we are selecting a standard torque motor that is very close to the actual torque in order to make the motor selection easier. One of the standard motors that we might consider is from Pololu. According to our design specifications, we could select a high-torque DC gear motor with an encoder working at 12 V DC and with a speed of 80 RPM.

The following image shows the selected motor for this robot. The motor comes with an integrated quadrature encoder with a resolution of 64 counts per revolution of the motor shaft, which corresponds to 8,400 counts per revolution of the gearbox's output shaft:



DC gear motor with encoder and wheel (see https://www.pololu.com/product/2827)

This motor has six differently colored pins . The descriptions of this motor's pins are given in the following table:

Color	**Function**
Red	Motor power (connects to one motor terminal)
Black	Motor power (connects to the other motor terminal)
Green	Encoder GND
Blue	Encoder Vcc (3.5 V-20 V)
Yellow	Encoder A output
White	Encoder B output

In accordance with our design specifications, we will choose a wheel diameter of 90 mm. Pololu provides a 90-mm wheel, which is available at `http://www.pololu.com/product/1439`. The preceding image showed the motor assembled with this wheel.

The other connectors needed to connect the motors and wheels together are available as follows:

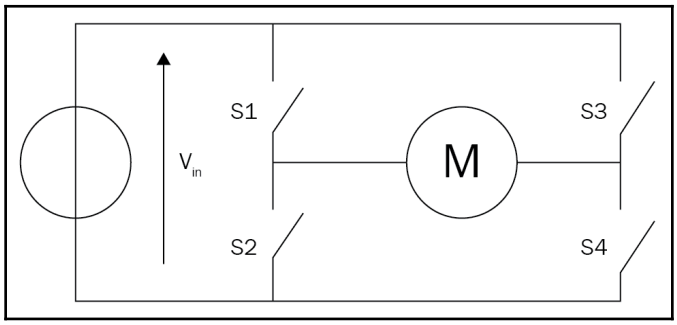
- The mounting hub required to mount the wheel to the motor shaft is available at `http://www.pololu.com/product/1083`.
- The L-bracket for the motor to mount onto the robot chassis is available at `http://www.pololu.com/product/1084`.

Motor driver

A **motor driver**, or **motor controller**, is a circuit that can control the speed of the motor. By controlling the motors, we mean that we can control the voltage across the motors and can also control the direction and speed of the motors. Motors can rotate clockwise or counter clockwise if we change the polarity of the motor terminal.

H-bridge circuits are commonly used in motor controllers. An **H-bridge** is an electronic circuit that can apply voltage in either direction of the load. It has high current-handling properties and can change the direction of the current flow.

The following diagram shows a basic H-bridge circuit using switches:



H -bridge circuit

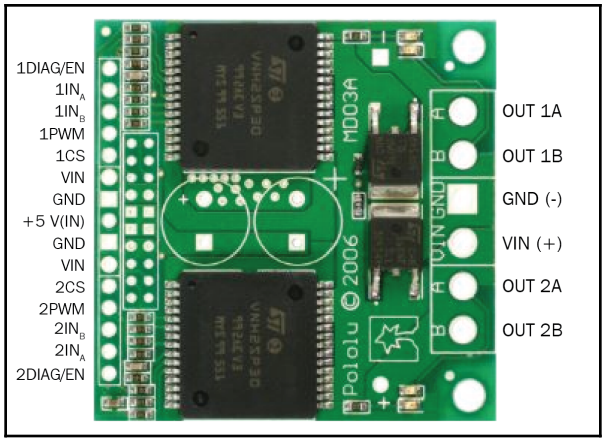
The direction of the motor according to the state of the four switches is given as follows:

S1	S2	S3	S4	Result
1	0	0	1	Motor moves right
0	1	1	0	Motor moves left
0	0	0	0	Motor free runs
0	1	0	1	Motor brakes
1	0	1	0	Motor brakes
1	1	0	0	Motor shoots through
0	0	1	1	Motor shoots through
1	1	1	1	Motor shoots through

We have seen the basics of an H-bridge circuit in the preceding motor driver circuit diagram. Now, we will select one of the motor drivers for our application and discuss how it works.

Selecting a motor driver/controller

There are some motor drivers available in Pololu that are compatible with the selected motor. The following image shows one of the motor drivers that we will use in our robot:



Dual VNH2SP30 motor driver carrier MD03A

This motor driver is available at `http://www.pololu.com/product/708`.

This driver can drive two motors with a maximum current rating of 30 A, and contains two integrated ICs for driving each of the motors. The pin description of this driver is given in the upcoming sections.

Input pins

The following pins are the input pins of the motor driver, through which we can control mainly the motor speed and direction:

Pin name	Function
1DIAG/EN, 2DIAG/EN	These monitor the fault conditions of motor drivers 1 and 2. In a normal operation, they will remain disconnected.
1INa, 1INb, 2INa, 2INb	These pins will control the direction of motors 1 and 2 in the following manner: • If INA = INB = 0, the motor will break • If INA = 1, INB = 0, the motor will rotate clockwise • If INA = 0, INB = 1, the motor will rotate counter clockwise • * If INA = INB = 1, the motor will break
1PWM, 2PWM	This will control the speed of motors 1 and 2 by turning them on and off at very high speed.
1CS, 2CS	This is the current sensing pin for each motor.

Output pins

The output pins of the motor driver will drive the two motors. The following are the output pins:

Pin name	Function
OUT 1A, OUT 1B	These pins can connect to motor 1's power terminals.
OUT 2A, OUT 2B	These pins can connect to motor 2's power terminals.

Power supply pins

The following are the power supply pins:

Pin name	Function
VIN (+), GND (-)	These are the supply pins of the two motors. The voltage ranges from 5.5 V to 16 V.
+5 VIN, GND (-)	This is the power supply of the motor driver. The voltage should be 5 V.

Embedded controller board

Controller boards are typically I/O boards that can send control signals in the form of digital pulses to the H-bridge/motor-driver board and can receive inputs from sensors, such as ultrasonic and IR sensors. We can also interface motor encoders with the control board in order to send data from the motor.

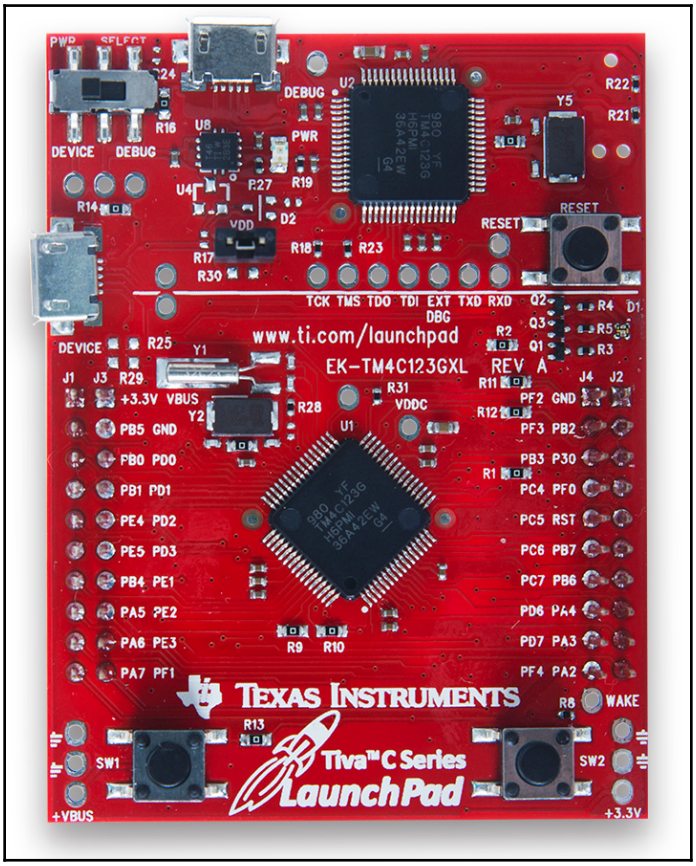
The main uses of the controller board in this robot are as follows:

- Interfacing the motor driver and encoder
- Interfacing the ultrasonic sound sensor
- Sending and receiving sensor values to and from the PC

We will deal with I/O boards and interfacing with different components in the upcoming chapters. Some of the more popular I/O boards are Arduino (arduino.cc) and Tiva-C LaunchPad (`http://www.ti.com/tool/EK-TM4C123GXL`) by Texas Instruments. We are selecting Tiva-C LaunchPad over Arduino because of the following factors:

- Tiva-C LaunchPad has a microcontroller based on a 32-bit ARM Cortex-M4 with 256 KB flash memory, 32 KB SRAM, and 80 MHz data transmission frequency; most Arduino boards run below these specifications.
- Outstanding processing performance combined with fast interrupt handling.
- 12 timers.
- 16 PWM outputs.
- 2 quadrature encoder inputs.
- 8 **universal asynchronous receiver/transmitters** (**UART**).
- 5 V-tolerant **general-purpose input/output** (**GPIO**).
- Low cost and size compared to Arduino boards.
- Easily programmable interface IDE called Energia (http://energia.nu/). The code written in Energia is Arduino-board compatible.

The following image shows Texas Instrument's Tiva-C LaunchPad:



Tiva-C LaunchPad 123 (http://www.ti.com/tool/EK-TM4C123GXL)

The pinout of Texas Instrument's LaunchPad series is given at `http://energia.nu/pin-maps/guide_stellarislaunchpad/`. This pinout map is compatible with all LaunchPad series releases. This can also be used while programming in the Energia IDE.

Ultrasonic sensors

Ultrasonic sensors, also called ping sensors and are mainly used to measure the distance from an object. The main application of ping sensors is to avoid obstacles. The ultrasonic sensor emits high-frequency sound waves and evaluates the echoes that it receives from the object. The sensor will calculate the delay between the sending and receiving of the echo and determine its distance to the object.

In our robot, collision-free navigation is an important part of the design specifications, otherwise there will be damage to the robot. You will see an image showing an ultrasonic sensor in the next section. This sensor can be installed on the sides of a robot to detect collisions at the sides and back of the robot. Kinect is also mainly used for obstacle detection and collision avoidance when used in robotics. Kinect can only be expected to be accurate at a range of 0.8 m, so the remaining distance from the 0.8 m-range limit can be detected using an ultrasonic sensor. In this case, the ultrasonic sensor is actually an add-on to our robot in order to increase its collision avoidance and detection abilities.

Selecting an ultrasonic sensor

One of the more popular and cheap ultrasonic sensors available is **HC-SR04**. We are selecting this sensor for our robot because of the following factors:

- Range of detection is from 2 cm to 4 m
- Working voltage is 5 V
- Working current is very low, typically 15 mA

We can use this sensor to accurately detect obstacles. It also works at 5 V. Here is an image of HC-SR04 and its pinout:

Ultrasonic sound sensor (https://www.makerfabs.com/index.php?route=product/product&product_id=72)

The pins and their functions are given as follows:

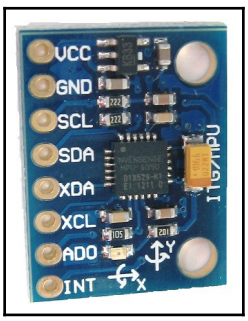
Pins	Function
Vcc, GND	These are the supply pins of the ultrasonic sensor. Usually, we need to apply 5 V for it to operate normally.
Trig	This is the input pin of the sensor. We need to apply a pulse with a particular duration to this pin to send the ultrasonic sound waves.
Echo	This is the output pin of the sensor. It will generate a pulse on this pin with a time duration according to the delay in receiving the triggered pulse.

Inertial measurement unit

We will use **the inertial measurement unit** (**IMU**) in this robot to get a good estimate of the odometry value and the robot's pose. The odometry values computed from the encoder alone may not be sufficient for efficient navigation, as they can contain errors. To compensate for the error during the robot's movement, especially rotation, we will use the IMU in this robot. We are selecting MPU 6050 for the IMU because of the following reasons:

- In MPU 6050, the accelerometer and gyroscope are integrated on a single chip
- It provides high accuracy and sensitivity
- We are able to interface with a magnetometer for better IMU performance
- The breakout board of MPU 6050 is very cheap
- MPU 6050 can directly interface with LaunchPad
- Both MPU 6050 and LaunchPad are 3.3 V-compatible
- Software libraries are also available for easier interfacing between MPU 6050 and LaunchPad

The following image shows the breakout board of MPU 6050:



The MPU 6050 device

The pins and their functions are given as follows:

Pins	Functions
VDD, GND	Supply voltage 2.3 V-3.4 V
INT	This pin will generate an interrupt when data comes to the device buffer
SCL, SDA	**Serial data line** (**SDA**) and **serial clock line** (SCL) are used for I2C communication
ASCL, ASDA	Auxiliary I2C for communication with the magnetometer

We can purchase the breakout board from Amazon at `http://a.co/9EBIquO`.

Kinect/Orbbec Astra

Kinect is a 3D-vision sensor, mainly used in 3D-vision applications and motion-based gaming. We are using Kinect for 3D vision. Using Kinect, the robot will get a 3D image of the surrounding. The 3D images are converted to finer points that are gathered to form a point cloud. The point cloud data will have all the 3D parameters that constitute the surrounding environment.

The main use of Kinect on the robot is to mock the functionality of a laser scanner. The laser scanner data is essential for the SLAM algorithm to build a map of the environment. The laser scanner is a very costly device, so instead of buying an expensive laser scanner, we can convert a Kinect into a virtual laser scanner. Kinect has officially stopped production, but it's still available from some vendors. One of the alternatives to Kinect is Orbbec Astra (`https://orbbec3d.com/product-astra/`). It will support the same software that is written for Kinect. The point-cloud-to-laser-data conversion is done with this software, so we only need to change the device driver if you are using Astra; the resetting of the software is the same. After generating the map of the environment, the robot can navigate the surroundings. The following image shows the Kinect sensor (A) and Orbbec Astra (B):

Kinect and Orbbec Astra

Kinect mainly has an IR camera and projector, as well as an RGB camera. The IR camera and projector generates the 3D point cloud of the surrounding area. It also has a mic array and motorized tilt for moving the Kinect up and down. The Astra is very similar to Kinect.

We can purchase Kinect from
`http://www.amazon.co.uk/Xbox-360-Kinect-Sensor-Adventures/dp/B0036DDW2G`.

We can purchase Astra from `https://orbbec3d.com/product-astra/`.

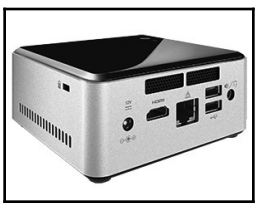
Central processing unit

The robot is mainly controlled by the navigational algorithm that is running on its PC. We can choose a laptop, mini PC, or net book to use for the robot's processing functionalities. Recently, Intel launched a mini PC called Intel **Next Unit of Computing** (**NUC**). It has an ultra-small form factor (size), is lightweight, and has a good computing processor with either Intel Celeron, Core i3, or Core i5. It can support up to 16 GB of RAM and has integrated Wi-Fi/Bluetooth. We are choosing Intel NUC because of its performance, ultra-small form factor, and its light weight. We are not going for a popular board, such as Raspberry Pi (`http://www.raspberrypi.org/`) or Beagle Bone (`http://beagleboard.org/`), because we require high-computing power, and this cannot be provided by these boards.

The NUC we are using is **Intel DN2820FYKH**. Here is the specification of this computer:

- Intel Celeron dual core processor with 2.39 GHz
- 4 GB RAM
- 500 GB hard disk
- Intel-integrated graphics
- Headphone/microphone jack
- 12 V supply

The following image shows the Intel NUC minicomputer:



Intel NUC DN2820FYKH

We can purchase NUC from Amazon at `http://a.co/2F2flYl`.

This model of NUC is an old model; if it is not available, you can check for a low-cost NUC using the following links shown:

- Intel NUC BOXNUC6CAYH (`https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc6cayh.html`)
- **Intel NUC KIT NUC7CJYH** (`https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc7cjyh.html`)
- **Intel NUC KIT NUC5CPYH** (`https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc5cpyh.html`)
- **Intel NUC KIT NUC7PJYH** (`https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc7pjyh.html`)

Speakers/mic

The main function of the robot is autonomous navigation. We will add an additional feature with which the robot can interact with users through speech. The robot can be given commands using voice input and can speak to the user using a **text-to-speech** (**TTS**) engine, which can convert text to speech format. A microphone and speakers are essential for this application. There is no particular selection that we will recommend for this hardware. If the speaker and mic are USB compatible, then that will be great. One of the other alternatives is a Bluetooth headset.

Power supply/battery

One of the most important hardware components is the power supply. We saw in the specification that the robot has to work for more than one hour. It will be good if the supply voltage of the battery is compatible with that required by the components. Also, if the size and weight of the battery is less than what we had in mind, it will not affect the robot's payload.

Another concern is that the maximum current needed for the entire circuit will not exceed the battery's maximum current, that it can source. The maximum voltage and current distribution of each part of the circuit is as follows:

Component	**Maximum current (in amperes)**
Intel NUC PC	12 V, 5 A
Kinect	12 V, 1 A
Motors	12 V, 0.7 A
Motor driver, ultrasonic sensors, IMU, speakers	5 V, < 0.5 A

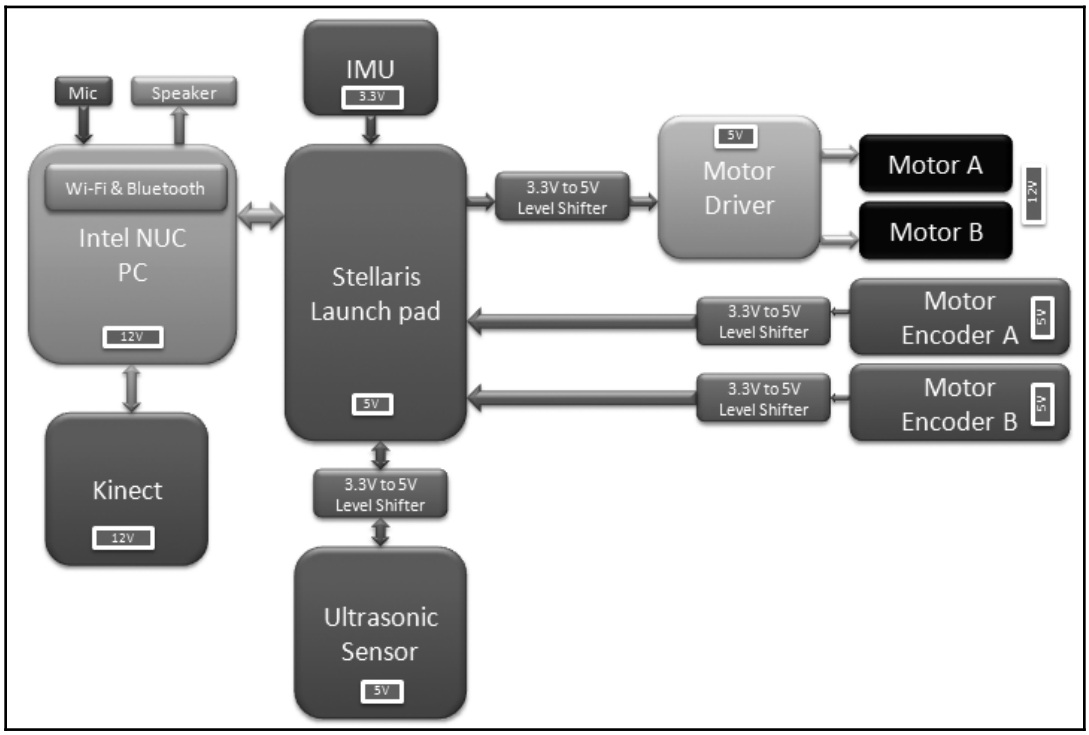
To meet these specifications, we are selecting a 12 V, 10 AH li-polymer or sealed lead acid (SLA) battery for our robot. Here is a typical low-cost SLA battery that we can use for this purpose:

Sealed lead acid battery

We can buy this battery from `http://a.co/iOaMuZe`. You can choose a battery based on how convenience it is for you, but it should satisfy the robot's power requirements.

How ChefBot's hardware works'?

We can explain how ChefBot's hardware works using the following block diagram. This is an improved version of our first block diagram, as it mentions the voltage of each component and its interconnection:



Detailed block diagram of Chefbot's hardware

The main aim of this chapter was to design the hardware for ChefBot, which included finding the appropriate hardware components and learning about the interconnection between each part. The main functionality of this robot is to perform autonomous navigation. The hardware design of the robot is optimized for autonomous navigation.

The robot drive is based on the differential-drive system, which consists of two motors and two wheels. There are caster wheels for supporting the main wheels. These two motors can move the robot to face in any direction on a 2D plane by adjusting their direction and speed of rotation.

For controlling the velocity and direction of the wheels, we have to interface a motor controller, which can perform these functions. The motor driver we choose should able to control two motors at the same time, and it should also be able to change their direction and speed.

The motor driver pins are interfaced with a microcontroller board called Tiva-C LaunchPad, which can send the commands to change the direction and speed of the motor. The motor driver is interfaced with LaunchPad with the help of a level shifter. The **level shifter** is a circuit that can shift voltage levels from 3.3 V to 5 V and vice versa. We are using a level shifter because the motor driver is operating at a level of 5 V, but the LaunchPad board is operating at 3.3 V.

Each motor has a rotation feedback sensor called an encoder, which can be used to estimate the robot's position. The encoders are interfaced with LaunchPad with the level shifter.

Other sensors that are interfaced with LaunchPad include an ultrasonic sound sensor and IMU. The ultrasonic sound sensor can detect objects that are close by, but that cannot be detected by the Kinect sensor. IMU is used along with encoders to get a good estimation of the robot's pose.

All sensor values are received on the LaunchPad and sent to the PC via USB. The LaunchPad board runs a firmware code that can receive all sensor values and send them to the PC.

The PC is interfaced with Kinect, the LaunchPad board, the speaker, and the mic. The PC has ROS running on it, and it will receive Kinect data and convert it to its equivalent laser scanner data. This data can be used to build a map of the environment using SLAM. The speaker and mic are used for communication between the user and robot. The speed commands generated in ROS nodes are sent to LaunchPad. LaunchPad will process the speed commands and send the appropriate PWM values to the motor driver circuit.

After designing and discussing the workings of the robot's hardware, we can discuss the detailed interfacing of each component and the firmware coding necessary for this interfacing in the next chapter.

Summary

In this chapter, we have looked at the features of the robot that we are going to design. The main feature of this robot is its autonomous navigation. The robot can navigate its surroundings by analyzing sensor readings. We looked at the robot's block diagram and discussed the role of each block, selecting the appropriate components that satisfy our requirements. We also suggested some economical components with which to build this robot. In the next chapter, we will take a closer look at actuators and the interfacing that we will use for them in this robot.

Questions

1. What is robot hardware design all about?
2. What is an H-bridge circuit and what are its functions?
3. What are the essential components for a robot's navigation algorithm?
4. What are the criteria that have to be kept in mind while selecting robotic components?
5. What are the main applications of Kinect as regards this robot?

Further reading

You can learn more about the Tiva-C LaunchPad board from the following link:

`http://processors.wiki.ti.com/index.php/Getting_Started_with_the_TIVA%E2%84%A2_C_Series_TM4C123G_LaunchPad`

6
Interfacing Actuators and Sensors to the Robot Controller

In the previous chapter, we discussed the selection of the hardware components needed to build our robot. The important components in a robot are actuators and sensors. Actuators provide mobility to the robot and sensors provide information about the robot environment. In this chapter, we will concentrate on the different types of actuators and sensors that we are going to use in this robot and how they can be interfaced with Tiva C LaunchPad, which is a 32 bit ARM micro controller board from Texas Instruments, working at 80 MHz. We will start by discussing actuators. The actuator that we are going to discuss first is a DC-geared motor with an encoder. A DC-geared motor works using direct current and has gear reduction to reduce the shaft speed and increase the torque of the final shaft. These kinds of motors are very economical and satisfy our robot design requirement. We will use this motor in our robot prototype.

In the first section of this chapter, we will deal with the design of our robot drive system. The **drive system** of our robot is a differential drive and consists of two DC-geared motors with encoders and a motor driver. The motor driver is controlled by Tiva C LaunchPad. We will look at the interfacing of the motor driver and quadrature encoder with Tiva C Launchpad. After that, we will look at some of the latest actuators that can replace the existing DC-geared motor with an encoder. If the desired robot needs more payload and accuracy, we have to switch to these kinds of actuators. Finally, we will look at some different sensors that are commonly used for robots.

In this chapter, we will cover:

- Interfacing a DC-geared motor with Tiva C LaunchPad
- Interfacing a quadrature encoder with Tiva C LaunchPad
- An explanation of interfacing code
- Interfacing Dynamixel actuators
- Interfacing ultrasonic sensors and IR proximity sensors
- Interfacing inertial measurement units (IMUs)

Technical requirements

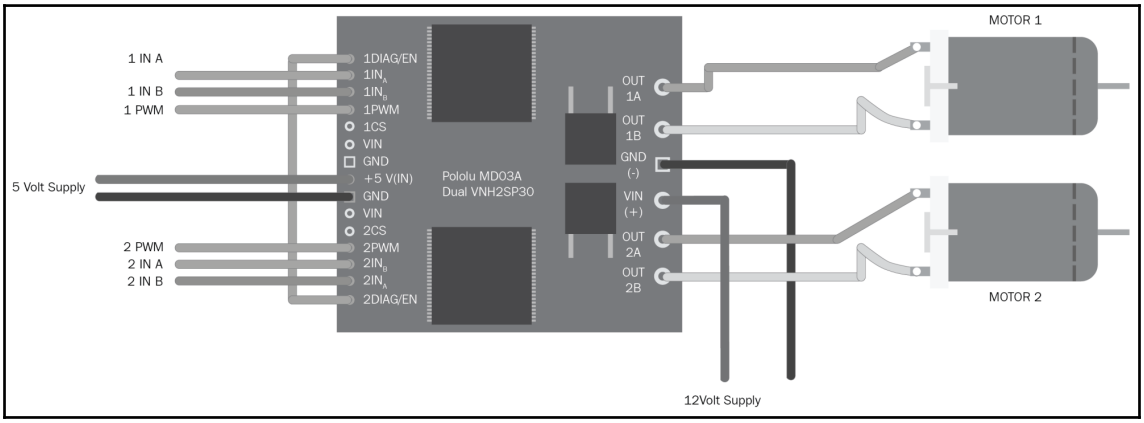
You will need the necessary robot hardware component and Energia IDE set up in Ubuntu 16.04 LTS.

Interfacing DC geared motor to Tiva C LaunchPad

In the previous chapter, we selected a DC-geared motor with an encoder from Pololu and an embedded board from Texas Instruments, called Tiva C LaunchPad. We need the following components to interface the motor with LaunchPad:

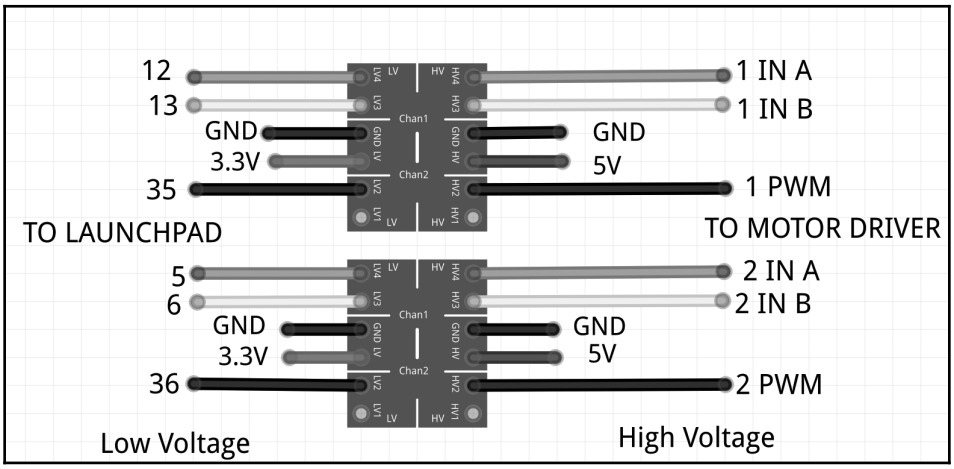
- Two Pololu metal gear motors, 37Dx73L mm with 64 counts per revolution encoder
- Pololu wheel, 90x10 mm and a matching hub
- Pololu dual VNH2SP30 motor driver carrier, MD03A
- A sealed lead acid/lithium ion battery of 12V
- A logic level convertor of 3.3V to 5V; visit `https://www.sparkfun.com/products/11978`.
- A Tiva C LaunchPad and its compatible interfacing wires

The following diagram shows the interfacing circuit of two motors using Pololu H-Bridge:



Motor interfacing circuit

To interface with Launchpad, we have to connect a level shifter board in between these two motors. The motor driver works in 5V but the Launchpad works in 3.3V, so we have to connect a level shifter, as shown in the following diagram:



Level shifter circuit

The two geared DC motors are connected to **OUT1A**, **OUT1B**, and **OUT2A**, **OUT2B** of the motor driver. **VIN (+)** and **GND (-)** are the supply voltage of the motor. These DC motors can work with a 12V supply, so we give 12V as the input voltage. The motor driver will support an input voltage ranging from 5.5V to 16V.

The control signals/input pins of the motor drivers are on the left side of the driver. The first pin is **1DIAG/EN**; in most cases, we leave this pin disconnected. These pins are externally pulled high in the driver board itself. The main use of this pin is to enable or disable the H-bridge chip. It is also used to monitor the faulty condition of the H-Bridge IC. Pins **1INA** and **1INB** control the direction of the rotation of the motor. The **1PWM** pin will switch the motor to the ON and OFF state. We achieve speed control using **PWM** pins. The **CS** pin will sense the output current. It will output 0.13V per Ampere of the output current. The **VIN** and **GND** pins give the same input voltage that we supplied for the motor. We are not using these pins here. The **+5V(IN)** and **GND** pins are the supply for the motor driver IC. The supply to the motor driver and motors are different.

The following table shows the truth table of the input and output combinations:

INA	INB	DIAGA/ENA	DIAGB/ENB	OUTA	OUTB	CS	Operating mode
1	1	1	1	H	H	High Imp	Brake to Vcc
1	0	1	1	H	L	Isense = Iout / K	Clockwise (CW)
0	1	1	1	L	H	Isense = Iout / K	Counterclockwise (CCW)
0	0	1	1	L	L	High Imp	Breaker to GND

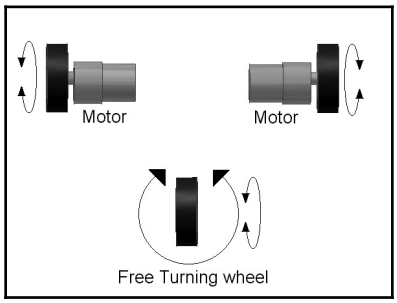
The value **DIAG/EN** pins are always high because these pins are externally pulled high in the driver board itself. Using the aforementioned signal combinations, we can move the robot in any direction and by adjusting the PWM signal, we can adjust the speed of the motor too. This is the basic logic behind controlling a DC motor using an H-Bridge circuit.

While interfacing motors to Launchpad, we may require a level shifter. This is because the output pins of Launchpad can only supply 3.3V but the motor driver needs 5V to trigger; so, we have to connect 3.3V to the 5V logic level convertor to start working.

The two motors work in a differential drive mechanism. The following section discusses the differential drive and its operation.

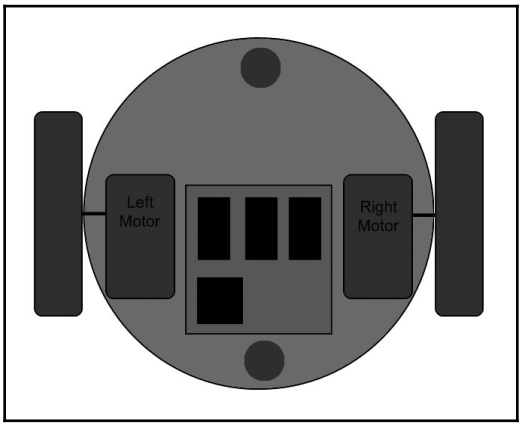
Differential wheeled robot

The robot we have designed is a differential wheeled/drive robot. In a differential wheeled robot, the movement is based on two separately driven wheels placed on either side of the robot's body. It can change its direction by changing the relative rate of rotation of its wheels, and hence, doesn't require additional steering motion. To balance the robot, a free turning wheel or caster wheels may be added. The following diagram shows a typical representation of a differential drive:



Differential wheeled robot

If the two motors are in the same direction, the robot will move forward or backward. If one motor has more speed than the other, then the robot turns to the slower motor side; so, to turn left, stop the left motor and move the right motor. The following diagram shows how we connect the two motors in our robot. The two motors are mounted on the opposite sides of the base plate and we put two casters in the front and back of the robot for balancing:



Top view of robot base

Next, we can program the motor controller using Launchpad according to the truth table data. Programming is done using an IDE called **Energia** (`http://energia.nu/`). We are programming Launchpad using the C++ language, very similar to Arduino boards (`http://energia.nu/Reference_Index.html`).

