1
Getting Started with ROS

In this chapter, we will introduce the **Robot Operating System** (**ROS**), which is a collection of software packages to aid researchers and developers using robotic systems. After we discuss the instructions to install ROS on your computer system using the Ubuntu operating system, the ROS architecture and many of its components are discussed. This will aid you in understanding the use of ROS to develop software for robotic applications.

ROS will be introduced in terms of its elements and their functions. An understanding of the ROS vocabulary is necessary to become proficient in using ROS to create programs for the control of real or simulated robots as well as devices, such as cameras.

To make the discussion more concrete, the **turtlesim simulator** will be presented with various examples of the ROS command usage. This simulator is part of ROS and it provides an excellent introduction to the capabilities of ROS.

In this chapter, we will cover the following topics:

- What ROS is and which robots use ROS
- How to install and launch ROS on your computer
- How to navigate the ROS directories
- An introduction to ROS packages, nodes, and topics
- Examples of useful ROS commands
- How to use ROS commands with the turtlesim simulator

What does ROS do and what are the benefits of learning ROS?

ROS is sometimes called a meta operating system because it performs many functions of an operating system, but it requires a computer's operating system such as Linux. One of its main purposes is to provide communication between the user, the computer's operating system, and equipment external to the computer. This equipment can include sensors, cameras, as well as robots. As with any operating system, the benefit of ROS is the hardware abstraction and its ability to control a robot without the user having to know all of the details of the robot.

For example, to move a robot's arms, a ROS command is issued, or scripts in Python or C++ written by the robot designers cause the robot to respond as commanded. The scripts can, in turn, call various control programs that cause the actual motion of the robot's arms. It is also possible to design and simulate your own robot using ROS. These subjects and many others will be considered in this book.

In this book, you will learn a set of concepts, software, and tools that apply to an ever-increasing and diverse army of robots. For example, the navigation software of one mobile robot can be used, with a few changes, to work in another mobile robot. The flight navigation of an aerial robot is similar to that of the ground robot and so on. All across the broad spectrum of robotics, system interfaces are standardized or upgraded to support increased complexity. There are readily available libraries for commonly used robotics functions. ROS not only applies to the central processing of robotics but also to sensors and other subsystems. ROS hardware abstraction combined with low-level device control speeds the upgrade toward the latest technology.

ROS is an open source robotic software system that can be used without licensing fees by universities, government agencies, and commercial companies. The advantages of open source software is that the source code for the system is available and can be modified according to a user's needs. More importantly for some users, the software can be used in a commercial product as long as the appropriate licenses are cited. The software can be improved and modules can be added by users and companies.

ROS is used by many thousands of users worldwide and knowledge can be shared between users. The users range from hobbyists to professional developers of commercial robots. In addition to the large group of ROS researchers, there is a ROS-Industrial group dedicated to applying ROS software to robots for manufacturing.

Who controls ROS?

A ROS distribution is a set of ROS software packages that can be downloaded to your computer. These packages are supported by the **Open Source Robotics Foundation** (**OSRF**), a nonprofit organization. The distributions are updated periodically and given different names by the ROS organization. More details about the ROS organization are available at:

`http://www.ros.org/about-ros/`

This book is written using Ubuntu 14.04 as the operating system and **ROS Indigo** as the version of the ROS distribution. Always make sure that you check for any updates for the Ubuntu or ROS versions you are using.

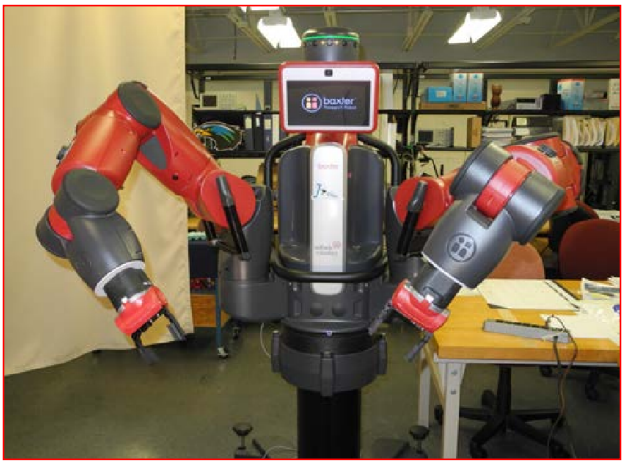
Which robots are using ROS?

There is a long list of robots on the ROS wiki website `http://wiki.ros.org/Robots`, which use ROS. For example, we are using four different robots in this book to provide you with an experience of a wide range of ROS capabilities. These robots are as follows:

- TurtleBot, a mobile robot
- Baxter, a friendly two-armed robot
- Crazyflie and Bebop, flying robots

The images of these robots are in the following figures:

TurtleBot 2



Baxter in the authors' laboratory

Of course, not everyone has the opportunity to use real robots such as Baxter. However, there is good news! Using the ROS Gazebo software, you can simulate Baxter as well as many other robots whose models are provided for Gazebo. We will simulate TurtleBot using Gazebo and actually design our own mobile robot in the upcoming chapters of this book.

Bebop and Crazyflie

Installing and launching ROS

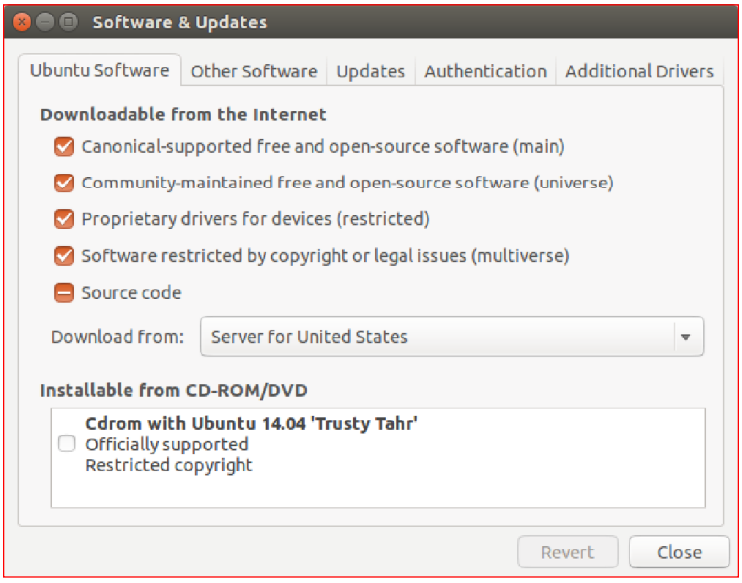
For this book, we assume the reader has a computer with Ubuntu Saucy 13.10 or Trusty 14.04 installed. The examples in this book have been developed using ROS Indigo and this version of ROS is only supported by these two versions of Ubuntu. The instructions for ROS installation provided in this section are for installing Debian (binary) packages. This method is the most efficient and preferred way to install ROS.

If you wish to install the ROS Indigo source code and build the software, refer to instructions at `http://wiki.ros.org/indigo/Installation/Source`. The instructions presented here to install ROS Indigo with Debian packages can also be found at `http://wiki.ros.org/indigo/Installation/Ubuntu`.

If you have any problems installing ROS, refer to this site and the ROS forum at `http://answers.ros.org`.

Configuring your Ubuntu repositories

To begin, configure the Ubuntu repositories to allow **restricted**, **universe**, and **multiverse**. Click on the Ubuntu Software Center icon in the launch menu on the left side of your desktop. From the Software Center's top menu bar, navigate to **Edit,** then to the drop-down menu, and select **Software Sources**. On the **Software & Updates** screen, select checkboxes to match the following screenshot:



Ubuntu Software Center Screen

Setting up your sources.list

Open a terminal window to set up the `sources.list` file on your computer to accept software from the ROS software repository at `http://packages.ros.org` which is the authorized site for the ROS software.

At the `$` command prompt, type the following command as one long command:

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

This step allows the operating system to know where to download programs that need to be installed on your system. When updates are made to ROS Indigo, your operating system will be made aware of these updates.

Setting up your keys

Keys confirm the origin of the code and verify that unauthorized modifications to the code have not been made without the knowledge of the owner. A repository and the keys of that repository are added the operating system's trusted software list. Type the following command:

```
$ sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net:80 --
recv-key 0xB01FA116
```

Installing ROS Indigo

Before you begin with the installation, the current system software must be up to date to avoid problems with libraries and wrong versions of software. To make sure your Debian package index is up-to-date, type the following command:

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

Warning: If you are using Ubuntu Trusty14.04.2 and experience dependency issues during the ROS installation, you may have to install some additional system dependencies.

⚠ Do not install these packages if you are using 14.04, it will destroy your X server:

```
$ sudo apt-get install xserver-xorg-dev-lts-utopic mesa-common-dev-lts-utopic libxatracker-dev-lts-utopic libopenvg1-mesa-dev-lts-utopic libgles2-mesa-dev-lts-utopic libgles1-mesa-dev-lts-utopic libgl1-mesa-dev-lts-utopic libgbm-dev-lts-utopic libegl1-mesa-dev-lts-utopic
```

Alternatively, try installing just this to fix dependency issues:

```
$ sudo apt-get install libgl1-mesa-dev-lts-utopic
```

For more information on this issue, refer to the following sites:

`http://answers.ros.org/question/203610/ubuntu-14042-unmet-dependencies/`

`https://bugs.launchpad.net/ubuntu/+source/mesa-lts-utopic/+bug/1424059`

Install the **desktop-full** configuration of ROS. `Desktop-full` includes ROS, **rqt**, **rviz**, robot-generic libraries, 2D/3D simulators, navigation, and 2D/3D perception. In this book, we will be using rqt and rviz for visualization and also the Gazebo 3D simulator, as well as the ROS navigation and perception packages. To install, type the following command:

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

ROS Indigo is installed on your computer system when the installation is complete!

Initialize rosdep

The ROS system may depend on software packages that are not loaded initially. These software packages external to ROS are provided by the operating system. The ROS environment command `rosdep` is used to download and install these external packages. Type the following command:

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

Environment setup

Your terminal session must now be made aware of these ROS files so that it knows what to do when you attempt to execute ROS command-line commands. Running this script will set up the ROS environment variables:

```
$ source /opt/ros/indigo/setup.bash
```

Alternately, it is convenient if the ROS environment variables are automatically added to your terminal session every time a new shell is launched. If you are using bash for your terminal shell, do this by typing the following command:

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

Now when a new terminal session is launched, the bash shell is automatically aware of the ROS environment variables.

Getting rosinstall

The `rosinstall` command is a command-line tool in ROS that allows you to download ROS packages with one command.

To install this tool on Ubuntu, type the following command:

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

Troubleshooting – examining your ROS environment

The ROS environment is set up through a number of variables that tell the system where to find ROS packages. Two main variables are `ROS_ROOT` and `ROS_PACKAGE_PATH` that enable ROS to locate packages in the filesystem.

To check whether the ROS environment variables are set correctly, use the `export` command in the following form that lists the ROS environment variables:

```
$ export | grep ROS
```

The output of the preceding command is as follows:

declare -x ROSLISP_PACKAGE_DIRECTORIES=""

declare -x ROS_DISTRO="indigo"

declare -x ROS_ETC_DIR="/opt/ros/indigo/etc/ros"

declare -x ROS_MASTER_URI="http://localhost:11311"

declare -x ROS_PACKAGE_PATH="/opt/ros/indigo/share:/opt/ros/indigo/stacks"

declare -x ROS_ROOT="/opt/ros/indigo/share/ros"

If the variables are not set correctly, you will need to source your `setup.bash` file as described in the *Environment setup* section of this chapter. Check whether the `ROS_DISTRO= "indigo"` and the `ROS_PACKAGE_PATH` variables are correct, as shown previously.

The tutorial that discusses the ROS environment can be found at `http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment`.

Creating a catkin workspace

The next step is to create a catkin workspace. A catkin workspace is a directory (folder) in which you can create or modify existing catkin packages. The catkin structure simplifies the build and installation process for your ROS packages. The ROS wiki website is `http://wiki.ros.org/catkin/Tutorials/create_a_workspace`.

A catkin workspace can contain up to three or more different subdirectories (`/build`, `/devel`, and `/src`) which each serve a different role in the software development process.

We will label our catkin workspace `catkin_ws`. To create the catkin workspace, type the following commands:

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

Even though the workspace is empty (there are no packages in the `src` folder, just a single `CMakeLists.txt` link), you can still build the workspace by typing the following command:

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

The `catkin_make` command creates the catkin workspaces. If you view your current directory contents, you should now have the `build` and `devel` folders. Inside the `devel` folder there are now several setup.`*sh` files. We will source the `setup.bash` file to overlay this workspace on top of your ROS environment:

```
$ source ~/catkin_ws/devel/setup.bash
```

Remember to add this source command to your `.bashrc` file by typing the following command:

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

To make sure your workspace is properly overlaid by the setup script, make sure the `ROS_PACKAGE_PATH` environment variable includes the directory you're in by typing the following command:

```
$ echo $ROS_PACKAGE_PATH
```

The output of the preceding command should be as follows:

```
/home/<username>/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indigo/stacks
```

Here, `<username>` is the name you chose as user when Ubuntu was installed.

ROS packages and manifest

The ROS software is divided into **packages** that can contain various types of programs, images, data, and even tutorials. The specific contents depend on the application for the package. The site `http://wiki.ros.org/Packages` discusses ROS packages.

A package can contain programs written in Python or C++ to control a robot or another device. For the turtlesim simulator package for example, the package contains the executable code used to change the background color or move a turtle around on the screen. This package also contains images of a turtle for display and files used to create the simulator.

There is another class of packages in ROS called **metapackages** that are specialized packages that only contain a `package.xml` manifest. Their purpose is to reference one or more related packages, which are loosely grouped together.

ROS manifest

Each package contains a **manifest** named `package.xml` that describes the package in the **Extensible Markup Language** (**XML**) format. In addition to providing a minimal specification describing the package, the manifest defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other packages.

Exploring the ROS packages

Occasionally, we would like to find packages that we wish to use and display the files involved. This section introduces several useful ROS commands:

- `rospack` used for information about a package
- `roscd` used to navigate the ROS directories
- `rosls` used to list directories and files in a package directory

The `rospack` command can be used to list ROS packages, locate packages by name, and determine if a package depends on another package, among other uses. For more information use the following command with the `help` or `-h` option in the form:

```
$ rospack help | less
```

There are many options for this command, so piping with the `less` option shows one screen at a time. For many commands, using the power of Linux to make the output readable is necessary. Use the *Q* key to quit (exit) the mode.

We will use the `turtlesim` package for the examples here. To change directories to the location of turtlesim, use the following command:

```
$ roscd turtlesim
```

This yields the location to one of the author's laptops as follows:

harman@Laptop-M1210:/opt/ros/indigo/share/turtlesim$

On your computer, the `$` command prompt will be preceded by the information about your computer. Generally, that information for our computers will be deleted in our examples using ROS commands. Once you are in the turtlesim directory, the standard Linux commands can be used with the subdirectories or files, or the ROS commands can be used. To determine the directories and files in the turtlesim directory but without changing to the turtlesim directory, use the following command:

```
$ rosls turtlesim
```

Here is the result from the home directory of the author's laptop with ROS installed:

cmake images msg package.xml srv

To see the filenames of the images loaded with `turtlesim`, specify the `images` directory in the package as follows:

```
$ rosls turtlesim/images
```

The output of the preceding command is as follows:

box-turtle.png fuerte.png hydro.svg palette.png turtle.png

diamondback.png groovy.png indigo.png robot-turtle.png

electric.png hydro.png indigo.svg sea-turtle.png

There are various turtle images that can be used. The `rosls turtlesim` command will also work to show the contents of the turtlesim subdirectories: `/msg` for messages and `/srv` for services. These files will be discussed later. To see the manifest, type the following command:

```
$ roscd turtlesim
$ cat package.xml
```

This will also show the dependencies such as `roscpp` for C++ programs.

rospack find packages

The `rospack find <package name>` command returns the path to the package named `<package name>`. For example, type the following command:

```
$ rospack find turtlesim
```

The preceding command displays the path to the turtlesim directory.

rospack list

Execute the following command:

```
$ rospack list
```

This lists the ROS package names and their directories on the computer. In the case of the laptop mentioned earlier, there are 225 ROS packages listed!

If you really want to see all the ROS packages and their locations, use the following command form:

```
$ rospack list | less
```

This form allows paging of the long list of names and directories for the packages. Press *Q* to quit.

Alternatively, this is the form of the `rospack` command:

```
$ rospack list-names
```

This lists only the names of the packages without the directories. After such a long list, it is a good idea to open a new terminal window or clear the window with the `clear` command.

This is the form of the `rospack` command:

```
$ rospack list-names | grep turtle
```

This lists the four packages with turtle in the name.

More information on commands that are useful to navigate the ROS filesystem is available at the ROS website `http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem`.

ROS nodes and ROS Master

One of the primary purposes of ROS is to facilitate communication between the ROS modules called nodes. These nodes represent the executable code and the code can reside entirely on one computer, or nodes can be distributed between computers or between computers and robots. The advantage of this distributed structure is that each node can control one aspect of a system. One node can capture and output camera images, and another node can control a robot's manipulator in response to the camera view. We will see that the turtlesim simulator has two nodes that are created when turtlesim is executed.

The ROS site `http://www.ros.org/core-components/` describes the communication and robot-specific features of ROS. Here, we will explore some of the main components of a ROS system including ROS nodes and the ROS Master. It is important for you to understand the ROS nodes and Master, so they will be defined later through discussions and examples.