Installing Energia IDE

We can download the latest version of Energia from the following link:

`http://energia.nu/download/`

We will discuss the installation procedure mainly on Ubuntu 16.04 LTS, 64-bit. The Energia version that we will use is 0101E0018:

1. Download Energia for Linux 64-bit from the preceding link.
2. Extract the Energia compressed file into the Home folder of the user.
3. The instructions for setting the Tiva C Launchpad boards are given in the following link: `http://energia.nu/guide/guide_linux/`
4. You have to download the `71-ti-permissions.rules` file from the following link: `http://energia.nu/files/71-ti-permissions.rules`
5. The rules file will give permission to the user for reading and writing to the Launchpad board. You have to save the file as 71-ti-permissions.rules and execute the following command from the current path to copy the rules files into a system folder to get the permission:

```
$ sudo mv 71-ti-permissions.rules /etc/udev/rules.d/
```

6. After copying the file, execute the following command to activate the rules:

```
$ sudo service udev restart
```

7. You can plug the Tiva C Launchpad to your PC now and execute the `dmesg` command in the Linux Terminal to see the Linux kernel log. If it is created, a serial port device will show at the end of the messages, as shown in the following screenshot:

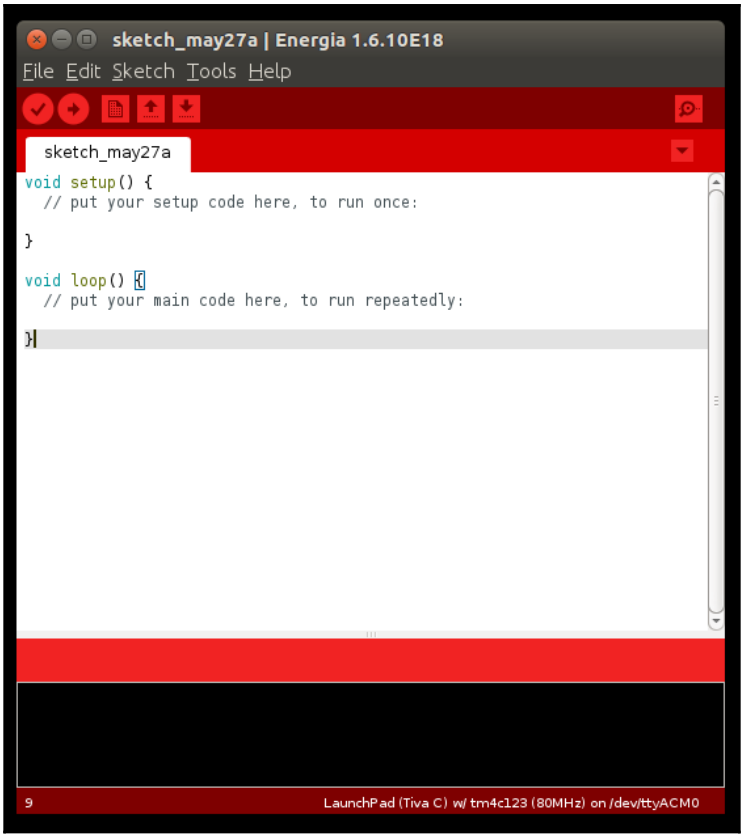
```
[  569.441209] usb 1-5: New USB device found, idVendor=1cbe, idProduct=00fd
[  569.441215] usb 1-5: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  569.441218] usb 1-5: Product: In-Circuit Debug Interface
[  569.441222] usb 1-5: Manufacturer: Texas Instruments
[  569.441225] usb 1-5: SerialNumber: 0E2258F8
[  569.461748] cdc_acm 1-5:1.0: ttyACM0: USB ACM device
[  569.461943] usbcore: registered new interface driver cdc_acm
[  569.461944] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
```

Top view of robot base

8. If you can see the serial port device, then start Energia using the following command inside the folder:

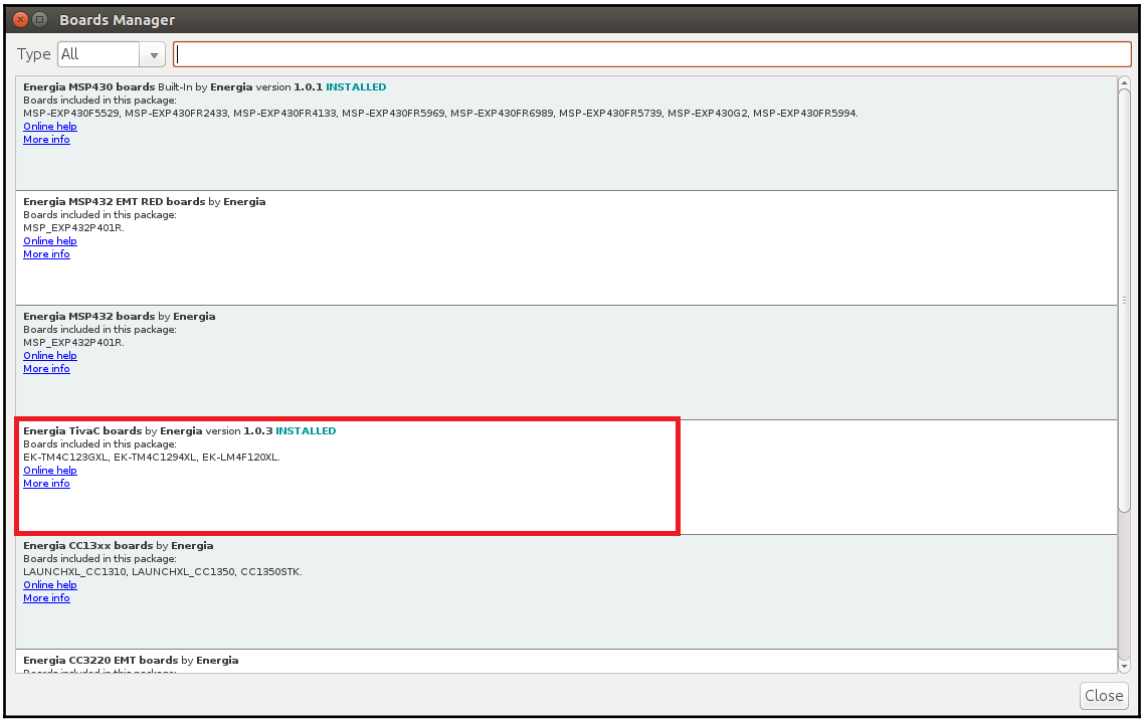
```
$./energia
```

The following screenshot shows the Energia IDE:



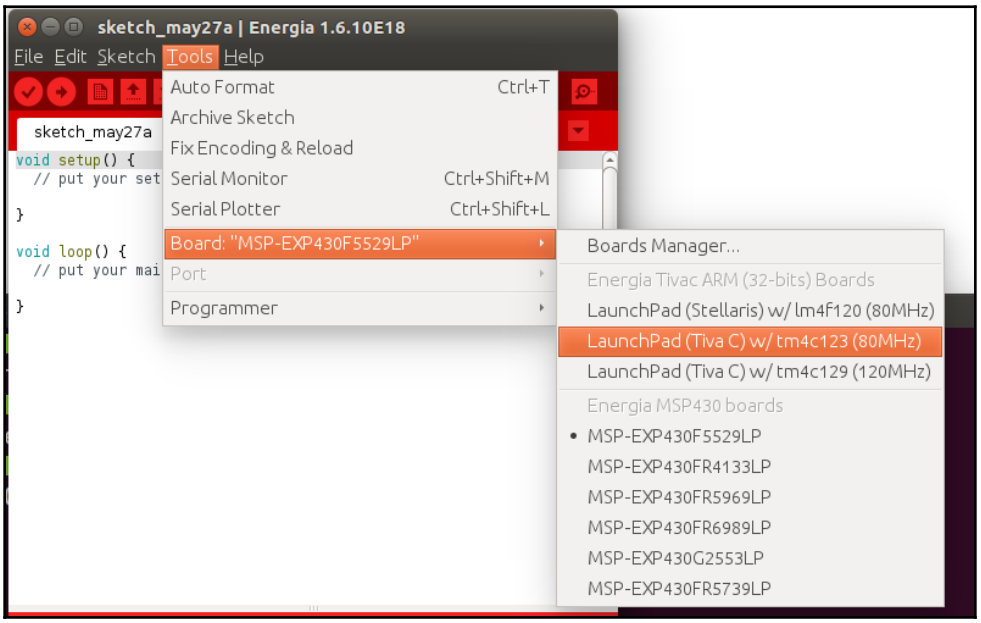
Energia IDE

Now, we have to select the board tm4c123 in the IDE for compiling the code specific for this board. To do so, we have to install the packages of this board. You can select the option **Tools** | **Boards** | **Boards Manager** to install the packages.



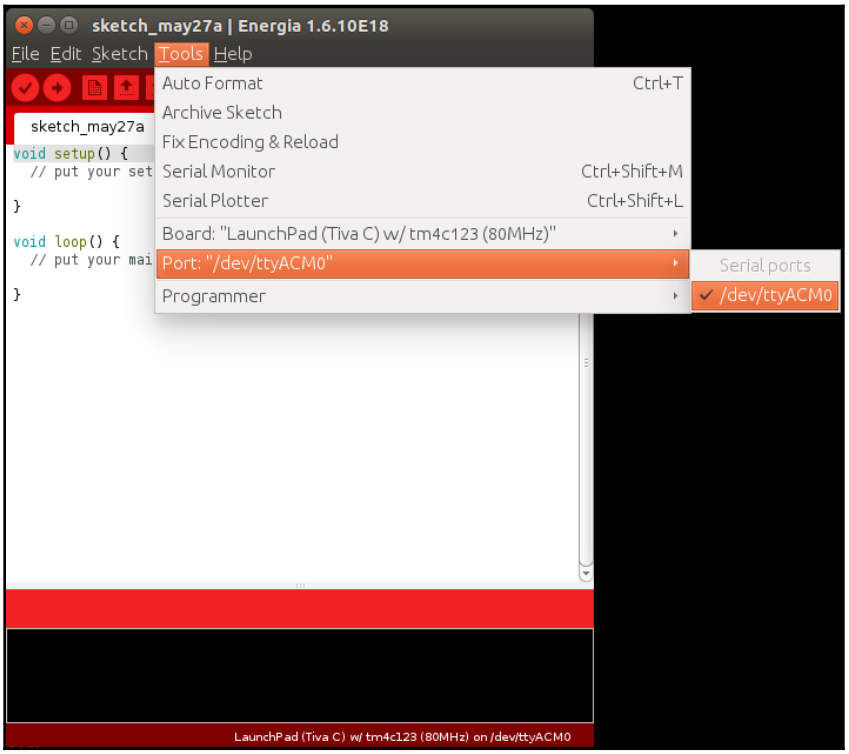
Board Manager of Energia

9. After installing the packages, you can select the board by navigating to **Tools** | **Boards** | **Launchpad (Tiva C) w/tm4c123 (80MHz),** as shown in the following screenshot:



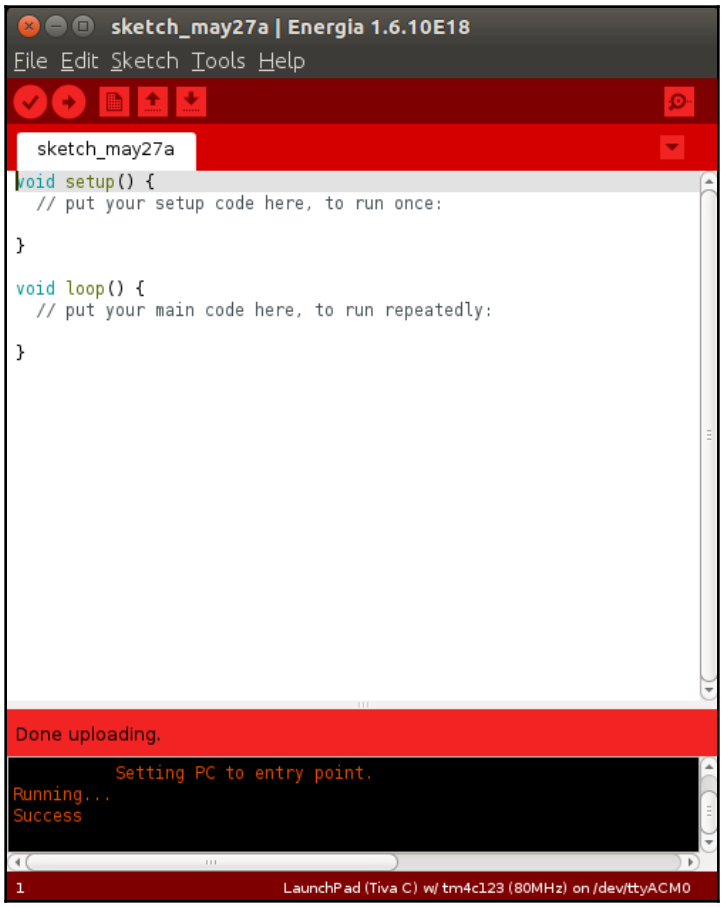
Energia board selection

10. Then, select the serial port by navigating to **Tools** | **Serial Port** | **/dev/ttyACM0**, as shown in the following screenshot:



Energia serial port selection

11. Compile and upload the code by using the **Upload** button. The Upload button will do both the processes. The following screenshot illustrates a successful upload:



Energia serial port selection

Visit the following links to install Energia on Linux, macOS X, and Windows:

- Refer to `http://energia.nu/guide/guide_linux/` for Linux
- Refer to `http://energia.nu/Guide_MacOSX.html` for macOS X
- Refer to `http://energia.nu/Guide_Windows.html` for Windows

Motor interfacing code

The following code in Energia can be used to test the two motors in the differential drive configuration. This code can move the robot forward for 5 seconds and backward for 5 seconds. Then, it moves the robot to the left for 5 seconds and right for 5 seconds. After each movement, the robot will stop for 1 second.

At the start of the code, we define pins for INA, INB, and PWM of the two motors, as follows:

```
///Left Motor  Pins
#define INA_1 12
#define INB_1 13
#define PWM_1 PC_6

///Right Motor Pins
#define INA_2 5
#define INB_2 6
#define PWM_2 PC_5
```

The pinout for Launchpad is given at: `http://energia.nu/pin-maps/guide_tm4c123launchpad/`

The following code shows the five functions to move the robot forward, backward, left, and right. The fifth function is to stop the robot. We will use the `digitalWrite()` function to write a digital value to a pin. The first argument of `digitalWrite()` is the pin number and the second argument is the value to be written to the pin. The value can be `HIGH` or `LOW`. We will use the `analogWrite()` function to write a PWM value to a pin. The first argument of this function is the pin number and the second is the PWM value. The range of this value is from 0-255. At high PWM, the motor driver will switch fast and have more speed. At low PWM, switching inside the motor driver will be slow, so the motor will also be slow. Currently, we are running at full speed:

```
void move_forward()
{
    //Setting CW rotation to and Left Motor  and CCW to Right Motor
    //Left Motor
```

```
    digitalWrite(INA_1,HIGH);
    digitalWrite(INB_1,LOW);
    analogWrite(PWM_1,255);
    //Right Motor
    digitalWrite(INA_2,LOW);
    digitalWrite(INB_2,HIGH);
    analogWrite(PWM_2,255);
}

//////////////////////////////////////////////////////

void move_left()
{
    //Left Motor
    digitalWrite(INA_1,HIGH);
    digitalWrite(INB_1,HIGH);
    analogWrite(PWM_1,0);
    //Right Motor
    digitalWrite(INA_2,LOW);
    digitalWrite(INB_2,HIGH);
    analogWrite(PWM_2,255);
}

//////////////////////////////////////////////////////

void move_right()
{
      //Left Motor
      digitalWrite(INA_1,HIGH);
      digitalWrite(INB_1,LOW);
      analogWrite(PWM_1,255);
      //Right Motor
      digitalWrite(INA_2,HIGH);
      digitalWrite(INB_2,HIGH);
      analogWrite(PWM_2,0);
}

//////////////////////////////////////////////////////

void stop()
{
    //Left Motor
    digitalWrite(INA_1,HIGH);
    digitalWrite(INB_1,HIGH);
    analogWrite(PWM_1,0);
    //Right Motor
    digitalWrite(INA_2,HIGH);
    digitalWrite(INB_2,HIGH);
```

```
    analogWrite(PWM_2,0);
}

////////////////////////////////////////////////

void move_backward()

{
    //Left Motor
    digitalWrite(INA_1,LOW);
    digitalWrite(INB_1,HIGH);
    analogWrite(PWM_1,255);
    //Right Motor
    digitalWrite(INA_2,HIGH);
    digitalWrite(INB_2,LOW);
    analogWrite(PWM_2,255);
}
```

We first set the `INA` and `INB` pins of the two motors to the `OUTPUT` mode, so that we can write `HIGH` or `LOW` values to these pins. The `pinMode()` function is used to set the mode of the I/O pin. The first argument of `pinMode()` is the pin number and the second argument is the mode. We can set a pin as input or output. To set a pin as output, give the `OUTPUT` argument as the second argument; to set it as input, give `INPUT` as the second argument, as shown in following code. There is no need to set the PWM pin as the output because `analogWrite()` writes the PWM signal without setting `pinMode()`:

```
void setup()
{
   //Setting Left Motor pin as OUTPUT
    pinMode(INA_1,OUTPUT);
    pinMode(INB_1,OUTPUT);
    pinMode(PWM_1,OUTPUT);

   //Setting Right Motor pin as OUTPUT
    pinMode(INA_2,OUTPUT);
    pinMode(INB_2,OUTPUT);
    pinMode(PWM_2,OUTPUT);
}
```

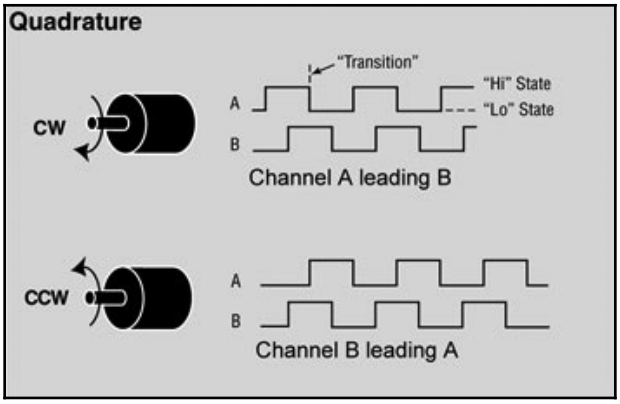
The following snippet is the main loop of the code. It will call each function, such as `move forward(),move_backward(),move_left()`, and `move_right()`, for `5` seconds. After calling each function, the robot stops for `1` second:

```
void loop()
{
  //Move forward for 5 sec
move_forward();
delay(5000);
  //Stop for 1 sec
stop();
delay(1000);

  //Move backward for 5 sec
move_backward();
delay(5000);
  //Stop for 1 sec
stop();
delay(1000);

  //Move left for 5 sec
move_left();
delay(5000);
  //Stop for 1 sec
stop();
delay(1000);

  //Move right for 5 sec
move_right();
delay(5000);
  //Stop for 1 sec
stop();
delay(1000);
}
```

Interfacing quadrature encoder with Tiva C Launchpad

The wheel encoder is a sensor attached to the motor to sense the number of rotations of the wheel. If we know the number of rotations, we can compute the velocity and displacement of the wheel.

For this robot, we have chosen a motor with an in-built encoder. This encoder is a quadrature type, which can sense both the direction and speed of the motor. Encoders use different types of sensors, such as optical and hall sensors, to detect these parameters. This encoder uses the hall effect to sense the rotation. The quadrature encoder has two channels, namely **Channel A** and **Channel B**. Each channel will generate digital signals with a 90-degree phase shift. The following diagram shows the wave form of a typical quadrature encoder:



Quadrature encoder waveforms

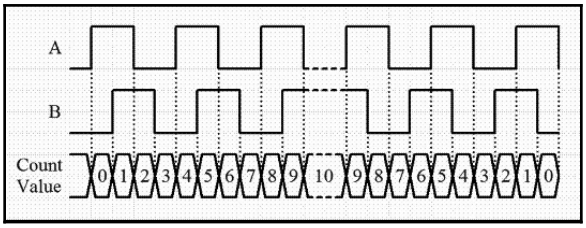
If the motor rotates in a clockwise direction, **Channel A** will lead **Channel B,** and if the motor rotates counterclockwise, **Channel B** will lead **Channel A**. This reading will be useful to sense the direction of rotation of the motor. The following section discusses how we can translate the encoder output to useful measurements, such as displacement and velocity.

Processing encoder data

Encoder data is a two-channel pulse out with 90 degrees out of phase. Using this data, we can find the direction of rotation and how many times the motor has rotated, and thereby find the displacement and velocity.

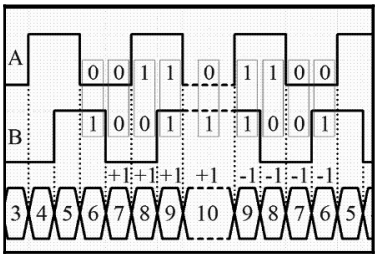
Some of the terms that specify encoder resolution are **pulses per revolution** (**PPR**) or **lines per revolution** (**LPR**) and **counts per revolution** (**CPR**). PPR specifies how many electrical pulses (0 to 1 transitions) there will be during one revolution of the motor final shaft. Some manufactures use the name CPR instead of PPR, because each pulse will contain two edges (rising and falling) and there are two pulse channels (A and B) with 90-degree phase shift; the total number of edges will be four times the number of PPR. Most quadrature receivers use the so-called 4X decoding to count all the edges from encoder A and B channels yielding 4X resolution compared to the raw PPR value.

In our motor, Pololu specifies that the CPR is 64 for the motor shaft, which corresponds to 8,400 CPR of the gearbox's output shaft. In effect, we get 8,400 counts from the gearbox output shaft when the motor's final shaft completes one revolution. The following diagram shows how we can compute the count from the encoder pulses:



Encoder pulses with count value

In this encoder specification, the count per revolution is given; it is calculated by the encoder channel edge transitions. One pulse of an encoder channel corresponds to four counts. So, to get 8,400 counts in our motor, the PPR will be *8,400 / 4 = 2,100*. From the preceding diagram, we will be able to calculate the number of counts in one revolution, but we also need to sense the direction of movement. This is because irrespective of whether the robot moves forward or backward, the counts that we get will be same; so, sensing the direction is important in order to decode the signal. The following diagram shows how we can decode the encoder pulses:



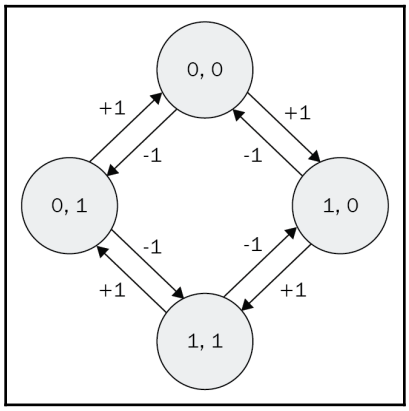
Detecting motor direction from encoder pulses

If we observe the code pattern, we can understand that it follows the 2-bit Gray code. A Gray code is the encoding of numbers, so that adjacent numbers have a single digit differing by 1. Gray code (`http://en.wikipedia.org/wiki/Gray_code`) is commonly used in rotary encoders for efficient coding.

We can predict the direction of rotation of a motor by state transitions. The state transition table is as follows:

State	Clockwise transition	Counterclockwise transition
0,0	0,1 to 0,0	1,0 to 0,0
1,0	0,0 to 1,0	1,1 to 1,0
1,1	1,0 to 1,1	0,1 to 1,1
0,1	1,1 to 0,1	0,0 to 0,1

It will be more convenient if we represent it in a state transition diagram:

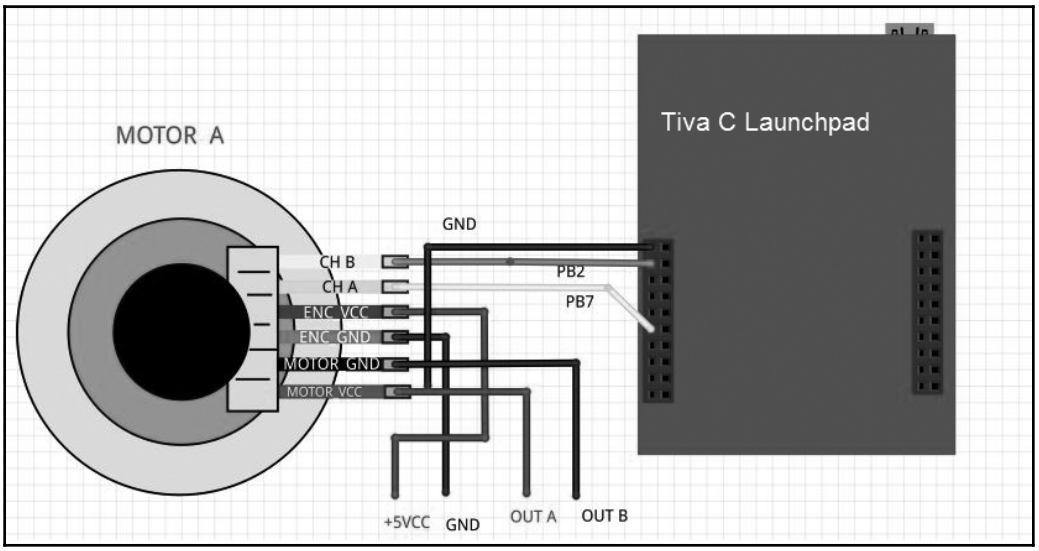


State transition diagram of encoders

After getting this Gray code, we can process the pulses using a microcontroller. The channel pins of the motor have to be connected to the interrupt pins of the microcontroller. So, when the channel has edge transitions, it will generate an interrupt or trigger in the pins, and if any interrupts arrives in that pin, an interrupt service routine, or simply a function, will be executed inside the microcontroller program. It can read the current state of the two pins. According to the current state of the pins and previous values, we can determine the direction of rotation and can decide whether we have to increment or decrement the count. This is the basic logic for encoder handling.

After getting the count, we can calculate the angle of rotation (in degrees) using *Angle = (Count Value / CPR) * 360*. Here, if we substitute CPR with 8,400, the equation becomes *Angle = 0.04285 * Count Value*; that is, for turning one degree, 24 counts have to be received or six encoded channel pulses have to come.

The following diagram shows the interfacing circuit of one motor encoder with Tiva C LaunchPad:



Interfacing encoder to Launchpad

From the above diagram, you can find motor pins CH A and CH B which are the output from the motor encoders. These pins are interfaced to PB2 and PB7 pins of the Tiva C Launchpad. The pins ENC VCC and ENC GND are the power pins of the encoder, so we have to provide +5V and GND to these pins. The next set of pins are for powering the motors. The MOTOR VCC and MOTOR GND are marked as OUTA and OUTB which is directly going to Motor driver in order to control the motor speed.

The maximum voltage level of output pulse is in between 0V to 5V from the encoder. In this case, we can directly interface the encoder with Launchpad because it can receive input up to 5V, or we can use a 3.3V to 5V level shifter like we used for motor driver interfacing earlier.

In the next section, we will upload code in Energia to test the quadrature encoder signal. We need to check whether we get a proper count from the encoder.

Quadrature encoder interfacing code

This code will print the count of the left and right motor encoder via a serial port. The two encoders are in a 2X decoding scheme, so we will get 4,200 CPR. In the first section of the code, we are defining pins for two channel outputs of two encoders and we are declaring the count variable for two encoders. The encoder variable uses a volatile keyword before the variable data type. The main use of `volatile` is that the variable with the `volatile` keyword will store in RAM memory, whereas normal variables are in CPU registers. Encoder values will change very quickly, so using an ordinary variable will not be accurate. In order to get accuracy, we will use `volatile` for encoder variables, as follows:

```
//Encoder pins definition

// Left encoder

#define Left_Encoder_PinA 31
#define Left_Encoder_PinB 32

volatile long Left_Encoder_Ticks = 0;

//Variable to read current state of left encoder pin
volatile bool LeftEncoderBSet;

//Right Encoder

#define Right_Encoder_PinA 33
#define Right_Encoder_PinB 34
volatile long Right_Encoder_Ticks = 0;
//Variable to read current state of right encoder pin
volatile bool RightEncoderBSet;
```

The following code snippet is the definition of the `setup()` function. In Energia, `setup()` is a built-in function used for initialization and for one-time execution of variables and functions. Inside `setup()`, we initialize the serial data communication with a baud rate of `115200` and call a user-defined `SetupEncoders()` function to initialize pins of the encoders. The serial data communication is mainly done to check the encoder count via the serial terminal:

```
void setup()
{
    //Init Serial port with 115200 buad rate
  Serial.begin(115200);
  SetupEncoders();
}
```

The definition of `SetupEncoders()` is given in the code that follows. To receive the encoder pulse, we need two pins in Launchpad as the input. Configure the encoder pins to Launchpad as the input and activate its pull-up resistor. The `attachInterrupt ()` function will configure one of the encoder pins as an interrupt. The `attachInterrupt ()` function has three arguments. The first argument is the pin number, the second argument is the **interrupt service routine** (**ISR**), and the third argument is the interrupt condition, that is, the condition in which the interrupt has to fire ISR. In this code, we are configuring `PinA` of the left and right encoder pins as the interrupt; it calls the ISR when there is a rise in the pulse:

```
void SetupEncoders()
{
  // Quadrature encoders
  // Left encoder
  pinMode(Left_Encoder_PinA, INPUT_PULLUP);
  // sets pin A as input
  pinMode(Left_Encoder_PinB, INPUT_PULLUP);
  // sets pin B as input
  attachInterrupt(Left_Encoder_PinA, do_Left_Encoder, RISING);

  // Right encoder
  pinMode(Right_Encoder_PinA, INPUT_PULLUP);
  // sets pin A as input
  pinMode(Right_Encoder_PinB, INPUT_PULLUP);
  // sets pin B as input

  attachInterrupt(Right_Encoder_PinA, do_Right_Encoder, RISING);
}
```

The following code is the built-in `loop()` function in Energia. The `loop()` function is an infinite loop where we put our main code. In this code, we call the `Update_Encoders()` function to print the encoder value continuously through the serial Terminal:

```
void loop()
{
  Update_Encoders();
}
```

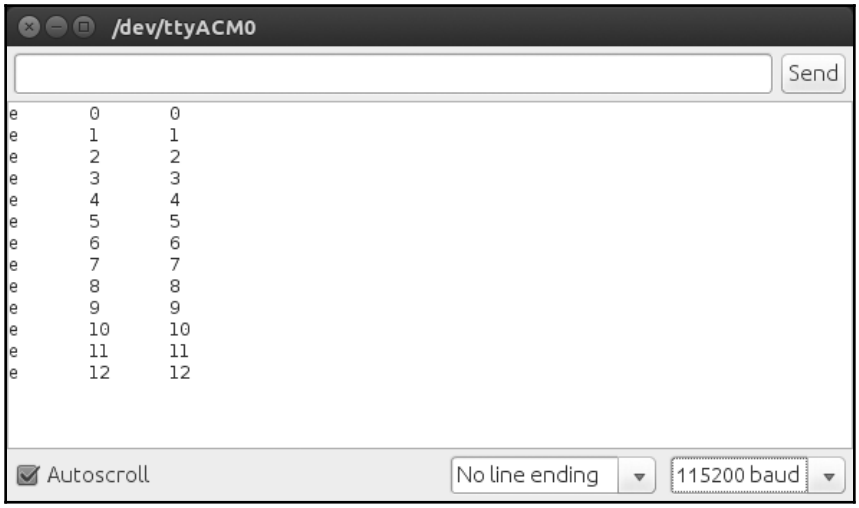
The following code is the function definition of the `Update_Encoders()` function. It prints two encoder values in a line with a starting character `e` and the values are separated by tab spaces. The `Serial.print()` function is a built-in function that will print the character/string given as the argument:

```
void Update_Encoders()
{
  Serial.print("e");
  Serial.print("t");
  Serial.print(Left_Encoder_Ticks);
  Serial.print("t");
  Serial.print(Right_Encoder_Ticks);
  Serial.print("n");
 }
```

The following code is the ISR definition of the left and right encoders. When a rising edge is detected on each of the pins, one of the ISRs will be called. The current interrupt pins are `PinA` of each of the encoders. After getting the interrupt, we can assume that the rising `PinA` has a higher value state, so there is no need to read that pin. Read `PinB` of both the encoders and store the pin state to `LeftEncoderBSet` or `RightEncoderBSet`. The current state is compared to the previous state of `PinB` and can detect the direction and decide whether the count has to be incremented or decremented according to the state transition table:

```
void do_Left_Encoder()
{
  LeftEncoderBSet = digitalRead(Left_Encoder_PinB);
  // read the input pin
  Left_Encoder_Ticks -= LeftEncoderBSet ? -1 : +1;
}

void do_Right_Encoder()
{
  RightEncoderBSet = digitalRead(Right_Encoder_PinB);
  // read the input pin
  Right_Encoder_Ticks += RightEncoderBSet ? -1 : +1;
}
```

Upload the sketch and view the output using the serial monitor in Energia. Navigate to **Tools** | **Serial monitor**. Move the two motors manually and you will see the count changing. Set the baud rate in the serial monitor, which is the same as that initialized in the code; in this case, it is **115200**. The output will look like this:



Interfacing encoder to Launchpad

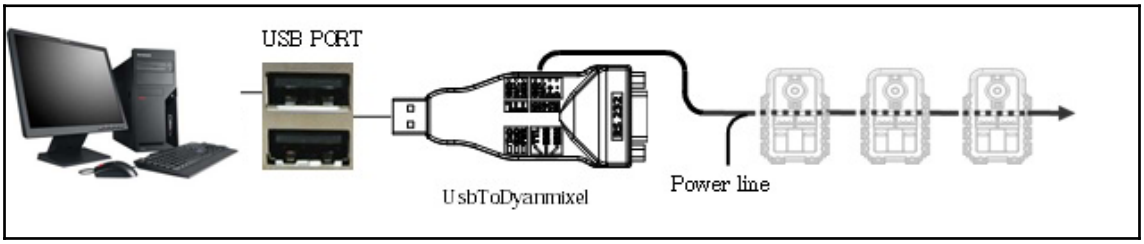
If we want to upgrade the robot to high accuracy and payload, we have to consider high quality actuators, such as Dynamixel. Dynamixel servos are intelligent actuators, which have in-built PID control and monitoring of the servo and encoder parameters, such as torque, position, and so on. In this robot, we are not using Dynamixel.

Working with Dynamixel actuators

Dynamixel is a kind of networked actuator for robots developed by the Korean manufacture, ROBOTIS. It is widely used by companies, universities, and hobbyists due to its versatile expansion capability, power feedback function, position, speed, internal temperature, input voltage, and so on.

The Dynamixel servos can be connected in a daisy chain; it is a method of connecting devices in a serial fashion, connecting one device to another through the connected devices, and can control all the connected servos from one controller. Dynamixel servos communicate via RS485 or TTL. The list of available Dynamixel servos is given at `http://www.robotis.com/xe/dynamixel_en`.

The interfacing of Dynamixel is very easy. Dynamixel comes with a controller called USB2Dyanmixel, which will convert a USB to Dynamixel compatible TTL/RS485 levels. The following diagram shows the interfacing diagram of Dynamixel:



Interfacing Dynamixel actuators to a PC

ROBOTIS provides Dynamixel SDK for accessing motor registers; we can read and write values to Dynamixel registers and retrieve data, such as position, temperature, voltage, and so on.

The instructions to set USB2Dynamixel and Dynamixel SDK are given at `support.robotis.com/en/`.

Dynamixel can be programed using Python libraries. One of the Python libraries for handling Dynamixel servos is **pydynamixel**. This package is available for Windows and Linux. Pydynamixel will support RX, MX, and EX series servos.

We can download the the pydynamixel Python package from `https://pypi.python.org/pypi/dynamixel/`.

Download the package and extract it to the `home` folder. Open a terminal/DOS prompt and execute the following command:

```
sudo python setup.py install
```

After installing the package, we can try the following Python example, which will detect the servo attached to the USB2Dynamixel and write some random position to the servo. This example will work with RX and MX servos:

```
#!/usr/bin/env python
```

The following code will import the necessary Python modules required for this example. This includes Dynamixel Python modules too:

```
import os
import dynamixel
import time
import random
```

The following code defines the main parameters needed for Dynamixel communication parameters. The `nServos` variable denotes the number of Dynamixel servos connected to the bus. The `portName` variable indicates the serial port of USB2Dynamixel to which Dynamixel servos are connected. The `baudRate` variable is the communication speed of USB2Dynamixel and Dynamixel:

```
# The number of Dynamixels on our bus.
nServos = 1

# Set your serial port accordingly.
if os.name == "posix":
    portName = "/dev/ttyUSB0"
else:
    portName = "COM6"
# Default baud rate of the USB2Dynamixel device.
baudRate = 1000000
```

The following code is the Dynamixel Python function to connect to Dynamixel servos. If it is connected, the program will print it and scan the communication bus to find the number of servos starting from ID `1` to `255`. The servo ID is the identification of each servo. We are given `nServos` as `1`, so it will stop scanning after getting one servo on the bus:

```
# Connect to the serial port
print "Connecting to serial port", portName, '...',
serial = dynamixel.serial_stream.SerialStream( port=portName,
baudrate=baudRate, timeout=1)
print "Connected!"
net = dynamixel.dynamixel_network.DynamixelNetwork( serial )
net.scan( 1, nServos )
```

The following code will append the Dynamixel ID and the servo object to the `myActutors` list. We can write servo values to each servo using the servo ID and servo object. We can use the `myActutors` list for further processing:

```
# A list to hold the dynamixels
myActuators = list()
print myActuators

This will create a list for storing  dynamixel actuators details.

print "Scanning for Dynamixels...",
```

```
for dyn in net.get_dynamixels():
    print dyn.id,
    myActuators.append(net[dyn.id])
print "...Done"
```

The following code will write random positions from 450 to 600 to each Dynamixel actuator that is available on the bus. The range of positions in Dynamixel is 0 to 1,023. This will set the servo parameters, such as `speed`, `torque`, `torque_limt`, `max_torque`, and so on:

```
# Set the default speed and torque
for actuator in myActuators:
    actuator.moving_speed = 50
    actuator.synchronized = True
    actuator.torque_enable = True
    actuator.torque_limit = 800
    actuator.max_torque = 800
```

The following code will print the current position of the current actuator:

```
# Move the servos randomly and print out their current positions
while True:
    for actuator in myActuators:
        actuator.goal_position = random.randrange(450, 600)
    net.synchronize()
```

The following code will read all data from the actuators:

```
    for actuator in myActuators:
        actuator.read_all()
        time.sleep(0.01)

    for actuator in myActuators:
        print actuator.cache[dynamixel.defs.REGISTER['Id']],
actuator.cache[dynamixel.defs.REGISTER['CurrentPosition']]

    time.sleep(2)
```

Working with ultrasonic distance sensors

One of the important capabilities of a mobile robot is navigation. An ideal navigation means a robot can plan its path from its current position to the destination and can move without any obstacles. We use ultrasonic distance sensors in this robot for detecting near objects that can't be detected using the Kinect sensor. A combination of Kinect and ultrasonic sound sensors provides ideal collision avoidance on this robot.

Ultrasonic distance sensors work in the following manner. The transmitter will send an ultrasonic sound that is not audible to human ears. After sending an ultrasonic wave, it will wait for an echo of the transmitted wave. If there is no echo, it means there are no obstacles in front of the robot. If the receiving sensor receives any echo, a pulse will be generated on the receiver, and it can calculate the total time the wave will take to travel to the object and return to the receiver sensors. If we get this time, we can compute the distance to the obstacle using the following formula:

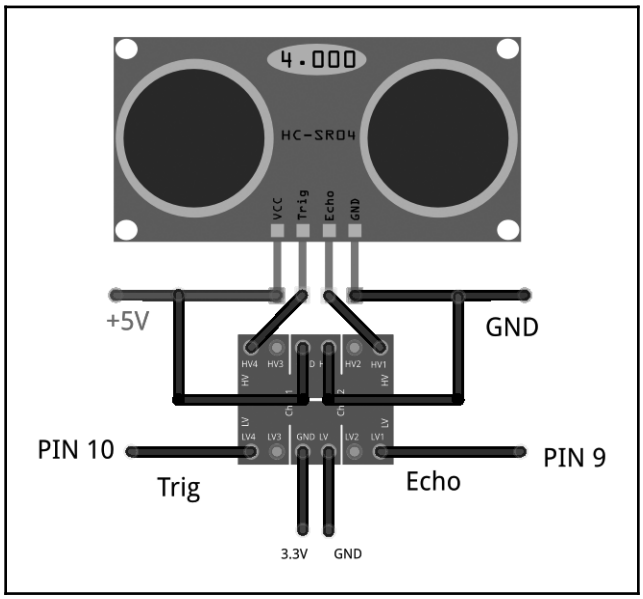
*Speed of Sound * Time Passed /2 = Distance from Object*.

Here, the speed of sound can be taken as 340 m/s.

Most of the ultrasonic range sensors have a distance range from 2 cm to 400 cm. In this robot, we use a sensor module called HC-SR04. We look at how to interface HC-SR04 with Tiva C LaunchPad to get the distance from the obstacles.

Interfacing HC-SR04 to Tiva C LaunchPad

The following diagram illustrates the interfacing circuit of the HC-SR04 ultrasonic sound sensor with Tiva C LaunchPad:



Interfacing ultrasonic sound sensors to Launchpad

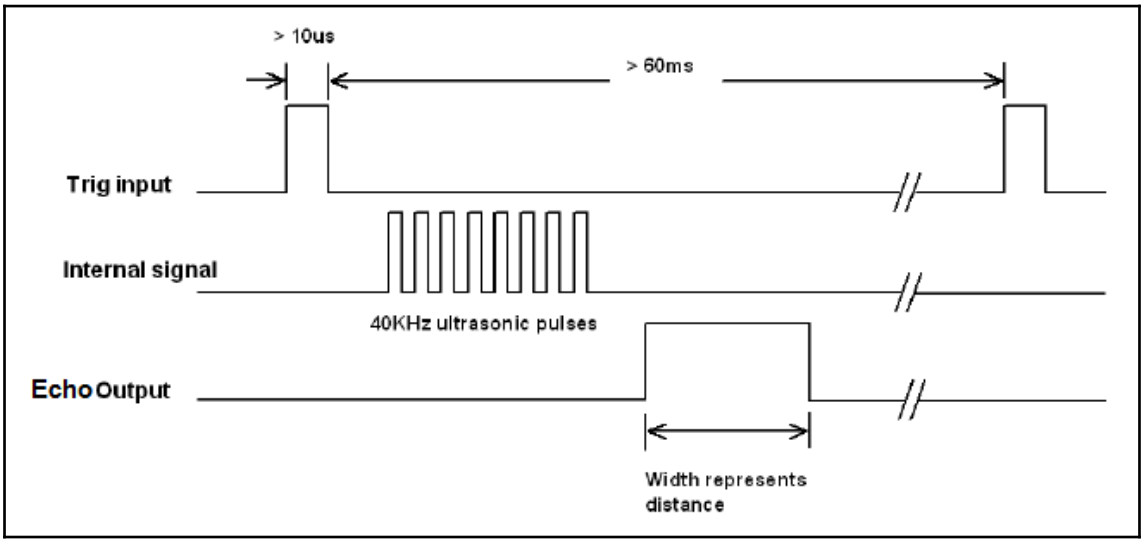
The working voltage of the ultrasonic sensor is 5V and the input/output of this sensor is also 5V, so we need a level shifter on the **Trig** and **Echo** pins for interfacing into the **3.3V** level Launchpad. In the level shifter, we need to apply high voltage, that is, 5V, and low voltage, that is, 3.3V, as shown in the preceding diagram, to switch from one level to another level. **Trig** and **Echo** pins are connected on the high voltage side of the level shifter and the low voltage side pins are connected to Launchpad. The **Trig** pin and **Echo** are connected to the 10th and 9th pins of Launchpad. After interfacing the sensor, we can see how to program the two I/O pins.

Working of HC-SR04

The timing of the waveform on each pin is shown in the following diagram. We need to apply a short 10 μs pulse to the trigger input to start the ranging and then the module will send out an eight-cycle burst of ultrasound at 40 KHz and raise its echo. The echo is a distance object that is the pulse width and the range in proportion. You can calculate the range through the time interval between sending trigger signals and receiving echo signals using the following formula:

*Range = high level time of echo pin output * velocity (340 M/S) / 2.*

It will be better to use a delay of 60 ms before each trigger to avoid overlapping between trigger and echo:



Input and output waveform of ultrasound sensor

Interfacing Code of Tiva C Launchpad

The following Energia code for Launchpad reads values from the ultrasound sensor and monitors the values through a serial port.

The following code defines the pins in Launchpad to handle ultrasonic echo and trigger pins and also defines variables for the duration of the pulse and the distance in centimeters:

```
const int echo = 9, Trig = 10;
long duration, cm;
```

The following code snippet is the `setup()` function. The `setup()` function is called when the program starts. Use this to initialize variables, pin modes, to start using libraries, and so on. The setup function will only run once, after each power up or reset of the Launchpad board. Inside `setup()`, we initialize serial communication with a baud rate of 115200 and set up the mode of ultrasonic handling pins by calling a `SetupUltrasonic();function:`

```
void setup()
{
  //Init Serial port with 115200 baud rate
  Serial.begin(115200);
  SetupUltrasonic();
}
```

The following is the setup function for the ultrasonic sensor; it will configure the `Trigger` pin as `OUTPUT` and the `Echo` pin as `INPUT`. The `pinMode()` function is used to set the pin as `INPUT` or `OUTPUT`:

```
void SetupUltrasonic()
{
 pinMode(Trig, OUTPUT);
 pinMode(echo, INPUT);
}
```

After creating a `setup()` function, which initializes and sets the initial values, the `loop()` function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Launchpad board.

The main loop of this is in the following code. This function is an infinite loop and calls the `Update_Ultra_Sonic()` function to update and print the ultrasonic readings through a serial port:

```
void loop()
{
    Update_Ultra_Sonic();
    delay(200);
}
```

The following code is the definition of the `Update_Ultra_Sonic()` function. This function will do the following operations. First it will take the trigger pin to the `LOW` state for `2` microseconds and `HIGH` for `10` microseconds. After `10` microseconds, it will again return the pin to the `LOW` state. This is according to the timing diagram. We already saw that 10 µs is the trigger pulse width.

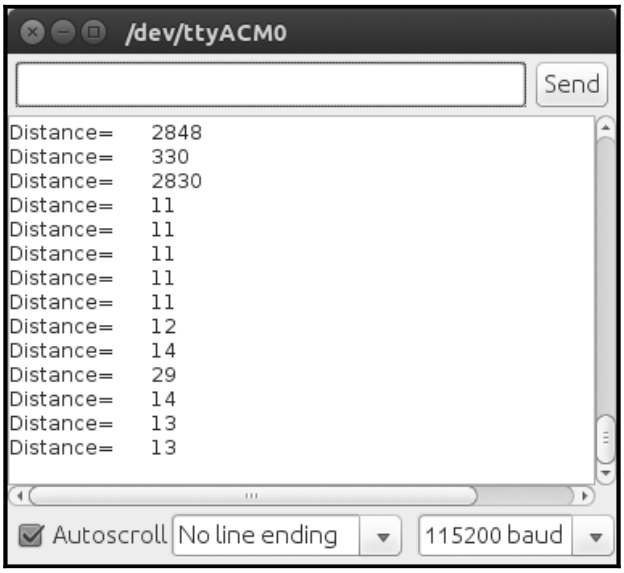
After triggering with 10 µs, we have to read the time duration from the Echo pin. The time duration is the time taken for the sound to travel from the sensor to the object and from the object to the sensor receiver. We can read the pulse duration by using the `pulseIn()` function. After getting the time duration, we can convert the time into centimeters by using the `microsecondsToCentimeters()` function, as shown in the following code:

```
void Update_Ultra_Sonic()
{
  digitalWrite(Trig, LOW);
  delayMicroseconds(2);
  digitalWrite(Trig, HIGH);
  delayMicroseconds(10);
  digitalWrite(Trig, LOW);

  duration = pulseIn(echo, HIGH);
  // convert the time into a distance
  cm = microsecondsToCentimeters(duration);
  //Sending through serial port
  Serial.print("distance=");
  Serial.print("t");
  Serial.print(cm);
  Serial.print("n");
}
```

The following code is the conversion function from microseconds to distance in centimeters. The speed of sound is 340 m/s, that is, 29 microseconds per centimeter. So, we get the total distance by diving the total microseconds by 29/2:

```
long microsecondsToCentimeters(long microseconds)
{
return microseconds / 29 / 2;
}
```

After uploading the code, open the serial monitor from the Energia menu under **Tools** | **Serial Monitor** and change the baud rate to **115200**. The values from the ultrasonic sensor are shown in the following screenshot:



Output of the ultrasonic distance sensor in Energia serial monitor

Interfacing Tiva C LaunchPad with Python

In this section, we will look at how to connect Tiva C LaunchPad with Python to receive data from Launchpad in a PC.

The **PySerial** module can be used for interfacing Launchpad to Python. The detailed documentation of PySerial and its installation procedure for Windows, Linux, and OS X can be found here : `http://pyserial.sourceforge.net/pyserial.html`

PySerial is available in the Ubuntu package manager and it can be easily installed in Ubuntu using the following command in the Terminal:

```
$ sudo apt-get install python-serial
```

After installing the `python-serial` package, we can write Python code to interface Launchpad. The interfacing code is given in the following section.

The following code imports the Python `serial` module and the `sys` module. The `serial` module handles the serial ports of Launchpad and performs operations such as reading, writing, and so on. The `sys` module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available:

```
#!/usr/bin/env python
import serial
import sys
```

When we plug Launchpad to the computer, the device registers on the OS as a virtual serial port. In Ubuntu, the device name looks like `/dev/ttyACMx`. Here, **x** can be a number; if there is only one device, it will probably be 0. To interact with Launchpad, we need to handle this device file only.

The following code will try to open the serial port `/dev/ttyACM0` of Launchpad with a baud rate of `115200`. If it fails, it will print `Unable to open serial port`:

```
try:
    ser = serial.Serial('/dev/ttyACM0',115200)
except:
    print "Unable to open serial port"
```

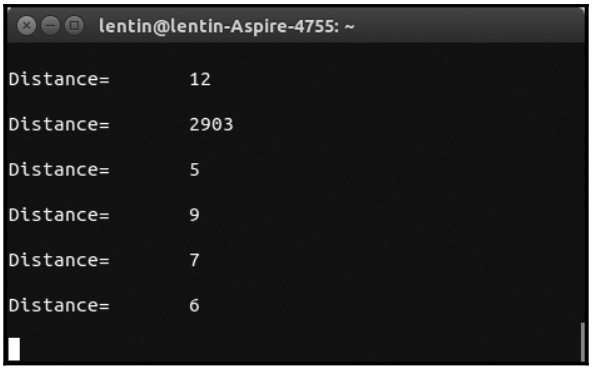
The following code will read the serial data until the serial character becomes a new line (`'n'`) and prints it on the Terminal. If we press Ctrl + C on the keyboard, to quit the program, it will exit by calling `sys.exit(0)`:

```
while True:
    try:
        line = ser.readline()
        print line
    except:
        print "Unable to read from device"
        sys.exit(0)
```

After saving the file, change the permission of the file to executable using the following command:

```
$ sudo chmod +X script_name
$ ./ script_name
```

The output of the script will look like this:



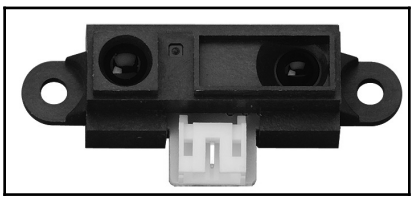
Output of the ultrasonic distance sensor in Energia serial monitor

Working with the IR proximity sensor

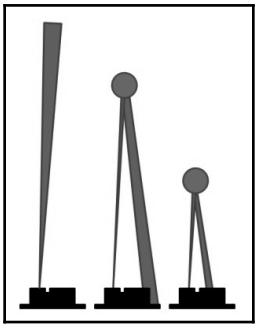
Infrared sensors are another method to find obstacles and the distance from the robot. The principle of infrared distance sensors is based on the infrared light that is reflected from a surface when hitting an obstacle. An IR receiver will capture the reflected light and the voltage is measured based on the amount of light received.

One of the popular IR range sensors is Sharp GP2D12. The product link can be found here: `http://www.robotshop.com/en/sharp-gp2y0a21yk0f-ir-range-sensor.html`

The following image shows the Sharp GP2D12 sensor:

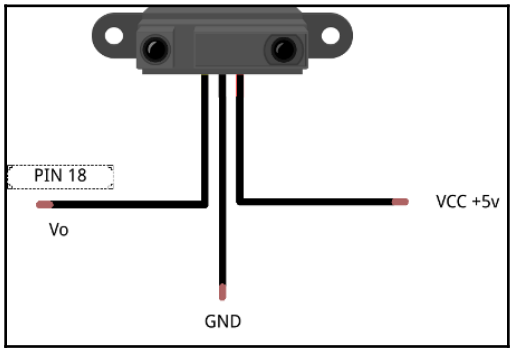


The sensor sends out a beam of IR light and uses triangulation to measure the distance. The detection range of the GP2D12 is between 10 cm and 80 cm. The beam is 6 cm wide at a distance of 80 cm. The transmission and reflection of the IR light sensor is illustrated in the following diagram:



Obstacle sensing using IR light sensor

On the left of the sensor is an IR transmitter, which continuously sends IR radiation. After hitting into some objects, the IR light will reflect and it will be received by the IR receiver. The interfacing circuit of the IR sensor is shown here:



Pinout of Sharp IR sensor

The analog out pin **Vo** can be connected to the ADC pin of Launchpad. The interfacing code of the Sharp distance sensor with the Tiva C Launchpad is discussed further in this section. In this code, we select the 18th pin of Launchpad and set it to the ADC mode and read the voltage levels from the Sharp distance sensor. The range equation of the GP2D12 IR sensor is given as follows:

Range = (6,787 / (Volt - 3)) - 4

Here, *Volt* is the analog voltage value from the ADC of the Volt pin.

In this first section of the code, we set the 18th pin of Tiva C LaunchPad as the input pin and start a serial communication at a baud rate of 115200:

```
int IR_SENSOR = 18; // Sensor is connected to the analog A3
int intSensorResult = 0; //Sensor result
float fltSensorCalc = 0; //Calculated value

void setup()
{
Serial.begin(115200); // Setup communication with computer
   to present results serial monitor
}
```

In the following section of code, the controller continuously reads the analog pin and converts it to the distance value in centimeters:

```
void loop()
{

// read the value from the ir sensor
intSensorResult = analogRead(IR_SENSOR); //Get sensor value

//Calculate distance in cm according to the range equation
fltSensorCalc = (6787.0 / (intSensorResult - 3.0)) - 4.0;

Serial.print(fltSensorCalc); //Send distance to computer
Serial.println(" cm"); //Add cm to result
delay(200); //Wait
}
```

This is the basic code to interface a sharp distance sensor. There are some drawbacks with IR sensors. Some of them are as follows:

- We can't use them in direct or indirect sunlight, so it's difficult to use them in an outdoor robot
- They may not work if the object is not reflective
- The range equation only works within the range

In the next section, we will discuss the IMU and its interfacing with Tiva C LaunchPad. An IMU can give the odometry data and it can be used as the input to navigation algorithms.

Working with Inertial Measurement Units

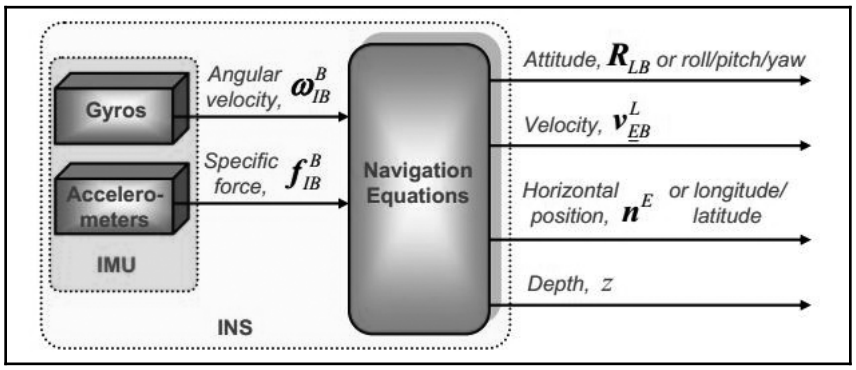
An **inertial measurement unit** (**IMU**) is an electronic device that measures velocity, orientation, and gravitational forces using a combination of accelerometers, gyroscopes, and magnetometers. IMUs have a lot of applications in robotics; some of the applications are applied in balancing of **unmanned aerial vehicles** (**UAVs**) and robot navigation.

In this section, we will discuss the role of IMUs in mobile robot navigation and some of the latest IMUs on the market and their interfacing with Launchpad.

Inertial navigation

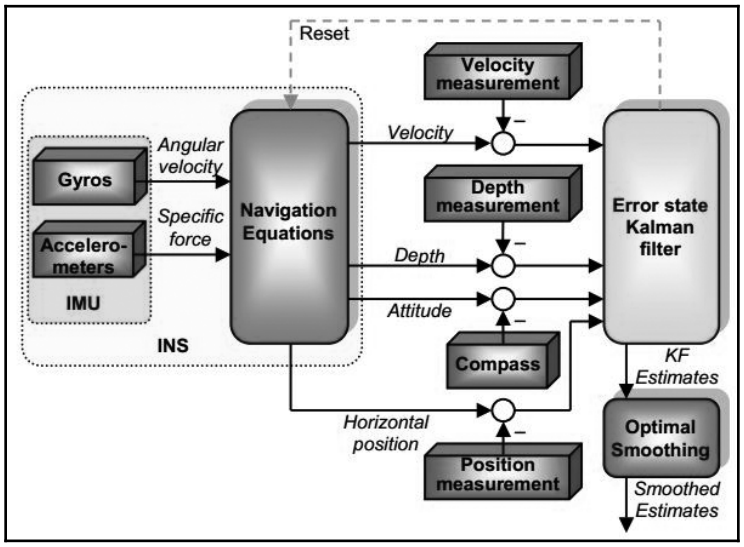
An IMU provides acceleration and orientation relative to inertial space. If you know the initial position, velocity, and orientation, you can calculate the velocity by integrating the sensed acceleration and the second integration gives the position. To get the correct direction of the robot, the orientation of the robot is required; this can be obtained by integrating sensed angular velocity from a gyroscope.

The following diagram illustrates an inertial navigation system, which will convert IMU values to odometry data:



Block diagram of IMU

The values we get from the IMU are converted into navigational information using navigation equations and feeding them into estimation filters, such as the Kalman filter. The **Kalman** filter is an algorithm that estimates the state of a system from the measured data (`http://en.wikipedia.org/wiki/Kalman_filter`). The data from an **inertial navigation system** (**INS**) will have some drift because of the error from the accelerometer and gyroscope. To limit the drift, an INS is usually aided by other sensors that provide direct measurements of the integrated quantities. Based on the measurements and sensor error models, the Kalman filter estimates errors in the navigation equations and all the colored sensors' errors. The following diagram illustrates an aided inertial navigation system using the Kalman filter:



IMU with inertial navigation system

Along with the motor encoders, the value from the IMU can be taken as the odometer value and it can be used for **dead reckoning**, the process of finding the current position of a moving object by using a previously determined position.

In the next section, we are going to look at one of the most popular IMUs from InvenSense, called **MPU 6050**.

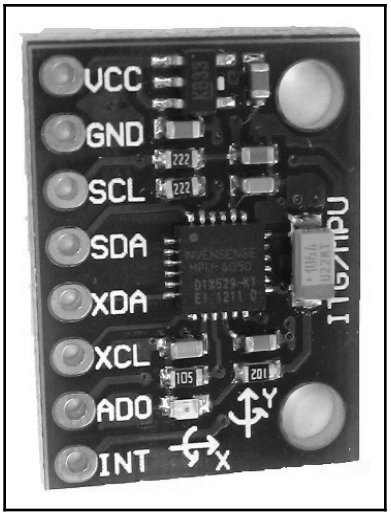
Interfacing MPU 6050 with Tiva C LaunchPad

The MPU-6000/MPU-6050 family of parts is the world's first and only six-axis motion tracking devices designed for the low power, low cost, and high-performance requirements of smartphones, tablets, wearable sensors, and robotics.

The MPU-6000/6050 devices combine a three-axis gyroscope and three-axis accelerometer on the silicon die together with an onboard digital motion processor capable of processing complex nine-axis motion fusion algorithms. The following diagram shows the system diagram of MPU 6050 and break out of MPU 6050:

Block diagram of MPU 6050

The breakout board of MPU 6050 is shown in the following diagram and it can be purchased from here `http://a.co/7C3yL96`:

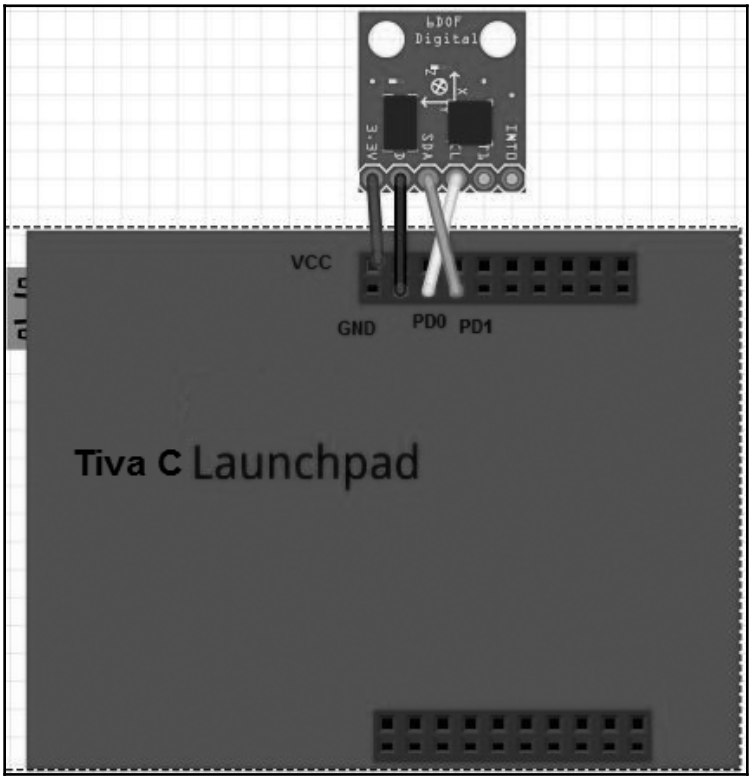


MPU 6050 breakout board

The connection from Launchpad to MPU 6050 is given in the following table. The remaining pins can be left disconnected:

Launchpad pins	MPU6050 pins
+3.3V	VCC/VDD
GND	GND
PD0	SCL
PD1	SDA

The following diagram shows the interfacing of MPU 6050 and Tiva C Launchpad:



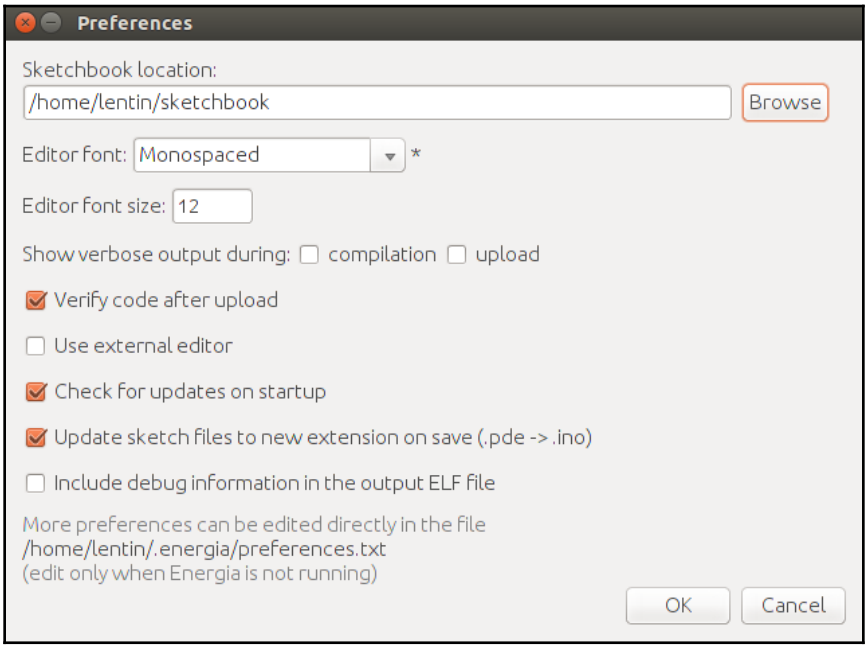
Interfacing MPU 6050 breakout board to Launchpad

MPU 6050 and Launchpad communicate using the I2C protocol. The supply voltage is 3.3V and it is taken from Launchpad.

Setting the MPU 6050 library in Energia

The interfacing code of Energia is discussed in this section. The interfacing code uses the `https://github.com/jrowberg/i2cdevlib/zipball/master` library for interfacing MPU 6050.