ROS nodes

Basically, nodes are processes that perform some action. The nodes themselves are really software modules but with the capability to register with the ROS Master node and communicate with other nodes in the system. The ROS design idea is that each node is an independent module that interacts with other nodes using the ROS communication capability.

The nodes can be created in various ways. From a terminal window a node can be created directly by typing a command after the command prompt, as shown in the examples to follow. Alternatively, nodes can be created as part of a program written in Python or C++. In this book, we will use both the ROS commands in a terminal window and Python programs to create nodes.

Nodes can publish and nodes can subscribe

One of the strengths of ROS is that a particular task, such as controlling a wheeled mobile robot, can be separated into a series of simpler tasks. The tasks can include the perception of the environment using a camera or laser scanner, map making, planning a route, monitoring the battery level of the robot's battery, and controlling the motors driving the wheels of the robot. Each of these actions might consist of a ROS node or series of nodes to accomplish the specific tasks.

A node can independently execute code to perform its task but can communicate with other nodes by sending or receiving messages. The messages can consist of data, or commands, or other information necessary for the application.

Some nodes provide information for other nodes, as a camera feed would do, for example. Such a node is said to publish information that can be received by other nodes. The information in ROS is called a **topic**, which will be explained later. Continuing with the camera example, the camera node can publish the image on the `camera/image_raw` topic.

Image data from the `camera/image_raw` topic can be used by a node that shows the image on the computer screen. The node that receives the information is said to subscribe to the topic being published, in this case `camera/image_raw`.

In some cases, a node can both publish and subscribe to one or more topics. A number of examples later in the book will demonstrate this functionality.

ROS Master

The ROS nodes are typically small and independent programs that can run concurrently on several systems. Communication is established between the nodes by the ROS Master. The ROS Master provides naming and registration services to the nodes in the ROS system. It tracks publishers and subscribers to the topics. The role of the Master is to enable individual ROS nodes to locate one another. The most often used protocol for connection is the standard **Transmission Control Protocol/Internet Protocol** (**TCP/IP**) or Internet Protocol called TCPROS in ROS. Once these nodes are able to locate one another, they can communicate with each other peer-to-peer.

One responsibility of the Master is to keep track of nodes when new nodes are executed and come into the system. Thus, the Master provides a dynamic allocation of connections. The nodes cannot communicate however, until the Master notifies the nodes of each others' existence. A simple example is shown at: `http://wiki.ros.org/Master`.

Invoking the ROS Master using roscore

`roscore` is a collection of nodes and programs that you **must** have running for ROS nodes to communicate. After it is launched, `roscore` will start the following:

- A ROS Master
- A ROS Parameter Server
- A `rosout` logging node

The `roscore` command starts ROS and creates the Master so that nodes can register with the Master. You can view the ROS tutorial for `roscore` at `http://wiki.ros.org/roscore`.

Issue the following command to start the Master in a new terminal window and observe the output:

```
$ roscore
```

The output of the preceding command is as follows:

... logging to /home/harman/.ros/log/94248b4a-3f05-11e5-b5ce-00197d37ddd2/roslaunch-Laptop-M1210-2322.log

Checking log directory for disk usage. This may take awhile.

Press Ctrl-C to interrupt

Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://Laptop-M1210:46614/

ros_comm version 1.11.13

SUMMARY

========

PARAMETERS

*** /rosdistro: indigo**

*** /rosversion: 1.11.13**

NODES

auto-starting new master

process[master]: started with pid [2334]

ROS_MASTER_URI=http://Laptop-M1210:11311/

setting /run_id to 94248b4a-3f05-11e5-b5ce-00197d37ddd2

process[rosout-1]: started with pid [2347]

started core service [/rosout]

In the preceding screen output, you will see information about the computer, parameters that list the name (`indigo`) and version number of the ROS distribution, and other information. The Master is defined by its **Uniform Resource Identifier** (**URI**). This identifies the location of the Master; in this case, it is running on the laptop and is used to execute the `roscore` command.

Parameter Server

The Parameter Server is a shared dictionary of parameters that nodes store and retrieve at runtime. The Parameter Server runs inside the Master and parameters are globally viewable so that nodes can access the parameters.

In the preceding screen output from the `roscore` command, the parameters associated with the Master are as follows:

*** /rosdistro: indigo**

*** /rosversion: 1.11.13**

Indigo is the ROS distribution release that we are using. As Indigo is changed or packages are added, numbered versions such as `1.11.13` are released. Issuing the `roscore` command is a way to determine the release of ROS running on your computer.

Whenever ROS is executing, it is possible to list the nodes that are active and the topics that are used for communication. We will explore the information in the `roscore` output in more detail by invoking useful ROS terminal commands.

ROS commands to determine the nodes and topics

Three commands used extensively in ROS are as follows:

- `roscore` to start the Master and allow nodes to communicate
- `rosnode list` to list the active nodes
- `rostopic list` to list the topics associated with ROS nodes

After the `roscore` command is executed, the terminal window used to execute `roscore` must remain active, but it can be minimized. In another terminal window, the `rosnode list` command will cause a list of the ROS nodes that are active to be displayed on the screen. After the command for `roscore` is executed, only one node `rosout` will be listed as an active node if you type the following command:

```
$ rosnode list
```

The output of the preceding command is as follows:

/rosout

In the second terminal window, list the active topics by typing:

```
$ rostopic list
```

The output of the preceding command is as follows:

/rosout

/rosout_agg

Notice that the `/rosout` node and the `/rosout` topic have the same designation. In ROS terms, the `rosout` node subscribes to the `/rosout` topic. All the active nodes publish their debug messages to the `/rosout` topic. We will not be concerned with these messages here; however, they can be useful to debug a program. For some explanation, refer to the ROS wiki at `http://wiki.ros.org/rosout`.

The `rosout` node is connected to every other active node in the system. The `/rosout_agg` topic receives messages also, but just from the `rosout` node so it does not have to connect to all of the nodes and thus saves time at system startup.

The `rostopic` and `rosnode` terminal commands have a number of options that will be demonstrated by various examples in this book.

Most of the ROS commands have help screens that are usually helpful. Type the following command for the command options:

```
$ rosnode -h
```

For more detailed usage, use the subcommand name, for example:

```
$ rosnode list -h
```

will list the subcommands and the options for the `rosnode` command.

There are a number of other important ROS terminal commands that you should know. They are introduced and explained using the turtlesim simulator in the upcoming section.

Turtlesim, the first ROS robot simulation

A simple way to learn the basics of ROS is to use the turtlesim simulator that is part of the ROS installation. The simulation consists of a graphical window that shows a turtle-shaped robot. The background color for the turtle's world can be changed using the Parameter Server. The turtle can be moved around on the screen by ROS commands or using the keyboard.

Turtlesim is a ROS package, and the basic concepts of package management were presented in the *Exploring the ROS packages* section, as discussed earlier. We suggest that you refer to this section before continuing.

We will illustrate a number of ROS commands that explore the nodes, topics, messages, and services used by the turtle simulator. We have already covered the `roscore`, `rosnode`, and `rostopic` commands. These commands will be used with turtlesim also.

Other important ROS terminal commands that will be covered in this section are as follows:

- `rosrun` that finds and starts a requested node in a package
- `rosmsg` that shows information about messages
- `rosservice` that displays runtime information about nodes and can pass data between nodes in a request/response mode
- `rosparam` that is used to get and set parameters (data) used by nodes

Starting turtlesim nodes

To start turtlesim with ROS commands, we need to open two separate terminal windows. First, issue the following command in the first window if the Master is not already running:

```
$ roscore
```

Wait for the Master to complete startup. You can minimize this window but do not close it because the Master must run to allow the nodes to communicate.

The result on your screen will resemble the output discussed previously in the *Invoking the ROS Master using roscore* section, where `roscore` was described.

rosrun command

To display the turtle on the screen, use the `rosrun` command. It takes the arguments `[package name] [executable name]`, and in this case, `turtlesim` as the package and `turtlesim_node` as the executable program.

In the second terminal window, issue the following command:

```
$ rosrun turtlesim turtlesim_node
```

You will see an output similar to this:

[INFO] [1427212356.117628994]: Starting turtlesim with node name /turtlesim

[INFO] [1427212356.121407419]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]

Wait for the display screen to appear with the image of a turtle at the center, as shown in the turtlesim screen in the following screenshot. The terminal window can be minimized, but keep the turtle display screen in view. The turtle is called `turtle1` since this is the first and only turtle in our display.

After you have started turtlesim by executing the `rosrun` command, you will see information about the turtle's position on the screen. The `/turtlesim` node creates the screen image and the turtle. Here, the turtle is in the center at about $x = 5.5$, $y = 5.5$ with no rotation since angle theta is zero. The origin (0, 0) is at the lower-left corner of the screen:

Turtlesim screen

Let's study the properties of the nodes, topics, services, and messages available with the `turtlesim` package in another terminal window. Thus, at this point, you will have three windows active but the first two can be minimized or dragged off to the side or the bottom. They should not be closed.

Turtlesim nodes

In the third window, issue the `rosnode` command to determine information about any node. First, list the active nodes, using the following command:

```
$ rosnode list
```

The output is as follows:

/rosout
/turtlesim

We will concentrate on the `/turtlesim` node. Note the difference in notation between the `/turtlesim` node and the `turtlesim` package.

To see the publications, subscriptions, and services of the turtlesim node, type the following command:

```
$ rosnode info /turtlesim
```

The output of the preceding command is as follows:

--

Node [/turtlesim]

Publications:

*** /turtle1/color_sensor [turtlesim/Color]**

*** /rosout [rosgraph_msgs/Log]**

*** /turtle1/pose [turtlesim/Pose]**

Subscriptions:

*** /turtle1/cmd_vel [unknown type]**

Services:

*** /turtle1/teleport_absolute**

*** /turtlesim/get_loggers**

*** /turtlesim/set_logger_level**

*** /reset**

*** /spawn**

*** /clear**

*** /turtle1/set_pen**

*** /turtle1/teleport_relative**

*** /kill**

contacting node http://Laptop-M1210:38993/ ...

Pid: 2364

Connections:

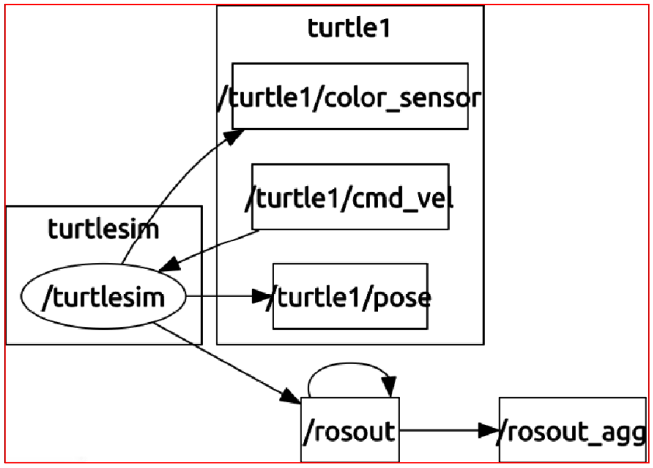
*** topic: /rosout**

*** to: /rosout**

*** direction: outbound**

*** transport: TCPROS**

The following diagram represents a graphical illustration of the relationship of the turtlesim node in elliptical shapes and topics in rectangular boxes:



/turtlesim node

The graph was created using the `rqt_graph` command as described in the *Introducing RQT tools* section in *Chapter 3, Driving Around with TurtleBot*.

We read the output of the `rosnode info` command and the graph of the turtlesim node and topics in the preceding diagram as follows (ignoring the logging services and the `/rosout` and `/rosout_agg` nodes):

- The `/turtlesim` node publishes to two topics. These topics control the color of the turtle's screen and the position of the turtle on the screen when messages are sent from `/turtlesim`:
  - `/turtle1/color_sensor` with the message type `[turtlesim/Color]`
  - `/turtle1/pose` with the message type `[turtlesim/Pose]`
- The `/turtlesim` node subscribes to the `turtle1/cmd_vel` topic of an unknown type. The type is unknown because another node that publishes on this topic has not yet been executed.
- There are a number of services associated with the `/turtlesim` node. The services can be used to move the turtle around (teleport), clear the screen, kill the nodes, and perform other functions. The services will be explained later in this section.

Turtlesim topics and messages

A ROS message is a strictly typed data structure. In fact, the message type is not associated with the message contents. For example, one message type is `string`, which is just text. Another type of message is `uint8`, which means that it is an unsigned 8-bit integer. These are part of the `std_msg` package or standard messages. The command form `rosmsg list` lists the type of messages on your system; it is a long list! There are packages for messages of the type control, type geometry, and type navigation, among others to control robot actions. There are sensor message types used with laser scanners, cameras, and joysticks to name just a few of the sensors or input devices possible with ROS. Turtlesim uses several of the message types to control its simulated robot—the turtle.

For these exercises, keep the `roscore` and `/turtlesim` windows active. Open other terminal windows as needed. We will concentrate on the `turtle1/color_sensor` topic first. You will be typing in the third window.

If the screen gets too cluttered, remember the `$ clear` command.
Use the `$ history` command to see the commands you have used.

rostopic list

For the `/turtlesim` node, the topics are listed using the following command:

```
$ rostopic list
```

The output of the preceding command is as follows:

/rosout

/rosout_agg

/turtle1/cmd_vel

/turtle1/color_sensor

/turtle1/pose

rostopic type

The topic type can be determined for each topic by typing the following command:

```
$ rostopic type /turtle1/color_sensor
```

turtlesim/Color

The message type in the case of the `/turtle1/color_sensor` topic is `turtlesim/Color`. This is the format of ROS message type names:

```
[package name]/[message type]
```

If a node publishes a message, we can determine the message type and read the message.

rosmsg list

The `turtlesim` package has two message types that are found with the following command:

```
$ rosmsg list | grep turtlesim
```

The output is as follows:

turtlesim/Color
turtlesim/Pose

The `rosmsg list` command displays all of the messages. Adding `| grep turtlesim` shows all messages of the `turtlesim` package. There are only two in the `turtlesim` package. From the message type, we can find the format of the message. Make sure that you note that `Color` in the message type starts with a capital letter.

rosmsg show

The `rosmsg show <message type>` command displays the fields in a ROS message type. For the `turtlesim/Color` type, the fields are integers:

```
$ rosmsg show turtlesim/Color
```

Output of the preceding command is as follows:

uint8 r
uint8 g
uint8 b

The format of values designating colors red (`r`), green (`g`), and blue (`b`) is unsigned 8-bit integer.

To understand the message, it is necessary to find the message type. For example, `turtlesim/Color` is a message type for turtlesim that has three elements that define the color of the background. For example, the red color in the background is defined by `uint8 r`. This indicates that if we wish to modify the red value, an 8-bit unsigned integer is needed. The amount of red in the background is in the range of 0-255.

In general, the formats of numerical data include integers of 8, 16, 32, or 64 bits, floating point numbers, and other formats.

rostopic echo

To determine the color mixture of red, green, and blue in the background of our turtle, use the `rostopic echo [topic name]` command in the form, as follows:

```
$ rostopic echo /turtle1/color_sensor
```

Output of the preceding command is as follows:

r: 69

g: 86

b: 255

Press *Ctrl* + *C* to stop the output.

The website describing the RGB Color Codes Chart and the meaning of the numerical color values can be found at `http://www.rapidtables.com/web/color/RGB_Color.htm`.

The chart explains how to mix the **red, green, and blue** (**rgb**) color values to achieve any desired color. An example, which is discussed later, will show you how to change the background color for the turtle. The color values are parameters that can be changed.

A simple table can clarify the relationship between the topics and the messages for the `/turtlesim` node:

Topics and Messages for /turtlesim node			
Topics name	**Topic type**	**Message format**	**Message**
`$ rostopic list`	`$ rostopic type [topic name]`	`$ rosmsg show [topic type]`	`$ rostopic echo [topic name]`

`/turtle1 /color_sensor`	`turtlesim/Color`	`uint8 r`
		`uint8 g`
		`uint8 b`
			`r: 69`
			`g: 86`
			`b: 255`

`/turtle1/pose`	`turtlesim/Pose`	`float32 x`
		`float32 y`
		`float32 theta`
		`float32 linear_velocity`
		`float32 angular_velocity`
			`x: 5.54444456`
			`y: 5.54444456`
			`theta: 0.0`
			`linear_velocity: 0.0`
			`angular_velocity: 0.0`

The table of **Topics and Messages** shows the topics, types, message formats, and data values for the two topics of the `/turtlesim` node that we explored. The commands to determine the information are also shown in the table.

Parameter Server of turtlesim

The Parameter Server maintains a dictionary of the parameters that are used to configure the screen of turtlesim. We will see that these parameters are used to define the color of the background for the turtle. Thus, the turtlesim node can read and write parameters held by the Parameter Server.

rosparam help

Use the `help` option to determine the form of the `rosparam` command:

```
$ rosparam help
```

Output of the preceding command is as follows:

rosparam is a command-line tool for getting, setting, and deleting parameters from the ROS Parameter Server. Commands:

rosparam set set parameter

rosparam get get parameter

rosparam list list parameter names

<Edited>

rosparam list for /turtlesim node

To list the parameters for the `/turtlesim` node, we will use the following command:

```
$ rosparam list
```

Output of the preceding code is as follows:

/background_r

/background_g

/background_b

/rosdistro

/roslaunch/uris/host_d125_43873__51759

/rosversion

/run_id

Note that the last four parameters were created by invoking the Master with the `roscore` command, as discussed previously. Also, the list defines the characteristics of the parameter but not the data value.