Download the ZIP file from the preceding link and navigate to **Preference from File** | **Preference in Energia**, as shown in the following screenshot:



Interfacing MPU 6050 breakout board to Launchpad

Go to **Sketchbook location** under **Preferences**, as seen in the preceding screenshot, and create a folder called `libraries`. Extract the files inside the **Arduino** folder inside the ZIP file to the `sketchbook/libraries` location. The Arduino packages in this repository are also compatible with Launchpad. The extracted files contain the `I2Cdev`, `Wire`, and `MPU6050` packages that are required for the interfacing of the MPU 6050 sensor. There are other sensor packages that are present in the `libraries` folder but we are not using them now.

The preceding procedure is done in Ubuntu, but it is the same for Windows and macOS X.

Interfacing code of Energia

This code is used to read the raw value from MPU 6050 to Launchpad. It uses an MPU 6050 third-party library compatible with Energia IDE. The following are the explanations of each block of the code.

In this first section of code, we include the necessary headers for interfacing MPU 6050 to Launchpad, such as `I2C`, `Wire` and the `MPU6050` library, and create an object of `MPU6050` with the name `accelgyro`. The `MPU6050.h` library contains a class named `MPU6050` to send and receive data to and from the sensor:

```
#include "Wire.h"

#include "I2Cdev.h"
#include "MPU6050.h"

MPU6050 accelgyro;
```

In the following section, we start the I2C and serial communication to communicate with MPU 6050 and print sensor values through the serial port. The serial communication baud rate is `115200` and `Setup_MPU6050()` is the custom function to initialize the MPU 6050 communication:

```
void setup()
{
  //Init Serial port with 115200 buad rate
  Serial.begin(115200);
  Setup_MPU6050();
}
```

The following section is the definition of the `Setup_MPU6050()` function. The `Wire` library allows you to communicate with the I2C devices. MPU 6050 can communicate using I2C. The `Wire.begin()` function will start the I2C communication between MPU 6050 and Launchpad; also, it will initialize the MPU 6050 device using the `initialize()` method defined in the `MPU6050` class. If everything is successful, it will print **connection successful;** otherwise, it will print **connection failed**:

```
void Setup_MPU6050()
{
   Wire.begin();
      // initialize device
    Serial.println("Initializing I2C devices...");
    accelgyro.initialize();

    // verify connection
```

```
    Serial.println("Testing device connections...");
    Serial.println(accelgyro.testConnection() ? "MPU6050 connection
successful" : "MPU6050 connection failed");
}
```

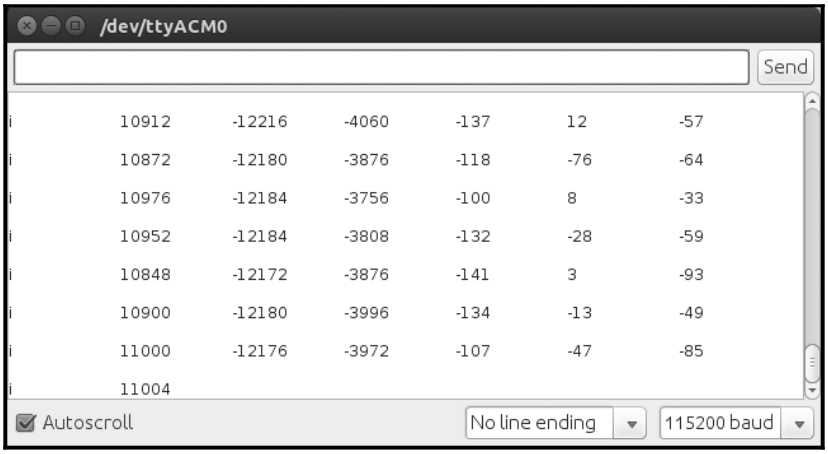
The following code is the `loop()` function, which continuously reads the sensor value and prints its values through the serial port: The `Update_MPU6050()` custom function is responsible for printing the updated value from MPU 6050:

```
void loop()
{

   //Update MPU 6050
    Update_MPU6050();
}
```

The definition of `Update_MPU6050()` is given as follows. It declares six variables to handle the accelerometer and gyroscope value in three-axis. The `getMotion6()` function in the MPU 6050 class is responsible for reading the new values from the sensor. After reading them, it will print via the serial port:

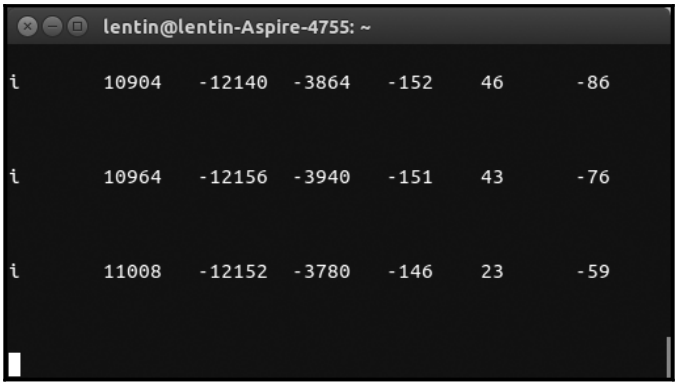
```
void Update_MPU6050()
{
   int16_t ax, ay, az;
  int16_t gx, gy, gz;

      // read raw accel/gyro measurements from device
    accelgyro.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

    // display tab-separated accel/gyro x/y/z values
    Serial.print("i");Serial.print("t");
    Serial.print(ax); Serial.print("t");
    Serial.print(ay); Serial.print("t");
    Serial.print(az); Serial.print("t");
    Serial.print(gx); Serial.print("t");
    Serial.print(gy); Serial.print("t");
    Serial.println(gz);
    Serial.print("n");
}
```

The output from the serial monitor is shown here:



Output from MPU 6050 in the serial monitor

We can read these values using the Python code that we used for ultrasonic. The following is the screenshot of the Terminal when we run the Python script:



Output from MPU 6050 in the Linux Terminal

Summary

In this chapter, we have discussed the interfacing of the motors that we are using in our robot. We have looked at motor and encoder interfacing with a controller board called Tiva C Launchpad. We have discussed the controller code for interfacing motors and encoders. In the future, if the robot requires high accuracy and torque, we have looked at Dynamixel servos that can substitute current DC motors. We have also looked at the robotic sensors that can be used in our robot. The sensors we discussed are ultrasonic distance sensors, IR proximity sensors, and IMUs. These three sensors help in the navigation of the robot. We also discussed the basic code to interface these sensors to Tiva C LaunchPad. We will discuss the vision sensors used in this robot further in the next chapter.

Questions

1. What is the H-Bridge circuit?
2. What is a quadrature encoder?
3. What is the 4X encoding scheme?
4. How do we calculate displacement from encoder data?
5. What are the features of the Dynamixel actuator?
6. What are ultrasonic sensors and how do they work?
7. How do you calculate distance from the ultrasonic sensor?
8. What is the IR proximity sensor and how does it work?

Further reading

Read more about Energia programming at the following link:

`http://energia.nu/guide/`

7
Interfacing Vision Sensors with ROS

In the previous chapter, we looked at actuators and how to interface the robot's sensors using the Tiva-C LaunchPad board. In this chapter, we will mainly look at vision sensors and the interface that they use with our robot.

The robot we are designing will have a 3D vision sensor, and we will be able to interface it with vision libraries such as **Open Source Computer Vision** (**OpenCV**), **Open Natural Interaction (OpenNI)**, and **Point Cloud Library** (**PCL**). The main application of the 3D vision sensor in our robot is autonomous navigation.

We will also look at how to interface the vision sensors with ROS and process the images that it senses using vision libraries such as OpenCV. In the last section of this chapter, we will look at the mapping and localization algorithm that we will use in our robot, called **SLAM** (**simultaneous localization and mapping**), and its implementation using a 3D vision sensor, ROS, and image-processing libraries.

In this chapter, we will cover the following topics:

- List of robotic vision sensors and image libraries
- Introduction to OpenCV, OpenNI, and PCL
- The ROS-OpenCV interface
- Point cloud processing using the PCL-ROS interface
- Conversion of point cloud data to laser scan data
- Introduction to SLAM

Technical requirements

You will need an Ubuntu 16.04 system with ROS Kinetic installed, as well as a web camera and a depth camera in order to try out the example in this chapter.

In the first section, we will look at the 2D and 3D vision sensors that are available in the market that can be used in different robots.

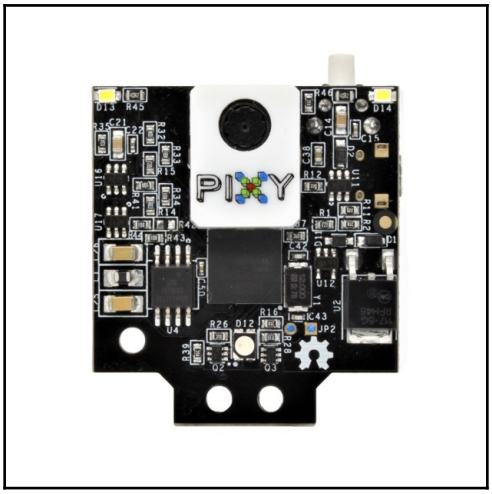
List of robotic vision sensors and image libraries

A 2D vision sensor or an ordinary camera delivers 2D image frames of the surroundings, whereas a 3D vision sensor delivers 2D image frames and an additional parameter called the depth of each image point. We can find the *x*, *y*, and *z* distance of each point from the 3D sensor with respect to the sensor's axis.

There are quite a few vision sensors available on the market. Some of the 2D and 3D vision sensors that can be used in our robot are mentioned in this chapter.

Pixy2/CMUcam5

The following picture shows the latest 2D vision sensor, called Pixy2/CMUcam5 (`https://pixycam.com/pixy-cmucam5/`), which is able to detect color objects with high speed and accuracy, and can be interfaced with an Arduino board. Pixy can be used for fast object detection, and the user can teach it which object it needs to track. The Pixy module has a CMOS sensor and NXP LPC4330 (`http://www.nxp.com/`) based on Arm Cortex M4/M0 cores for picture processing. The following image shows the Pixy/CMUcam5:



Pixy/CMUcam5 (`http://a.co/fZtPqck`)

The most commonly available 2D vision sensors are webcams. They contain a CMOS sensor and USB interface, but they do not have any inbuilt vision-processing capabilities like Pixy has.

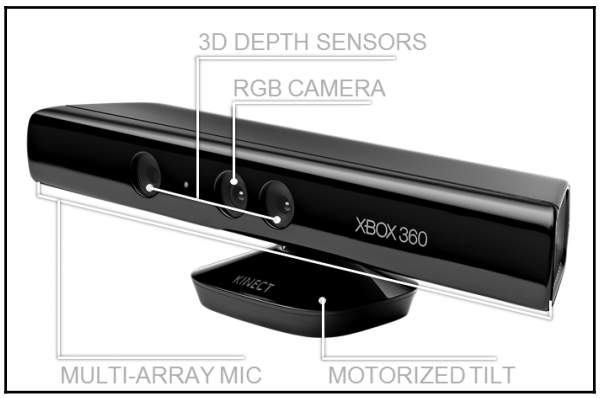
Logitech C920 webcam

The following picture shows a popular webcam from Logitech that can capture pictures of up to 5-megapixel resolution and HD videos:

Logitech HD C920 webcam (http://a.co/02DUUYd)

Kinect 360

We will now take a look at some of the 3D vision sensors available on the market. Some of the more popular sensors are Kinect, the Intel RealSense D400 series, and Orbbec Astra.



Kinect sensor

Kinect is a 3D vision sensor originally developed for the Microsoft Xbox 360 game console. It mainly contains an RGB camera, an infrared projector, an IR depth camera, a microphone array, and a motor to alter its tilt. The RGB camera and depth camera capture images at a resolution of 640 x 480 at 30 Hz. The RGB camera captures 2D color images, whereas the depth camera captures monochrome depth images. Kinect has a depth-sensing range of between 0.8 m and 4 m.

Some of the applications of Kinect are 3D motion capture, skeleton tracking, face recognition, and voice recognition.

Kinect can be interfaced with a PC using the USB 2.0 interface and programmed using Kinect SDK, OpenNI, and OpenCV. Kinect SDK is only available for Windows platforms, and SDK is developed and supplied by Microsoft. The other two libraries are open source and available for all platforms. The Kinect we are using here is the first version of Kinect; the latest versions of Kinect only support Kinect SDK when it is running on Windows (see `https://www.microsoft.com/en-us/download/details.aspx?id=40278` for more details).

The production of Kinect series sensors is discontinued, but you can still find the sensor on Amazon and eBay.

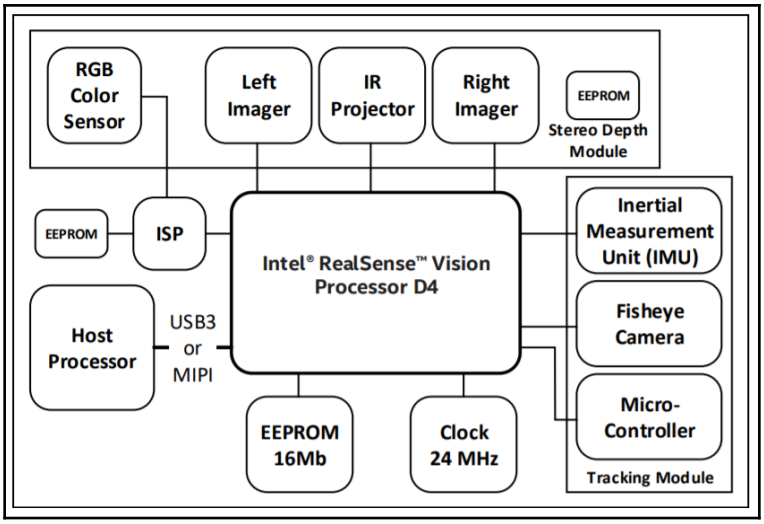
Intel RealSense D400 series

Intel RealSense D400 series (https://realsense.intel.com/)

The Intel RealSense D400 depth sensors are stereo cameras that come with an IR projector to enhance the depth data (see `https://software.intel.com/en-us/realsense/d400` for more details), as shown in Figure 4. The more popular sensor models in the D400 series are D415 and D435. In Figure 4, the sensor on the left is D415 and the sensor on the right is D435. Each consists of a stereo camera pair, an RGB camera, and an IR projector. The stereo camera pair computes the depth of the environment with the help of the IR projector.

The major features of this depth camera are that it can work in an indoor and outdoor environment. It can deliver the depth image stream with 1280 x 720 resolution at 90 fps, and the RGB camera can deliver a resolution of up to 1920 x 1080. It has a USB-C interface, which enables fast data transfer between the sensor and the computer. It has a small form factor and is lightweight, which is ideal for a robotics vision application.

The applications of Kinect and Intel RealSense are the same, except for speech recognition. They will work in Windows, Linux, and Mac. We can develop applications by using ROS, OpenNI, and OpenCV. The following diagram shows the block diagram of the D400 series camera:

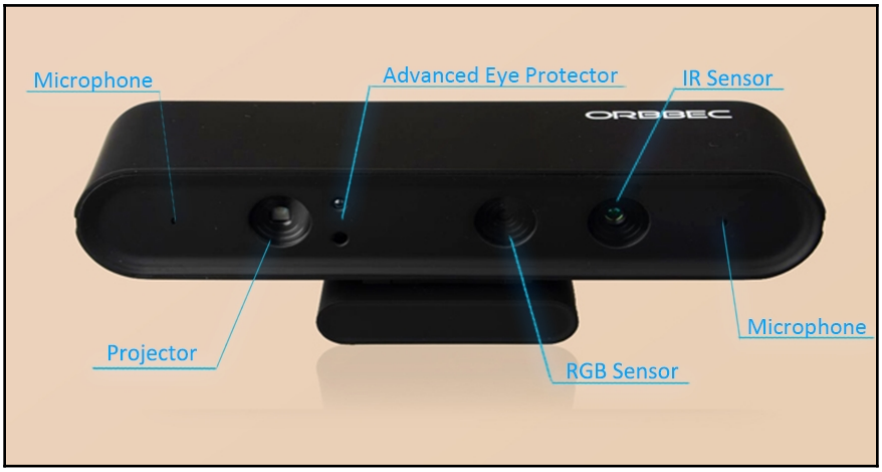


Block diagram of the Intel RealSense D400 series

You can find the datasheet of the Intel RealSense series at the following link:
`https://software.intel.com/sites/default/files/Intel_RealSense_Depth_Cam_D400_Series_Datasheet.pdf`
A research paper about Intel RealSense's depth sensor can be found at the following link:
`https://arxiv.org/abs/1705.05548`
You can find the Intel RealSense SDK at the following link:
`https://github.com/IntelRealSense/librealsense`

Orbbec Astra depth sensor

The new Orbbec Astra sensor is one of the alternatives to Kinect available on the market. It has similar specs compared to Kinect and uses similar technology to obtain depth information. Similar to Kinect, it has an IR projector, RGB camera, and IR sensor. It also comes with a microphone, which helps for voice recognition applications. The following image shows all parts of the Orbbec Astra depth sensor:



Orbbec Astra depth sensor (https://orbbec3d.com/product-astra/)

The Astra sensor comes in two models: Astra and Astra S. The main difference between these two models is the depth range. The Astra has a depth range of 0.6-8 m, whereas the Astra S has a range of 0.4-2 m. The Astra S is best suited for 3D scanning, whereas the Astra can be used in robotics applications. The size and weight of Astra is much lower than that of Kinect. These two models can both deliver depth data and an RGB image of 640 x 480 resolution at 30 fps. You can use a higher resolution, such as 1280 x 960, but it may reduce the frame rate. They also have the ability to track skeletons, like Kinect.

The sensor is compliant with the OpenNI framework, so an application built using OpenNI can also work using this sensor. We are going to use this sensor in our robot.

The SDK is compatible with Windows, Linux, and Mac OS X. For more information, you can go to the sensor's development website at `https://orbbec3d.com/develop/`.

One of the sensors you can also refer to is the ZED Camera (https://www.stereolabs.com/zed/). It is a stereo vision camera system which can able to deliver high resolution with good frame rate. The price is around 450 USD which is higher than above sensors. This can be used for high-end robotics applications required good accuracy from sensors.

We can see the ROS interfacing for this sensor in the upcoming section.

Introduction to OpenCV, OpenNI, and PCL

Let's look at the software frameworks and libraries that we will be using in our robots. First, let's look at OpenCV. This is one of the libraries that we are going to use in this robot for object detection and other image-processing capabilities.

What is OpenCV?

OpenCV is an open source, BSD-licensed computer vision library that includes the implementations of hundreds of computer-vision algorithms. The library, mainly intended for real-time computer vision, was developed by Intel Russia's research, and is now actively supported by Itseez (`https://github.com/Itseez`). In 2016, Intel acquired Itseez.

OpenCV is written mainly in C and C++, and its primary interface is in C++. It also has good interfaces in Python, Java, and MATLAB/Octave, and also has wrappers in other languages (such as C# and Ruby).

In the latest version of OpenCV, there is support for CUDA and OpenCL to enable GPU acceleration (`http://www.nvidia.com/object/cuda_home_new.html`).

OpenCV will run on most OS platforms (such as Windows, Linux, Mac OS X, Android, FreeBSD, OpenBSD, iOS, and BlackBerry).

In Ubuntu, OpenCV, the Python wrapper, and the ROS wrapper are already installed when we install the `ros-kinetic-desktop-full` or `ros-melodic-desktop-full` package. The following commands install the OpenCV-ROS package individually.

In Kinetic:

```
$ sudo apt-get install ros-kinetic-vision-opencv
```

In Melodic:

```
$ sudo apt-get install ros-melodic-vision-opencv
```

If you want to verify that the OpenCV-Python module is installed on your system, take a Linux Terminal, and enter the *python* command. You should then see the Python interpreter. Try to execute the following commands in the Python terminal to verify the OpenCV installation:

```
>>> import cv2
>>> cv2.__version__
```

If this command is successful, this version of OpenCV will be installed on your system. The version might be either 3.3.x or 3.2.x.

If you want to try OpenCV in Windows, you can try the following link: `https://docs.opencv.org/3.3.1/d5/de5/tutorial_py_setup_in_windows.html`
The following link will guide you through the installation process of OpenCV on Mac OS X: `https://www.learnopencv.com/install-opencv3-on-macos/`

The main applications of OpenCV are in the following fields:

- Object detection
- Gesture recognition
- Human-computer interaction
- Mobile robotics
- Motion tracking
- Facial-recognition systems

Installation of OpenCV from the source code in Ubuntu

The OpenCV installation can be customized. If you want to customize your OpenCV installation, you can try to install it from the source code. You can find out how to do this installation at `https://docs.opencv.org/trunk/d7/d9f/tutorial_linux_install.html`.

To work with the examples in this chapter, it's best that you work with OpenCV installed, along with ROS.

Reading and displaying an image using the Python-OpenCV interface

The first example will load an image in grayscale and display it on the screen.

In the following section of code, we will import the `numpy` module for handling the image array. The `cv2` module is the OpenCV wrapper for Python, which we can use to access OpenCV Python APIs. NumPy is an extension to the Python programming language, adding support for large multidimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays (see `https://pypi.python.org/pypi/numpy` for more information):

```
#!/usr/bin/env python
import numpy as np
import cv2
```

The following function will read the `robot.jpg` image and load this image in grayscale. The first argument of the `cv2.imread()` function is the name of the image and the next argument is a flag that specifies the color type of the loaded image. If the flag is greater than 0, the image returns a three-channel RGB color image; if the flag is 0, the loaded image will be a grayscale image; and if the flag is less than 0, it will return the same image as was loaded:

```
img = cv2.imread('robot.jpg',0)
```

The following section of code will show the read image using the `imshow()` function. The `cv2.waitKey(0)` function is a keyboard-binding function. Its argument is time in milliseconds. If it's 0, it will wait indefinitely for a key stroke:

```
cv2.imshow('image', img)
cv2.waitKey(0)
```

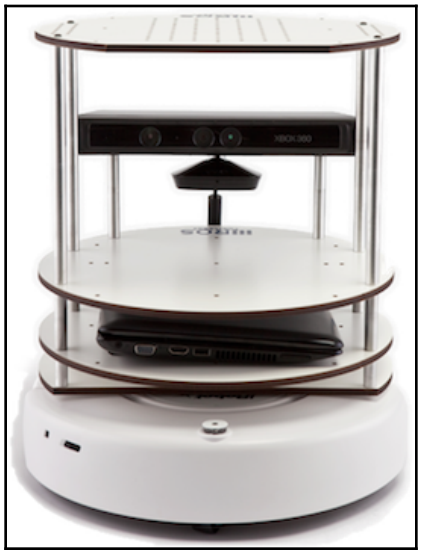
The `cv2.destroyAllWindows()` function simply destroys all the windows we created:

```
cv2.destroyAllWindows()
```

Save the preceding code as `image_read.py` and copy a JPG file and name it `robot.jpg`. Execute the code using the following command:

```
$python image_read.py
```

The output will load an image in grayscale because we used `0` as the value in the `imread()` function:



Output of read image code

The following example will try to use an open webcam. The program will quit when the user presses any button.

Capturing from the web camera

The following code will capture an image using the webcam with the device name `/dev/video0` or `/dev/video1`.

We need to import the *numpy* and *cv2* modules for capturing an image from a camera:

```
#!/usr/bin/env python
import numpy as np
import cv2
```

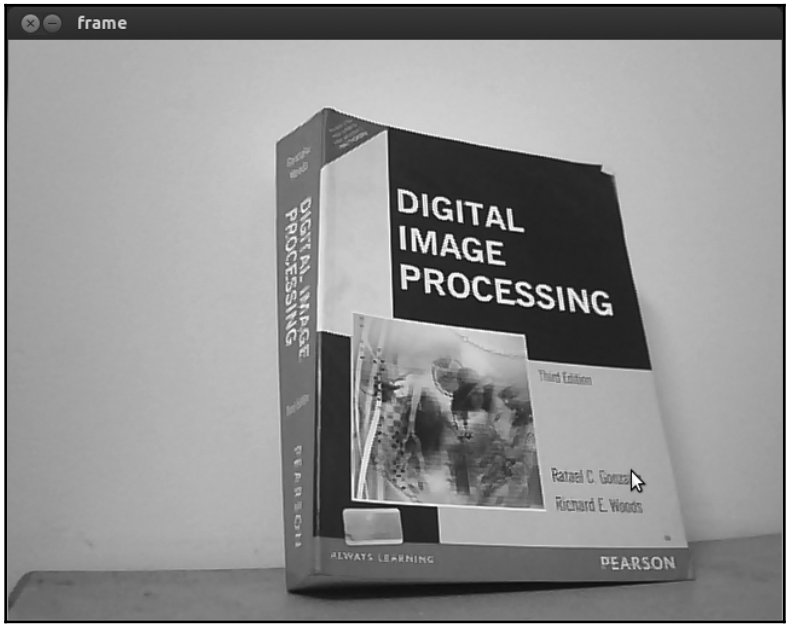
The following function will create a `VideoCapture` object. The `VideoCapture` class is used to capture videos from video files or cameras. The initialization argument of the `VideoCapture` class is the index of a camera or the name of a video file. The device index is just a number that is used to specify the camera. The first camera index is 0, and has the device name `/dev/video0`-that's why we will put `0` in the following code:

```
cap = cv2.VideoCapture(0)
```

The following section of code is looped to read image frames from the `VideoCapture` object, and shows each frame. It will quit when any key is pressed:

```
while(True):
    # Capture frame-by-frame
    ret, frame = cap.read()
    # Display the resulting frame
    cv2.imshow('frame', frame)
    k = cv2.waitKey(30)
    if k > 0:
        break
```

The following is a screenshot of the program output:



Output of the video capture

You can explore more OpenCV-Python tutorials at

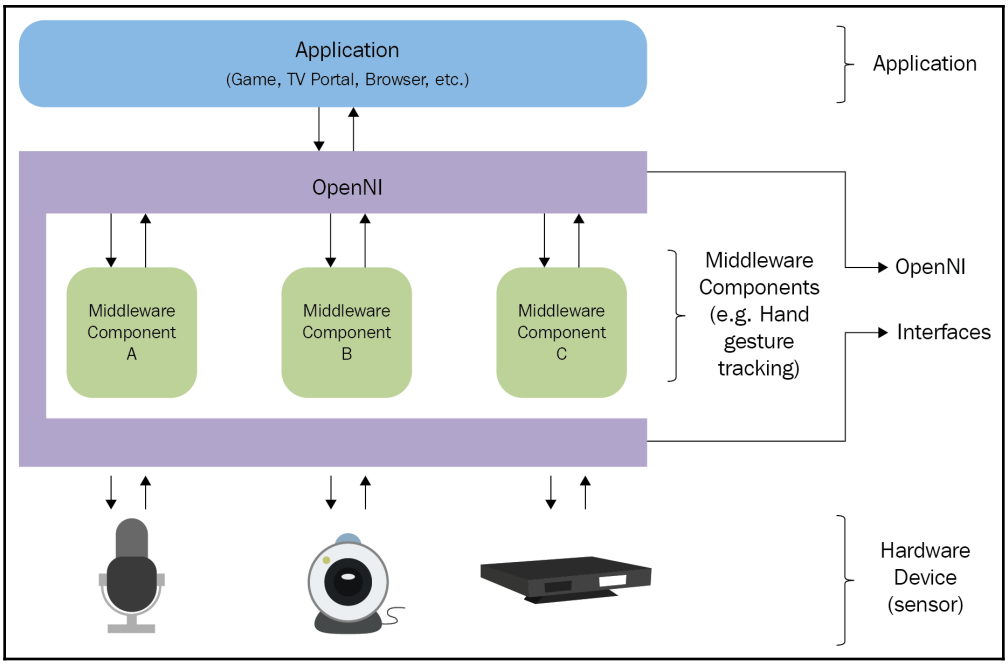
`http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_tutorials.html`.

In the next section, we will look at the OpenNI library and its application.

What is OpenNI?

OpenNI is a multilanguage, cross-platform framework that defines APIs in order to write applications using **natural interaction** (**NI**) (see `https://structure.io/openni` for more information). Natural interaction refers to the way in which people naturally communicate through gestures, expressions, and movements, and discover the world by looking around and manipulating physical objects and materials.

OpenNI APIs are composed of a set of interfaces that are used to write NI applications. The following figure shows a three-layered view of the OpenNI library:



OpenNI framework software architecture

The top layer represents the application layer that implements the natural interaction-based application. The middle layer is the OpenNI layer, and it will provide communication interfaces that interact with sensors and middleware components that analyze the data from the sensor. Middleware can be used for full-body analysis, hand-point analysis, gesture detection, and so on. One example of a middle layer component is NITE (`http://www.openni.ru/files/nite/index.html`), which can detect gestures and skeletons.

The bottom layer contains the hardware devices that capture the visual and audio elements of the scene. It can include 3D sensors, RGB cameras, IR cameras, and microphones.

The latest version of OpenNI is OpenNI 2, which support sensors such as Asus Xtion Pro, and Primesense Carmine. The first version of OpenNI mainly supports the Kinect 360 sensor.

OpenNI is cross platform, and has been successfully compiled and deployed on Linux, Mac OS X, and Windows.

In the next section, we will see how we to install OpenNI in Ubuntu.

Installing OpenNI in Ubuntu

We can install the OpenNI library along with ROS packages. ROS is already interfaced with OpenNI, but the ROS desktop full installation may not install OpenNI packages; if so, we need to install it from the package manager.

The following command will install the ROS-OpenNI library (which is mainly supported by the Kinect Xbox 360 sensor) in Kinetic and Melodic:

```
$ sudo apt-get install ros-<version>-openni-launch
```

The following command will install the ROS-OpenNI 2 library (which is mainly supported by Asus Xtion Pro and Primesense Carmine):

```
$ sudo apt-get install ros-<version>-openni2-launch
```

The source code and latest build of OpenNI for Windows, Linux, and MacOS X is available at `http://structure.io/openni`.

In the next section, we will look at how to install PCL.

What is PCL?

A **point cloud** is a set of data points in space that represent a 3D object or an environment. Generally, a point cloud is generated from depth sensors, such as Kinect and LIDAR. PCL (Point Cloud Library) is a large scale, open project for 2D/3D images and point-cloud processing. The PCL framework contains numerous algorithms that perform filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation. Using these methods, we can process the point cloud, extract key descriptors to recognize objects in the world based on their geometric appearance, create surfaces from the point clouds, and visualize them.

PCL is released under the BSD license. It's open source, free for commercial use, and free for research use. PCL is cross platform and has been successfully compiled and deployed on Linux, macOS X, Windows, and Android/iOS.

You can download PCL at `http://pointclouds.org/downloads/`.

PCL is already integrated into ROS. The PCL library and its ROS interface are included in a ROS full desktop installation. PCL is the 3D-processing backbone of ROS. Refer to http://wiki.ros.org/pcl for details on the ROS-PCL package.

Programming Kinect with Python using ROS, OpenCV, and OpenNI

Let's look at how we can interface and work with the Kinect sensor in ROS. ROS is bundled with the OpenNI driver, which can fetch the RGB and depth image of Kinect. The OpenNI and OpenNI 2 package in ROS can be used for interfacing with Microsoft Kinect, Primesense Carmine, Asus Xtion Pro, and Pro Live.

When we install ROS's `openni_launch` package, it will also install its dependent packages, such as `openni_camera`. The `openni_camera` package is the Kinect driver that publishes raw data and sensor information, whereas the `openni_launch` package contains ROS launch files. These launch files launch multiple nodes at a time and publish data such as the raw depth, RGB, and IR images, and the point cloud.

How to launch the OpenNI driver

You can connect the Kinect sensor to your computer using a USB interface and make sure it is detected on your PC using the `dmesg` command in the terminal. After setting up Kinect, we can start ROS's OpenNI driver to get data from the device.

The following command will open the OpenNI device and load all nodelets (see `http://wiki.ros.org/nodelet` for more information) to convert raw depth/RGB/IR streams to depth images, disparity images, and point clouds. The ROS `nodelet` package is designed to provide a way to run multiple algorithms in the same process with zero copy transport between algorithms:

```
$ roslaunch openni_launch openni.launch
```

After starting the driver, you can list out the various topics published by the driver using the following command:

```
$ rostopic list
```

You can view the RGB image using a ROS tool called `image_view`:

```
$ rosrun image_view image_view image:=/camera/rgb/image_color
```

In the next section, we will learn how to interface these images with OpenCV for image processing.

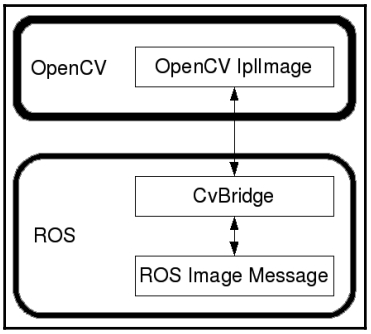
The ROS interface with OpenCV

OpenCV is also integrated into ROS, mainly for image processing. The `vision_opencv` ROS stack includes the complete OpenCV library and the interface with ROS.

The `vision_opencv` meta package consists of individual packages:

- `cv_bridge`: This contains the `CvBridge` class. This class converts ROS image messages to the OpenCV image data type and vice versa.
- `image_geometry`: This contains a collection of methods to handle image and pixel geometry.

The following diagram shows how OpenCV is interfaced with ROS:



OpenCV-ROS interfacing

The image data types of OpenCV are `IplImage` and `Mat`. If we want to work with OpenCV in ROS, we have to convert `IplImage` or `Mat` to ROS image messages. The ROS package `vision_opencv` has the `CvBridge` class; this class can convert `IplImage` to a ROS image and vice versa. Once we get the ROS image topics from any kind of vision sensor, we can use ROS CvBridge in order to convert it from ROS topic to Mat or IplImage format.

The following section shows you how to create a ROS package; this package contains a node to subscribe to RGB and depth images, process RGB images to detect edges and display all images after converting them to an image type equivalent to OpenCV.

Creating a ROS package with OpenCV support

We can create a package called `sample_opencv_pkg` with the following dependencies: `sensor_msgs`, `cv_bridge`, `rospy`, and `std_msgs`. The `sensor_msgs` dependency defines ROS messages for commonly used sensors, including cameras and scanning-laser rangefinders. The `cv_bridge` dependency is the OpenCV interface of ROS.

The following command will create the ROS package with the aforementioned dependencies:

```
    $ catkin-create-pkg sample_opencv_pkg sensor_msgs cv_bridge
rospy std_msgs
```

After creating the package, create a `scripts` folder inside the package; we will use it as a location in which to save the code that will be mentioned in the next section.

Displaying Kinect images using Python, ROS, and cv_bridge

The first section of the Python code is given in the following code fragment. It mainly involves importing `rospy`, `sys`, `cv2`, `sensor_msgs`, `cv_bridge`, and the `numpy` module. The `sensor_msgs` dependency imports the ROS data type of both image and camera information type. The `cv_bridge` module imports the `CvBridge` class for converting the ROS image data type to the OpenCV data type and vice versa:

```
import rospy
import sys
import cv2
from sensor_msgs.msg import Image, CameraInfo
from cv_bridge import CvBridge, CvBridgeError
from std_msgs.msg import String
import numpy as np
```

The following section of code is a class definition in Python that we will use to demonstrate `CvBridge` functions. The class is called `cvBridgeDemo`:

```
class cvBridgeDemo():
    def __init__(self):
        self.node_name = "cv_bridge_demo"
        #Initialize the ros node
        rospy.init_node(self.node_name)

        # What we do during shutdown
        rospy.on_shutdown(self.cleanup)

        # Create the cv_bridge object
        self.bridge = CvBridge()

        # Subscribe to the camera image and depth topics and set
        # the appropriate callbacks
        self.image_sub =
 rospy.Subscriber("/camera/rgb/image_raw", Image,
 self.image_callback)        self.depth_sub =
 rospy.Subscriber("/camera/depth/image_raw", Image,
 self.depth_callback)

 #Callback executed when the timer timeout
      rospy.Timer(rospy.Duration(0.03), self.show_img_cb)

      rospy.loginfo("Waiting for image topics...")
```

Here is the callback to visualize the actual RGB image, processed RGB image, and depth image:

```
def show_img_cb(self,event):
     try:


         cv2.namedWindow("RGB_Image", cv2.WINDOW_NORMAL)
         cv2.moveWindow("RGB_Image", 25, 75)
         cv2.namedWindow("Processed_Image", cv2.WINDOW_NORMAL)
         cv2.moveWindow("Processed_Image", 500, 75)

         # And one for the depth image
         cv2.moveWindow("Depth_Image", 950, 75)
         cv2.namedWindow("Depth_Image", cv2.WINDOW_NORMAL)

         cv2.imshow("RGB_Image",self.frame)
         cv2.imshow("Processed_Image",self.display_image)
         cv2.imshow("Depth_Image",self.depth_display_image)
         cv2.waitKey(3)
     except:
         pass
```

The following code gives a callback function of the color image from Kinect. When a color image is received on the `/camera/rgb/image_raw` topic, it will call this function. This function will process the color frame for edge detection and show the edge detected and the raw color image:

```
def image_callback(self, ros_image):
    # Use cv_bridge() to convert the ROS image to OpenCV format
    try:
        self.frame = self.bridge.imgmsg_to_cv2(ros_image, "bgr8")
    except CvBridgeError, e:
        print e
   pass

    # Convert the image to a Numpy array since most cv2 functions
    # require Numpy arrays.
    frame = np.array(self.frame, dtype=np.uint8)
    # Process the frame using the process_image() function
    self.display_image = self.process_image(frame)
```

The following code gives a callback function of the depth image from Kinect. When a depth image is received on the `/camera/depth/raw_image` topic, it will call this function. This function will show the raw depth image:

```
def depth_callback(self, ros_image):
 # Use cv_bridge() to convert the ROS image to OpenCV format
 try:
     # The depth image is a single-channel float32 image
     depth_image = self.bridge.imgmsg_to_cv2(ros_image, "32FC1")
 except CvBridgeError, e:
     print e
pass
 # Convert the depth image to a Numpy array since most cv2 functions
 # require Numpy arrays.
 depth_array = np.array(depth_image, dtype=np.float32)
 # Normalize the depth image to fall between 0 (black) and 1 (white)
 cv2.normalize(depth_array, depth_array, 0, 1, cv2.NORM_MINMAX)
 # Process the depth image
 self.depth_display_image = self.process_depth_image(depth_array)
```

The following function is called `process_image()`, and will convert the color image to grayscale, then blur the image, and find the edges using the canny edge filter:

```
def process_image(self, frame):
    # Convert to grayscale
    grey = cv2.cvtColor(frame, cv.CV_BGR2GRAY)

    # Blur the image
    grey = cv2.blur(grey, (7, 7))

    # Compute edges using the Canny edge filter
    edges = cv2.Canny(grey, 15.0, 30.0)

    return edges
```

The following function is called `process_depth_image()`. It simply returns the depth frame:

```
def process_depth_image(self, frame):
    # Just return the raw image for this demo
    return frame
```

The following function will close the image window when the node shuts down:

```
def cleanup(self):
    print "Shutting down vision node."
    cv2.destroyAllWindows()
```

The following code is the `main()` function. It will initialize the `cvBridgeDemo()` class and call the `rospy.spin()` function:

```
def main(args):
    try:
        cvBridgeDemo()
        rospy.spin()
    except KeyboardInterrupt:
        print "Shutting down vision node."
        cv.DestroyAllWindows()

if __name__ == '__main__':
    main(sys.argv)
```

Save the preceding code as `cv_bridge_demo.py` and change the permission of the node using the following command. The nodes are only visible to the `rosrun` command if we give it executable permission:

```
$ chmod +X cv_bridge_demo.py
```

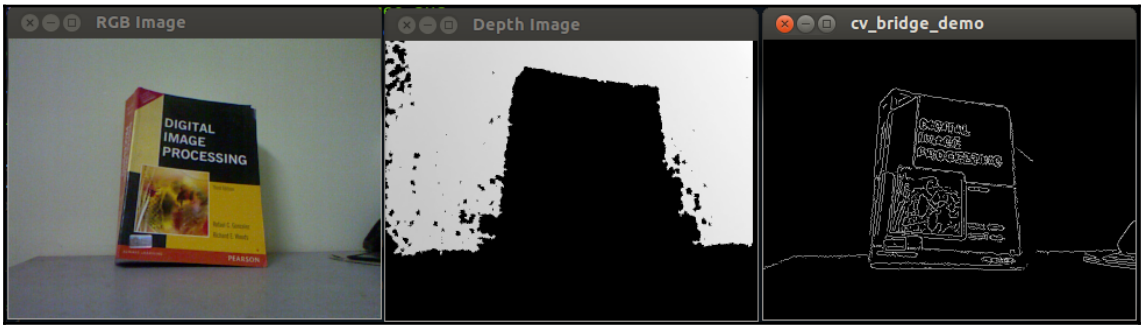
The following are the commands to start the driver and node. Start the Kinect driver using the following command:

```
$ roslaunch openni_launch openni.launch
```

Run the node using the following command:

```
$ rosrun sample_opencv_pkg cv_bridge_demo.py
```

The following is a screenshot of the output:



RGB, depth, and edge images

Interfacing Orbbec Astra with ROS

One of the alternatives to Kinect is Orbbec Astra. There are ROS drivers available for Astra, and we can see how to set up that driver and get the image, depth, and point cloud from this sensor.

Installing the Astra–ROS driver

The complete instructions to set up the Astra-ROS driver in Ubuntu are mentioned at `https://github.com/orbbec/ros_astra_camera` and `http://wiki.ros.org/Sensors/OrbbecAstra`. After installing the driver, you can launch it using the following command:

```
$ roslaunch astra_launch astra.launch
```

You can also install the Astra driver from the ROS package repository. Here is the command to install those packages:

```
$ sudo apt-get install ros-kinetic-astra-camera
$ sudo apt-get install ros-kinetic-astra-launch
```

After installing these packages, you have to set the permission of the device in order to work with the device, as described at `http://wiki.ros.org/astra_camera`. You can check the ROS topics that are generated from this driver using the `rostopic` list command in the terminal. In addition, we can use the same Python code for image processing that we mentioned in the previous section.

Working with point clouds using Kinect, ROS, OpenNI, and PCL

A 3D point cloud is a way of representing a 3D environment and 3D objects as collection points along the x, y, and z axes. We can get a point cloud from various sources: Either we can create our point cloud by writing a program or we can generate it from depth sensors or laser scanners.

PCL supports the OpenNI 3D interfaces natively; thus, it can acquire and process data from devices (such as Prime Sensor's 3D cameras, Microsoft Kinect, or Asus Xtion Pro).

PCL will be included in the ROS full desktop installation. Let's see how we can generate and visualize a point cloud in RViz, a data visualization tool in ROS.

Opening the device and generating a point cloud

Open a new terminal and launch the ROS-OpenNI driver, along with the point cloud generator nodes, using the following command:

```
$ roslaunch openni_launch openni.launch
```

This command will activate the Kinect driver and process the raw data into convenient outputs, such as a point cloud.

If you are using Orbbec Astra, you can use the following command:

```
$ roslaunch astra_launch astra.launch
```

We will use the RViz 3D visualization tool to view our point clouds.

The following command will start the RViz tool:

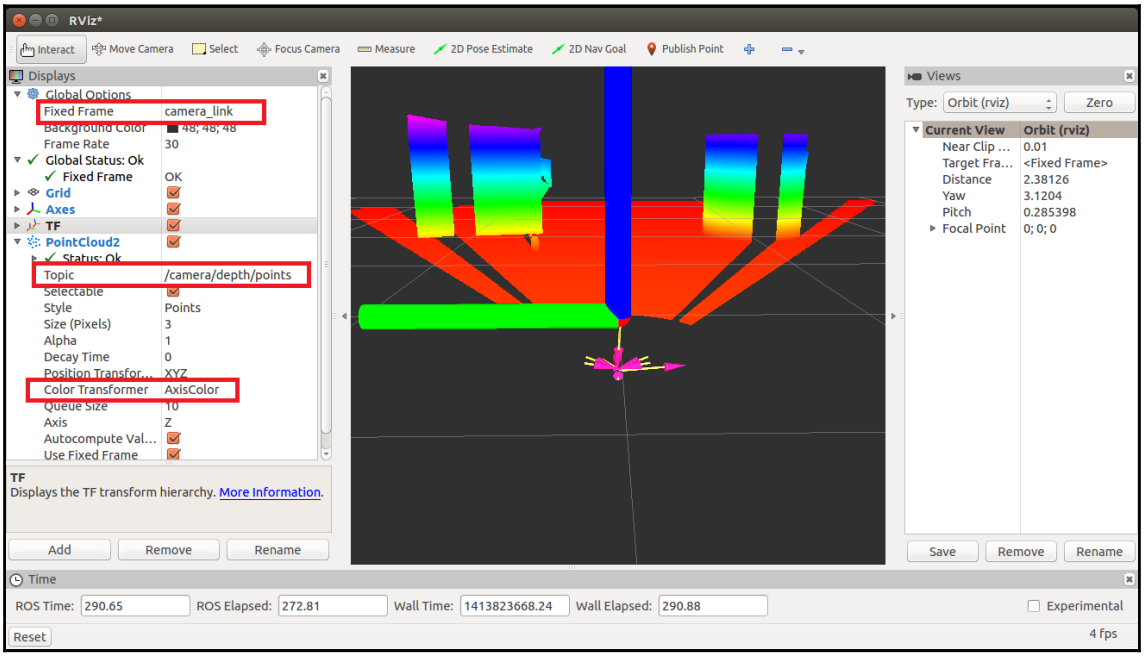
```
$ rosrun rviz rviz
```

Set the RViz options for **Fixed Frame** (at the top of the **Displays** panel under **Global Options**) to **camera_link**.

From the left-hand side panel of the **RViz** panel, click on the **Add** button and choose the **PointCloud2** display option. Set its topic to **/camera/depth/points** (this is the topic for Kinect; it will be different for other sensors)

Change the **Color Transformer** of **PointCloud2** to **AxisColor**.

The following screenshot shows a screenshot of the RViz point cloud data. You can see the nearest objects are marked in red and the farthest objects are marked in violet and blue. The objects in front of the Kinect are represented as a cylinder and cube:



Visualizing point cloud data in Rviz

Conversion of point cloud data to laser scan data

We are using Astra in this robot to replicate the function of an expensive laser range scanner. The depth image is processed and converted to the data equivalent of a laser scanner using ROS's `depthimage_to_laserscan` package (see `http://wiki.ros.org/depthimage_to_laserscan` for more information).

You can either install this package from the source code or use the Ubuntu package manager. Here is the command to install this package from the Ubuntu package manager

```
$ sudo apt-get install ros-<version>-depthimage-to-laserscan
```

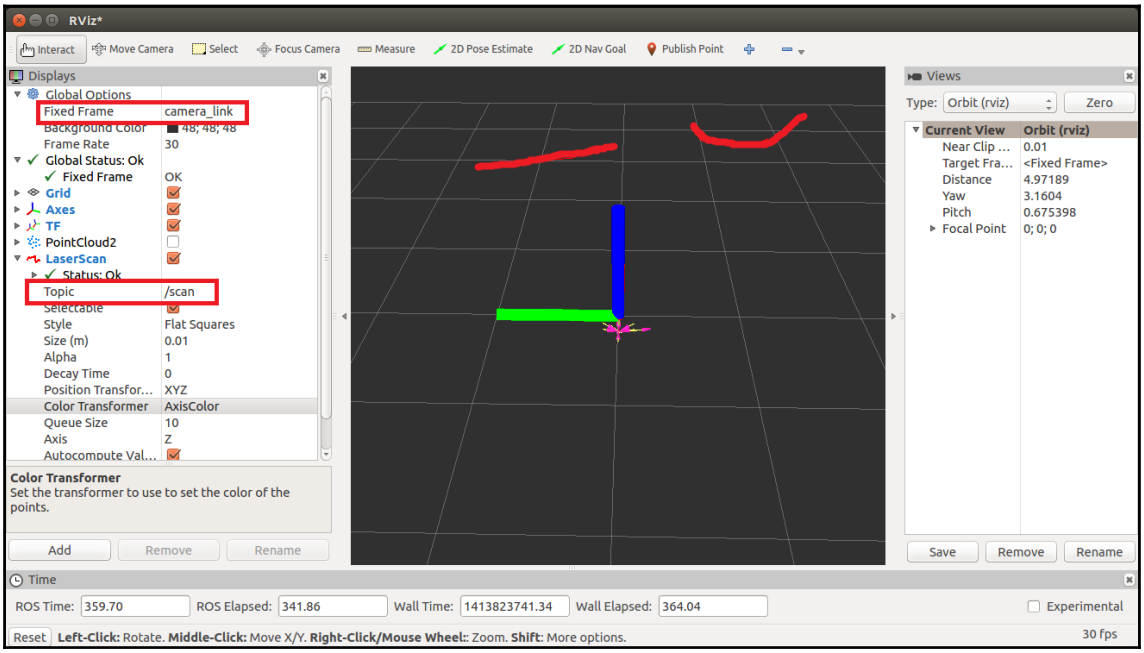
The main function of this package is to slice a section of the depth image and convert it to an equivalent laser scan data type. The ROS `sensor_msgs/LaserScan` message type is used for publishing the laser scan data. This `depthimage_to_laserscan` package will perform this conversion and fake the laser scanner data. The laser scanner output can be viewed using RViz. In order to run the conversion, we have to start the convertor nodelets that will perform this operation. We have to specify this in our launch file in order to start the conversion. The following is the required code in the launch file to start the `depthimage_to_laserscan` conversion:

```
    <!-- Fake laser -->
    <node pkg="nodelet" type="nodelet"
   name="laserscan_nodelet_manager" args="manager"/>  <node pkg="nodelet"
  type="nodelet"
   name="depthimage_to_laserscan"        args="load
  depthimage_to_laserscan/DepthImageToLaserScanNodelet
   laserscan_nodelet_manager">
      <param name="scan_height" value="10"/>
      <param name="output_frame_id" value="/camera_depth_frame"/>
      <param name="range_min" value="0.45"/>
      <remap from="image" to="/camera/depth/image_raw"/>
      <remap from="scan" to="/scan"/>
    </node>
```

The topic of the depth image can be changed in each sensor; you have to change the topic name according to your depth image topic.

As well as starting the nodelet, we need to set certain parameters of the nodelet for better conversion. Refer to `http://wiki.ros.org/depthimage_to_laserscan` for a detailed explanation of each parameter.

The laser scan of the preceding view is given in the following screenshot. To view the laser scan, add the **LaserScan** option. This is similar to how we add the **PointCloud2** option and change the **Topic** value of **LaserSan** to **/scan**:



Visualizing laser scan data in Rviz

Working with SLAM using ROS and Kinect

The main aim of deploying vision sensors in our robot is to detect objects and navigate the robot through an environment. SLAM is a algorithm that is used in mobile robots to build up a map of an unknown environment or update a map within a known environment by tracking the current location of the robot.

Maps are used to plan the robot's trajectory and to navigate through this path. Using maps, the robot will get an idea about the environment. The two main challenges in mobile robot navigation are mapping and localization.

Mapping involves generating a profile of obstacles around the robot. Through mapping, the robot will understand what the world looks like. Localization is the process of estimating the position of the robot relative to the map we build.

SLAM fetches data from different sensors and uses it to build maps. The 2D/3D vision sensor can be used to input data into SLAM. 2D vision sensors, such as web cameras, and 3D sensors, such as Kinect, are mainly used as inputs for the SLAM algorithm.

A SLAM library called OpenSlam (`http://openslam.org/gmapping.html`) is integrated with ROS as a package called gmapping. The `gmapping` package provides a node to perform laser-based SLAM processing, called `slam_gmapping`. This can create a 2D map from the laser and position data collected by the mobile robot.

The `gmapping` package is available at `http://wiki.ros.org/gmapping`.

To use the `slam_gmapping` node, we have to input the odometry data of the robot and the laser scan output from the laser range finder, which is mounted horizontally on the robot.

The `slam_gmapping` node subscribes to the `sensor_msgs/LaserScan` messages and `nav_msgs/Odometry` messages to build the map (`nav_msgs/OccupancyGrid`). The generated map can be retrieved via a ROS topic or service.

We have used the following launch file to use SLAM in our Chefbot. This launch file launches the `slam_gmapping` node and contains the necessary parameters to start mapping the robot's environment:

```
$ roslaunch chefbot_gazebo gmapping_demo.launch
```

Summary

In this chapter, we looked at the various vision sensors that can be used in Chefbot. We used Kinect and Astra in our robot and learned about OpenCV, OpenNI, PCL, and their application. We also discussed the role of vision sensors in robot navigation, the popular SLAM technique, and its application using ROS. In the next chapter, we will see the complete interfacing of the robot and learn how to perform autonomous navigation with our Chefbot.

Questions

1. What are 3D sensors and how are they different from ordinary cameras?
2. What are the main features of ROS?
3. What are the applications of OpenCV, OpenNI, and PCL?
4. What is SLAM?
5. What is RGB-D SLAM and how does it work?

Further reading

You can read more about the robotic vision package in ROS at the following links:

- `http://wiki.ros.org/vision_opencv`
- `http://wiki.ros.org/pcl`

8
Building ChefBot Hardware and the Integration of Software

In `Chapter 3`, *Modeling a Differential Robot Using ROS and URDF*, we looked at the ChefBot chassis design. In this chapter, we will learn how to assemble this robot using those parts. We will also look at the final interfacing of the sensors and other electronic components of this robot with Tiva-C LaunchPad. After the interfacing, we will learn how to interface the robot with the PC and implement autonomous navigation using SLAM and AMCL in the real robot.

The following topics will be covered in this chapter:

- Building ChefBot hardware
- Configuring the ChefBot PC and packages
- Interfacing the ChefBot sensors with Tiva-C Launchpad
- Embedded code for ChefBot
- Understanding ChefBot ROS packages
- Implementing SLAM on ChefBot
- Autonomous navigation in ChefBot

Technical requirements

To test the application and codes in this chapter, you will need an Ubuntu 16.04 LTS PC/laptop with ROS Kinetic installed.

You will also need fabricated robot chassis parts for assembling the robot.

You should have all the sensors and other hardware components that can be integrated in the robot.

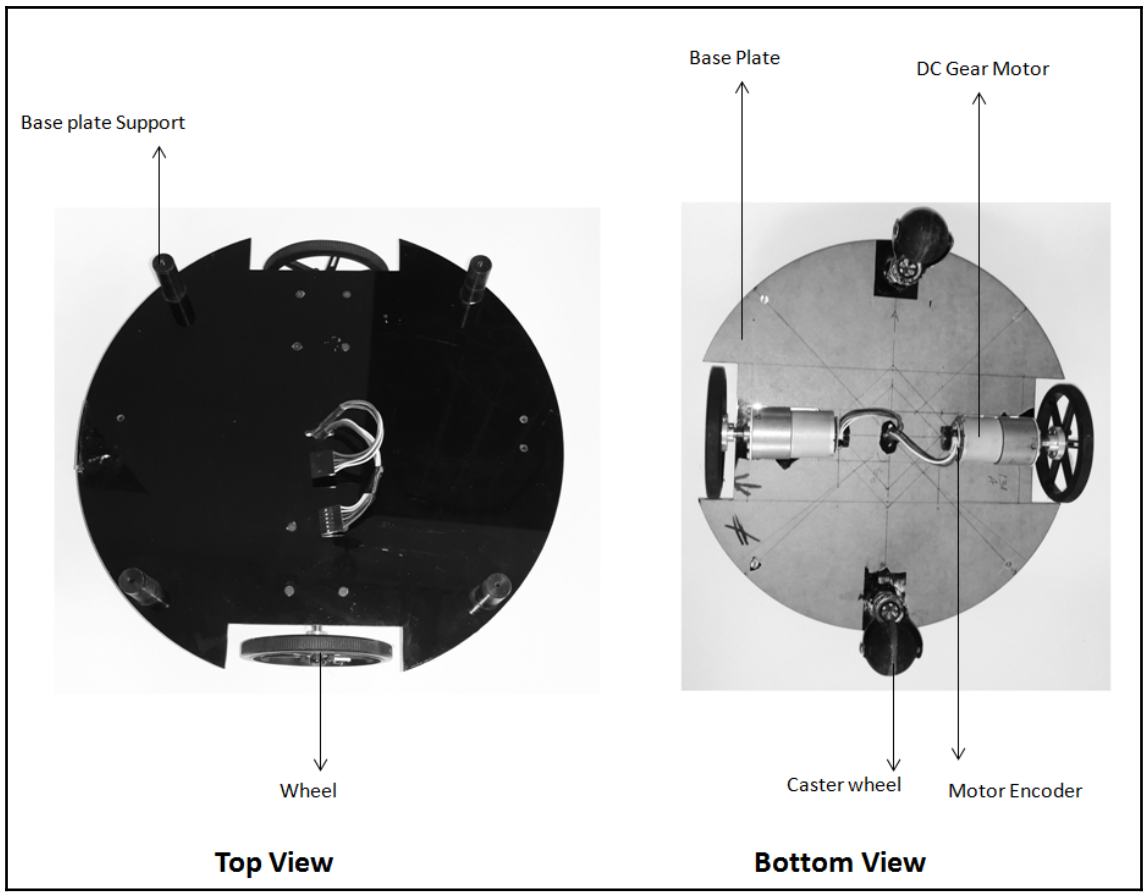
We have already discussed interfacing individual robot components and sensors with Launchpad. In this chapter, we will try to interface the necessary robotic components and sensors of ChefBot and program it in such a way that it will receive the values from all sensors and control the information from the PC. Launchpad will send all sensor values to the PC via a serial port and also receive control information (such as reset commands, speed data, and so on) from the PC.

After receiving Serial port data from the Tiva C Launchpad, a ROS Python node will receive the serial values and convert them to ROS topics. There are other ROS nodes present in the PC that subscribe to these sensor topics and compute robot odometry. The data from the wheel encoders and IMU values combine to calculate the odometry of the robot. The robot detects obstacles by subscribing to the ultrasonic sensor topic and laser scan and controls the speed of the wheel motors using the PID node. This node converts the linear velocity command to a differential wheel velocity command. After running these nodes, we can run SLAM to map the area, and after running SLAM, we can run the AMCL nodes for localization and autonomous navigation.

In the first section of this chapter, *Building ChefBot hardware*, we will learn how to assemble the ChefBot hardware using the body parts and electronic components of the robot.

Building ChefBot hardware

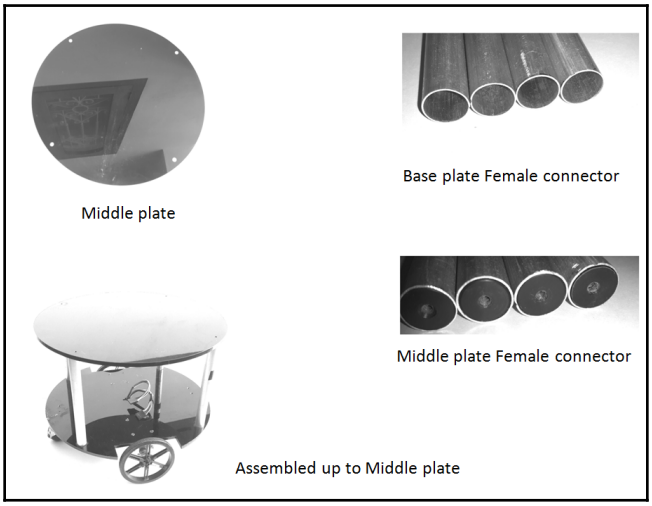
The first section of the robot that needs to be configured is the base plate. The base plate consists of two motors and their attached wheels, the caster wheels, and the base plate supports. The following image shows the top and bottom view of the base plate:



Base plate with motors, wheels, and caster wheels

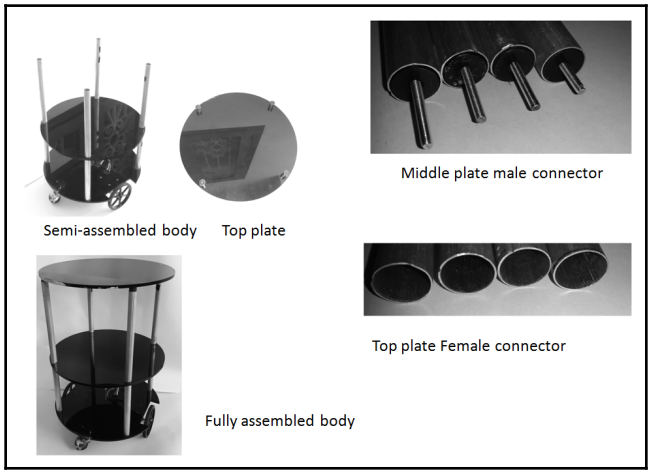
The base plate has a radius of 15 cm, and the motors and their attached wheels are mounted on the opposite sides of the plate by cutting two sections from the base plate. Two rubber caster wheels are mounted on opposite sides of the base plate to achieve a good balance and support for the robot. We can either choose ball caster wheels or rubber caster wheels for this robot. The wires of the two motors are taken to the top of the base plate through a hole in the center of the base plate. To extend the layers of the robot, we will put base plate supports to connect the following layers. Now, let's look at the next layer with the middle plate and connecting tubes. There are hollow tubes to connect the base plate and the middle plate. The hollow tubes can be connected to the base plate support.

The following image shows the middle plate and connecting tubes:



Middle plate with connecting tubes

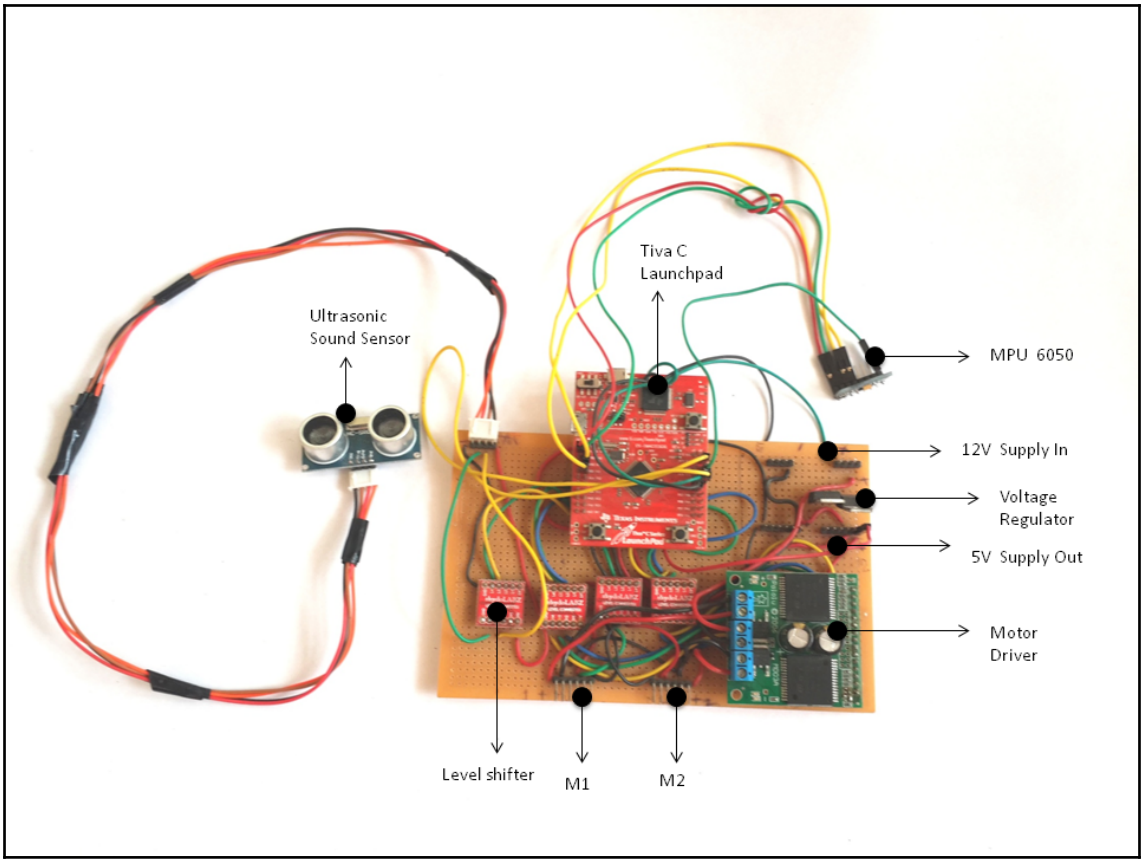
The connecting tube will connect the base plate and the middle plate. There are four hollow tubes to connect the base plate to the middle plate. One end of these tubes is hollow, which can fit the base plate support, and the other end has a hard plastic fitting with a hole for a screw. The middle plate has no support, except for four holes for the connecting tubes:



Fully assembled robot body

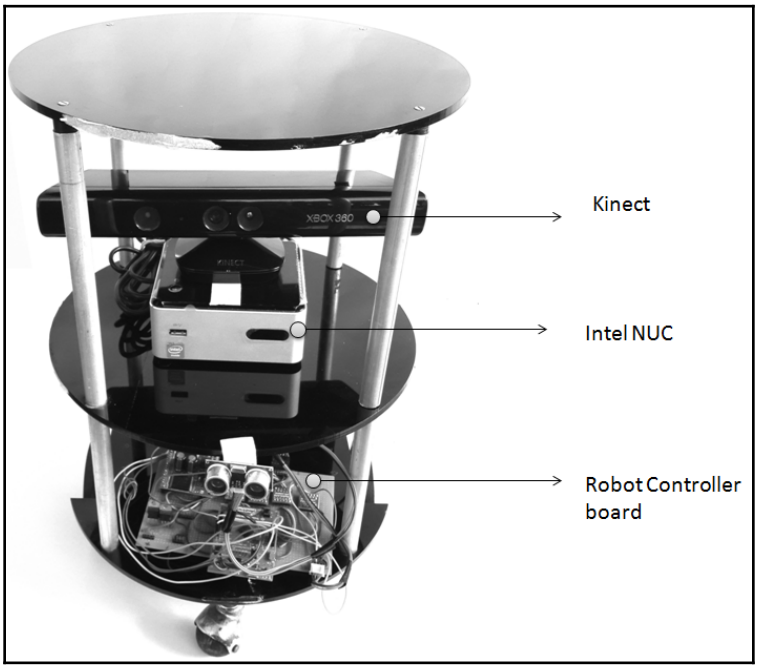
The middle plate male connector helps to connect the middle plate and the top of the base plate tubes. We can fit the top plate at the top of the middle plate tubes using the four supports on the back of the top plate. We can insert the top plate's female connector into the top plate support. Now we have the fully assembled body of the robot.