Change parameters for the color of the turtle's background

To change the color parameters for turtlesim, let's change the turtle's background to red. To do this, make the blue and green data values equal to zero and saturate red = 255 using the `rosparam set` command. Note that the `clear` option from `rosservice` must be executed before the screen changes color.

rosparam get

The default turtle screen is blue. You can use `rosparam get /` to show the data contents of the entire Parameter Server:

```
$ rosparam get /
```

Output of the preceding command is as follows:

background_b: 255

background_g: 86

background_r: 69

rosdistro: 'indigo

roslaunch:

uris: {host_d125_43873__60512: 'http://D125-43873:60512/'}

rosversion: '1.11.13

run_id: 2429b792-d23c-11e4-b9ee-3417ebbca982

rosparam set

You can change the colors of the turtle's screen to a full red background using the `rosparam set` command:

```
$ rosparam set background_b 0
$ rosparam set background_g 0
$ rosparam set background_r 255
$ rosservice call /clear
```

You will see a red background on the turtle screen. To check the numerical results, use the `rosparam get /` command.

ROS services to move turtle

Another capability of nodes is to provide what in ROS terms is called a service. This feature is used when a node requests information from another node. Thus, there is a two-way communication between the nodes.

You can check the turtle's pose using the `/turtle1/pose` topic and the message type, `/turtlesim/Pose`. Carefully note the different notations and meanings.
To determine the the message type, run the following command:

```
$ rostopic type /turtle1/pose
```

The output is as follows:

turtlesim/Pose

```
$ rosmsg show turtlesim/Pose
```

The output is as follows:

float32 x

float32 y

float32 theta

float32 linear_velocity

float32 angular_velocity

We can find the turtle's position, orientation in angle (`theta`), and its velocity using the `rostopic echo` command:

```
$ rostopic echo /turtle1/pose
```

The output is as follows:

x: 5.544444561

y: 5.544444561

theta: 0.0

linear_velocity: 0.0

angular_velocity: 0.0

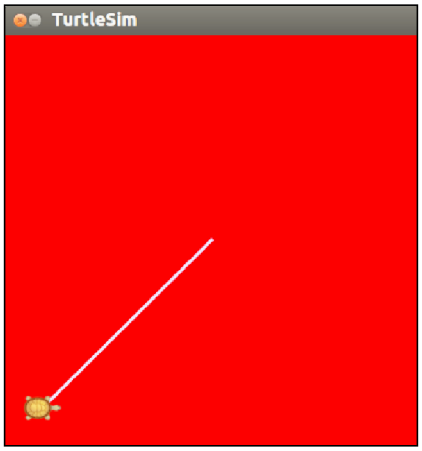
This command outputs the result continuously and is stopped by pressing the *Ctrl +C* keys. The result will show that the turtle is at the center of its screen with no rotation at angle zero and no movement since the velocities are zero.

rosservice call

The turtle can be moved using the `rosservice` teleport option. The format of the command is `rosservice call <service name> <service arguments>`. The arguments here will be the turtle's position and orientation as `x`, `y`, and `theta`. The turtle is moved to position [`1,1`] with theta=`0` by running the following command:

```
$ rosservice call /turtle1/teleport_absolute 1 1 0
```

The result can be seen in the following screenshot:



turtle after an absolute move

The relative teleport option moves the turtle with respect to its present position. The arguments are [linear distance , angle]. Here, the rotation angle is zero. The command for relative movement is as follows:

```
$ rosservice call /turtle1/teleport_relative 1 0
```

Your turtle should now move to *x=2* and *y=1*.

ROS commands summary

If you are communicating with ROS via the terminal window, it is possible to issue commands to ROS to explore or control nodes in a package from the command prompt, as listed in the following table:

Command	**Action**	**Example usage and subcommand examples**
`roscore`	This starts the Master	`$ roscore`
`rosrun`	This runs an executable program and creates nodes	`$ rosrun [package name] [executable name]`

`rosnode`	This shows information about nodes and lists the active nodes	`$ rosnode info [node name]`
		`$ rosnode <subcommand>`
		Subcommand: `list`

`rostopic`	This shows information about ROS topics	`$ rostopic <subcommand> <topic name>` Subcommands: `echo`, `info`, and `type`
`rosmsg`	This shows information about the message types	`$ rosmsg <subcommand> [package name]/ [message type]` Subcommands: `show`, `type`, and `list`
`rosservice`	This displays the runtime information about various services and allows the display of messages being sent to a topic	`$ rosservice <subcommand> [service name]` Subcommands: `args`, `call`, `find`, `info`, `list`, and `type`
`rosparam`	This is used to get and set parameters (data) used by nodes	`$ rosparam <subcommand> [parameter]` Subcommands: `get`, `set`, `list`, and `delete`

The website (`http://wiki.ros.org/ROS/CommandLineTools`) describes many ROS commands. The table lists some important ones. However, these examples only cover a few of the possible variations of the commands.

Summary

In this chapter, you first learned how to install and launch ROS. We discussed the ROS architecture and ROS packages, nodes, topics, messages, and services. To apply the knowledge, the turtlesim simulator was used to illustrate many ROS commands.

In *Chapter 2, Creating Your First Two-Wheeled ROS Robot (in Simulation)*, we will show you how to build a robot model that ROS uses to display the robot and allows you to control it in a simulation. The chapter introduces the visualization tool called rviz to display the robot and the simulation tool Gazebo that includes the physics of the robot as you move it around in a simulated environment.

2
Creating Your First Two-Wheeled ROS Robot (in Simulation)

Your first robot will be created in simulation so that even if you do not have a robot to learn ROS on, you will be able to follow along and do the exercises in this book. We will build a simple two-wheeled robot named `dd_robot` (dd is short for a differential drive). We will build a **Unified Robot Description Format** (**URDF**) file for the robot that will describe the main components of our robot and enable it to be visualized and controlled by ROS tools, such as rviz and Gazebo. Rviz is a visualization tool in which we will view our `dd_robot` URDF file as we build it in increments. When the visual model is complete, we will modify the URDF file for use in the Gazebo simulator. In Gazebo, we can view the effects of physics on our model as we move our model around the 3D environment.

In this chapter, we will cover the following topics:

- An introduction to rviz, installation instructions, and instructions for use
- How to create and build a ROS package
- An incremental approach to develop a URDF file and visualize it in rviz
- URDF tools to verify the URDF file
- An introduction to Gazebo, installation instructions, and instructions for use
- Modifications necessary to visualize the URDF file in Gazebo
- Tools to verify your Gazebo URDF/**Simulation Description Format** (**SDF**) file
- A simple way to control a robot in Gazebo

We begin by learning about rviz.

Rviz

Rviz, abbreviation for ROS visualization, is a powerful 3D visualization tool for ROS. It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information. By visualizing what the robot is seeing, thinking, and doing, the user can debug a robot application from sensor inputs to planned (or unplanned) actions.

Rviz displays 3D sensor data from stereo cameras, lasers, Kinects, and other 3D devices in the form of point clouds or depth images. 2D sensor data from webcams, RGB cameras, and 2D laser rangefinders can be viewed in rviz as image data.

If an actual robot is communicating with a workstation that is running rviz, rviz will display the robot's current configuration on the virtual robot model. ROS topics will be displayed as live representations based on the sensor data published by any cameras, infrared sensors, and laser scanners that are part of the robot's system. This can be useful to develop and debug robot systems and controllers. Rviz provides a configurable **Graphical User Interface** (**GUI**) to allow the user to display only information that is pertinent to the present task.

In this chapter, we will use rviz to visualize our progress in creating a two-wheeled robot model. Rviz will use the URDF file that we create for our robot and display the visual representation.

We will begin by checking whether rviz has been downloaded and installed on your system.

Installing and launching rviz

To run rviz, you require a powerful graphics card and the appropriate drivers need to be installed on your computer.

If you have trouble with running rviz, refer to `http://wiki.ros.org/rviz/Troubleshooting` or search the ROS forum at `http://answers.ros.org/questions/`.

Rviz should have been installed on your computer as part of the ros-indigo-desktop-full installation, as described in the *Installing and launching ROS* section in *Chapter 1*, *Getting Started with ROS*.

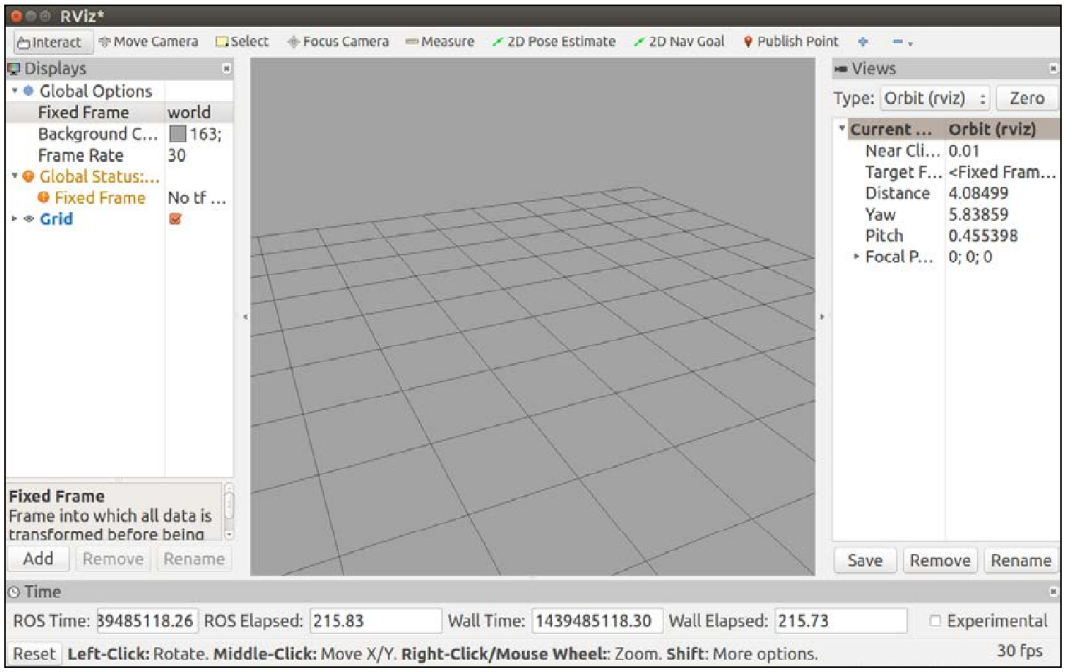
To test whether rviz has been installed correctly, open a terminal window and start the ROS Master by typing the following command:

```
$ roscore
```

Next, open a second terminal window and type the following command:

```
$ rosrun rviz rviz
```

This will display an environment similar to the following screenshot:



rviz main screen

If the `$ rosrun rviz rviz` command generates a warning message, make sure that you have `source ~/catkin_ws/devel/setup.bash` in your `.bashrc` file, or this command is entered at the terminal window prompt. The `.bashrc` file resides in your home directory but cannot be seen unless you use the `$ ls -la` command option to list the directory and files. This option shows the hidden files that are preceded by a dot (`.`).

If rviz has not been installed, then install it from the Debian repository using the following command:

```
$ sudo apt-get install ros-indigo-rviz
```

If you wish to install rviz from a source, refer to the rviz user guide at `http://wiki.ros.org/rviz/UserGuide`. This guide is also a good reference to learn additional features of rviz that are not covered in this book.

Using rviz

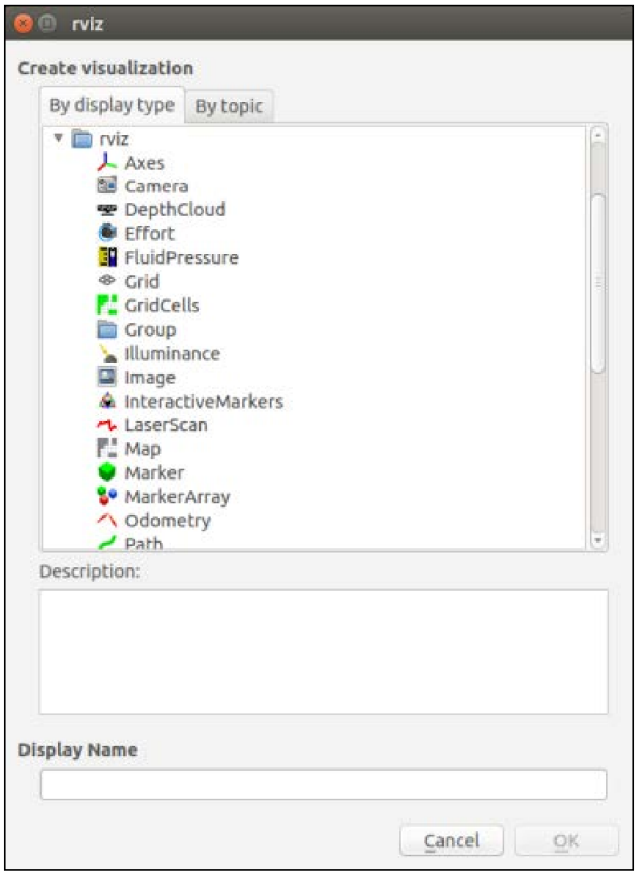
The central window on the rviz main screen provides the world view of a 3D environment. Typically, only the grid is displayed in the center window or the window is blank.

The main screen is divided into four main display areas: the central window, the **Displays** panel to the left, the **Views** panel to the right, and the **Time** panel at the bottom. Across the top of these display areas are the toolbar and the main screen menu bar. Each of these areas of the rviz main screen is described in the following sections. This overview is provided so that you can gain familiarity with the rviz GUI.

Displays panel

On the left panel of the rviz main screen is the **Displays** panel, where the user can add, remove, or rename the visualization elements in the 3D environment.

By clicking on the **Add** button on the **Displays** panel, the **Add** menu appears, as shown in the following screenshot. This menu displays the visualization elements that can be added to the environment, such as a camera image, point cloud, robot state, and so on. A brief description of each item is provided at the bottom of the window when that item is highlighted. A unique display name can be entered for the item to be added to the environment. For further details on the display types, go to `http://wiki.ros.org/rviz/DisplayTypes`. This site also identifies the ROS messages that provides the data for the display:



Rviz 'Add' Display menu

Clicking on the triangle symbol on the left side of a panel item will expand or hide the details of the item.

The **Displays** panel also shows access to **Global Options**, such as **Background Color** and **Frame Rate**. The options under the **Grid** element allow the user to tailor the grid lines by changing **the number of grid cells**, **line width**, **line color**, and so on.

Views and Time panels

The **Views** panel on the right side of the rviz main screen and the **Time** panel at the bottom are not important at this point, and so we have removed them from the rest of the screenshots in this chapter. We will work in the orbit view, which is the default camera view for rviz. In this view, the orbital camera will rotate around a focal point visualized as a small yellow disc in the 3D world view.

Mouse control

To move around the 3D world in the orbit view, use the mouse and keyboard as follows:

- **Left mouse button**: Click and drag to rotate around the focal point.
- **Middle mouse button** (if available): Click and drag to move the focal point in the plane formed by the camera's up and right vectors. (The *Shift* key and left mouse button combination also invokes this mode.)
- **Right mouse button**: Click and drag to zoom in/out of the focal point. Dragging up zooms in and down zooms out. (The scroll wheel also invokes this mode.)

Toolbar

The toolbar at the top of the rviz main screen provides the following functionalities:

- **Interact**: This shows interactive markers when present.
- **Move camera** (default mode): This is a 3D view that responds to the mouse/keyboard, as described for the **Views** panel.
- **Select**: This allows items to be selected by a mouse click or drag box selection. The selected item will have a wireframe box placed around it.
- **Focus camera**: A mouse click on a specific spot in the 3D view becomes the focal point of the camera.
- **2D Nav Goal** and **2D Pose Estimate**: These selections will be discussed in *Chapter 4*, *Navigating the World with TurtleBot*.

Main window menu bar

The top-most main window menu bar provides options under the basic **File**, **Panels**, and **Help** headings, as shown here:

- **File**: **Open Config**, **Save Config**, **Save Config As**, **Recent Configs**, **Save Image**, and **Quit**
- **Panels**: **Add Panel** and **Delete Panel**

  Other options for panels are: **Tools**, **Displays**, **Selection**, **Tool Properties**, **Views**, and **Time**

- **Help**: **Show Help panel**, **Open rviz wiki in browser**, and **About**

These selections allow the user to customize the panels to be displayed for rviz. This custom configuration of rviz can be saved and reused.

In this chapter, we use rviz to visualize the construction of our two-wheeled robot model in 3D. We will show you how to use rviz to visualize odometry data for navigation purposes in *Chapter 3*, *Driving Around with TurtleBot*.

For more in-depth tutorials on rviz, go to `http://wiki.ros.org/rviz/Tutorials`.

At this point, rviz can be exited by navigating to **File | Quit**. In the next section, we create and build a ROS package to hold our URDF code and launch files.

Creating and building a ROS package

Before we begin to design and build our robot model in simulation, we should create our first ROS package. In *Chapter 1*, *Getting Started with ROS*, we created a ROS catkin workspace under `/home/<username>/catkin_ws`. The structure of a catkin workspace looks like this:

```
catkin_ws/          -- WORKSPACE
  build/            -- BUILD SPACE
  devel/            -- DEVEL SPACE
  src/              -- SOURCE SPACE
  CMakeLists.txt    -- 'Toplevel' CMake file, provided by catkin
```

Make sure that you have `source ~/catkin_ws/devel/setup.bash` in your `.bashrc` file, or this command is entered at the terminal window prompt.

We begin by moving to your catkin workspace source directory:

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

Now, let's create our first ROS package, `ros_robotics`:

```
$ catkin_create_pkg ros_robotics
```

This command will create a `/ros_robotics` directory under the `~/catkin_ws/src` directory. The `/ros_robotics` directory will contain a `package.xml` file and a `CMakeLists.txt` file. These files contain information generated from the `$ catkin_create_pkg` command execution.