The bottom layer of the robot can be used to put the **printed circuit board** (**PCB**) and battery. In the middle layer, we can put the Kinect/Orbecc and Intel NUC. We can put a speaker and mic if needed. We can use the top plate to carry food. The following image shows the PCB prototype of the robot; it consists of Tiva-C LaunchPad, a motor driver, level shifters, and provisions to connect two motors, ultrasonic sensors, and IMU:



ChefBot PCB prototype

The board is powered by a 12 V battery placed on the base plate. The two motors can be directly connected to the M1 and M2 male connectors. The NUC PC and Kinect are placed on the middle plate. The LaunchPad board and Kinect should be connected to the NUC PC via USB. The PC and Kinect are powered using the same 12 V battery itself. We can use a lead-acid or lithium-polymer battery. Here, we are using a lead-acid cell for testing purposes. Later, we will migrate to a lithium-polymer battery for better performance and better backup. The following image shows a diagram of the complete, assembled ChefBot:



Fully assembled robot body

After assembling all the parts of the robot, we will start working with the robot software. ChefBot's embedded code and ROS packages are available in the codes under `chapter_8`. Let's get that code and start working with the software.

Configuring ChefBot PC and setting ChefBot ROS packages

In ChefBot, we are using Intel's NUC PC to handle the robot sensor data and the processing of the data. After procuring the NUC PC, we have to install Ubuntu 16.04 LTS. After the installation of Ubuntu, install the complete ROS and its packages that we mentioned in the previous chapters. We can configure this PC separately, and after the configuration of all the settings, we can put this into the robot. The following are the procedures to install the ChefBot packages on the NUC PC.

Clone ChefBot's software packages from GitHub using the following command:

```
$ git clone https://github.com/qboticslabs/learning_robotics_2nd_ed
```

We can clone this code in our laptop and copy the `ChefBot` folder to Intel's NUC PC. The `ChefBot` folder consists of the ROS packages of the ChefBot hardware. In the NUC PC, create a ROS catkin workspace, copy the `ChefBot` folder, and move it inside the `src` directory of the catkin workspace.

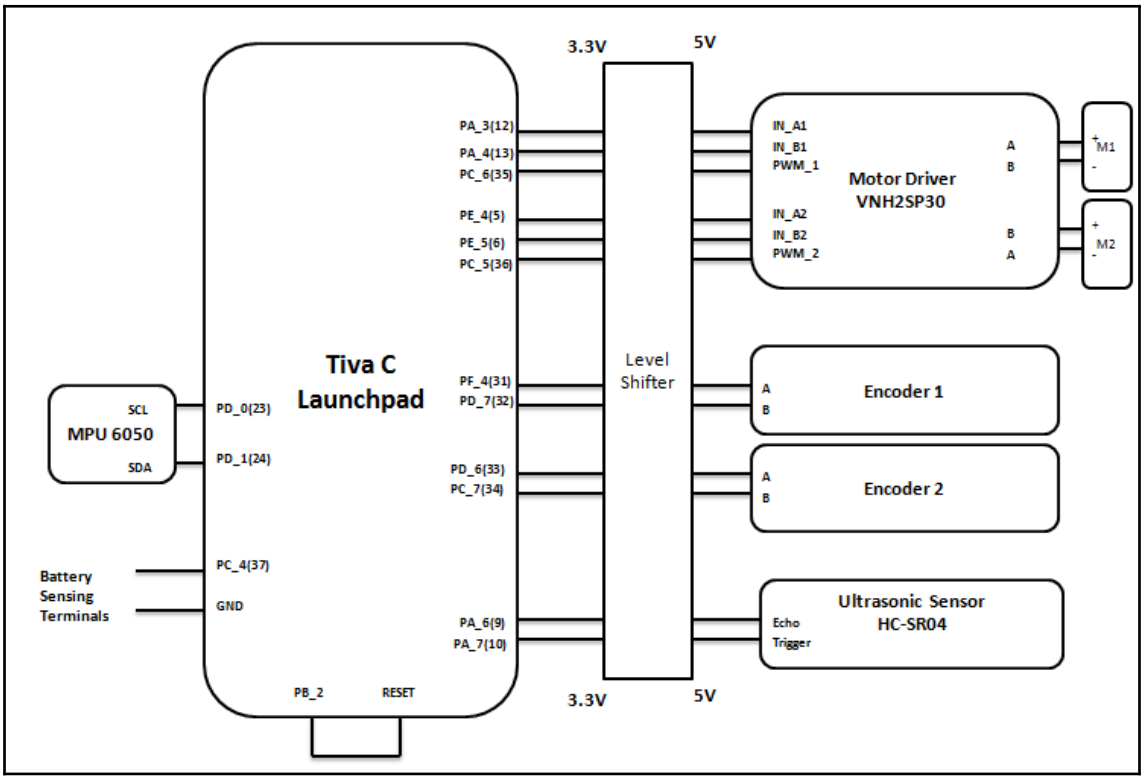
Build and install the source code of ChefBot by simply using the following command. This should be executed inside the `catkin` workspace we created:

```
$ catkin_make
```

If all dependencies are properly installed in the NUC, then the ChefBot packages will build and install in this system. After setting the ChefBot packages on the NUC PC, we can switch to the embedded code for ChefBot. Now, we can connect all the sensors in LaunchPad. After uploading the code in LaunchPad, we can again look at the ROS packages and how to run them. The cloned code from GitHub contains the Tiva-C LaunchPad code, which is going to be explained in the next section.

Interfacing ChefBot sensors to the Tiva-C LaunchPad

We have looked at the interfacing of the individual sensors that we are going to use in ChefBot. In this section, we will learn how to integrate sensors into the LaunchPad board. The Energia code to program Tiva-C LaunchPad is available in the cloned files at GitHub. The connection diagram showing the connection of the Tiva-C LaunchPad with the sensors is as follows. From this diagram, we learn how the sensors are interconnected with LaunchPad:



Sensor-interfacing diagram of ChefBot

M1 and M2 are two differential-drive motors that we are using in this robot. The kind of motor we are going to use here is a DC geared motor with an encoder from Pololu. The motor terminals are connected to the dual **VNH2SP30** motor driver from Pololu. One of the motors is connected with reverse polarity because in differential steering, one motor rotates in the opposite direction to the other. If we send the same control signal to both the motors, each motor will rotate in the opposite direction. To avoid this condition, we will swap the cables of one motor. The motor driver is connected to Tiva-C LaunchPad through a 3.3 V-5 V bidirectional level shifter. One of the level shifters we will use here is available at `https://www.sparkfun.com/products/12009`.

The two channels of each encoder are connected to LaunchPad using a level shifter. At the moment, we are using one ultrasonic distance sensor for obstacle detection. In future, we could increase the number of sensors if required. To get a good odometry estimate, we will put the IMU sensor MPU 6050 through an I2C interface. The pins are directly connected to LaunchPad because MPU6050 is 3.3 V compatible. To reset LaunchPad from the ROS nodes, we are allocating one pin as the output and connecting it to the reset pin of LaunchPad. When a specific character is sent to LaunchPad, it will set the output pin to high and reset the device. In some situations, the error from the calculation may accumulate and affect the navigation of the robot. We are resetting LaunchPad to clear this error. To monitor the battery level, we are allocating another pin to read the battery value. This feature is not currently implemented in the Energia code.

The code you downloaded from GitHub consists of the embedded code and the dependent libraries needed to compile this code. We can see the main section of the code here, and there is no need to explain all of the sections because we have already looked at them.

Embedded code for ChefBot

The main sections of the LaunchPad code are discussed in this section. The following are the header files used in the code:

```
//Library to communicate with I2C devices
#include "Wire.h"
//I2C communication library for MPU6050
#include "I2Cdev.h"
//MPU6050 interfacing library
#include "MPU6050_6Axis_MotionApps20.h"
//Processing incoming serial data
#include <Messenger.h>
//Contain definition of maximum limits of various data type
#include <limits.h>
```

The main libraries used in this code are for the purposes of communicating with MPU 6050 and processing the incoming serial data to LaunchPad. MPU 6050 can provide the orientation in quaternion or Euler values using the inbuilt **digital motion processor** (**DMP**). The functions to access DMP are written in `MPU6050_6Axis_MotionApps20.h`. This library has dependencies such as `I2Cdev.h` and `Wire.h`; that's why we are including this header as well. These two libraries are used for I2C communication. The `Messenger.h` library allows you to handle a stream of text data from any source and will help you to extract the data from it. The `limits.h` header contains the definitions of the maximum limits of various data types.

After we include the header files, we need to create an object to handle MPU6050 and process the incoming serial data using the `Messenger` class:

```
//Creating MPU6050 Object
MPU6050 accelgyro(0x68);
//Messenger object
Messenger Messenger_Handler = Messenger();
```

After declaring the messenger object, the main section deals with assigning pins for the motor driver, encoder, ultrasonic sensor, MPU 6050, reset, and battery pins. Once we have assigned the pins, we can look at the `setup()` function of the code. The definition of the `setup()` function is given in the following code:

```
//Setup serial, encoders, ultrasonic, MPU6050 and Reset functions
void setup()
{
  //Init Serial port with 115200 baud rate
  Serial.begin(115200);
  //Setup Encoders
  SetupEncoders();
  //Setup Motors
  SetupMotors();
  //Setup Ultrasonic
  SetupUltrasonic();
  //Setup MPU 6050
  Setup_MPU6050();
  //Setup Reset pins
  SetupReset();
  //Set up Messenger object handler
  Messenger_Handler.attach(OnMssageCompleted);
}
```

The preceding function contains a custom routine to configure and allocate pins for all of the sensors. This function will initialize serial communication with a 115,200 baud rate and set pins for the encoder, motor driver, ultrasonic sensors, and MPU6050. The `SetupReset()` function will assign a pin to reset the device, as shown in the preceding connection diagram. We have already seen the setup routines of each of the sensors in the previous chapters, so there is no need to explain the definition of each of these functions. The `Messenger` class handler is attached to a function called `OnMssageCompleted()`, which will be called when data is input to the `Messenger_Handler`.

The following is the main `loop()` function of the code. The main purpose of this function is to read and process serial data, as well as send available sensor values:

```
void loop()
{
    //Read from Serial port
    Read_From_Serial();
    //Send time information through serial port
    Update_Time();
    //Send encoders values through serial port
    Update_Encoders();
    //Send ultrasonic values through serial port
    Update_Ultra_Sonic();
    //Update motor speed values with corresponding speed received from PC
and send speed values through serial port
    Update_Motors();
    //Send MPU 6050 values through serial port
    Update_MPU6050();
    //Send battery values through serial port
    Update_Battery();
}
```

The `Read_From_Serial()` function will read serial data from the PC and feed data to the `Messenger_Handler` handler for processing purposes. The `Update_Time()` function will update the time after each operation in the embedded board. We can take this time value to be processed in the PC or use the PC's time instead.

We can compile the code in Energia's IDE and burn the code in LaunchPad. After uploading the code, we can look at the ROS nodes for handling the LaunchPad sensor values.

Writing a ROS Python driver for ChefBot

After uploading the embedded code to LaunchPad, the next step is to handle the serial data from LaunchPad and convert it to ROS topics for further processing. The `launchpad_node.py` ROS Python driver node interfaces Tiva-C LaunchPad with ROS. The `launchpad_node.py` file is in the `script` folder, which is inside the `ChefBot_bringup` package. The following is an explanation of the important code sections of `launchpad_node.py`:

```
#ROS Python client
import rospy
import sys
import time
import math

#This python module helps to receive values from serial port which execute
in a thread
from SerialDataGateway import SerialDataGateway
#Importing required ROS data types for the code
from std_msgs.msg import Int16,Int32, Int64, Float32,
 String, Header, UInt64
#Importing ROS data type for IMU
from sensor_msgs.msg import Imu
```

The `launchpad_node.py` file imports the preceding modules. The main module we can see is `SerialDataGateway`. This is a custom module written to receive serial data from the LaunchPad board in a thread. We also need some data types of ROS to handle the sensor data. The main function of the node is given in the following code snippet:

```
if __name__ =='__main__':
  rospy.init_node('launchpad_ros',anonymous=True)
  launchpad = Launchpad_Class()
  try:

    launchpad.Start()
    rospy.spin()
  except rospy.ROSInterruptException:
    rospy.logwarn("Error in main function")

  launchpad.Reset_Launchpad()
  launchpad.Stop()
```

The main class of this node is called `Launchpad_Class()`. This class contains all the methods to start, stop, and convert serial data to ROS topics. In the main function, we will create an object of the `Launchpad_Class()`. After creating the object, we will call the `Start()` method, which will start the serial communication between Tiva-C LaunchPad and the PC. If we interrupt the driver node by typing *Ctrl* + *C*, it will reset LaunchPad and stop the serial communication between the PC and LaunchPad.

The following code snippet is from the constructor function of `Launchpad_Class()`. In the following snippet, we will retrieve the port and baud rate of the LaunchPad board from the ROS parameters and initialize the `SerialDateGateway` object using these parameters. The `SerialDataGateway` object calls the `_HandleReceivedLine()` function inside this class when any incoming serial data arrives at the serial port. This function will process each line of serial data and extract, convert, and insert it in the appropriate headers of each ROS topic data type:

```
#Get serial port and baud rate of Tiva C Launchpad
port = rospy.get_param("~port", "/dev/ttyACM0")
baudRate = int(rospy.get_param("~baudRate", 115200))

################################################################
rospy.loginfo("Starting with serial port:
 " + port + ", baud rate: " + str(baudRate))#Initializing SerialDataGateway
object with serial port, baud
  rate and callback function to handle incoming serial
dataself._SerialDataGateway = SerialDataGateway(port,
 baudRate, self._HandleReceivedLine)
rospy.loginfo("Started serial communication")


#################################################################Subscrib
ers and Publishers

#Publisher for left and right wheel encoder values
self._Left_Encoder = rospy.Publisher('lwheel',Int64,queue_size
 = 10)self._Right_Encoder = rospy.Publisher('rwheel',Int64,queue_size
 = 10)
```

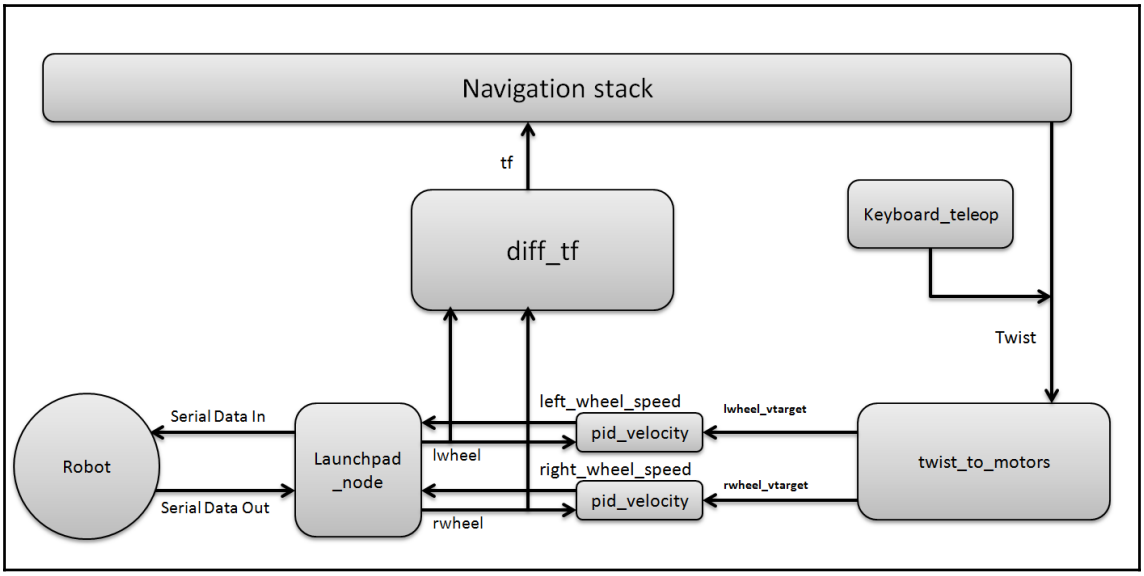
```
#Publisher for Battery level(for upgrade purpose)
self._Battery_Level =
 rospy.Publisher('battery_level',Float32,queue_size = 10)
#Publisher for Ultrasonic distance sensor
self._Ultrasonic_Value =
 rospy.Publisher('ultrasonic_distance',Float32,queue_size = 10)

#Publisher for IMU rotation quaternion values
self._qx_ = rospy.Publisher('qx',Float32,queue_size = 10)
self._qy_ = rospy.Publisher('qy',Float32,queue_size = 10)
self._qz_ = rospy.Publisher('qz',Float32,queue_size = 10)
self._qw_ = rospy.Publisher('qw',Float32,queue_size = 10)

#Publisher for entire serial data
self._SerialPublisher = rospy.Publisher('serial',
 String,queue_size=10)
```

We will create the ROS publisher object for sensors such as the encoder, IMU, and ultrasonic sensor, as well as for the entirety of the serial data for debugging purposes. We will also subscribe the speed commands to the left-hand side and right-hand side wheel of the robot. When a speed command arrives on the topic, it calls the respective callbacks to send speed commands to the robot's LaunchPad:

```
self._left_motor_speed =
rospy.Subscriber('left_wheel_speed',Float32,self._Update_Left_Speed)
self._right_motor_speed =
rospy.Subscriber('right_wheel_speed',Float32,self._Update_Right_Speed)
```

After setting the ChefBot driver node, we need to interface the robot with a ROS navigation stack in order to perform autonomous navigation. The basic requirement for doing autonomous navigation is that the robot driver nodes receive velocity commands from the ROS navigational stack. The robot can be controlled using teleoperation. In addition to these features, the robot must be able to compute its positional or odometry data and generate the tf data to be sent into the navigational stack. There must be a PID controller to control the robot's motor velocity. The following ROS package helps us to perform these functions. The `differential_drive` package contains nodes to perform the preceding operation. We are reusing these nodes in our package to implement these functionalities. You can find the `differential_drive` package in ROS at `http://wiki.ros.org/differential_drive`.

The following diagram shows how these nodes communicate with each other:



Block diagram of the robot showing the ROS nodes

The purpose of each node in the `ChefBot_bringup` package is as follows:

`twist_to_motors.py`: This node will convert a ROS `Twist` command or linear and angular velocity to an individual motor velocity target. The target velocities are published at a rate of the `~rate` (measured in Hertz) and the publish `timeout_ticks` time's velocity after the `Twist` message stops. The following are the topics and parameters that will be published and subscribed to by this node:

Publishing topics:

`lwheel_vtarget(std_msgs/Float32)`: This is the target velocity of the left wheel (measured in m/s).

`rwheel_vtarget` (`std_msgs/Float32`): This is the target velocity of the right wheel (measured in m/s).

Subscribing topics:

`Twist` (`geometry_msgs/Twist`): This is the target `Twist` command for the robot. The linear velocity in the x-direction and the angular velocity theta of the Twist messages are used in this robot.

Important ROS parameters:

`~base_width` (`float, default: 0.1`): This is the distance between the robot's two wheels in meters.

`~rate` (`int, default: 50`): This is the rate at which the velocity target is published (Hertz).

`~timeout_ticks` (`int, default:2`): This is the number of the velocity target message published after stopping the Twist messages.

`pid_velocity.py`: This is a simple PID controller to control the speed of each motor by taking feedback from the wheel encoders. In a differential drive system, we need one PID controller for each wheel. It will read the encoder data from each wheel and control the speed of each wheel.

Publishing topics:

`motor_cmd` (`Float32`): This is the final output of the PID controller that goes to the motor. We can change the range of the PID output using the `out_min` and `out_max` ROS parameter.

`wheel_vel` (`Float32`): This is the current velocity of the robot wheel in m/s.

Subscribing topics:

`wheel` (`Int16`): This topic is the output of a rotary encoder. There are individual topics for each encoder of the robot.

`wheel_vtarget` (`Float32`): This is the target velocity in m/s.

Important parameters:

`~Kp` (`float,default: 10`): This parameter is the proportional gain of the PID controller.

`~Ki` (`float, default: 10`): This parameter is the integral gain of the PID controller.

`~Kd` (`float, default: 0.001`): This parameter is the derivative gain of the PID controller.

`~out_min` (`float, default: 255`): This is the minimum limit of the velocity value to the motor. This parameter limits the velocity's value to the motor called the `wheel_vel` topic.

`~out_max` (`float, default: 255`): This is the maximum limit of the `wheel_vel` topic (measured in Hertz).

`~rate` (`float, default: 20`): This is the rate of publishing the `wheel_vel` topic.

`ticks_meter` (`float, default: 20`): This is the number of wheel encoder ticks per meter. This is a global parameter because it's used in other nodes too.

`vel_threshold` (`float, default: 0.001`): If the robot velocity drops below this parameter, we consider the wheel as stationary. If the velocity of the wheel is less than `vel_threshold`, we consider it as zero.

`encoder_min` (`int, default: 32768`): This is the minimum value of encoder reading.

`encoder_max` (`int, default: 32768`): This is the maximum value of encoder reading.

`wheel_low_wrap` (`int, default: 0.3 * (encoder_max - encoder_min) + encoder_min`): These values decide whether the odometry is in a negative or positive direction.

`wheel_high_wrap` (`int, default: 0.7 * (encoder_max - encoder_min) + encoder_min`): These values decide whether the odometry is in a negative or positive direction.

`diff_tf.py`: This node computes the transformation of odometry and broadcasts between the odometry frame and the robot's base frame.

Publishing topics:

`odom` (`nav_msgs/odometry`): This publishes the odometry (the current pose and twist of the robot).

`tf`: This provides the transformation between the odometry frame and the robot base link.

Subscribing topics:

`lwheel` (`std_msgs/Int16`), `rwheel` (`std_msgs/Int16`): These are the output values from the left and right encoders of the robot.

- `ChefBot_keyboard_teleop.py`: This node sends the `Twist` command using controls from the keyboard.

Publishing topics:

`cmd_vel_mux/input/teleop` (`geometry_msgs/Twist`): This publishes the Twist messages using keyboard commands.

Now that we have looked at the nodes in the `ChefBot_bringup` package, we will look at the functions of the launch files.

Understanding ChefBot ROS launch files

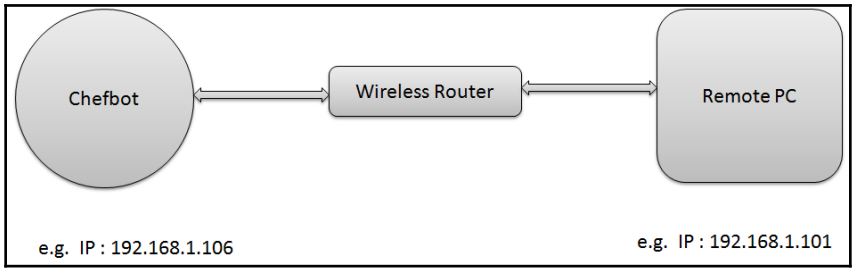
We will now look at the functions of each of the launch files of the `ChefBot_bringup` package:

- `robot_standalone.launch`: The main function of this launch file is to start nodes such as `launchpad_node`, `pid_velocity`, `diff_tf`, and `twist_to_motor` to get sensor values from the robot and to send the command velocity to the robot.
- `keyboard_teleop.launch`: This launch file will start teleoperation using the keyboard. It starts the `ChefBot_keyboard_teleop.py` node to perform the keyboard teleoperation.
- `3dsensor.launch` : This file will launch the Kinect OpenNI drivers and start publishing the RGB and depth stream. It will also start the depth-to-laser scanner node, which will convert point cloud data to laser scan data.
- `gmapping_demo.launch`: This launch file will start the SLAM gmapping nodes to map the area surrounding the robot.
- `amcl_demo.launch`: Using AMCL, the robot can localize and predict where it stands on the map. After localizing the robot on the map, we can command the robot to move to a position on the map. Then the robot can move autonomously from its current position to the goal position.
- `view_robot.launch`: This launch file displays the robot URDF model in RViz.
- `view_navigation.launch`: This launch file displays all the sensors necessary for the navigation of the robot.

Working with ChefBot Python nodes and launch files

We have already set ChefBot ROS packages in Intel's NUC PC and uploaded the embedded code to the LaunchPad board. The next step is to put the NUC PC on the robot, configure the remote connection from the laptop to the robot, test each node, and work with ChefBot's launch files to perform autonomous navigation.

The main device we should have before working with ChefBot is a good wireless router. The robot and the remote laptop have to connect across the same network. If the robot PC and remote laptop are on the same network, the user can connect from the remote laptop to the robot PC through SSH using its IP. Before putting the robot PC in the robot, we should connect the robot PC to the wireless network so that once it's connected to the wireless network, it will remember the connection details. When the robot powers up, the PC should automatically connect to the wireless network. Once the robot PC is connected to a wireless network, we can put it in the actual robot. The following diagram shows the connection diagram of the robot and remote PC:



Wireless connection diagram of the robot and remote PC

The preceding diagram assumes that the ChefBot's IP is `192.168.1.106` and the remote PC's IP is `192.168.1.101`.

We can remotely access the ChefBot terminal using SSH. We can use the following command to log in to ChefBot, where `robot` is the username of the ChefBot PC:

```
$ ssh robot@192.168.1.106
```

When you log in to the ChefBot PC, it will ask for the robot PC password. After entering the password of the robot PC, we can access the robot PC terminal. After logging in to the robot PC, we can start testing ChefBot's ROS nodes and test whether we receive the serial values from the LaunchPad board inside ChefBot. Note that you should log in to the ChefBot PC again through SSH if you are using a new terminal.

If the `ChefBot_bringup` package is properly installed on the PC, and if the LaunchPad board is connected, then before running the ROS driver node, we can run the `miniterm.py` tool to check whether the serial values come to the PC properly via USB. We can find the serial device name using the `dmesg` command. We can run `miniterm.py` using the following command:

```
$ miniterm.py /dev/ttyACM0 115200
```

If it shows the permission denied message, set the permission of the USB device by writing rules on the `udev` folder, which we did in `Chapter 6`, *Interfacing Actuators and Sensors to the Robot Controller*, or we can temporarily change the permission using the following command. Here, we are assuming that `ttyACM0` is the device name of LaunchPad. If the device name is different in your PC, then you have to use that name instead of `ttyACM0`:

```
$ sudo chmod 777 /dev/ttyACM0
```

If everything works fine, we will get values such as those shown in the following screenshot:

```
b       0.00
t       66458239        0.05
e       0       0
u       10
s       0.00    0.00
i       -0.68   -0.47   -0.40   0.40
b       0.00
t       66511681        0.05
e       0       0
u       10
s       0.00    0.00
i       -0.68   -0.47   -0.40   0.40
b       0.00
t       66566051        0.05
e       0       0
u       10
s       0.00    0.00
i       -0.68   -0.47   -0.40   0.40
b       0.00
t       66620423        0.05
e       0       0
u       10
s       0.00    0.00
```

Output of miniterm.py

The letter `b` is used to indicate the battery reading of the robot; currently, it's not implemented. The value is set to zero now. These values are coming from the Tiva C Launchpad. There are different approaches to sense the voltage using a microcontroller board. One of the approaches is given below (`http://www.instructables.com/id/Arduino-Battery-Voltage-Indicator/`). The letter `t` indicates the total time elapsed (in microseconds) after the robot starts running the embedded code. The second value is the time taken to complete one entire operation in LaunchPad (measured in seconds). We can use this value if we are performing real-time calculations of the parameters of the robot. At the moment, we are not using this value, but we may use it in the future. The letter `e` indicates the values of the left and right encoder respectively. Both the values are zero here because the robot is not moving. The letter `u` indicates the values from the ultrasonic distance sensor. The distance value we get is in centimeters. The letter `s` indicates the current wheel speed of the robot. This value is used for inspection purposes. Actually, speed is a control output from the PC itself.

To convert this serial data to ROS topics, we have to run the drive node called `launchpad_node.py`. The following code shows how to execute this node.

First, we have to run `roscore` before starting any nodes:

```
$ roscore
```

Run `launchpad_node.py` using the following command:

```
$ rosrun ChefBot_bringup launchpad_node.py
```

If everything works fine, we will get the following output in node in the running terminal:

```
robot@robot-desktop:~$ rosrun chefbot_bringup launchpad_node.py
Initializing Launchpad class
[INFO] [WallTime: 1424097603.219564] Starting with serial port: /dev/ttyACM0, bau
d rate: 115200
[INFO] [WallTime: 1424097603.220825] Started serial communication
```

Output of launchpad_node.py

After running `launchpad_node.py`, we will see the following topics generated, as shown in the following screenshot:

```
robot@robot-desktop:~$ rostopic list
/battery_level
/imu/data
/left_wheel_speed
/lwheel
/qw
/qx
/qy
/qz
/right_wheel_speed
/rosout
/rosout_agg
/rwheel
/serial
/ultrasonic distance
```

Topics generated by launchpad_node.py

We can view the serial data received by the driver node by subscribing to the `/serial` topic. We can use it for debugging purposes. If the serial topic shows the same data that we saw in `miniterm.py`, then we can confirm that the nodes are working fine. The following screenshot is the output of the `/serial` topic:

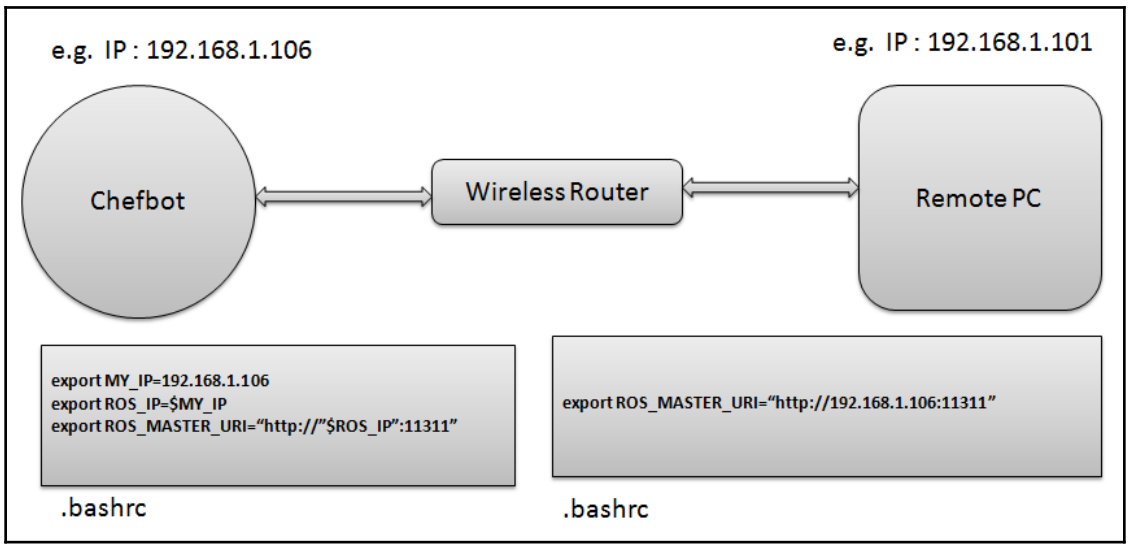
```
---
data: 16266, in:  e     1       -1
---
data: 16267, in:  u     10
---
data: 16268, in:  s     0.00    0.00
---
```

Output of the /serial topic published by the LaunchPad node

After setting the `ChefBot_bringup` package, we can start working with the autonomous navigation of ChefBot. Currently, we are accessing only the ChefBot PC's terminal. To visualize the robot's model, sensor data, maps, and so on, we have to use RViz in the user's PC. We have to do some configuration in the robot and user PC to perform this operation. It should be noted that the user's PC should have the same software setup as the ChefBot PC.

The first thing we have to do is to set the ChefBot PC as a ROS master. We can set the ChefBot PC as the ROS master by setting the `ROS_MASTER_URI` value. The `ROS_MASTER_URI` setting is a required setting; it informs the nodes about the **uniform resource identifier** (**URI**) of the ROS master. When you set the same `ROS_MASTER_URI` for the ChefBot PC and the remote PC, we can access the topics of the ChefBot PC in the remote PC. So, if we run RViz locally, then it will visualize the topics generated in the ChefBot PC.

Assume that the ChefBot PC IP is `192.168.1.106` and the remote PC IP is `192.168.1.10`. You can set a static IP for Chefbot PC and remote PC so that the IP will always be the same all test otherwise if it is automatic, you may get different IP in each test. To set `ROS_MASTER_URI` in each system, the following command should be included in the `.bashrc` file in the `home` folder. The following diagram shows the setup needed to include the `.bashrc` file in each system:



Network configuration for ChefBot

Add these lines at the bottom of `.bashrc` on each PC and change the IP address according to your network.

After we establish these settings, we can just start `roscore` on the ChefBot PC terminal and execute the `rostopic list` command on the remote PC.

If you see any topics, you are done with the settings. We can first run the robot using the keyboard teleoperation to check the robot's functioning and confirm whether we get the sensor values.

We can start the robot driver and other nodes using the following command. Note that this should execute in the ChefBot terminal after logging in using SSH:

```
$ roslaunch ChefBot_bringup robot_standalone.launch
```

After launching the robot driver and nodes, start the keyboard teleoperation using the following command. This also has to be done on the new terminal of the ChefBot PC:

```
$ roslaunch ChefBot_bringup keyboard_teleop.launch
```

To activate Kinect, execute the following command. This command is also executed on the ChefBot terminal:

```
$roslaunch ChefBot_bringup 3dsensor_kinect.launch
```

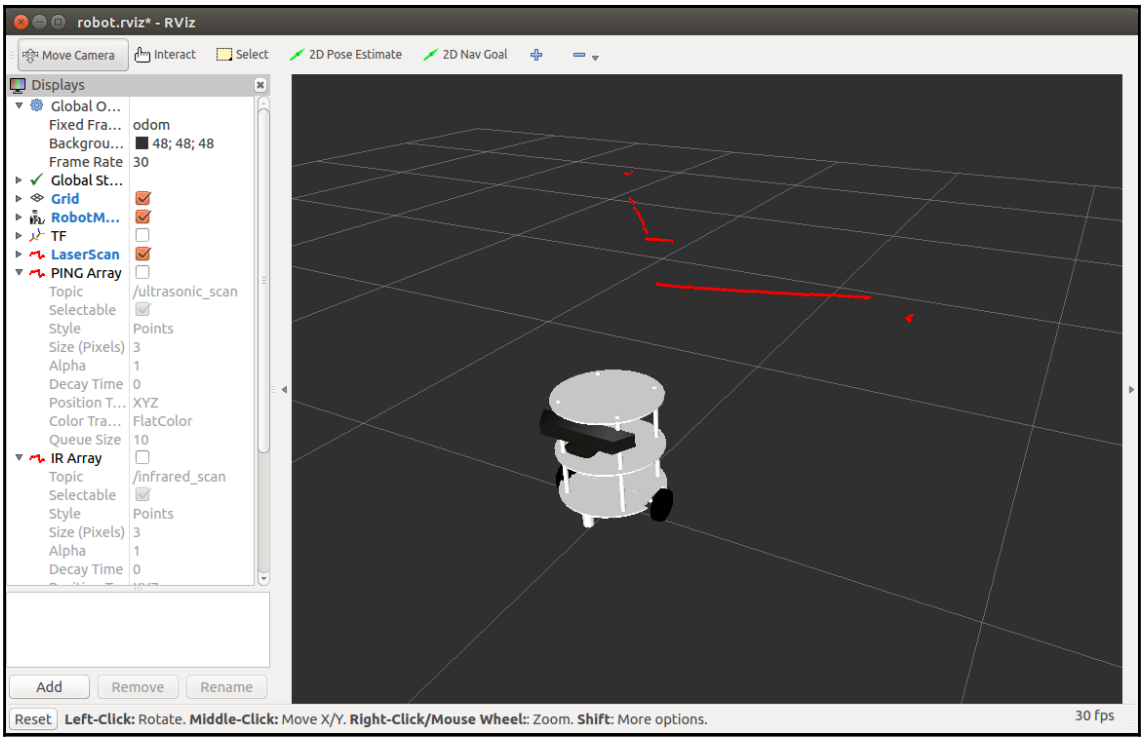
If you are using Orbecc Astra, use the following launch file to start the sensor:

```
$ roslaunch ChefBot_bringup 3d_sensor_astra.launch
```

To view the sensor data, we can execute the following command. This will view the robot model in RViz and should be executed in the remote PC. If we set up the `ChefBot_bringup` package in the remote PC, we can access the following command and visualize the robot model and sensor data from the ChefBot PC:

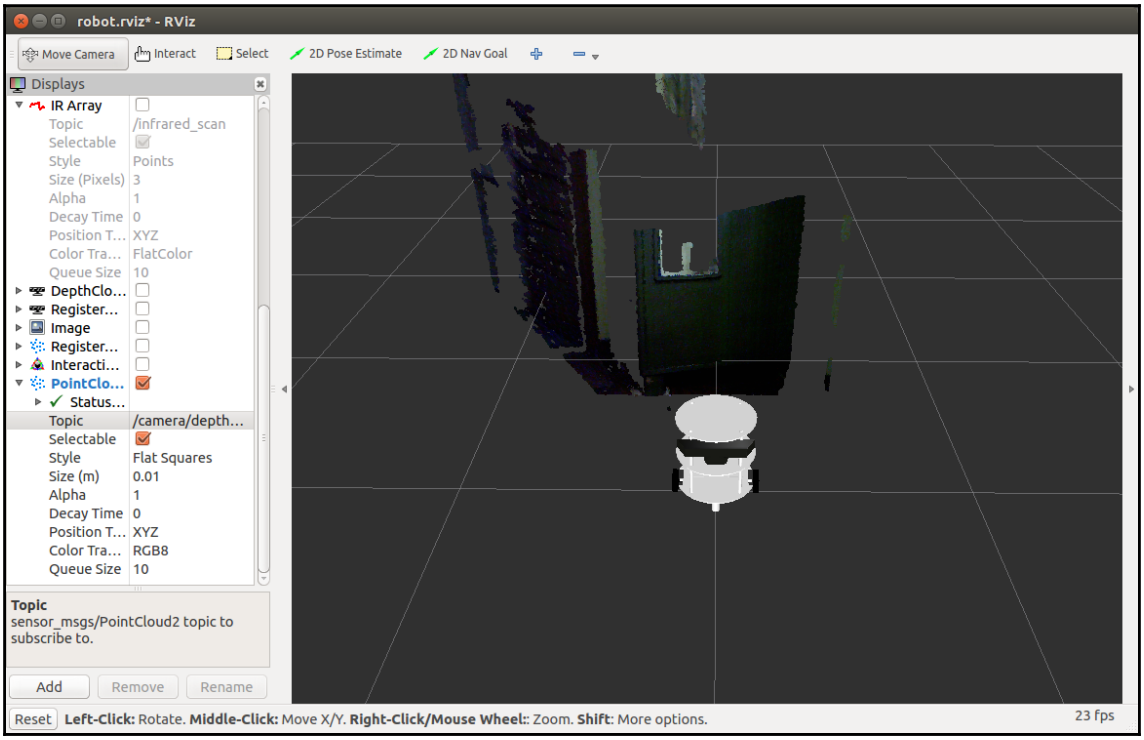
```
$ roslaunch ChefBot_bringup view_robot.launch
```

The following screenshot is the output of RViz. We can see the **LaserScan** and **PointCloud** mapped data in the screenshots:



ChefBot LaserScan data in RViz

The preceding screenshot shows **LaserScan** in RViz. We need to tick the **LaserScan** topic from the left-hand side section of RViz to show the laser scan data. The laser scan data is marked on the viewport. If you want to watch the point cloud data from Kinect/Astra, click on the **Add** button on the left-hand side of RViz and select **PointCloud2** from the pop-up window. Select **Topic** | `/camera/depth_registered` from the list and you will see an image similar to the one shown in the following screenshot:



ChefBot with PointCloud data

After working with sensors, we can perform SLAM to map the room. The following procedure helps us to start SLAM on this robot.

Working with SLAM on ROS to build a map of the room

To perform gmapping, we have to execute the following commands.

The following command starts the robot driver in the ChefBot terminal:

```
$ roslaunch ChefBot_bringup robot_standalone.launch
```

The following command starts the gmapping process. Note that it should be executed on the ChefBot terminal:

```
$ roslaunch ChefBot_bringup gmapping_demo.launch
```

Gmapping will only work if the odometry value that is received is proper. If the odometry value is received from the robot, we will receive the following message for the preceding command. If we get this message, we can confirm that gmapping will work fine:

```
[ INFO] [1422618733.585407153]: Created local_planner dwa_local_planner/DWAPlanner
ROS
[ INFO] [1422618733.604762090]: Sim period is set to 0.20
[ INFO] [1422618735.208493249]: odom received!
```

ChefBot with PointCloud data

To start the keyboard teleoperation, use the following command:

```
$ roslaunch ChefBot_bringup keyboard_teleop.launch
```

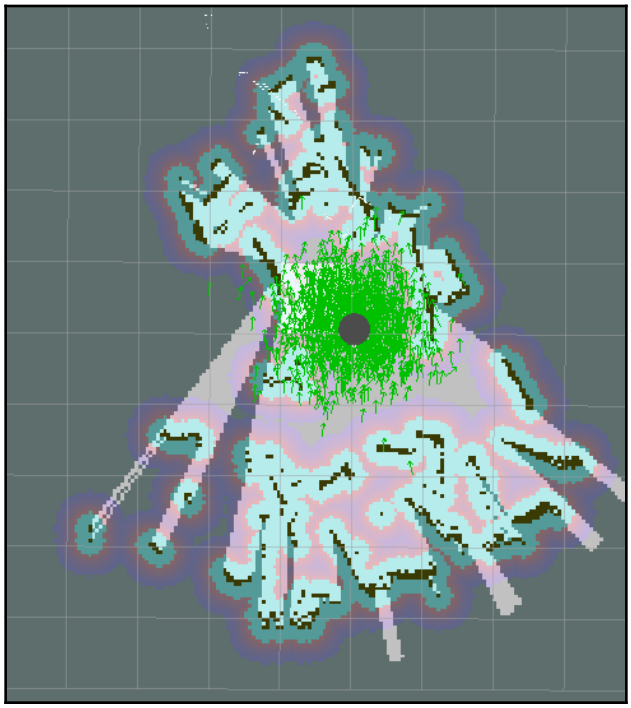
To view the map that is being created, we need to start RViz on the remote system using the following command:

```
$ roslaunch ChefBot_bringup view_navigation.launch
```

After viewing the robot in RViz, you can move the robot using the keyboard and see the map being created. When it has mapped the entire area, we can save the map using the following command on the ChefBot PC terminal:

```
$rosrun map_server map_saver -f ~/test_map
```

In the preceding code, `test_map` is the name of the map being stored in the `home` folder. The following screenshot shows the map of a room created by the robot:



Mapping a room

After the map is stored, we can work with the localization and autonomous navigation functionalities using ROS.

Working with ROS localization and navigation

After building the map, close all the applications and rerun the robot driver using the following command:

```
$ roslaunch ChefBot_bringup robot_standalone.launch
```

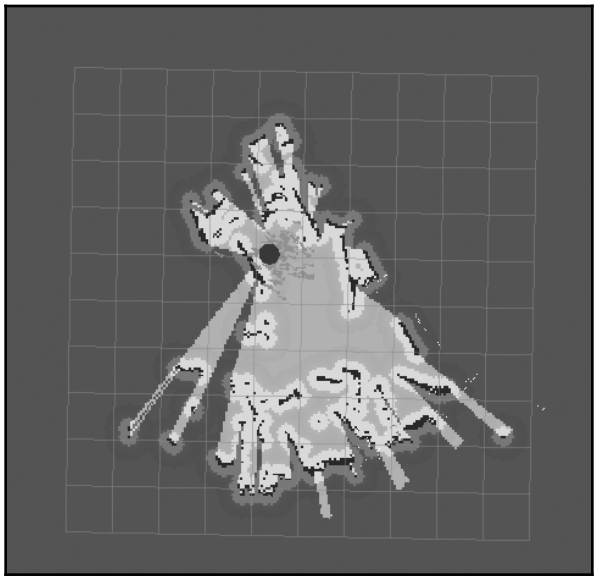
Start the localization and navigation on the stored map using the following command:

```
$ roslaunch ChefBot_bringup amcl_demo.launch map_file:=~/test_map.yaml
```

Start viewing the robot using the following command in the remote PC:

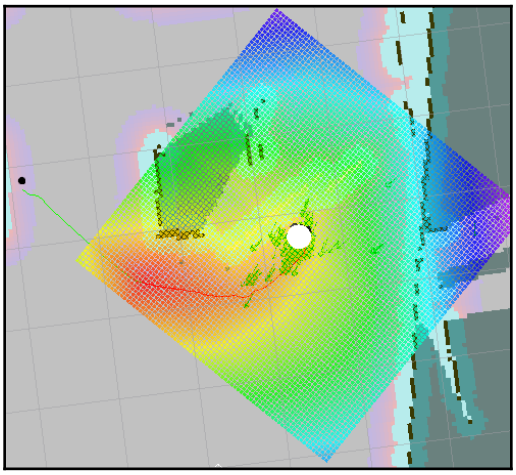
```
$ roslaunch ChefBot_bringup view_navigation.launch
```

In RViz, we may need to specify the initial pose of the robot using the **2D Pose Estimate** button. We can change the robot pose on the map using this button. If the robot is able to access the map, then we can use the **2D Nav Goal** button to command the robot to move to the desired position. When we start the localization, we can see the particle cloud around the robot by using the AMCL algorithm:



Localizing the robot using AMCL

The following is a screenshot of the robot as it navigates autonomously from its current position to the goal position. The goal position is marked as a black dot:



Autonomous navigation using a map

The black line from the robot to the black dot is the robot's planned path to reach the goal position. If the robot is not able to locate the map, we might need to fine-tune the parameter files in the `ChefBot_bringupparam` folder. For more fine-tuning details, you can go through the AMCL package on ROS at `http://wiki.ros.org/amcl`.

Summary

This chapter was about assembling the hardware of ChefBot and integrating the embedded and ROS code into the robot to perform autonomous navigation. We saw the robot hardware parts that were manufactured using the design from `Chapter 6`, *Interfacing Actuators and Sensors to the Robot Controller*. We assembled the individual sections of the robot and connected the prototype PCB we designed for the robot. This consisted of the LaunchPad board, motor driver, left shifter, ultrasonic sensor, and IMU. The LaunchPad board was flashed with the new embedded code, which can interface with all sensors in the robot and can send or receive data from the PC.

After looking at the embedded code, we configured the ROS Python driver node to interface with the serial data from the LaunchPad board. After interfacing with the LaunchPad board, we computed the odometry data and differential drive control using nodes from the `differential_drive` package that was in the ROS repository. We interfaced the robot with the ROS navigation stack. This enabled us to use SLAM and AMCL for autonomous navigation. We also looked at SLAM and AMCL, created a map, and commanded the robot to navigate autonomously.

Questions

1. What is the use of the robot ROS driver node?
2. What is the role of the PID controller in navigation?
3. How do you convert encoder data to odometry data?
4. What is the role of SLAM in robot navigation?
5. What is the role of AMCL in robot navigation?

Further reading

You can read more about the robotic vision package in ROS from the following links:

- `http://wiki.ros.org/gmapping`
- `http://wiki.ros.org/amcl`

9
Designing a GUI for a Robot Using Qt and Python

In the last chapter, we discussed the integration of robotic hardware components and software packages for performing autonomous navigation. After the integration, the next step is to build a GUI to control the robot. We are building a GUI that can act as a trigger for the underlying ROS commands. Instead of running all the commands on the Terminal, the user can work with the GUI buttons. The GUI we are going to design is for a typical hotel room with nine tables. The user can set a table position in the map of the hotel room and command the robot to go to a particular table to deliver food. After delivering the food, the user can command the robot to go to its home position.

The following topics will be covered in the chapter:

- Installing Qt on Ubuntu
- Introduction to PyQt and PySide
- Introduction to Qt Designer
- Qt signals and slots
- Converting a Qt UI file to a Python file
- Working with the ChefBot GUI application
- Introduction to rqt and its features

Technical requirements

To test the application and code in this chapter, you need an Ubuntu 16.04 LTS PC/laptop with ROS Kinetic installed.

You need to know Qt, PyQt, and rqt installed.

Two of the most popular GUI frameworks currently available are Qt (`http://qt.digia.com`) and GTK+ (`http://www.gtk.org/`).Qt and GTK+ are open source, cross-platform user interface toolkits and development platforms. These two software frameworks are widely used in Linux desktop environments, such as GNOME and KDE.

In this chapter, we will be using Python binding of the Qt framework to implement the GUI because Python binding of Qt is easier to develop compared to other methods. We will look at how to develop a GUI from scratch and program it using Python. After discussing basic Python and Qt programming, we will discuss ROS interfaces of Qt and Python, which are already available in ROS. We will first look at what the Qt UI framework is and how to install it on our PC.

Installing Qt on Ubuntu 16.04 LTS

Qt is a cross-platform application framework that is widely used to develop application software with a GUI interface as well as command line tools. Qt is available in almost all operating systems, such as Windows, macOS X, Android, and so on. The main programming language used for developing Qt applications is C++ but there are bindings available for languages such as Python, Ruby, Java, and so on. Let's take a look at how to install Qt SDK on Ubuntu 16.04. We will install Qt from the **Advance Packaging Tool** (**APT**) in Ubuntu. The APT already comes with Ubuntu installation. So, for installing Qt/Qt SDK, we can simply use the following command, which will install Qt SDK and its required dependencies from the Ubuntu package repository. We can install Qt version 4 using the following command:

```
$ sudo apt-get install qt-sdk
```

This command will install the entire Qt SDK and its libraries required for our project. The packages available on Ubuntu repositories may not be the latest versions. To get the latest version of Qt, we can download the online or offline installer of Qt for various OS platforms from the following link:

`http://qt-project.org/downloads`

After installing Qt on our system, we will see how we can develop GUI using Qt and interface with Python.

Working with Python bindings of Qt

Let's see how we can interface Python and Qt. In general, there are two modules available in Python for connecting to the Qt user interface. The two most popular frameworks are:

- PyQt
- PySide

PyQt

PyQt is one of the most popular Python bindings for Qt cross-platform. PyQt is developed and maintained by Riverbank Computing Limited. It provides binding for Qt version 4 and Qt version 5 and comes with GPL (version 2 or 3) along with a commercial license. PyQt is available for Qt version 4 and 5, called PyQt4 and PyQt5, respectively. These two modules are compatible with Python versions 2 and 3. PyQt contains more than 620 classes that cover the user interface, XML, network communication, web, and so on.

PyQt is available in Windows, Linux, and macOS X. It is a prerequisite to install Qt SDK and Python in order to install PyQt. The binaries for Windows and macOS X are available at the following link:

`http://www.riverbankcomputing.com/software/pyqt/download`

We will see how to install PyQt4 on Ubuntu 16.04 using Python 2.7.

Installing PyQt in Ubuntu 16.04 LTS

If you want to install PyQt on Ubuntu/Linux, use the following command. This command will install the PyQt library, its dependencies, and some Qt tools:

```
$ sudo apt-get install python-qt4 pyqt4-dev-tools
```

PySide

PySide is an open source software project that provides Python binding for the Qt framework. The PySide project was initiated by Nokia and offers a full set of Qt binding for multiple platforms. The technique used in PySide to wrap the Qt library is different from PyQt, but the API of both is similar. PySide is currently not supported on Qt 5. PySide is available for Windows, Linux, and macOS X. The following link will guide you to set up PySide on Windows and macOS X:

`http://qt-project.org/wiki/Category:LanguageBindings::PySide::Downloads`

The prerequisites of PySide are the same as PyQt. Let's see how we can install PySide on Ubuntu 16.04 LTS.

Installing PySide on Ubuntu 16.04 LTS

The PySide package is available on the Ubuntu package repository. The following command will install the PySide module and Qt tools on Ubuntu:

```
$ sudo apt-get install python-pyside pyside-tools
```

Let's work with both modules and see the differences between both.

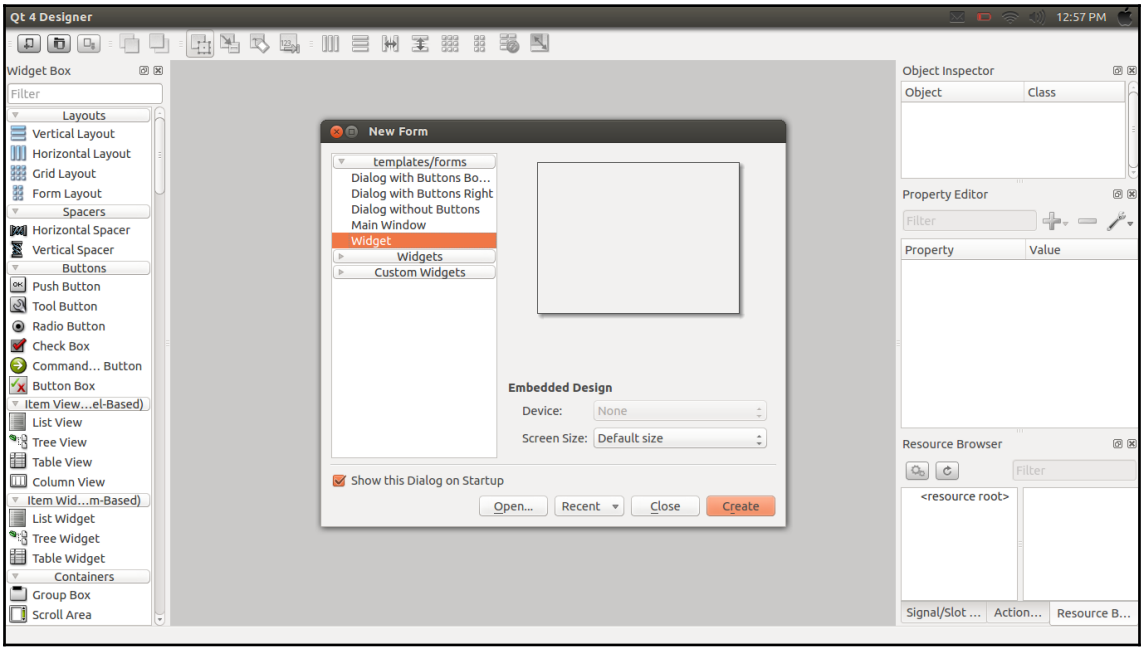
Working with PyQt and PySide

After installing the PyQt and PySide packages, we will look at how to write an **Hello World** GUI using PyQt and PySide. The main difference between PyQt and PySide is only in some commands; most of the steps are the same. Let's see how to make a Qt GUI and convert it into Python code.

Introducing Qt Designer

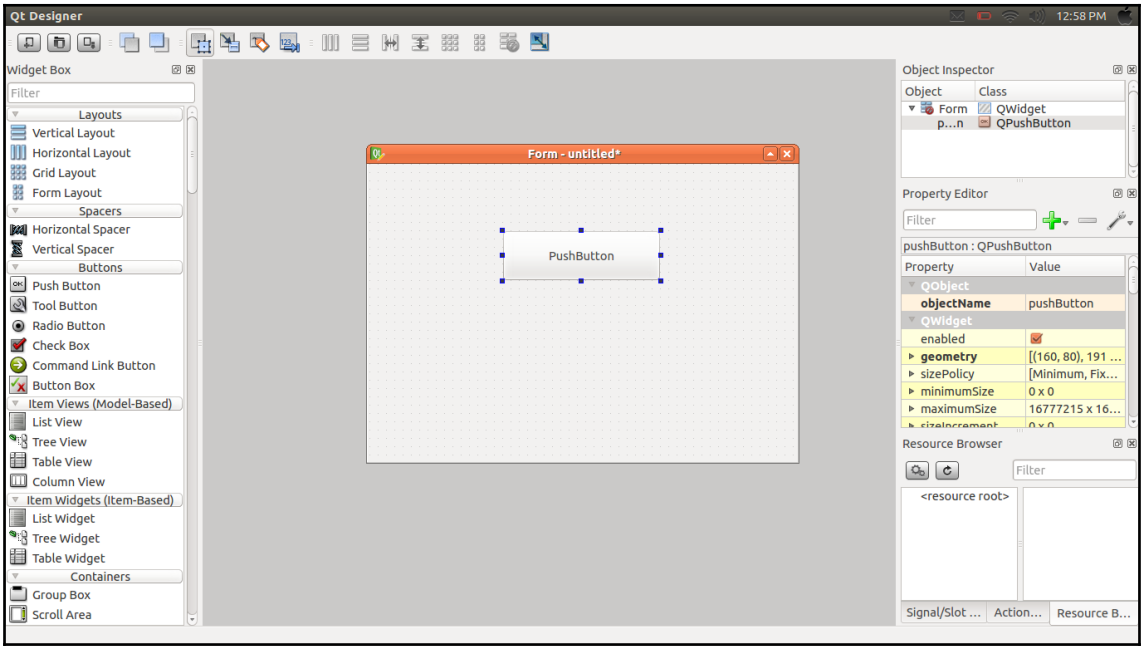
Qt Designer is the tool for designing and inserting control into the Qt GUI. The Qt GUI is basically an XML file that contains the information of its components and controls. The first step to work with the GUI relates to its design. The Qt Designer tool provides various options to make excellent GUIs.

Start Qt Designer by entering the `designer-qt4` command on the Terminal. The following screenshot shows what you will be able to see after running this command:



Qt 4 Designer

The preceding screenshot shows the Qt Designer interface. Select the **Widget** option from the **New Form** window and click on the **Create** button. This will create an empty widget; we can drag various GUI controls from the left-hand side of Qt 4 Designer to the empty widget. Qt widgets are the basic building blocks of the Qt GUI. The following screenshot shows a form with a **PushButton** dragged from the left-hand side window of Qt Designer:



Qt Designer widget form

The **Hello World** application that we are going to build will have a **PushButton**. When we click on the **PushButton**, an **Hello World** message will be printed on the Terminal. Before building the **Hello World** application, we need to understand what Qt signals and slots are, because we have to use these features for building the **Hello World** application.

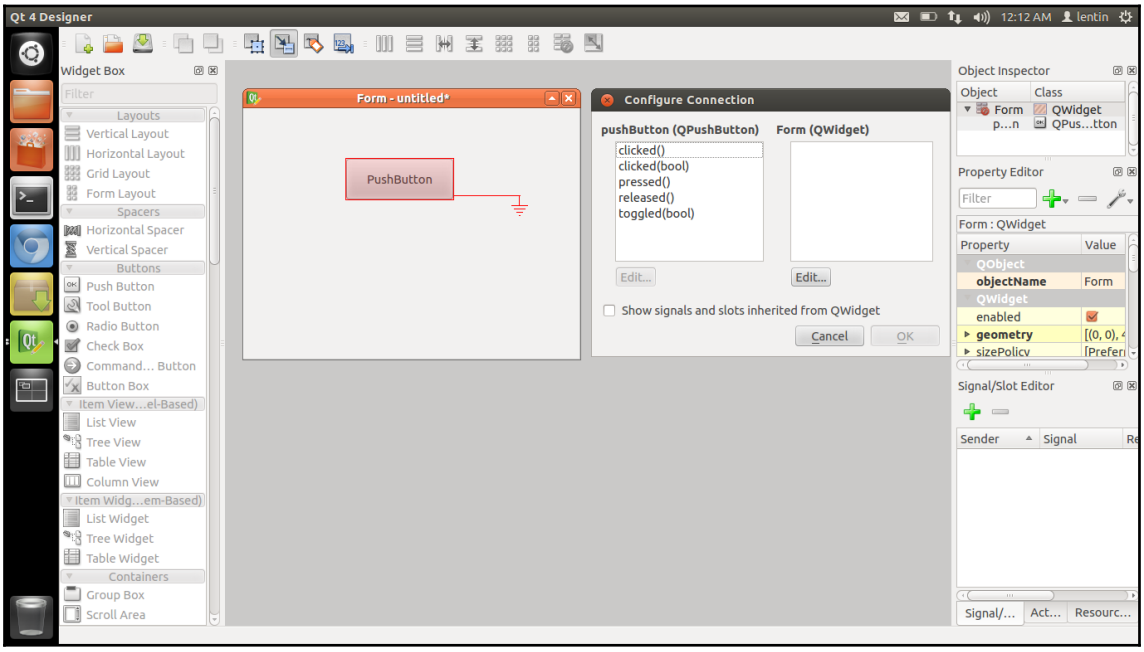
Qt signals and slots

In Qt, GUI events are handled using the signals and slots features. A signal is emitted from the GUI when an event occurs. Qt widgets have many predefined signals, and users can add custom signals for GUI events. A slot is a function that is called in response to a particular signal. In this example, we are using the `clicked()` signal of **PushButton** and creating a custom slot for this signal.

We can write our own code inside this custom function. Let's see how we can create a button, connect a signal to a slot, and convert the entire GUI to Python. Here are the steps involved in creating the **Hello World** GUI application:

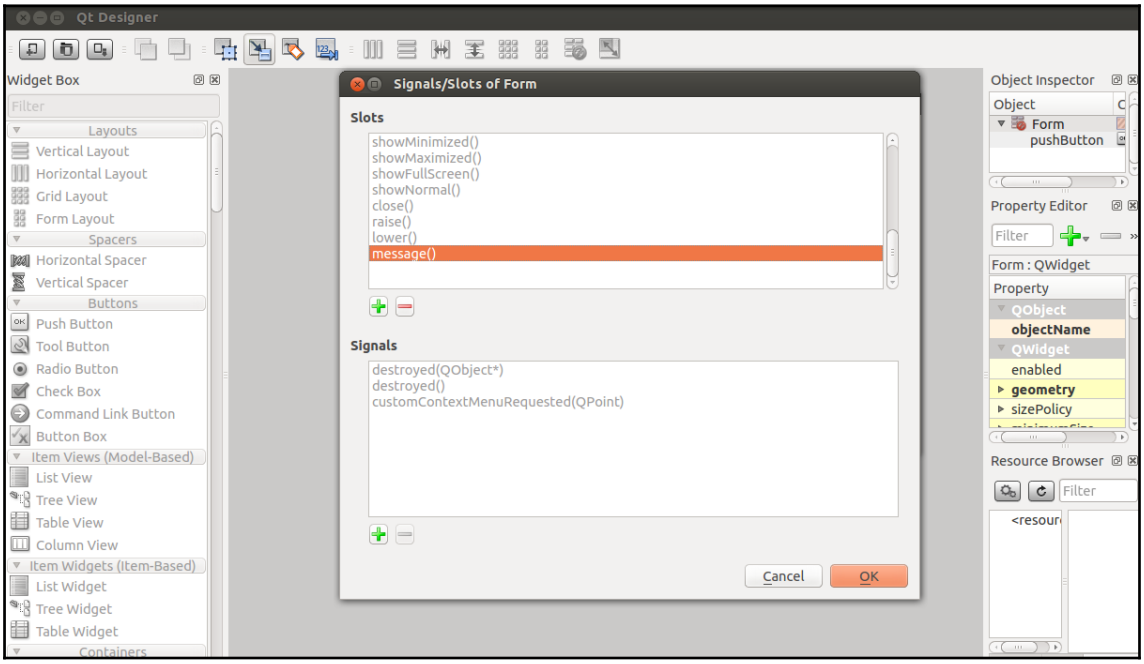
1. Drag and create a **PushButton** from Qt Designer to the empty **Form**.
2. Assign a slot for the button clicked event, which emits a signal called `clicked()`.
3. Save the designed UI file in the `.ui` extension.
4. Convert UI files to Python.
5. Write the definition of the custom slot.
6. Print the **Hello World** message inside the defined slot/function.

We have already dragged a button from Qt Designer to an empty **Form**. Press the F4 key to insert a slot on the button. When we press F4, the **PushButton** turns red, and we can drag a line from the button and place the ground symbol in the main window. This is shown in the following screenshot:



Assigning slots and signals in Qt 4 Designer

Select the **clicked()** signal from the left-hand side and click on the **Edit...** button to create a new custom slot. When we click on the **Edit...** button, another window will pop up to create a custom function. You can create a custom function by clicking on the **+** symbol. We created a custom slot called **message()**, as shown in the following screenshot:



Assigning slots and signals in Qt 4 Designer

Click on the **OK** button, save the UI file as `hello_world.ui`, and quit Qt Designer. After saving the UI file, let's see how we can convert a Qt UI file into a Python file.

Read more about Qt Signals and slots from the following link

https://doc.qt.io/qt-5/signalsandslots.html

Converting a UI file into Python code

After designing the UI file, we can convert the UI file into its equivalent Python code. The conversion is done using a `pyuic` compiler. We have already installed this tool while installing PyQt/PySide. The following are the commands to convert a Qt UI file into a Python file.

We have to use different commands for PyQt and PySide. The following command is to convert the UI into its PyQt equivalent file:

```
$ pyuic4 -x hello_world.ui -o hello_world.py
```

The `pyuic4` is a UI compiler to convert a UI file into its equivalent Python code. We need to mention the UI filename after the `-x` argument and mention the output filename after the `-o` argument.

There are not many changes to the PySide command. Instead of `pyuic4`, PySide uses `pyside-uic` to convert UI files into Python files. The remaining arguments are the same:

```
$ pyside-uic -x hello_world.ui -o hello_world.py
```

The preceding command will generate an equivalent Python code for the UI file. This will create a Python class that has the GUI components. The generated script will not have the definition of the custom function `message()`. We should add this custom function to generate the code. The following procedure will guide you to add the custom function; so when you click on the button, the custom function `message()` will be executed.

Adding a slot definition to PyQt code

The generated Python code from PyQt is given here. The code generated by `pyuic4` and `pyside-uic` are the same, except in importing module names. All other parts are the same. The explanation of the code generated using PyQt is also applicable to PySide code. The code generated from the preceding conversion is as follows. The code structure and parameters can change according to the UI file that you have designed:

```
from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s

class Ui_Form(object):

    def setupUi(self, Form):
        Form.setObjectName(_fromUtf8("Form"))
        Form.resize(514, 355)

        self.pushButton = QtGui.QPushButton(Form)
        self.pushButton.setGeometry(QtCore.QRect(150, 80, 191, 61))
        self.pushButton.setObjectName(_fromUtf8("pushButton"))
```

```
        self.retranslateUi(Form)
        QtCore.QObject.connect(self.pushButton,
QtCore.SIGNAL(_fromUtf8("clicked()")), Form.message)
        QtCore.QMetaObject.connectSlotsByName(Form)



    def retranslateUi(self, Form):
        Form.setWindowTitle(QtGui.QApplication.translate("Form", "Form",
None, QtGui.QApplication.UnicodeUTF8))
       self.pushButton.setText( QtGui.QApplication.translate("Form",
"Press", None, QtGui.QApplication.UnicodeUTF8))

#This following code should be added manually
if __name__ == "__main__":
   import sys
    app = QtGui.QApplication(sys.argv)
    Form = QtGui.QWidget()
    ui = Ui_Form()
    ui.setupUi(Form)
    Form.show()
    sys.exit(app.exec_())
```

The preceding code is the equivalent Python script of the Qt UI file that we designed in the Qt Designer application. Here is the step-by-step procedure of the working of this code:

1. The code will start executing from `if __name__ == "__main__":`. The first thing in the PyQt code is to create a `QApplication` object. A `QApplication` class manages the GUI application's control flow and main settings. The `QApplication` class contains the main event loop, where all events from the Windows system and other sources are processed and dispatched. It also handles initialization and finalization of an application. The `QApplication` class is inside the `QtGui` module. This code creates an object of `QApplication` called `app`. We have to add the main code manually.
2. The `Form = QtGui.QWidget()` line creates an object called `Form` of the `QWidget` class that is present inside the `QtGui` module. The `QWidget` class is the base class of all user interface objects of Qt. It can receive the mouse and keyboard event from the main Windows system.

3. The `ui = Ui_Form()` line creates an object called `ui` of the `Ui_Form()` class defined in the code. The `Ui_Form()` object can accept the `QWidget` class that we created in the previous line and it can add buttons, text, button control, and other UI components into this `QWidget` object. The `Ui_Form()` class contains two functions: `setupUi()` and `retranslateUi()`. We can pass the `QWidget` object to the function called `setupUi()`. This function will add UI components on this widget object, such as buttons, assigning slots for signals, and so on. The `retranslateUi()` function will translate the language of the UI to other languages if needed. For example, if we need translation from English to Spanish, we can mention the corresponding Spanish word in this function.
4. The `Form.show()` line displays the final window with buttons and text.

The next thing is to create the slot function, which prints the **Hello World** message. The slot definition is created inside the `Ui_Form()` class. The following steps insert the slot called `message()` into the `Ui_Form()` class.

The `message()` function definition is as follows:

```
def message(self):
print "Hello World"
```

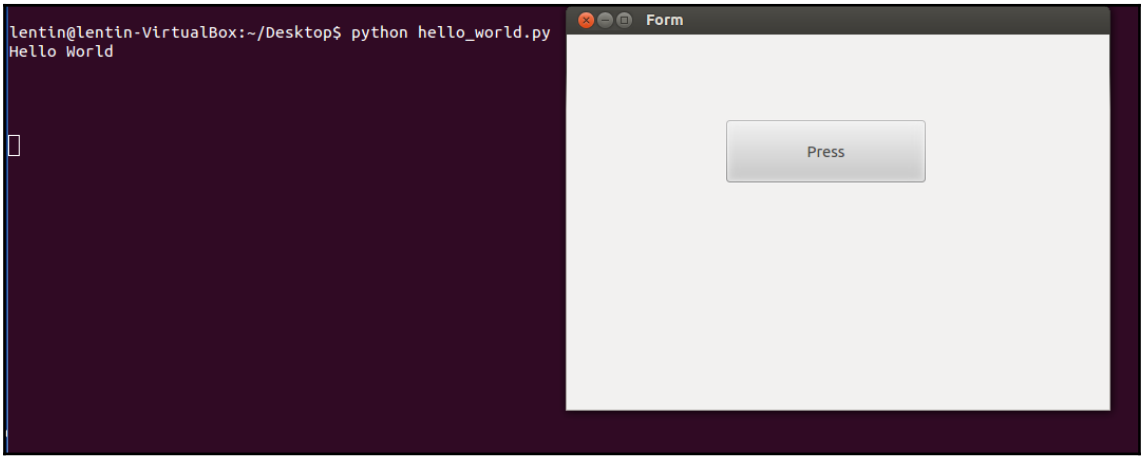
This should be inserted as a function inside the `Ui_Form()` class. Also, change the following line in the `setupUi()` function inside the `Ui_Form()` class:

```
QtCore.QObject.connect(self.pushButton,
QtCore.SIGNAL(_fromUtf8("clicked()")), Form.message)
```

The `Form.message` parameter should be replaced with the `self.message` parameter. The preceding line connects the **PushBbutton** signal, `clicked()`, to the `self.message()` slot that we already inserted in the `Ui_Form()` class.

Operation of the Hello World GUI application

After replacing the `Form.message` parameter with the `self.message` parameter, we can execute the code and the output will look like this:



Running Pyqt4 application

When we click on the **Press** the button, it will print the **Hello world** message. This is all about setting a custom GUI with Python and Qt.

In the next section, we will see the actual GUI that we are designing for the robot.

Working with ChefBot's control GUI

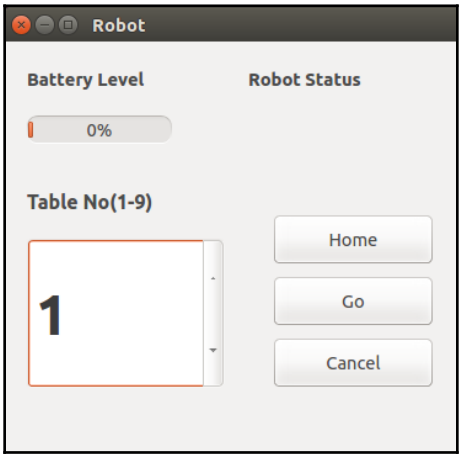
After completing the **Hello World** application in PyQt, we will now discuss a GUI for controlling ChefBot. The main use of building a GUI is to create an easier way to control the robot. For example, if the robot is deployed in a hotel to serve food, the person who controls this robot need not have knowledge about the complex commands to start and stop this robot; so, building a GUI for ChefBot can reduce the complexity and make it easier for the user. We are planning to build a GUI using PyQt, ROS, and the Python interface. The ChefBot ROS package is available on GitHub at the following link:
`https://github.com/qboticslabs/learning_robotics_2nd_ed`

If you haven't cloned the code yet, you can do so now using following command:

```
$ git clone https://github.com/qboticslabs/learning_robotics_2nd_ed.git
```

The GUI code named `robot_gui.py` is placed in the `scripts` folder, which is inside the `chefbot_bringup` package.

The following screenshot shows the GUI that we have designed for ChefBot:



Running Pyqt4 application

The GUI has the following features:

- It can monitor the robot battery status and robot status. The robot status indicates the working status of the robot. For example, if the robot encounters an error, it will indicate the error on this GUI.
- It can command the robot to move into a table position for delivering food. There is a spin box widget on the GUI to input the table position. Currently, we are planning this GUI for a room with nine tables, but we may expand it to any number according to the requirement. After inputting the table number, we can command the robot to go to that table by clicking on the **Go** button; the robot will get into that position. If we want to return the robot to the initial position, we can click on the **Home** button. If we want to cancel the current robot movement, click on **Cancel** to stop the robot. The working of this GUI application is as follows:

When we have to deploy ChefBot in a hotel, the first procedure that we have to do is to create a map of the room. After mapping the entire room properly, we have to save the map on the robot PC. The robot does the mapping only once. After mapping, we can run the localization and navigation routines and command the robot to get into a position on the map. The ChefBot ROS package comes with a map and simulation model of a hotel-like environment. We can now run this simulation and localization for testing the GUI and in the next chapter, we will discuss how to control the hardware using the GUI. If you install the ChefBot ROS packages on your local system, we can simulate a hotel environment and test the GUI.

Start the ChefBot simulation in a hotel-like arrangement using the following command:

```
$roslaunch chefbot_gazebo chefbot_hotel_world.launch
```

After starting the ChefBot simulation, we can run the localization and navigation routines using an already built map. The map is placed in the `chefbot_bringup` package. We can see a `map` folder inside this package. Here, we will use this map for performing this test. We can load the localization and navigation routine using the following command:

```
    $ roslaunch chefbot_gazebo amcl_demo.launch
map_file:=/home/<user_name>/catkin_ws/src/chefbot/chefbot_bringup/map/hotel
1.yaml
```

The path of the map file can change in different systems, so use the path in your system instead of this path.

If the path mentioned is correct, it will start running the ROS navigation stack. If we want to see the robot position on the map or manually set the initial position of the robot, we can use RViz using the following command:

```
$ roslaunch chefbot_bringup view_navigation.launch
```

In RViz, we can command the robot to go to any map coordinates using the **2D Nav Goal** button.

We can command the robot to go to any map coordinates using programming too. The ROS navigation stack works using the ROS `actionlib` library. The ROS `actionlib` library is for performing preemptable tasks; it is similar to ROS services. An advantage over ROS services is that we can cancel the request if we don't want it at that time.

In the GUI, we can command the robot to go to a map coordinate using the Python `actionlib` library. We can get the table position on the map using the following technique.

After starting the simulator and AMCL nodes, launch the keyboard teleoperation and move the robot near each table. Use the following command to get the translation and rotation of the robot:

```
$ rosrun tf tf_echo /map /base_link
```

When we click on the **Go** button, the position is fed to the navigation stack and the robot plans its path and reaches its goal. We can even cancel the task at any time. So, the ChefBot GUI acts as an `actionlib` client, which sends map coordinates to the `actionlib` server; that is, the navigation stack.

We can now run the robot GUI to control the robot using the following command:

```
$ rosrun chefbot_bringup robot_gui.py
```

We can select a table number and click on the **Go** button for moving the robot to each table.

Assuming that you cloned the files and got the `robot_gui.py` file, we will discuss the main slots we added into the `Ui_Form()` class for the `actionlib` client and to get values of the battery and robot status.

We need to import the following Python modules for this GUI application:

```
import rospy
import actionlib
from move_base_msgs.msg import *
import time
from PyQt4 import QtCore, QtGui
```

The additional modules we require are the ROS Python client `rospy`, and the `actionlib` module to send values to the navigation stack. The `move_base_msgs` module contains the message definition of the goal that needs to be sent to the navigation stack.

The robot position near each table is mentioned in a Python dictionary. The following code shows hardcode values of the robot's position near each table:

```
table_position = dict()
table_position[0] = (-0.465, 0.37, 0.010, 0, 0, 0.998, 0.069)
table_position[1] = (0.599, 1.03, 0.010, 0, 0, 1.00, -0.020)
table_position[2] = (4.415, 0.645, 0.010, 0, 0, -0.034, 0.999)
table_position[3] = (7.409, 0.812, 0.010, 0, 0, -0.119, 0.993)
table_position[4] = (1.757, 4.377, 0.010, 0, 0, -0.040, 0.999)
table_position[5] = (1.757, 4.377, 0.010, 0, 0, -0.040, 0.999)
```

```
table_position[6] = (1.757, 4.377, 0.010, 0, 0, -0.040, 0.999)
table_position[7] = (1.757, 4.377, 0.010, 0, 0, -0.040, 0.999)
table_position[8] = (1.757, 4.377, 0.010, 0, 0, -0.040, 0.999)
table_position[9] = (1.757, 4.377, 0.010, 0, 0, -0.040, 0.999)
```

We can access the position of the robot near each table by accessing this dictionary.

Currently, we have inserted only four values for the purpose of demonstration. You can add more values by finding the position of other tables.

We are assigning some variables to handle the table number, position of the robot, and the `actionlib` client inside the `Ui_Form()` class:

```
#Handle table number from spin box
self.table_no = 0
#Stores current table robot position
self.current_table_position = 0
#Creating Actionlib client
self.client = actionlib.SimpleActionClient('move_base',MoveBaseAction)
#Creating goal message definition
self.goal = MoveBaseGoal()
#Start this function for updating battery and robot status
self.update_values()
```

The following code shows the signals and slots assignment in this code for buttons and spin box widgets:

```
#Handle spinbox signal and assign to slot set_table_number()
QtCore.QObject.connect(self.spinBox,
QtCore.SIGNAL(_fromUtf8("valueChanged(int)")), self.set_table_number)

#Handle Home button signal and assign to slot Home()
QtCore.QObject.connect(self.pushButton_3,
QtCore.SIGNAL(_fromUtf8("clicked()")), self.Home)

#Handle Go button signal and assign to slot Go()
QtCore.QObject.connect(self.pushButton,
QtCore.SIGNAL(_fromUtf8("clicked()")), self.Go)

#Handle Cancel button signal and assign to slot Cancel()
QtCore.QObject.connect(self.pushButton_2,
QtCore.SIGNAL(_fromUtf8("clicked()")), self.Cancel)
```

The following slot handles the spin box value from the UI and assigns a table number. Also, it converts the table number to the corresponding robot position:

```
def set_table_number(self):
  self.table_no = self.spinBox.value()
  self.current_table_position = table_position[self.table_no]
```

Here is the definition of the **Go** slot for the **Go** button. This function will insert the robot position of the selected table in a goal message header and send it into the navigation stack:

```
def Go(self):

  #Assigning x,y,z pose and orientation to target_pose message
  self.goal.target_pose.pose.position.x=float(self.current_table
_position[0])

  self.goal.target_pose.pose.position.y=float(self.current_table
_position[1])
  self.goal.target_pose.pose.position.z=float(self.current_table
_position[2])

  self.goal.target_pose.pose.orientation.x =
float(self.current_table_position[3])
  self.goal.target_pose.pose.orientation.y=
float(self.current_table_position[4])
  self.goal.target_pose.pose.orientation.z=
float(self.current_table_position[5])

  #Frame id
  self.goal.target_pose.header.frame_id= 'map'

  #Time stamp
  self.goal.target_pose.header.stamp = rospy.Time.now()

  #Sending goal to navigation stack
  self.client.send_goal(self.goal)
```

The following code is the `Cancel()` slot definition. This will cancel all the robot paths that it was planning to perform at that time:

```
def Cancel(self):
  self.client.cancel_all_goals()
```

The following code is the definition of `Home()`. This will set the table position to zero, and call the `Go()` function. The table at position zero is the home position of the robot:

```
def Home(self):
  self.current_table_position = table_position[0]
  self.Go()
```

The following definitions are for the `update_values()` and `add()` functions. The `update_values()` method will start updating the battery level and robot status in a thread. The `add()` function will retrieve the ROS parameters of the battery status and robot status, and set them to the progress bar and label, respectively:

```
def update_values(self):
    self.thread = WorkThread()
    QtCore.QObject.connect( self.thread,
QtCore.SIGNAL("update(QString)"), self.add )
    self.thread.start()
def add(self,text):
  battery_value = rospy.get_param("battery_value")
  robot_status = rospy.get_param("robot_status")
   self.progressBar.setProperty("value", battery_value)
     self.label_4.setText(_fromUtf8(robot_status))
```

The `WorkThread()` class used in the preceding function is given here. The `WorkThread()` class is inherited from `QThread` provided by Qt for threading. The thread simply emits the signal `update(Qstring)` with a particular delay. In the preceding function, `update_values()`, the `update(QString)` signal is connected to the `self.add()` slot; so when a signal `update(QString)` is emitted from the thread, it will call the `add()` slot and update the battery and status value:

```
class WorkThread(QtCore.QThread):
  def __init__(self):
    QtCore.QThread.__init__(self)
   def __del__(self):
    self.wait()
   def run(self):
    while True:
      time.sleep(0.3) # artificial time delay
      self.emit( QtCore.SIGNAL('update(QString)'), " " )
      return
```

We have discussed how to make a GUI for ChefBot, but this GUI is only for the user who controls ChefBot. If someone wants to debug and inspect the robot data, we may have to go for other tools. ROS provides an excellent debugging tool to visualize data from the robot.

The rqt tool is a popular ROS tool. It is based on a Qt-based framework for GUI development for ROS. Let's discuss the rqt tool, installation procedure, and how we can inspect the sensor data from the robot.

Installing and working with rqt in Ubuntu 16.04 LTS

rqt is a software framework in ROS, which implements various GUI tools in the form of plugins. We can add plugins as dockable windows in rqt.

Installing rqt in Ubuntu 16.04 can be done using the following command. Before installing rqt, ensure that you have the full installation of ROS Indigo.

```
$ sudo apt-get install ros-<ros_version>-rqt
```

After installing the rqt packages, we can access the GUI implementation of rqt, called `rqt_gui`, in which we can dock `rqt plugins` in a single window.

Let's start using `rqt_gui`.

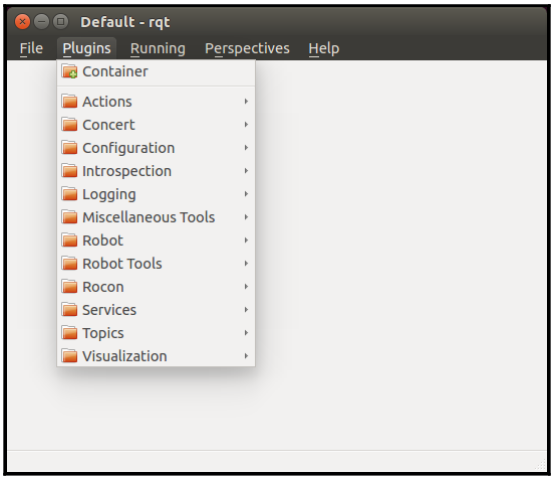
Run the `roscore` command before running `rqt_gui`:

```
$ roscore
```

Run the following command to start `rqt_gui`:

```
$ rosrun rqt_gui rqt_gui
```

We will get the following window if the commands work fine:



Running rqt

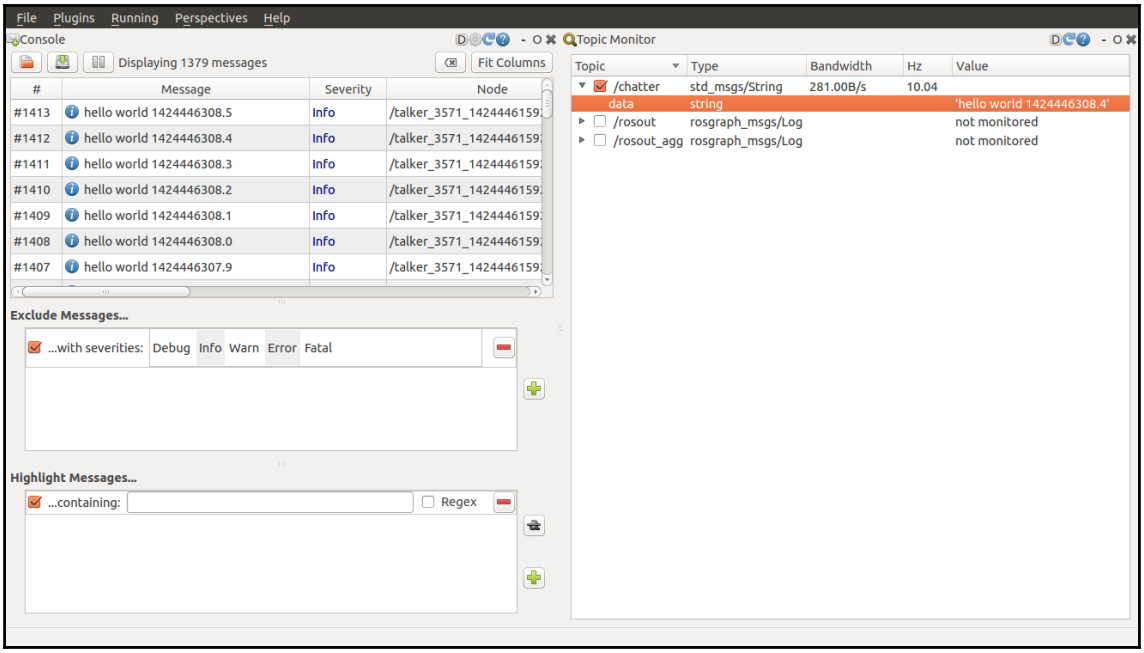
We can load and unload plugins at runtime. To analyze the ROS message log, we can load the **Console** plugin from **Plugins** | **Logging** | **Console**. In the following example, we load the **Console** plugin and run a talker node inside `rospy_tutorials`, which will send an **Hello World** message to a topic called `/chatter`.

Run the following command to start the node `talker.py`:

```
$rosrun rospy_tutorials talker.py
```

In the following screenshot, `rqt_gui` is loaded with two plugins named **Console** and **Topic Monitor**. The **Topic Monitor** plugin can be loaded from **Plugins** | **Topics** | **Topic Monitor**. The **Console** plugin monitors the messages printing on each node and their severity. It is very useful for debugging purposes. In the following screenshot, the left section of `rqt_gui` is loaded with the **Console** plugin and the right side is loaded with the **Topic Monitor**. The **Topic Monitor** will list the topics available and will monitor its values.

In the following screenshot, the **Console** plugin monitors the `talker.py` node's messages and their severity level and the **Topic Monitor** monitors the value inside the `/chatter` topic:



Running rqt with different plugins

We can also visualize data such as images and plot graphs on `rqt_gui`. For the robot's navigation and its inspection, there are plugins for embedding RViz on `rqt_gui`. The **Navigation viewer** plugin views from the `/map` topic. The visualization plugins are available in **Plugin** | **Visualization**.

We can also create the GUI using rqt. The instructions to create rqt plugins that can load in to rqt_gui can be found at:

`http://wiki.ros.org/rqt/Tutorials/Create%20your%20new%20rqt%20plugin`

Summary

In this chapter, we discussed creating a GUI for ChefBot that can be used by an ordinary user who doesn't have any idea about the internal workings of a robot. We used Python binding of Qt called PyQt to create this GUI. Before we looked at the main GUI design, we looked at an **Hello World** application to get an easier understanding of PyQt. The UI design was done using the Qt Designer tool and the UI file was converted into its equivalent Python script using the Python UI compiler. After designing the main GUI in Qt Designer, we converted the UI file into Python script and inserted the necessary slots in the generated script. The ChefBot GUI can start the robot, select a table number, and command the robot to get into that position. The position of each table comes from the generated map where we hardcoded the positions in this Python script for testing. When a table is selected, we set a goal position on the map, and when we click on the **Go** button, the robot will move into the intended position. The user can cancel the operation at any time and command the robot to come to the home position. The GUI can also receive the real-time status of the robot and its battery status. After discussing the robot GUI, we looked at the debugging GUI tool in ROS, called rqt. We saw some plugins used for debugging the data from the robot. In the next chapter, we will see the complete testing and calibration of the robot.

Questions

1. What are the popular UI toolkits available on the Linux platform?
2. What are the differences between PyQt and PySide Qt bindings?
3. How do you convert a Qt UI file into Python script?
4. What are Qt signals and slots?
5. What is rqt and what are its main applications?

Further reading

Read more about robotic vision packages in ROS at the following links:

- `http://wiki.ros.org/rqt/UserGuide`
- `http://wiki.ros.org/rqt/Tutorials`

Assessments

Chapter 1, Getting Started with the Robot Operating System

1. Here are the three main features of ROS:
    - Message passing interface to communicate with different programs
    - Off-the-shelf robotics algorithm to make the robot prototyping faster
    - Software tools to visualize robot data and debugging
2. The different levels of concepts in ROS are the ROS Filesystem level, ROS Computation Graph Level, and ROS Community Level.
3. The Catkin build system is built using CMake and Python scripts. This tool helps us build the ROS packages.
4. The ROS topic is a named bus in which one node can communicate to another node. The kind of message type used in the topics are ROS messages.
5. The different concepts of the ROS computation graph are ROS Nodes, ROS Topics, ROS Messages, ROS Master, ROS Services, and ROS Bags.
6. The ROS Master act as a mediator program to connect two ROS nodes to start communicating with each other.
7. The important features of Gazebo are:
    - Dynamic simulation: It includes physics engine like ODE, Bullet, Simbody and Dart
    - Advanced 3D graphics: It uses OGRE framework to create high-quality lighting, shadows, and textures
    - Plugin support: This will allow developers to add new robot, sensors, and environmental control
    - TCP/IP Transport: Controlling Gazebo using socket-based message passing interface

Chapter 2, Understanding the Basics of Differential Robots

1. Holonomic robots can freely move in any direction and the controllable degrees of freedom is equal to the total degrees of freedom. Omni wheel-based robots are an example of holonomic robots. Nonholonomic robots have constraints on its motion, so controllable degrees of freedom will not be equal to the total degrees of freedom. Differential driver configuration is an example of nonholonomic configuration.
2. Robot kinematics deals with the motion of the robot without considering the mass and inertia, whereas robot dynamics is the relationship between mass and inertia properties, motion, and associated torques.
3. ICC stands for Instantaneous Center of Curvature, which is an imaginary point on the robot wheel axis around which the robot is rotated.
4. It is the process of finding the robot's current position from the wheel velocity.
5. Finding the wheel velocity to reach a goal position.

Chapter 3, Modeling the Differential Drive Robot

1. Robot modeling is the process of creating the 2D and 3D representation of robot having all the parameters of the robot, which includes kinematic and dynamic parameters of the robot.
2. The 2D model mainly includes the exact dimension of robot parts, which helps us compute the kinematics of the robot as well as helps manufacture robot parts.
3. The 3D model of the robot is an exact replica of robot hardware having all parameters of the physical robot designed using a CAD software. This is used for creating robot simulation and 3D printing parts of the robot.
4. Creating a 3D model using Python scripting is much easier and accurate than manual modeling if you know the Blender scripting APIs.
5. URDF is the 3D robot model representation of robot in ROS. It is having kinematic and dynamic parameters of the robot.

Chapter 4, Simulating a Differential Drive Robot Using ROS

1. Sensor modeling in Gazebo can be done using Gazebo plugins. The sensor model can be written using C++, which can be plugged in to the Gazebo simulator.
2. ROS is interfaced to Gazebo using Gazebo ROS plugin. When we load this plugin into Gazebo, we can able to control Gazebo through ROS interface.
3. The important tags are `<inertia>`, `<collision>`, and `<gazebo>`.
4. The Gmapping package in ROS is an implementation of Fast SLAM algorithm, which can be used in robot to map the environment and localizing on it. Using Gmapping in ROS is a straightforward process, including the gmapping node with necessary parameters and topics such as odometry and laser scan.
5. The `Move_base` node has a provision to handle various navigation subsystem in a robot. It is having a provision to handle global and local planner, also the map of the robot. Once the node receives the goal position, which feed to the navigation subsystem in order to reach to that goal position.
6. AMCL stands for Adaptive Monte Carlo Localization, which is an algorithm to localize a robot on a given map. There is a ROS package in ROS for deploying AMCL in our robot. We can launch the amcl node with proper input and necessary parameters.

Chapter 5, Designing ChefBot Hardware and Circuits

1. It is the process for finding proper robot hardware components for the robot that is meeting the robot desired specification. It also involves circuit designing and computing the current flow of each components in order to ensure the stability of the robot components.
2. It's a switching circuit to control the direction and speed of an electric motor.
3. The main components are wheel encoder to compute wheel velocity and laser range finder or depth sensor to detect the obstacle around the robot.
4. We need to check whether it meets the specification of the robot.
5. Mapping, obstacle detection, object detection, and tracking.

Chapter 6, Interfacing Actuators and Sensors to the Robot Controller

1. Switching circuit in order to control the speed of motors in a robot.
2. A sensor that can detect the speed and direction of wheel rotation.
3. In the 4X encoding scheme, we are extracting the maximum transition between the encoder pulses in order to get more counts from a single rotation.
4. Using encoder count and distance per count, we can easily compute the displacement of the wheel.
5. It is a smart actuator having a motor and a microcontroller that can be directly interfaced to a PC and used to customize different settings of the actuator. It can be connected as daisy chain manner, which is appropriate for robotic arm.
6. It is the sensor for finding range and has one transmitter and one receiver. The transmitter transmits ultrasonic sound, and the receiver receives it. The delay between these process is used for distance measurement.
7. Range = high level time of echo pin output * velocity (340 M/S) / 2.
8. It is sending IR pulses and receive by an IR receiver. According to the distance, the voltage in the IR receiver changes, and we can compute the distance using the following equation:

    Range = (6787 / (Volt - 3)) – 4

Chapter 7, Interfacing Vision Sensors with ROS

1. Most of the 3D depth sensors have additional vision sensors to detect the depth. It may be using IR projection method or using stereo vision.
2. The message passing interface, tools to visualize and debug robots, off-the-shelf robot algorithms.
3. OpenCV is mainly having computer vision algorithm, OpenNI is having algorithm implementation for implementing NI applications, and PCL is having algorithm to process point cloud data.

4. It stands for Simultaneous Localization and Mapping. It is an algorithm commonly used to map the robot environment and localize on it at the same time.
5. It is an algorithm to map the robot environment in 3D.

Chapter 8, Building ChefBot Hardware and Integration of Software

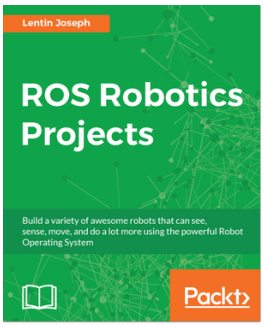
1. It is a mediator program between robot low-level controller and high-level controller such as PC. It converts the low-level data into ROS equivalent data.
2. PID is a control loop feedback mechanism to reach a robot goal position by taking feedback of robot position.
3. Using encoder data, we can compute the distance traversed by the robot using robot kinematic equations. Those values are odometry data.
4. It is mainly using for mapping the environment.
5. It is mainly using for localizing the robot in a static map.

Chapter 9, Designing a GUI for a Robot Using Qt and Python

1. Qt and GTK.
2. Both bindings are almost the same, only difference is in the name. The PyQt license is GPL and PySide comes with LGPL. Also, PySide is having much documentation about its APIs.
3. We can use Py UI compiler named pyuic.
4. Qt slots are functions in a program that can be triggered by Qt signal. For example, clicked is a signal that can invoke a function named `hello()`.
5. Rqt is one of the useful GUI tool in ROS. We can create rqt plugins and can insert in the rqt gui. There are existing plugins to do visualization, debugging, and so on in rqt.

Other Books You May Enjoy

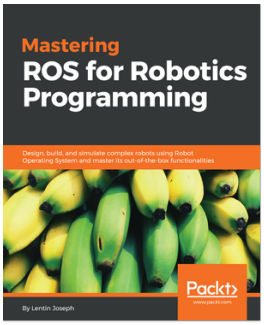
If you enjoyed this book, you may be interested in these other books by Packt:



ROS Robotics Projects
Lentin Joseph

ISBN: 978-1-78355-471-3

- Create your own self-driving car using ROS
- Build an intelligent robotic application using deep learning and ROS
- Master 3D object recognition
- Control a robot using virtual reality and ROS
- Build your own AI chatter-bot using ROS
- Get to know all about the autonomous navigation of robots using ROS
- Understand face detection and tracking using ROS
- Get to grips with teleoperating robots using hand gestures
- Build ROS-based applications using Matlab and Android
- Build interactive applications using TurtleBot



Mastering ROS for Robotics Programming
Lentin Joseph

ISBN: 978-1-78355-179-8

- Create a robot model of a Seven-DOF robotic arm and a differential wheeled mobile robot
- Work with motion planning of a Seven-DOF arm using MoveIt!
- Implement autonomous navigation in differential drive robots using SLAM and AMCL packages in ROS
- Dig deep into the ROS Pluginlib, ROS nodelets, and Gazebo plugins
- Interface I/O boards such as Arduino, Robot sensors, and High end actuators with ROS
- Simulation and motion planning of ABB and Universal arm using ROS Industrial
- Explore the ROS framework using its latest version

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

E

F

G

N

O

P

Q

R

S

T

U

V

W

X

Y