The catkin_create_pkg syntax

`catkin_create_pkg` requires a unique package name and, optionally, a list of dependencies for the package. The command format to create it is as follows:

```
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

`[depend1]`, `[depend2]`, and `[depend3]` specify software packages that are required to be present for this software package to be made.

We will not identify any dependencies for our `ros_robotics` package at this point.

Next, build the packages in the catkin workspace:

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

After the workspace has been built to include the `ros_robotics` package, the `~/catkin_ws/devel` subdirectory will have a structure similar to the structure under the `/opt/ros/indigo` directory.

Building a differential drive robot URDF

URDF is an XML format specifically defined to represent robot models down to their component level. These URDF files can become long and cumbersome on complex robot systems. **Xacro** (**XML Macros**) is an XML macro language created to make these robot description files easier to read and maintain. Xacro helps you reduce the duplication of information within the file.

For our first robot model, we will build a URDF file for a two-wheeled differential drive robot. The model will be created incrementally, and we will view the results at each step in rviz. When our simple two-wheeled robot is complete, we will add Gazebo formatting and view the model in Gazebo. In *Chapter 5*, *Creating Your First Robot Arm (in Simulation)*, we will expand our knowledge of URDF files and build a simple robot arm model using the Xacro notation.

Downloading the ros_robotics code

You can download the example code files and other support material for this book from `www.PacktPub.com`.

If you download the `ros_robotics` package from the Packt website, replace the entire `~/catkin_ws/src/ros_robotics` directory with the downloaded package. Instead, if you plan to enter the code from this book, begin by creating a `/urdf` directory under your `ros_robotics` package directory:

```
$ cd ~/catkin_ws/src/ros_robotics
$ mkdir urdf
$ cd urdf
```

Creating a robot chassis

Two basic URDF components are used to define a tree structure that describes a robot model. The **link** component describes a rigid body by its physical properties (dimensions, position of its origin, color, and so on). Links are connected together by **joint** components. Joint components describe the kinematic and dynamic properties of the connection (that is, links connected, types of joint, axis of rotation, amount of friction and damping, and so on). The URDF description is a set of these link elements and a set of the joint elements connecting the links together.

The first component of our robot is a simple chassis box. The downloaded `dd_robot.urdf` file contains the code for this exercise. Alternately, you can enter the code portion using your favorite editor and save the file as `dd_robot.urdf` to your `~/catkin_ws/src/ros_robotics/urdf` directory:

```
<?xml version='1.0'?>
<robot name="dd_robot">

  <!-- Base Link -->
  <link name="base_link">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
          <box size="0.5 0.5 0.25"/>
      </geometry>
    </visual>
  </link>

</robot>
```

XML comments are bracketed by `<!--` and `-->`.

This XML code defines a robot labeled `dd_robot` that has one link (also known as part) whose visual component is a box `0.5` meters long, `0.5` meters wide, and `0.25` meters tall. The box is centered at the origin (`0`, `0`, `0`) of the environment with no rotation in the **roll, pitch, or yaw** (**rpy**) axes. The link has been labeled `base_link`, and our model will use this box as the link on which our other links are defined. (A `base_link` link should be identified as the URDF root link to create the beginning of the robot's kinematic chain.) This XML is quite a bit of code for a simple box, but it will soon become even more complicated.

Using roslaunch

Roslaunch is a ROS tool that makes it easy to launch multiple ROS nodes as well as set parameters on the ROS Parameter Server. Roslaunch configuration files are written in XML and typically end in a `.launch` extension. In a distributed environment, the `.launch` files also indicate the processor the nodes should run on.

The `roslaunch` syntax is as follows:

```
$ roslaunch <package_name> <file.launch>
```

To use `roslaunch` for our URDF file, you will need to use one of the following ways:

- Download the `ddrobot_rviz.launch` file from the `ros_robotics/launch` directory from this book's website
- Create a `launch` directory under the `ros_robotics` package and create the `ddrobot_rviz.launch` file from the following XML code:

```
<launch>
  <!-- values passed by command line input -->
  <arg name="model" />
  <arg name="gui" default="False" />

  <!-- set these parameters on Parameter Server -->
  <param name="robot_description"
    textfile="$(find ros_robotics)/urdf/$(arg model)" />
  <param name="use_gui" value="$(arg gui)"/>

  <!-- Start 3 nodes: joint_state_publisher,
    robot_state_publisher and rviz -->
    <node name="joint_state_publisher"
    pkg="joint_state_publisher"
    type="joint_state_publisher" />

    <node name="robot_state_publisher"
    pkg="robot_state_publisher"
```

```
        type="state_publisher" />

        <node name="rviz" pkg="rviz" type="rviz"
        args="-d $(find ros_robotics)/urdf.rviz"
        required="true" />
    </launch>
```

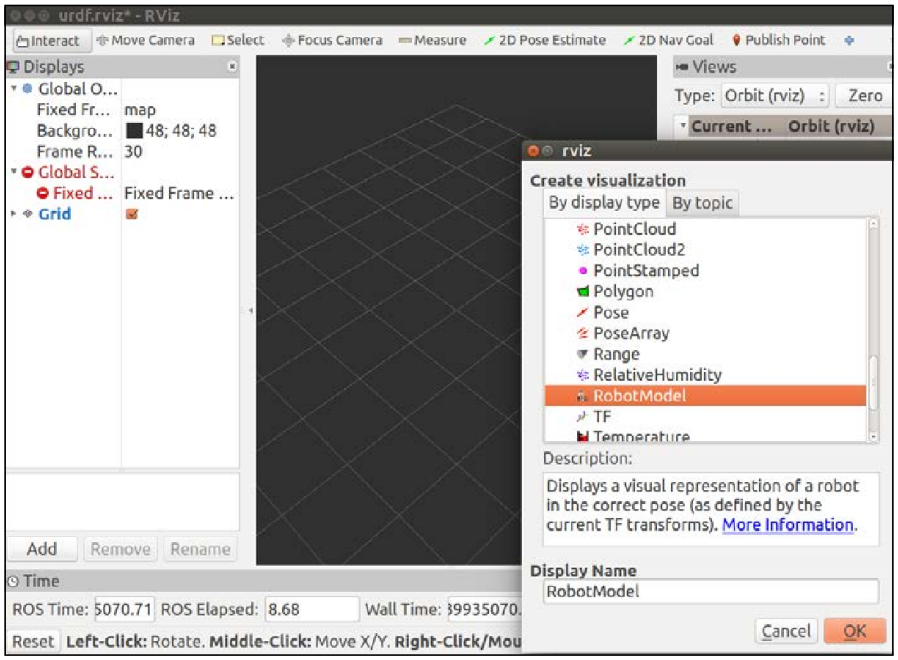
This `roslaunch` file performs the following:

- Loads the model specified in the command line into the Parameter Server.
- Starts nodes that publish the JointState and transforms (discussed later in this chapter).
- Starts rviz with a configuration file (`urdf.rviz`). It is important to use the `urdf.rviz` file that came with the book example code or save your own `urdf.rviz` file from rviz to be used with this launch file.

Type the following command to see the robot model in rviz:

```
$ roslaunch ros_robotics ddrobot_rviz.launch model:=dd_robot.urdf
```

At this point, your rviz screen should resemble one of the following two screenshots. Look carefully for a visual representation of your `dd_robot` box in the main screen and examine it under the **Displays** panel to decide how to proceed. Does your rviz screen look like the following screenshot:



Rviz screen without urdf.rviz file

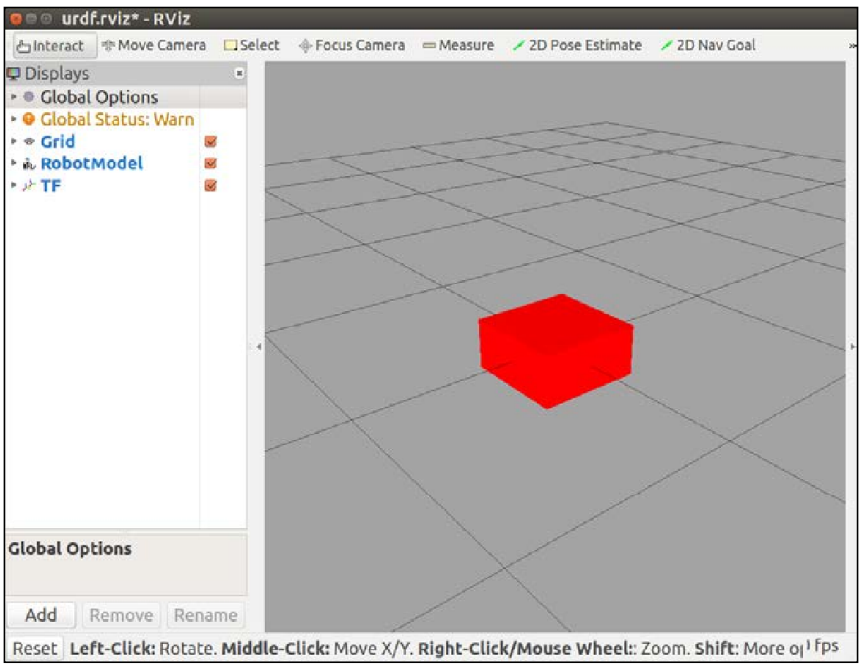
If your rviz screen looks like the the preceding screenshot with no box on the main screen and no **Robot Model** or **TF** under the **Displays** panel, then perform the following three steps in any order:

- Select the **Add** button under the **Displays** panel and add **RobotModel**
- Select the **Add** button under the **Displays** panel and add **TF**
- Select the field next to **Fixed Frame** (under **Global Options**), which in the preceding screenshot says **map**, and type in `base_link`

(The preceding screenshot shows the **Add** menu with the **RobotModel** selection highlighted.) When all the three steps are completed, your rviz screen will look similar to the following screenshot.

When you go to **File | Quit** to close your rviz session, you will be asked whether you want to save the configuration to a `urdf.rviz` file and it is recommended that you do. If you do not, you will have to perform the previous three steps each time to see your **RobotModel** and **TF** frames.

For the users who copied the `urdf.rviz` file from the book example code, the rviz screen will come up and and look like this:



dd_robot.urdf in rviz

Things to note:

- The fixed frame is a transform frame where the center (origin) of the grid is located.
- In URDF, the `<origin>` tag defines the reference frame of the visual element with respect to the reference frame of the link. In `dd_robot.urdf`, the visual element (the box) has its origin at the center of its geometry by default. Half of the box is above the grid plane and half is below.
- The rviz display configuration has been changed to remove the **View** and **Time** displays. This configuration is defined in the `urdf.rviz` file that comes with the book's example code (refer to the `.launch` file commands).
- The **RobotModel** and **TF** displays have been added under the **Displays** panel. Under **RobotModel**, notice the following:
    - Robot description: `robot_description` is the name of the ROS parameter where the URDF file is stored on the Parameter Server. The description of the links and joints and how they are connected is stored here.

Adding wheels

Now, let's add shapes and links for wheels on our robot. When we add link elements to the URDF file, we must add joints to describe the relationship between the links. Joint elements define whether the joint is flexible (movable) or inflexible (fixed). For flexible joints, the URDF describes the kinematics and dynamics of the joint as well as its safety limits. In URDF, there are six possible joint types, which are as follows:

- **Fixed**: This is not really a joint because it cannot move. All degrees of freedom are locked. This type of joint does not require the axis, calibration, dynamics, limits, or safety controller.
- **Revolute**: This joint rotates around one axis and has a range specified by the upper and lower limits.
- **Continuous**: This is a continuous hinge joint that rotates around the axis and has no upper and lower limits.
- **Prismatic**: This is a sliding joint that slides along the axis and has a limited range specified by the upper and lower limits.
- **Floating**: This joint allows motion for all six degrees of freedom.
- **Planar**: This joint allows motion in a plane perpendicular to the axis.

For our robot wheels, we require continuous joints, which means that they can respond to any rotation angle from negative infinity to positive infinity. They are modeled like this so that they can rotate in both directions forever.

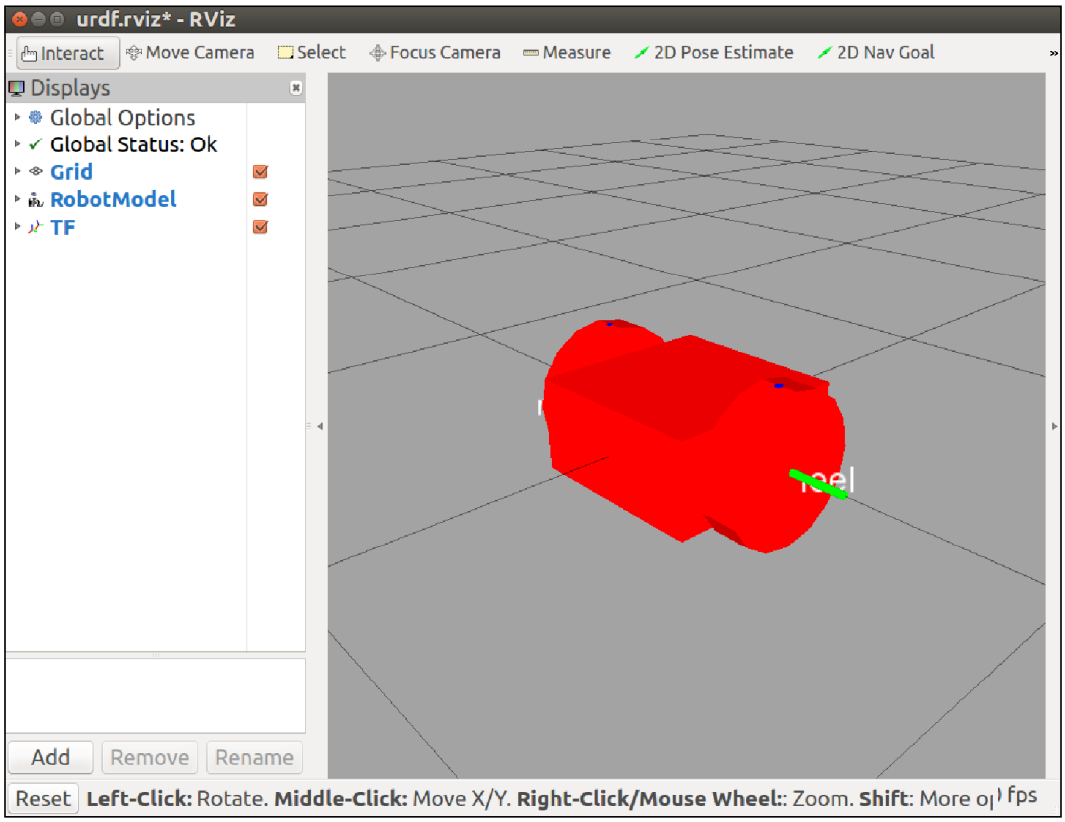
The downloaded `dd_robot2.urdf` file contains the XML code for this exercise. Alternately, you can enter the new code portion to your previous URDF file to create the two wheels (lines from the previous code have been left in or omitted and new code has been highlighted):

```
<?xml version='1.0'?>
<robot name="dd_robot">
  <!-- Base Link -->
  <link name="base_link">
  …
  </link>

  <!-- Right Wheel -->
  <link name="right_wheel">
    <visual>
      <origin xyz="0 0 0" rpy="1.570795 0 0" />
      <geometry>
          <cylinder length="0.1" radius="0.2" />
      </geometry>
    </visual>
  </link>
  <joint name="joint_right_wheel" type="continuous">
    <parent link="base_link"/>
    <child link="right_wheel"/>
    <origin xyz="0 -0.30 0" rpy="0 0 0" />
    <axis xyz="0 1 0" />
  </joint>

  <!-- Left Wheel -->
  <link name="left_wheel">
    <visual>
      <origin xyz="0 0 0" rpy="1.570795 0 0" />
      <geometry>
          <cylinder length="0.1" radius="0.2" />
      </geometry>
    </visual>
  </link>
  <joint name="joint_left_wheel" type="continuous">
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin xyz="0 0.30 0" rpy="0 0 0" />
    <axis xyz="0 1 0" />
  </joint>
</robot>
```

Run your rviz `roslaunch` command:

```
$ roslaunch ros_robotics ddrobot_rviz.launch model:=dd_robot2.urdf
```

Rviz should come up and look like this:



dd_robot2.urdf in rviz

Things to note in the URDF:

- Each wheel is defined visually as a cylinder of radius `0.2` meters and length of `0.1` meters. The wheel's visual origin defines where the center of the visual element should be, relative to its origin. Each wheel's origin is at (`0`, `0`, `0`) and is rotated by `1.560795` radians (= pi/2 = 90 degrees) about the *x* axis.
- The joint is defined in terms of a parent and a child. The URDF file is ultimately a tree structure with one root link. The `base_link` link is our robot's root link with the wheel's position dependent on the base_link's position.

- The wheel joint is defined in terms of the parent's reference frame. Therefore, the wheel's joint origin is `0.30` meters in the *x* direction for the left wheel and `-0.30` meters for the right wheel.
- The axis of rotation is specified by an `xyz` triplet, indicating that the wheel's joint axis of rotation is around the *y* axis.
- These `<joint>` elements define the complete kinematic model of our robot.

Adding a caster

In the next step, we will add a caster to the front of our robot in order to keep the robot chassis balanced. The castor will only be a visual element added to the chassis and not a joint. The caster will slide along the ground plane as the robot's wheels move.

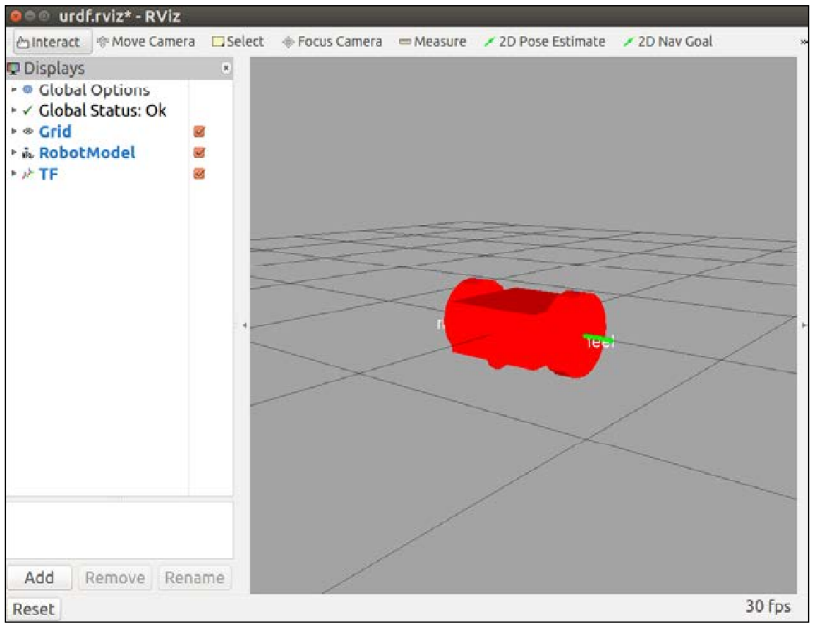
The downloaded `dd_robot3.urdf` file contains the XML code for this exercise. Alternately, you can enter the new code portion to your previous URDF file(new code has been highlighted):

```
<?xml version='1.0'?>
<robot name="dd_robot">

  <!-- Base Link -->
  …
    <!-- Caster -->
    <visual name="caster">
      <origin xyz="0.2 0 -0.125" rpy="0 0 0" />
      <geometry>
        <sphere radius="0.05" />
      </geometry>
    </visual>

  </link>
  <!-- Right Wheel -->
  …
  <!-- Left Wheel -->
  …
</robot>
```

Run your rviz `roslaunch` command:

```
$ roslaunch ros_robotics ddrobot_rviz.launch model:=dd_robot3.urdf
```

Rviz should come up and look like this:



dd_robot3.urdf in rviz

Things to note in the URDF file:

- The caster is defined visually as a sphere with a radius of `0.05` meters. The center of the caster is `0.2` meters in the *x* direction and `-0.125` meters in the *z* direction with respect to the origin of the `base_link`.

Adding color

A completely red robot has parts that are not distinctive enough; we will add some color to our model.

The downloaded `dd_robot4.urdf` file contains the XML code for this exercise. Alternately, you can enter the new code portions to your previous URDF file (new code has been highlighted):

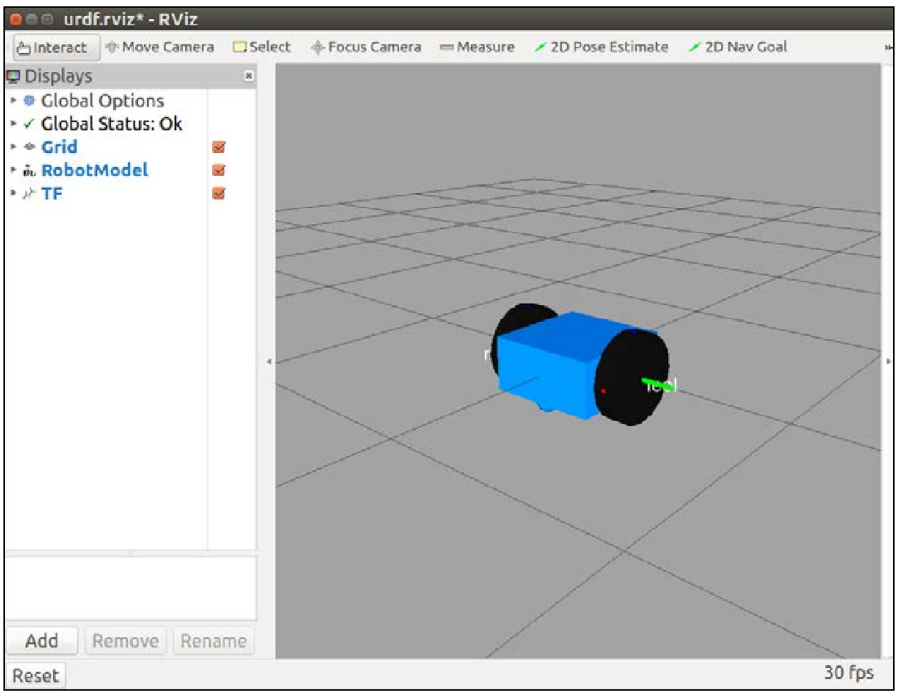
```
<?xml version='1.0'?>
<robot name="dd_robot">

  <!-- Base Link -->
  ...
      <material name="blue">
        <color rgba="0 0.5 1 1"/>
```

```
        </material>
      </visual>

      <!-- Caster -->
      …
  <!-- Right Wheel -->
  …
      <material name="black">
        <color rgba="0.05 0.05 0.05 1"/>
      </material>
    </visual>
…
  <!-- Left Wheel -->
  …
      <material name="black"/>
    </visual>
  …
</robot>
```

Run your rviz `roslaunch` command:

```
$ roslaunch ros_robotics ddrobot_rviz.launch model:=dd_robot4.urdf
```

Rviz should look like the following screenshot:



dd_robot4.urdf in rviz

Things to note in the URDF file:

- The `<material>` tag can define `<color>` in terms of red/green/blue/alpha, each in the range of [0,1]. Alpha is the transparency level of the color. An alpha value of 1 is opaque and 0 is transparent. Once specified and labeled with a name, the material name can be reused without specifying the color values. (For example, note that the left wheel does not have a `<color rgba>` tag because it has been defined in the right wheel visual link.)
- Although the book may show this picture in shades of gray, the chassis of the robot is now blue and the wheels are black.

Adding collisions

Next, we will add the `<collision>` properties to each of our `<link>` elements. Even though we have defined the visual properties of the elements, Gazebo's collision detection engine uses the collision property to identify the boundaries of the object. If an object has complex visual properties (such as a mesh), a simplified collision property should be defined in order to improve the collision detection performance.

The downloaded `dd_robot5.urdf` file contains the XML code for this exercise. Alternately, you can enter the new code portions to your previous URDF file (new code has been highlighted):

```
<?xml version='1.0'?>
<robot name="dd_robot">

  <!-- Base Link -->
  …
    <!-- Base collision -->
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
          <box size="0.5 0.5 0.25"/>
      </geometry>
    </collision>

    <!-- Caster -->
    …
    <!-- Caster collision -->
    <collision>
      <origin xyz="0.2 0 -0.125" rpy="0 0 0" />
      <geometry>
        <sphere radius="0.05" />
      </geometry>
    </collision>
```

```
    </link>

    <!-- Right Wheel -->
    …
      <!-- Right Wheel collision -->
      <collision>
        <origin xyz="0 0 0" rpy="1.570795 0 0" />
        <geometry>
            <cylinder length="0.1" radius="0.2" />
        </geometry>
      </collision>
    …

    <!-- Left Wheel -->
    …
      <!-- Left Wheel collision -->
      <collision>
        <origin xyz="0 0 0" rpy="1.570795 0 0" />
        <geometry>
            <cylinder length="0.1" radius="0.2" />
        </geometry>
      </collision>
  </robot>
```

Adding the `<collision>` property does not change the visual model of the robot, and the rviz display will look the same as in the previous screenshot.

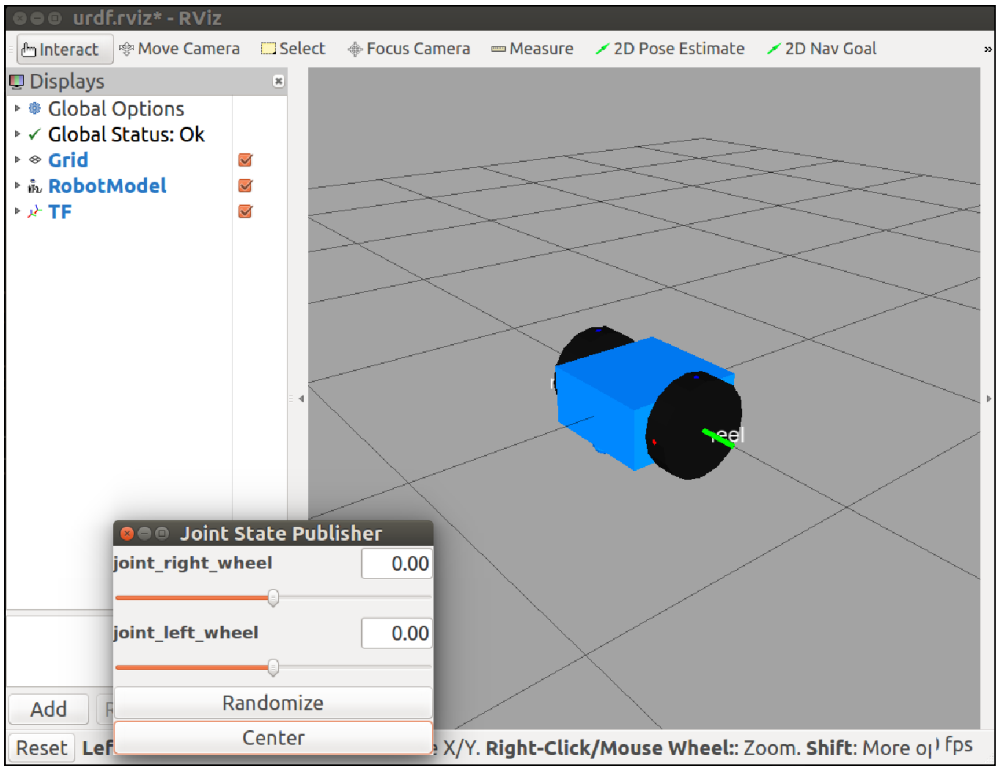
Moving the wheels

Now that we have the right and left wheel joints defined and we can see them clearly, we will bring up the GUI pop-up screen to control these joints. In the `ddrobot_rviz.launch` file, we start three ROS nodes: `joint_state_publisher`, `robot_state_publisher`, and rviz. The `joint_state_publisher` node finds all of the non-fixed joints and publishes a JointState message with all those joints defined. So far, the values in the JointState message have been constant, keeping the wheels from rotating. We bring up a GUI interface in rviz to change the value of each JointState and watch the wheels rotate.

Add the `gui` field to the rviz `roslaunch` command:

```
$ roslaunch ros_robotics ddrobot_rviz.launch model:=dd_robot5.urdf
gui:=True
```

Rviz should look like the following screenshot:



dd_robot5.urdf in rviz

Things to note:

- The joint positions in the window are sliders. The wheel joints are defined as continuous but this GUI limits each slider's value from –Pi to +Pi. Play with the sliders and see how the wheels move.
- The **Randomize** button will select a random value for both the joints.
- The **Center** button will move both the joints to the zero position. (Visually, the blue dot on both the wheels should be at the top).

A word about tf and robot_state_publisher

A robotic system is made up of a collection of 3D coordinate frames for every component in the system. In our `dd_robot` model, there is a base coordinate frame and a frame for each wheel that relates back to the base coordinate frame. The model's coordinate frames are also related to the world coordinate frame of the 3D environment. The tf package is the central ROS package used to relate the coordinate frames of our robot to the 3D simulated environment (or a real robot to its real environment).

The `robot_state_publisher` node subscribes to the JointState message and publishes the state of the robot to the tf transform library. The tf transform library maintains the relationships between the coordinate frames of each component in the system over time. The `robot_state_publisher` node receives the robot's joint angles as inputs and computes and publishes the 3D poses of the robot links. Internally, the `robot_state_publisher` node uses a kinematic tree model of the robot built from its URDF. Once the robot's state gets published, it is available to all components in the system that also use tf.

Adding physical properties

With the additional physical properties of mass and inertia, our robot will be ready to be launched in the Gazebo simulator. These properties are needed by Gazebo's physics engine. Specifically, every `<link>` element that is being simulated needs an `<inertial>` tag.

The two sub-elements of the inertial element we will use are as follows:

- `<mass>`: This is the weight defined in kilograms.
- `<inertia>`: This frame is a 3 x 3 rotational inertia matrix. Because this matrix is symmetrical, it can be represented by only six elements. The six highlighted elements are the six element `<inertia>` values. The other three values are not used.

ixx	**ixy**	**ixz**
ixy	**iyy**	**iyz**
ixz	iyz	**izz**

Using Wikipedia's list of moment of inertia tensors (`https://en.wikipedia.org/wiki/List_of_moments_of_inertia`), which provides the equations for the inertia of simple geometric primitives, such as a cylinder, box, and sphere. We use these equations to compute the inertia values for the model's chassis, caster, and wheels.

Do not use inertia elements of zero (or almost zero) because real-time controllers can cause the robot model to collapse without warning, and all links will appear with their origins coinciding with the world origin.

The downloaded `dd_robot6.urdf` file contains the XML code for this exercise. Alternately, you can enter the new code portions in your previous URDF file (new code has been highlighted):

```
<?xml version='1.0'?>
<robot name="dd_robot">

  <!-- Base Link -->
  …
    <inertial>
      <mass value="5"/>
      <inertia ixx="0.13" ixy="0.0" ixz="0.0"
               iyy="0.21" iyz="0.0" izz="0.13"/>
    </inertial>

    <!-- Caster -->
    …
    <inertial>
      <mass value="0.5"/>
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
               iyy="0.0001" iyz="0.0" izz="0.0001"/>
    </inertial>
  </link>

  <!-- Right Wheel -->
  …
    <inertial>
      <mass value="0.5"/>
      <inertia ixx="0.01"  ixy="0.0" ixz="0.0"
               iyy="0.005" iyz="0.0" izz="0.005"/>
    </inertial>
  …
  <!-- Left Wheel -->
  …
    <inertial>
      <mass value="0.5"/>
      <inertia ixx="0.01" ixy="0.0" ixz="0.0"
               iyy="0.005" iyz="0.0" izz="0.005"/>
    </inertial>
    …
</robot>
```

Adding the `<inertial>` property does not change the visual model of the robot, and the rviz display will look the same as the preceding screenshot.

Trying URDF tools

ROS provides command-line tools that can help verify and visualize information about your URDF. We will try out these tools on our robot URDF but first the tools must be installed on your computer system. Type the following command:

```
$ sudo apt-get install liburdfdom-tools
```

check_urdf

`check_urdf` attempts to parse a URDF file description and either prints a description of the resulting kinematic chain or an error message:

```
$ check_urdf dd_robot6.urdf
```

The output of the preceding command is as follows:

robot name is: dd_robot

---------- Successfully Parsed XML ----------

root Link: base_link has 2 child(ren)

child(1): left_wheel

child(2): right_wheel

urdf_to_graphiz

The `urdf_to_graphiz` tool creates a graphviz diagram of a URDF file and a diagram in the `.pdf` format. Graphviz is an open source graph visualization software.
To execute urdf_to_graphiz, type:

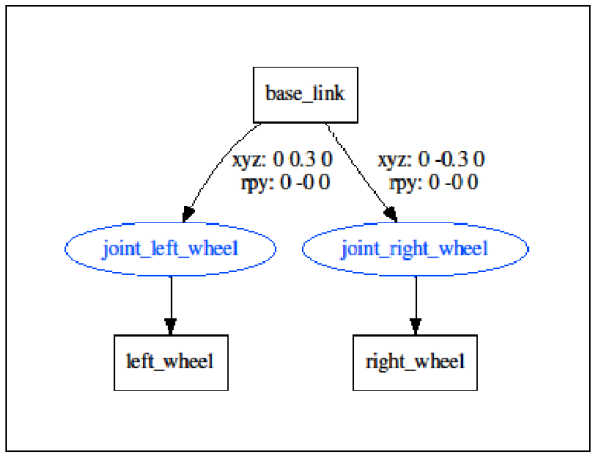
```
$urdf_to_graphiz dd_robot6.urdf
```

The output is as follows:

Created file dd_robot.gv

Created file dd_robot.pdf

The `dd_robot.pdf` file is shown as follows:



dd_robot.pdf

Now that we have a working URDF model of our two-wheeled robot, we are ready to launch it into Gazebo and move it around. First, we must make some modifications to our URDF file to add simulation-specific tags so that it properly works in Gazebo. Gazebo uses SDF, which is similar to URDF, but by adding specific Gazebo information, we can convert our `dd_robot` model file into an SDF-type format.

Gazebo

Gazebo is a free and open source robot simulation environment developed by Willow Garage. As a multifunctional tool for ROS robot developers, Gazebo supports the following:

- Designing of robot models
- Rapid prototyping and testing of algorithms
- Regression testing using realistic scenarios
- Simulation of indoor and outdoor environments
- Simulation of sensor data for laser range finders, 2D/3D cameras, kinect-style sensors, contact sensors, force-torque, and more
- Advanced 3D objects and environments utilizing **Object-Oriented Graphics Rendering Engine** (**OGRE**)
- Several high-performance physics engines (**Open Dynamics Engine** (**ODE**), Bullet, Simbody, and **Dynamic Animation and Robotics Toolkit** (**DART**)) to model the real-world dynamics

In this section, we will load our two-wheeled robot URDF into Gazebo to visualize it in a 3D environment. Gazebo allows you to take control of some aspects of our model without an external control program. In the later chapters, we will be using simulated versions of robots in Gazebo to control joints, visualize sensor data, and test control algorithms.

Installing and launching Gazebo

To run Gazebo requires a powerful graphics card and the appropriate drivers be installed on your computer.

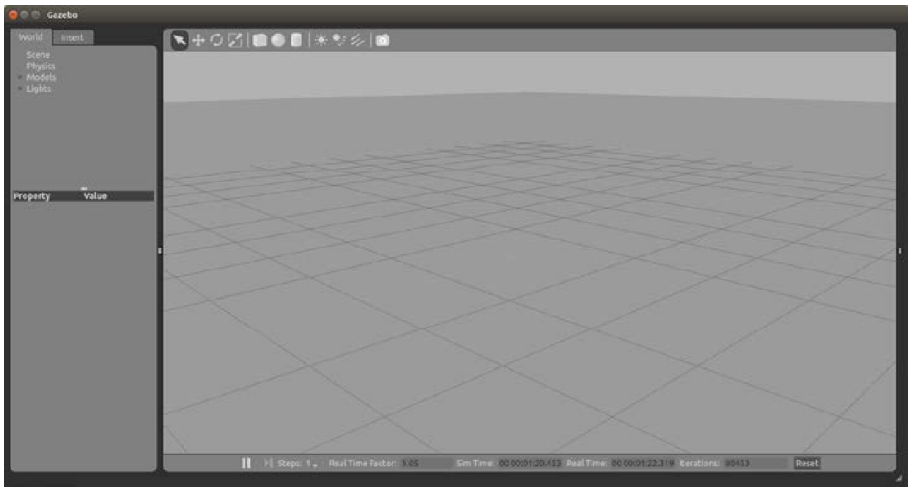
If you have trouble with running Gazebo, refer to `http://answers.gazebosim.org/questions/` or search the ROS forum at `http://answers.ros.org/questions/`.

Gazebo should have been installed on your computer as part of the ros-indigo-desktop-full installation, as described in the *Installing and launching ROS* section in *Chapter 1*, *Getting Started with ROS*. Gazebo 2 is the default version of Gazebo for ROS-Indigo/Ubuntu-Trusty and is the version recommended for the exercises in this book.

To test whether Gazebo has been installed correctly, open a terminal window and type the following command:

```
$ gazebo
```

This should display an environment similar to the following screenshot:



Gazebo main screen

If Gazebo has not been installed, refer to the ROS-Indigo installation instructions at `http://wiki.ros.org/indigo/Installation/Ubuntu` or the general Gazebo installation instructions at `http://gazebosim.org/tutorials?cat=install`. Make sure that you install the Gazebo 2 version.

The `$ gazebo` command runs two different executables: the Gazebo server and the Gazebo client. The Gazebo server `gzserver` will execute the simulation process, including the physics update loop and sensor data generation. This process is the core of Gazebo and can be used independently of any graphical interface. The Gazebo client `gzclient` command runs the Gazebo GUI. This GUI provides a nice visualization of simulation and handy controls for an assortment of simulation properties.

Tutorials for Gazebo can be found at `http://gazebosim.org/tutorials`.

Use the *Ctrl* + *C* keys to kill the terminal window process after you have closed the Gazebo window.

Important commands: If at any time, your command generates a warning or error command, type `$ rosnode list` to determine whether there are any active nodes still lingering after you have attempted to shut down Gazebo. If there are still active nodes, use the `$ rosnode kill` command to list the active nodes. Next, select the number of the ROS nodes that you wish to kill.

Using roslaunch with Gazebo

Roslaunch is a standard method used to start Gazebo with world files and robot URDF models. To perform a basic test of Gazebo, an empty Gazebo world can be brought up for the following command:

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

This test will verify that Gazebo has been installed properly. If you wish to try other demo worlds, including the `gazebo_ros` package, try substituting one of the following with `empty_world.launch` in the previous command:

- `willowgarage_world.launch`
- `mud_world.launch`
- `shapes_world.launch`
- `rubble_world.launch`

For the exercises in this chapter, we created our own world, `ddrobot.world`. This world consists of a ground plane and two construction cones for you to drive the robot around. You will find this file in the `ros_robotics` package under `ros_robotics/worlds`. We will launch our `dd_robot` model into this world using the `ddrobot_gazebo.launch` launch file found in the `ros_robotics/launch` directory.

Using Gazebo

The Gazebo GUI is similar to rviz in many ways. The central window provides the view for Gazebo's 3D world environment. The grid is typically configured to be the ground plane of the environment on which all the models are held due to gravity in the environment.

Gazebo also has the same cursor/mouse control as rviz, described in the *Using rviz* section. For Gazebo keyboard shortcuts, visit `http://gazebosim.org/hotkeys`.

Double-clicking on a spot in the environment will cause the display to be zoomed in to that spot.

For Gazebo, the standard units of measurement are in terms of meters and kilograms (like rviz).

The main screen of Gazebo is divided into four main display areas: the central window, the **World** and **Insert** panels to the left, the **Joints** panel to the right, and the **Simulation** panel at the bottom. Across the top of these display areas are the **Environment** toolbar and the main screen menu bar. Each of the display areas of Gazebo will be described in the following sections. This overview should provide basic familiarity with the Gazebo GUI.

Environment toolbar

The toolbar at the top of the Gazebo environment display provides the following functionalities:

- **Selection Mode** (*Esc*): This selects a model in the environment when the cursor is clicked on it or a drag box is wrapped around it. The **World** panel displays the model's properties. When selected, a white outline 3D box is drawn around the model. A yellow disc is placed where the cursor is clicked and becomes the focal point when the mouse controls are used to move around the scene. (Hitting the *Esc* key will also activate this mode.)

- **Translation Mode** (*T*): This selects a model in the environment when the cursor is clicked on it or a drag box is wrapped around it. A 3D axis (red-green-blue) is drawn and centered on the model. Use the cursor and left mouse button to move the model anywhere in the `x`, `y`, and `z` plane. (Hitting the *T* key will also activate this mode.)
- **Rotation Mode** (*R*): This selects a model in the environment when the cursor is clicked on it or a drag box is wrapped around it. A 3D sphere (red-green-blue) is drawn and centered around the model. Use the cursor and left mouse button to rotate the model in the roll, pitch, or yaw directions using one of the rings. (Hitting the *R* key will also activate this mode.)
- **Scale Mode** (*S*): This selects a model in the environment when the cursor is clicked on it or a drag box is wrapped around it. A 3D axis (red-green-blue) with square endpoints is drawn and centered on the model. Scaling of a model is currently limited to only simple shapes. (A warning message will be displayed in the terminal window if the user attempts to scale other models.)
- The next set of icons is used to create simple shapes in Gazebo: **Box**, **Sphere**, and **Cylinders**. Click on the icon and place the image shape anywhere in the 3D environment. The scale mode can then be used to resize the object.
- The next set of icons is used for lighting: **Point Light**, **Spot Light**, and **Directional Light**. Explore these if you wish to change the lighting and shadows in your environment.
- The last icon of the camera will take a **Screenshot** and save it to your `/home/<username>/.gazebo/pictures` directory.

World and Insert panels

The **World** panel to the left of the 3D environment provides access to all of the environment elements. These environment elements are **Scene**, **Physics**, **Models**, and **Lights**. By clicking on any of these labels, a properties panel will appear below the **World** panel with a properties list specific for that element.

The **Scene** selections allow the user to alter the **ambient environment**, the **background** and the **shadows**. The **Physics** selections check whether the physics engine is enabled. If the physics engine is enabled, the user can control the real-time **update rate**, **gravity**, and **constraints** under the **Physics** tab as well as other properties. The **Models** list will display all models active in the environment. When the `$ gazebo` command is used, the only active model will be the `ground_plane` model. By clicking on the **ground_plane** label, the properties displayed will be the model name, a checkbox to make the model **static**, the model's **pose** (**x**, **y**, **z**, **roll**, **pitch**, and **yaw**) in the environment, and its link information.

These details are lengthy so you can explore them at your leisure. The last element, **Lights**, displays all the light sources for the environment. For our default environment, the sun is the only source. The properties of the sun are its **pose**, **diffuse**, **specular**, **range**, and **attenuation**. Our primary interest for this book will be the **Models**.

The **Insert** tab is behind the **World** panel. The **Insert** panel accesses two locations to allow the user to select from a number of predefined models to be added to the environment. The first location, `/home/<username>/.gazebo/models`, is the user's repository of Gazebo models that they have selected from the main Gazebo repository. This repository is the second selection available at `http://gazebosim.org/models/`.

Joints panel

The panels to the right and left of the center display can be revealed or hidden using the three tiny rectangles in the black vertical strip. On the right side, the user can click and drag these three tiny rectangles to reveal the **Joints** panel. Under the Joints panel, there are three tabs labeled: **Force**, **Position**, and **Velocity**. There is also a **Reset** button to return your model to its original state (if possible).

To use these controls, the model must be selected to reveal the available model joints. For the joint control, we have the following values:

- **Force** values are in Newton meters
- **Position** values are in radians or degrees (make a selection from the drop-down window), **P Gain** (for proportional gain), **I Gain** (for integral gain), and **D Gain** (for derivative gain)
- **Velocity** values are in m/s (for meters per second), **P Gain** (for proportional gain), **I Gain** (for integral gain), and **D Gain** (for derivative gain)

The slider bar at the bottom of the **Joints** panel will help you see the information to the right of the display.

Main window menu bar

The top-most main window menu bar provides options under the basic **File**, **Edit**, **View**, **Window**, and **Help** headings as shown here:

- **File**: **Save World**, **Save World As**, and **Quit**
- **Edit**: **Reset Model Poses**, **Reset World**, and **Building Editor**

- **View**: **Grid**, **Transparent**, **Wireframe**, **Collisions**, **Joints**, **Center of Mass/Inertia**, **Contacts**, **Reset Camera**, **Full Screen**, and **Orbital View Control**
- **Window**: **Topic Visualization** and **Log Data**
- **Help**: **About**

These selections can be very useful. For example, if you wish to check the center of mass for your URDF in Gazebo, click on the **View** heading and select both the **Wireframe** and the **Center of Mass** options.

Simulation panel

At the bottom of the environment display is a handy tool used to run simulation scripts. It is useful when recording and playing back simulation runs.

Before we can load our `dd_robot` model into Gazebo, we must make a few modifications to the URDF file.

Modifications to the robot URDF

Gazebo expects the robot model file to be in SDF format. SDF is similar to the URDF, using some of the same XML descriptive tags. With the following modifications, Gazebo will automatically convert the URDF code into an SDF robot description. The following sections will describe the steps to be taken.

Adding the Gazebo tag

The `<gazebo>` tag must be added to the URDF to specify additional elements needed for simulation in Gazebo. This tag allows for identifying elements found in the SDF format that are not found in the URDF format. If a `<gazebo>` tag is used without a `reference=""` property, it is assumed that the `<gazebo>` elements refer to the whole robot model. The reference parameter usually refers to a specific robot link.

Other `<gazebo>` elements for both the links and joints can be applied to your robot but are not described in this book because of the extensive list and explanations of how they applied to the physics of Gazebo.

Refer to the Gazebo tutorial at `http://gazebosim.org/tutorials/?tut=ros_urdf` for a list of these elements and their usage.

Specifying color in Gazebo

The method of specifying link colors in rviz does not work in Gazebo since Gazebo has adopted OGRE's material scripts for coloring and texturing links. Therefore, **a Gazebo <material> tag must be specified for each link**. These tags can be placed in the model file just before the ending `</robot>` tag:

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

<gazebo reference="right_wheel">
  <material>Gazebo/Black</material>
</gazebo>

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

A word about <visual> and <collision> elements in Gazebo

Gazebo will not use `<visual>` elements the same as `<collision>` elements if you do not explicitly specify them for each link. Instead, Gazebo will treat your link as invisible to laser scanners and collision checking. If your model ends up partially embedded in Gazebo's ground plane, you should check your `<collision>` elements.

Verifying a Gazebo model

The `dd_robot` URDF has been updated with the `<gazebo>` tags and `<material>` elements, as described earlier, and is stored in the downloaded file, `dd_robot.gazebo`. The `.gazebo` extension is used by the author to signify that this file is ready for use in Gazebo. An easy tool exists to check whether your URDF can be properly converted into an SDF. Simply run the following command:

```
$ gzsdf print dd_robot.gazebo
```

or

```
$ gzsdf print $(rospack find ros_robotics)/urdf/dd_robot.gazebo
```

This command outputs the entire SDF to the screen so you may wish to redirect the output to a file. The output will show you the SDF that has been generated from your input URDF as well as any warnings about the missing information required to generate the SDF.

Viewing the URDF in Gazebo

Viewing the `dd_robot` model in Gazebo requires a launch file obtained or created by one of the following ways:

- Using the downloaded `ddrobot_gazebo.launch` file from the `ros_robotics/launch` directory from the book's website
- Creating the `ddrobot_gazebo.launch` file from the following XML code:

```
<launch>
  <!-- We resume the logic in gazebo_ros package
empty_world.launch,
         changing only the name of the world to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name"
           value="$(find ros_robotics)/worlds/ddrobot.world"/>

    <arg name="paused" default="false"/>
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="debug" default="false"/>
  </include>

  <!-- Spawn dd_robot into Gazebo -->
  <node name="spawn_urdf" pkg="gazebo_ros"
        type="spawn_model" output="screen"
        args="-file $(find ros_robotics)/urdf/dd_robot.gazebo
              -urdf -model ddrobot" />

</launch>
```

This launch file inherits most of the necessary functionality from `empty_world.launch` from the `gazebo_ros` package. The only parameter that is changed is the `world_name` parameter by substituting the `ddrobot.world` world file. In addition to this, our URDF-based `dd_robot` model is launched into Gazebo using the ROS `spawn_model` service from the `gazebo_ros` ROS node. If you plan to reuse this code or share it, it is recommended that you add the dependency to your `package.xml` file for the `ros_robotics` package. The following statement should be added under dependencies:

```
<exec_depend>gazebo_ros</exec_depend>
```

The `ddrobot.world` world file contains a ground plane and two construction cones. This file can be found in the `ros_robotics/worlds` directory on the book's website, or you can create the `ddrobot.world` file from the following code:

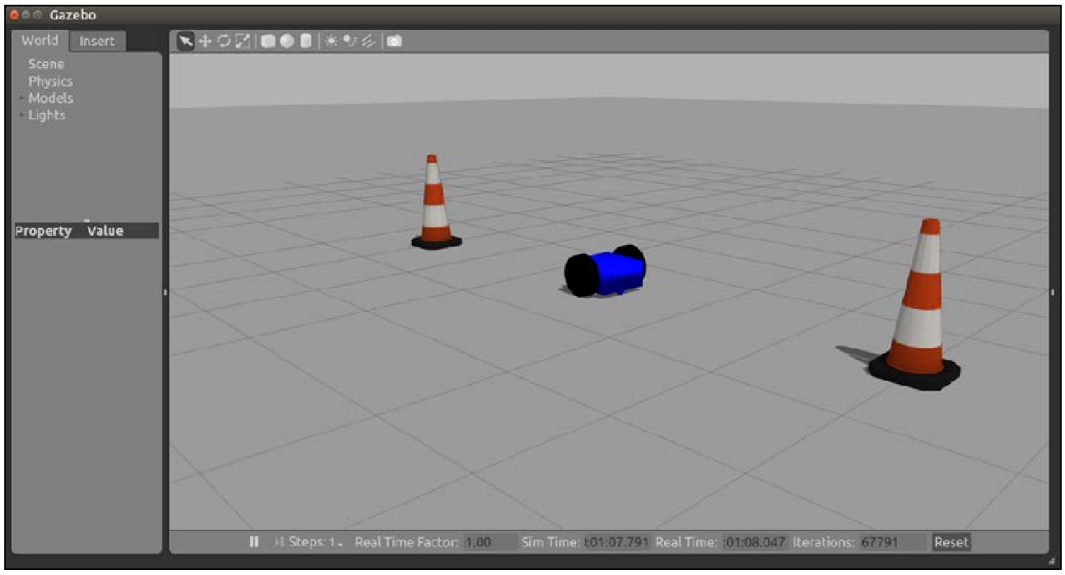
```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://sun</uri>
    </include>
    <include>
      <uri>model://construction_cone</uri>
      <name>construction_cone</name>
      <pose>-3.0 0 0 0 0 0</pose>
    </include>
    <include>
      <uri>model://construction_cone</uri>
      <name>construction_cone</name>
      <pose>3.0 0 0 0 0 0</pose>
    </include>
  </world>
</sdf>
```

The `ddrobot_gazebo.launch` file should be found in the `/launch` directory and `ddrobot.world` should be found in the `/worlds` directory of the `ros_robotics` ROS package.

Now we are ready to launch our `dd_robot` model in Gazebo by typing the following command:

```
$ roslaunch ros_robotics ddrobot_gazebo.launch
```

This command will launch both the Gazebo server and GUI client with the `dd_robot` model and world automatically launched inside the Gazebo environment. Gazebo will look similar to the following screenshot:



dd_robot.gazebo in Gazebo

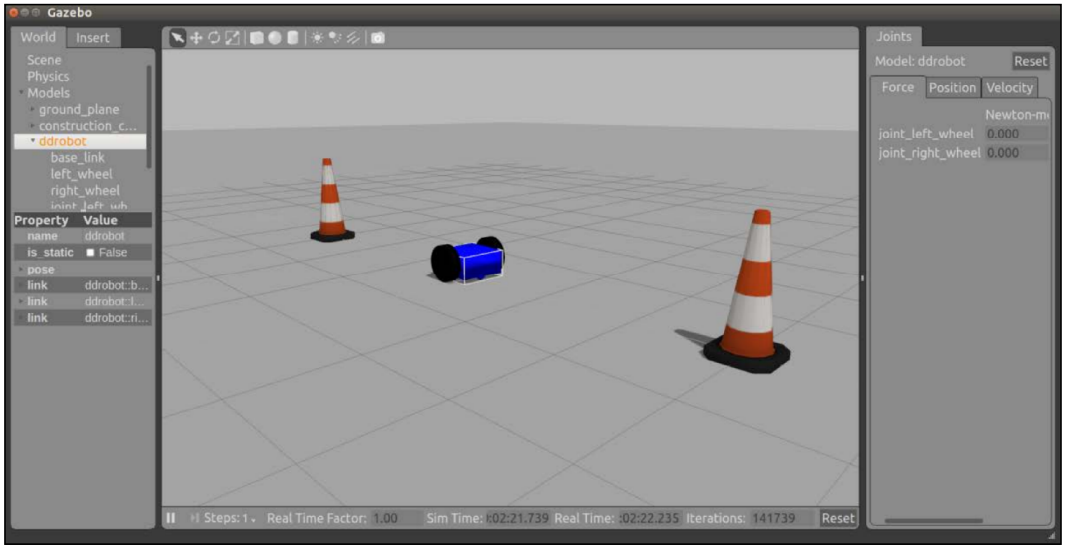
Tweaking your model

If your robot model behaves unexpectedly within Gazebo, it is likely because your model URDF needs further tuning to accurately represent its physics in Gazebo. Refer to the SDF user guide at `http://sdformat.org/spec` for more information on various properties available in Gazebo, which are also available in the URDF via the `<gazebo>` tag.

Moving your model around

To understand the physics of Gazebo, it is important to play with your model in Gazebo. Use the **Selection**, **Translation**, and **Rotation** modes on the Environment toolbar to move your model to different positions, and then watch how the gravity model works. If you are brave, you can even manipulate the environment to experiment with the relationship of the elements. For example, remove the ground plane and see what happens.

Simple joint control of our `dd_robot` model is possible by using the **Joints** panel, which is to the right of the center environment. In selection mode, click on the **ddrobot** model and the model will be highlighted with a white outline box. The **joint_left_wheel** and **joint_right_wheel** will appear under the tabbed sections with a value of **0.000** for each of the input windows. We will experiment by changing the values of the left and right wheel joints to see the `dd_robot` model move around on the ground plane. Values for **Force** and **Position** cause the robot to move; **Velocity** causes the robot to collapse. The following screenshot shows our `dd_robot` ready to be controlled via the **Joints** panel:



dd_robot.gazebo in Gazebo with the Joints panel

A greater control of our model can be achieved by adding transmission blocks to the URDF for the model joints. Gazebo plugins are also needed to simulate controllers that publish ROS messages for motor commands. A discussion of these advanced topics will be delayed until *Chapter 5*, *Creating Your First Robot Arm (in Simulation)* when the reader has a better understanding of ROS messages for control of mobile robots. *Chapter 5*, *Creating Your First Robot Arm (in Simulation)* will walk you through the construction of a URDF/SDF for a robot arm with a joint control implemented via Gazebo plugins. The implementation of transmission blocks and plugins for our `dd_robot` model is left as an exercise on completion of *Chapter 5*, *Creating Your First Robot Arm (in Simulation)*.

Other ROS simulation environments

Gazebo is only one simulator that can interface to ROS and ROS models. A list of other simulators, both open source and commercial, is provided along with a website reference:

- MATLAB with Simulink is a commercially available, multi-domain simulation and modeling design package for dynamic systems. It provides support for ROS through its Robotics System Toolbox (`http://www.mathworks.com/hardware-support/robot-operating-system.html`).
- Stage is an open source 2D simulator for mobile robots and sensors (`http://playerstage.sourceforge.net/doc/Stage-3.2.1/`).
- **Virtual Robot Experimentation Platform** (**V-REP**) is a commercially available robot simulator with an integrated development environment. Developed by Coppelia Robotics, V-REP lends itself to many robotic applications (`http://www.coppeliarobotics.com/`).

Summary

Your first robot design has been a simple two-wheeled differential drive model defined in URDF. There are many other properties that can be defined in the URDF file, and you are free to extend the `dd_robot` model. This introductory exercise was provided so that the elements of simulation can be understood by the reader. In *Chapter 3, Driving Around with TurtleBot*, we will use a simulated and a real TurtleBot to explore a variety of ROS control methods for mobile robot navigation. The rqt toolset will be introduced and used to monitor and control the TurtleBot's movements.

In *Chapter 5, Creating Your First Robot Arm (in Simulation)*, we will extend our understanding of URDF by learning more about Xacro. We will build a Xacro file to define a robot model for a robot arm.

3
Driving Around with TurtleBot

It is time for a real ROS robot! A robot called TurtleBot will be discussed and described both in simulation and as the real robot. In this chapter, you will learn how to move TurtleBot as a simulated robot and as the real robot. Even if you do not have a real TurtleBot, the examples in this chapter will teach you the techniques to control a mobile robot.

After introducing the TurtleBot, we will cover the following topics:

- Loading the TurtleBot simulation software and using Gazebo with TurtleBot
- Setting up your system to control a real TurtleBot from its own netbook computer or wirelessly from a remote computer
- Controlling the movement of the TurtleBot with ROS terminal commands or using the keyboard for control in teleoperation
- Creating a Python script which, when executed, moves the TurtleBot
- Using rqt tools to provide a GUI that aids the user in analyzing robot programs and also monitoring and controlling the robot
- Exploring an environment using TurtleBot's odometry data
- Executing the automatic docking program of TurtleBot

Introducing TurtleBot

TurtleBot is a mobile robot that can be purchased as a kit or fully assembled. Several companies in North America and around the world sell TurtleBots. The TurtleBot2 model is shown in the following image:

Turtlebot 2

A list of manufacturers can be found at: `http://www.turtlebot.com/manufacturers/`

The main items that comprise the TurtleBot 2 model from bottom to top in the preceding image are as follows:

- A mobile base that also serves as support for the upper stages of the robot
- A netbook resting on a module plate
- Another module plate used to hold items
- A vision sensor with a color camera and 3D depth sensor
- The top most module plate used to hold items

We will discuss the main items briefly here and provide more details for the base and the netbook later in this chapter. Overall, the TurtleBot model stands about 420 mm (16.5 inches) high and the base is approximately 355 mm (14 inches) in diameter.

The particular base in the authors' TurtleBot is a Kobuki mobile platform produced by the Yujin Robot company. TurtleBot rests on the floor on two wheels and a caster. The base is configured as a differential drive base, which means that when the TurtleBot is moving, the rotational velocity of the wheels can be controlled independently. So, for example, TurtleBot can move back and forth in a straight line when the wheels are driven in the same direction, **clockwise** (**CW**) or **counterclockwise** (**CCW**), with the same rotational velocity. If the wheels turn at different rotational velocities, TurtleBot can make turns as the velocity of the wheels is controlled. More details are available at: `http://kobuki.yujinrobot.com/home-en/about/`.

A model of a differential drive robot was built in the *Building a differential drive robot URDF* section in *Chapter 2*, *Creating Your First Two-Wheeled ROS Robot (in Simulation)*. The base has a battery pack and various power connections for accessories, including a USB connection and power plug for the netbook. TurtleBot also comes with a separate docking station for charging the Kobuki base.

The netbook is essentially a laptop computer but is lightweight with a small screen compared to many laptops. The netbook purchased with TurtleBot has Ubuntu and ROS packages installed. For a remote control, the netbook is connected via Wi-Fi to a network and a remote computer. There are USB ports used to plug in the vision sensor or other accessories. The setup of the network is described in the *Networking the netbook and remote computer* section of this chapter.

The vision sensor, as shown in the preceding image of TurtleBot2, is an Xbox 360 Kinect sensor manufactured by Microsoft. Originally designed for video games, the Kinect sensor is a popular vision and depth sensor for robotics.

The ROS wiki has a series of tutorials that cover TurtleBot, including the Gazebo simulator, at `http://wiki.ros.org/Robots/TurtleBot`.

Loading TurtleBot simulator software

This section deals with loading software packages for the TurtleBot simulator. The physical TurtleBot is not involved because these software packages are loaded on your laptop or desktop computer. It is assumed that Ubuntu 14.04 and ROS Indigo software are installed on the computer that you will use for the simulation. This installation is described in the *Installing and launching ROS* section in *Chapter 1, Getting Started with ROS*.

The procedure to load the TurtleBot software on your computer is documented in the tutorial at: `http://wiki.ros.org/turtlebot/Tutorials/indigo/PC%20Installation`

The installation described on the website will apply to the real TurtleBot netbook also, but we will only consider the simulation here. In a terminal window, type the following command:

```
$ sudo apt-get install ros-indigo-turtlebot ros-indigo-turtlebot-apps
ros-indigo-turtlebot-interactions ros-indigo-turtlebot-simulator ros-
indigo-kobuki-ftdi ros-indigo-rocon-remocon ros-indigo-rocon-qt-
library ros-indigo-ar-track-alvar-msgs
```

A large number of ROS packages are loaded by the `sudo apt-get` command. The groups are as follows:

- The TurtleBot software has ROS packages to simulate the TurtleBot and control the real TurtleBot. The TurtleBot simulator download includes the `turtlebot_gazebo` package.
- The Kobuki software consists of ROS packages used to drive or simulate the mobile base.
- The **rocon** (**robots in concert**) software is used for multirobot applications.
- The alvar software is used to track markers guiding a robot.

To view the TurtleBot packages downloaded in each category, type the following command:

```
$ rospack list | grep turtlebot
```

To see the packages that apply to the base, type the following command:

```
$ rospack list | grep kobuki
```

The other software can be viewed in a similar way.

Launching TurtleBot simulator in Gazebo

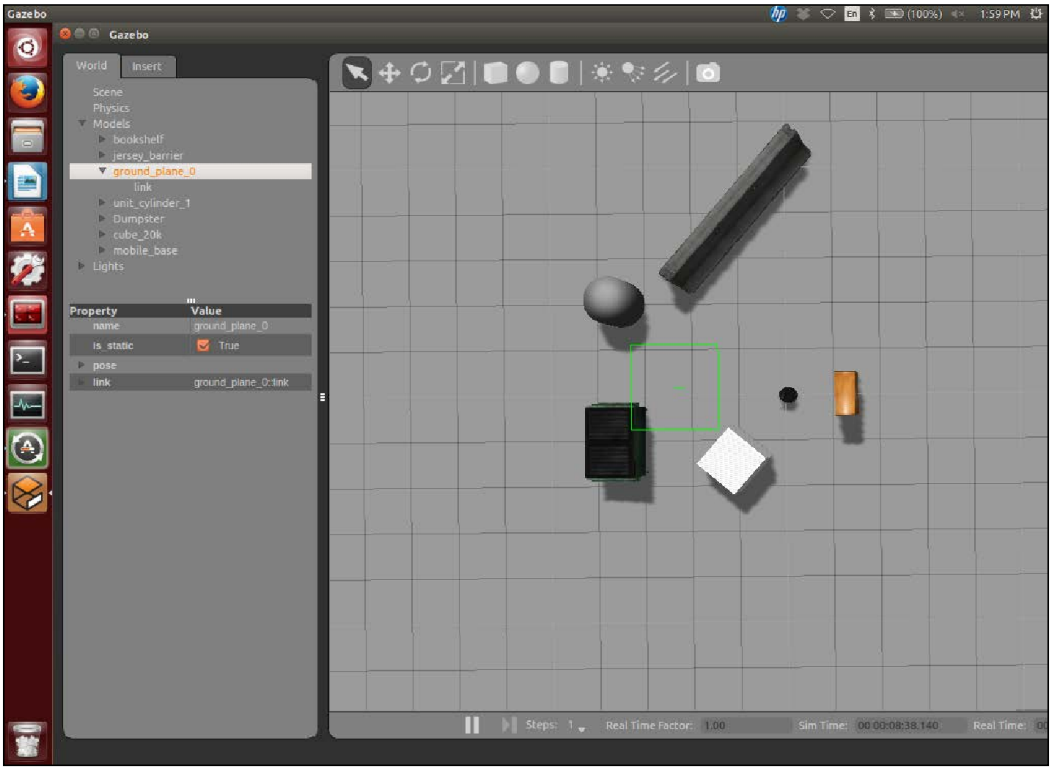
The simulator package Gazebo was introduced in *Chapter 2, Creating Your First Two-Wheeled ROS Robot (in Simulation)*. If you run the examples there using the differential drive robot, `dd_robot`, you should have a good understanding of Gazebo, including how to load models and worlds and manipulate the environment.

To run the simulator, you need to install the TurtleBot software, as described in the previous section. If you view the tutorials of TurtleBot Gazebo from the ROS wiki, make sure that you select Indigo since that should be your distribution of ROS.

To start the simulation, open a new terminal window and type the following command:

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

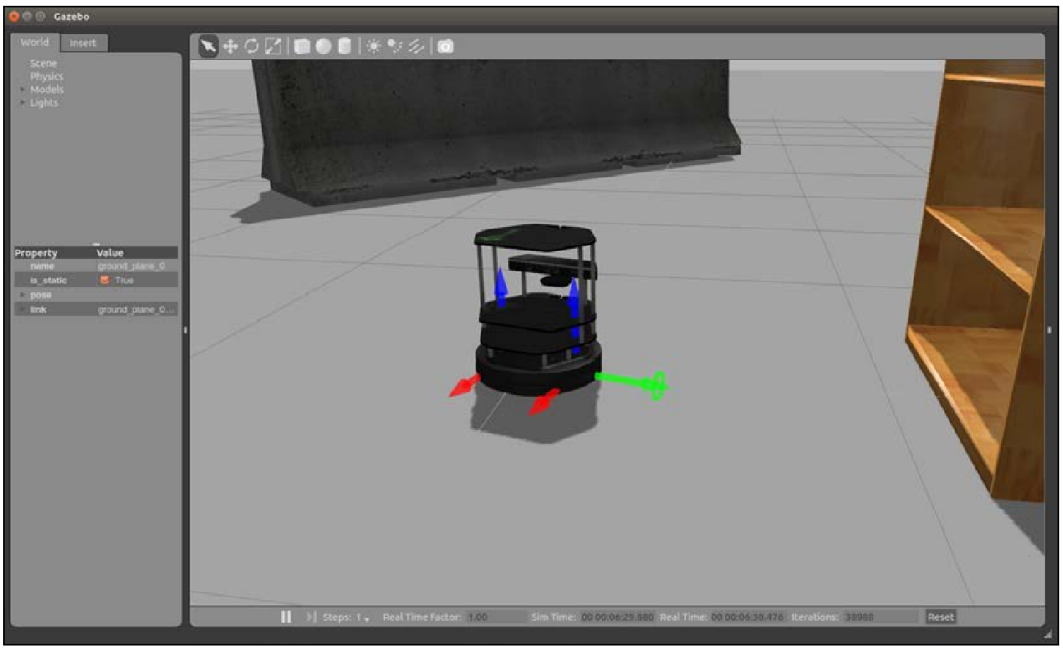
If all goes well, you will see a screenshot similar to this one:



TurtleBot simulated in Gazebo

If you do not see Gazebo start, refer to the following *Problems and troubleshooting* section. If that is the case, there are a few issues that may help you if you are having problems with the simulation and the use of Gazebo. TurtleBot is in the approximate center of the world view, as seen from an overhead camera.

Right-click on the scene over TurtleBot, and from the menu that appears, select **View** and **Joints** to see the *x*, *y*, and *z* axes used by the TurtleBot simulator. Of course, TurtleBot will only move along its own *x* axis and rotate about its *z* axis. The following screenshot shows the result of zooming in on the TurtleBot and displaying the axes from choosing **Joints**:



TurtleBot simulated with joint axes shown

If you have modified the scene, you can take a picture of it by left-clicking on the camera icon on the toolbar. You can save the world using the **Edit** tab on the menu bar.

Roll the cursor over the symbols on the Gazebo toolbar to obtain their meaning.

Problems and troubleshooting

The authors have tried their best to present the material in a clear manner so that you can follow along and achieve the same results. However, computers may differ in their abilities to run simulations that rely heavily on graphics, as Gazebo does.

We have run the code on relatively new laptops, older laptops, and on a powerful workstation. On a laptop, the response to commands from the keyboard may be slow, sometimes painfully so! Be patient: if the software is working, the TurtleBot will respond if commanded to move.

Some serious problems that may occur are as follows:

- On some older laptops, the hardware accelerator will not allow Gazebo to run, but this can be fixed by adding the following statement to `.bashrc` file, which disables the hardware accelerator:

  ```
  export LIBGL_ALWAYS_SOFTWARE=1
  ```

- You may need to execute the `roslaunch` command several times if Gazebo fails to come up on the first or second try
- Sometimes, it is necessary to close all the windows and start over if the system does not respond
- If you execute the `roscore` or `roslaunch` command, and you have changed your ROS Master address using an `export` command to be linked to the TurtleBot network, as described later in this chapter, you may receive an error message similar to this:

  ERROR: unable to contact ROS master at [http://<IP Address>:11311]

  The traceback for the exception was written to the log file

  It probably means that the ROS Master address is incorrect for your local machine. Usually, the problem can be fixed by issuing the following command:

  ```
  $ export ROS_MASTER_URI=http://localhost:11311
  ```

  This returns the ROS Master control to your local computer to run the simulator. You must run this `export` command in each new terminal window that is opened.

  Check the results for these environment variables with the following command:

  ```
  $ env | grep ROS
  ```

  Make sure that the `ROS_MASTER_URI` variable points to the proper location.

For more information on computer and network addresses, refer to the *Networking the netbook and remote computer* section in this chapter.

ROS commands and Gazebo

In the left-hand side pane of the Gazebo window, the list of models will appear when you click on **Models**. Notice, particularly, the `mobile_base` link. You can find the position and orientation of the base with the `rosservice` command. In a new terminal window, type the following command:

```
$ rosservice call gazebo/get_model_state '{model_name: mobile_base}'
```

The output of the preceding command is as follows:

```
pose:
  position:
    x: 0.00161336508139
    y: 0.0091790167961
    z: -0.00113098620237
  orientation:
    x: -5.20108036968e-05
    y: -0.00399736084462
    z: -0.0191615228716
    w: 0.999808408868
twist:
  linear:
    x: 9.00012388429e-06
    y: 6.54279879125e-05
    z: -1.4365465304e-05
  angular:
    x: -0.000449167550145
    y: 0.000197996689198
    z: -0.000470014447946
success: True
status_message: GetModelState: got properties
```

Looking at the position and orientation, we can see that the TurtleBot base is approximately at the center (*x=0*, *y=0*, *z=0*) of the grid as you can see by zooming out in the world view. Since so many decimal places are shown, it appears that the TurtleBot is off center.

However, if you notice, the first two decimal places in the position are zeros, and you can see that the values are very small, near zero. The orientation is also near zero and is represented in a special notation called a **quaternion**.

To see the complete list of services, type the following command:

```
$ rosservice list
```

You can also use the `rosnode list` or `rosmsg list` ROS commands, as was shown in *Chapter 1*, *Getting Started with ROS*, to list the nodes or messages.

With ROS commands, you can move the TurtleBot as we did with the turtle in Turtlesim in *Chapter 1*, *Getting Started with ROS*. First, find the topic that will control the `mobile_base` link since that is the name given in Gazebo's left panel:

```
$ rostopic list | grep mobile_base
```

The output is as follows:

/mobile_base/commands/motor_power

/mobile_base/commands/reset_odometry

/mobile_base/commands/velocity

/mobile_base/events/bumper

/mobile_base/events/cliff

/mobile_base/sensors/bumper_pointcloud

/mobile_base/sensors/core

/mobile_base/sensors/imu_data

/mobile_base_nodelet_manager/bond

Now you can find the message type published by the rostopic `/mobile_base/commands/velocity` that moves the base by typing the following command:

```
$ rostopic type /mobile_base/commands/velocity
```

The output is as follows:

geometry_msgs/Twist

From the previously shown screen printout of the `rosservice` command to call `gazebo/get_model_state`, you can see that the twist is a six-dimensional value although all six need not be specified. The values represent velocities, which in the case of the TurtleBot represent the linear velocity along its forward *x* axis and the angular velocity about the vertical *z* axis. A reference is available at

`https://en.wikipedia.org/wiki/Screw_theory`.

If you drive the turtle with a command, the possible motions are **linear** along its *x* direction and **angular rotation** about the *z* axis since the TurtleBot moves on the xy plane and cannot fly. To drive it forward, run the following command:

```
$ rostopic pub -r 10 /mobile_base/commands/velocity
\geometry_msgs/Twist '{linear: {x: 0.2}}'
```

Notice that the TurtleBot moves forward slowly until you stop it or it drives off the screen or it hits one of the objects in the environment. To stop its motion, press *Ctrl* + *C*. To bring the TurtleBot back, change the value of `x` to `x: -0.2` in the `rostopic` command and execute it.

If things go wrong, click on the **Edit** menu at the top of the Gazebo window and select **Reset** to reset the model pose.

There are many other features of Gazebo that can be explored, and you are encouraged to try various selections on the menu bar (**File**, **Edit**, **View**, **Window**, and **Help**). Also, you can open the rightmost third panel and change the values of **Force**, **Position**, or **Velocity** for the TurtleBot simulator.

Keyboard teleoperation of TurtleBot in simulation

A command to launch the teleop mode using the keyboard keys to move TurtleBot on the screen is as follows:

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

This command allows keyboard keys to maneuver the TurtleBot on the screen. The keys to command the motion are as follows:

Control Your Turtlebot!

Moving around:

u i o

j k l

m , .

q/z : increase/decrease max speeds by 10%

w/x : increase/decrease only linear speed by 10%

e/c : increase/decrease only angular speed by 10%

space key, k : force stop

anything else : stop smoothly

CTRL-C to quit

currently: speed 0.2 turn 1

Think of the letter `k` as the center of TurtleBot looking down on it. Start with the letter `i` to move the TurtleBot straight ahead along its *x* axis and try the other keys.

Remember to click on the window in which you executed the `roslaunch` command to move TurtleBot. This is termed **focusing** on the window.

The reference for TurtleBot teleoperation is found at:

`http://wiki.ros.org/turtlebot_gazebo/Tutorials/indigo/Explore%20the%20Gazebo%20world`

For now, we leave Gazebo and concentrate on installing software to control the real TurtleBot. However, even if you do not have access to a real TurtleBot, many of the commands and scripts that will be presented can also be used with the simulated TurtleBot. In fact, ideally, the Gazebo simulation should reflect the motion of the real TurtleBot in its environment.

For example, we later present a Python script that moves the real TurtleBot forward. You can use the command to run the script with Gazebo also.

Setting up to control a real TurtleBot

The TurtleBot system consists of the TurtleBot base and its netbook that rides along with the TurtleBot and a separate remote computer that is used to control the robot. The netbook and computer communicate wirelessly once a network connection is established. This section describes the setup of the system, including the network.

A brief overview of the steps to set up and test the TurtleBot are as follows:

1. Set up the netbook with Ubuntu 14.04 and ROS Indigo, and then load the TurtleBot software packages.
2. Set up the remote computer with similar software.
3. Test the TurtleBot in the standalone mode to assure proper operation.
4. Create the network of computers, being careful to define the TurtleBot netbook as the ROS Master to the remote computer.
5. Test the TurtleBot by communicating with commands wirelessly from the remote computer to the netbook of the TurtleBot.

The ROS wiki describes the installation of the ROS software for the TurtleBot netbook at:

```
http://wiki.ros.org/turtlebot/Tutorials/indigo/Turtlebot%20
Installation
```

With Ubuntu installed and the ros-indigo-desktop-full installation, the packages for the TurtleBot are installed with the following command:

```
$ sudo apt-get install ros-indigo-turtlebot ros-indigo-turtlebot-apps
ros-indigo-turtlebot-interactions ros-indigo-turtlebot-simulator ros-
indigo-kobuki-ftdi ros-indigo-rocon-remocon ros-indigo-rocon-qt-
library ros-indigo-ar-track-alvar-msgs
```

To link the Kobuki base to the device folder of Ubuntu, visit:

`http://wiki.ros.org/turtlebot/Tutorials/indigo/Kobuki%20Base`

To set up your netbook battery monitor for the TurtleBot, visit:

`http://wiki.ros.org/turtlebot/Tutorials/indigo/Netbook%20Battery%20Setup`

There are many tutorials that cover TurtleBot. There is a website devoted to the TurtleBot at `http://learn.turtlebot.com/`, with many interesting tutorials that cover the various aspects of the TurtleBot with details of setup, testing, and applications.

TurtleBot standalone test

Before we make an attempt to network the TurtleBot to a remote computer, it is wise to test the TurtleBot in the standalone mode to determine whether the software has been installed properly. Now disconnect the netbook from any networks. Once the TurtleBot and its netbook are powered up, you can test software by opening a new terminal window on the netbook and executing the following command:

```
$ roscore
```

This should respond with a screen output that ends with the following message:

started core service [/rosout]

If there are no errors indicated in the screen output the netbook is set up correctly with ROS.

After this, press *Ctrl* + *C* and close this terminal window. Open a new one to move the TurtleBot around, as was done previously in simulation. On the netbook, initialize the TurtleBot by typing the `roslaunch` command in the new window:

```
$ roslaunch turtlebot_bringup minimal.launch
```

Quite a bit of information is shown on the screen as the minimal launch proceeds, but most of this output is not of any concern now. The ROS Master is the netbook indicated by the following lines:

auto-starting new master

ROS_MASTER_URI=http://localhost:11311

This will launch `roscore` and initialize the TurtleBot for control when the movement commands are issued. The importance of the `ROS_MASTER_URI` variable will be explained when networking the TurtleBot is discussed.

To move the TurtleBot, open a **new** terminal window, and launch teleoperation by typing the following command:

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

Among other things, you will see a diagram of the control keys used to control the robot on the screen:

Control Your Turtlebot!

Moving around:

u i o

j k l

m , .

These are the same keys as discussed previously for the TurtleBot simulator. If all goes well, the TurtleBot will move forward, backward, or turn according to the key pressed on the netbook.

Tutorials devoted to the TurtleBot can be found at:

`http://learn.turtlebot.com/`

Of course, controlling the TurtleBot from its netbook is not very satisfying. It is done only to see that the TurtleBot software is set up correctly. In the next section, we describe the setup of a network so that the robot can be controlled by a remote computer.

In particular, as discussed in the *Using keyboard teleoperation to move TurtleBot* section of this chapter, the keyboard keys of a remote computer are used to move TurtleBot after it is connected to a network.

Networking the netbook and remote computer

ROS has the ability to allow multiple computers to communicate and share nodes, topics, and services. In the case of TurtleBot, the netbook has limited capabilities for graphics applications, such as rviz. It is better to run rviz and other visualization software on a desktop computer or a powerful laptop, both of which will be called a remote computer here to distinguish it from the netbook that rides along with the TurtleBot.

The approach is to designate one computer in the network to run the ROS Master identified by the `ROS_MASTER_URI` variable and launch the `roscore` process from that computer. The choice is to set up TurtleBot's netbook as the Master since many applications of the TurtleBot require autonomous motion without the intervention of the remote computer.

Any other remote computer on the network will have its own IP address as the `ROS_IP address` but its `ROS_MASTER_URI` variable will be TurtletBot's netbook IP address.

Types of networks

Networks between computers can be set up in various ways. To link to the TurtleBot from a remote computer, there are several common ways to network wirelessly:

- Use a network with a server computer that allows access to the Internet with Wi-Fi access, as might be found in a university or any other large organization.
- Use a router that allows local communication via Wi-Fi between the netbook and the remote computer. This is commonly used when setting up a private network to connect devices to each other wirelessly and to the Internet.

The network system in an organization may have security limitations that cover computers that can access their network. It is best to check any such requirements. Also, many such networks have network addresses assigned by a server using **Dynamic Host Configuration Protocol** (**DHCP**), which means that the IP address of a computer connected to a network can change if the computer is disconnected from the network and then reconnected. If the IP address changes, it is important to assign the ROS Master address as the new IP address of the TurtleBot's netbook connected to the wireless network.

Network addresses

A network identifies each computer on the network in one of the several ways, but each computer connected to the network has a unique identity. If the computers communicate through the Internet, you can refer to any Internet-connected machine by its **Internet Protocol** (**IP)**address, which is a four-part number string (such as `192.168.11.xxx`) in which the first part identifies the specific network to which the machine is connected. Another way to refer to the computer is by its hostname, which is usually a text string that consists of the machine name and the domain name.

You can determine the hostname of your computer with the `hostname` command and the username using the `whoami` command in the forms:

```
$ hostname
$ whoami
```

In a ROS network, the Master is designated by a URI used to identify the name of the Master on a network. For example, the `ROS_MASTER_URI` variable for the TurtleBot in the authors' laboratory has the following address:

ROS_MASTER_URI=http://192.168.11.123:11311

The IP address in this case is `192.168.11.123`. The IP address of a computer on the network can be determined by the following Ubuntu command:

```
$ ifconfig
```

This command will list the communication properties of the computer. The screen output will typically show an Ethernet connection (`eth0`) if any, a local loopback address (`lo`), and the wireless IP address (`wlan`), which is designated as `inet addr`. The digits `11311` represent the port used by the ROS Master for communication on the computer.

The description of the ROS networking requirements can be viewed on the following websites:

- `http://wiki.ros.org/ROS/NetworkSetup`
- `http://wiki.ros.org/ROS/Tutorials/MultipleMachines`

There must be a connection between the machines. Using the IP addresses of the machines to identify the machines is sufficient. Only one machine in the network can be the Master.

`ROS_IP` and `ROS_HOSTNAME` are environment variables that set the declared network address of a ROS node. The convention is to use `ROS_IP` if you are specifying an IP address, and `ROS_HOSTNAME` if you are specifying a hostname. The `ROS_HOSTNAME` variable takes precedence over the `ROS_IP` variable.

The `ROS_MASTER_URI`, `ROS_IP`, and `ROS_HOSTNAME` variables are described in the tutorial at: `http://wiki.ros.org/ROS/EnvironmentVariables`

In the case of TurtleBot, the ROS Master resides on the netbook and the netbook's IP address must be indicated to the remote computer. On the remote computer, the `ROS_MASTER_URI` variable must be set to the address of the netbook so that its nodes can register with the Master. Once that is done, the nodes can communicate with the Master and other nodes wirelessly.

Remote computer network setup

To link the remote computer and the TurtleBot's netbook, make sure that both the computers communicate on the same network. This may involve changing the network choice of the computers if there are several networks available.

For your setup of the remote computer, determine the IP addresses of the netbook and your remote computer using the `ifconfig` command. Your commands will use your specific addresses, and you will use the following commands:

```
$ export ROS_MASTER_URI=http://<IP address of TurtleBot>:11311
$ export ROS_IP=<IP address of remote computer>
```

We recommend that you add these commands to the `.bashrc` file so that the TurtleBot is the ROS Master every time you open a new window.

To be more specific, on our remote computer, we edited our `.bashrc` file and added the following commands to create these environment variables for the TurtleBot:

```
export ROS_MASTER_URI=http://192.168.11.123:11311
export ROS_IP=192.168.11.120
```

The ROS Master address points to the TurtleBot netbook, and the `ROS_IP` variable is the IP address of our remote computer's wireless card used in this example. The examples just shown using the network addresses were taken from the actual computers used in the authors' laboratory to run TurtleBot. Of course, your addresses will be different.

To check the variables on the remote computer, type the following command to check the IP addresses of the ROS Master and the remote computer:

```
$ env | grep ROS
```

Netbook network setup

The netbook setup instructions are found at the ROS wiki location at:

`http://wiki.ros.org/turtlebot/Tutorials/indigo/Network%20Configuration`

To set up the netbook addresses, you can type the following command at the netbook terminal window:

```
$ echo export ROS_MASTER_URI=http://<IP address of TurtleBot>:11311
>> ~/.bashrc
$ echo export ROS_IP=<IP address of TurtleBot> >> ~/.bashrc
```

where `<IP address of TurtleBot>` is replaced with the IP address of the TurtleBot netbook, which is normally called the IP address of the TurtleBot. This sets the TurtleBot as the Master.

Secure Shell (SSH) connection

The Secure Shell (SSH) will be used to allow remote login to the TurtleBot's netbook from the remote computer. Check the SSH status with the following command:

```
$ sudo service ssh status
```

If the SSH service is not present, install it according to the instructions available at: `http://learn.turtlebot.com/` in the *Setting Up Networking* section.

For the authors' TurtleBot netbook, our username is `turtlebot`. Your username can be found by running the following command:

```
$ whoami
```

To communicate with the TurtleBot, on the remote computer, type the `ssh` command in the following form and enter your TurtleBot password when prompted:

```
$ ssh <username>@<IP address of TurtleBot>
```

Summary of network setup

In summary, to set up the communication between the TurtleBot and the remote computer to control the robot, check the following on both the computers:

- TurtleBot's netbook hosts the ROS Master with

  ```
  ROS_MASTER_URI= http://<IP address of TurtleBot>:11311
  ```

  and

  ```
  ROS_IP=http://<IP address of TurtleBot>
  ```

- The remote computer has

  ```
  ROS_MASTER_URI = http://<IP address of TurtleBot>:11311
  ```

  and

  ```
  ROS_IP==http://<IP address of remote computer>
  ```

The addresses here are assumed to be the addresses of the TurtleBot netbook and the remote computer on a wireless network.

Troubleshooting your network connection

Many problems in networking ROS occur because the IP addresses of the netbook and the remote computer are not set correctly. Perform the following steps:

1. Check the computers' network settings.
2. Make sure that the network is working by communicating with the server or router for the network.
3. Use the `ifconfig` and `env | grep ROS` commands to check whether the network addresses are set correctly.
4. If your network has DHCP, see if the assigned IP addresses have changed.

Some information about networks that may be helpful can be found at the following sites:

`http://compnetworking.about.com/cs/protocolsdhcp/g/bldef_dhcp.htm`

`http://compnetworking.about.com/od/homenetworking/a/routernetworks.htm`

Testing the TurtleBot system

To test the system and the communication, perform the following steps:

1. Make sure that the TurtleBot base battery and the netbook battery are charged.
2. Plug in the netbook to the base and power up the base.
3. Turn on the netbook and log on using the netbook's password and then connect to your network.
4. Give the TurtleBot room to move without obstacles in the way.
5. Log on to the remote computer and start communicating with the TurtleBot through your network.

This procedure is used to command the robot from the remote computer by typing the `ssh` command at the remote computer terminal and entering the TurtleBot password. The first example in *Using keyboard teleoperation to move TurtleBot* section will allow you to control the TurtleBot using several keyboard keys.

To start communication with the TurtleBot from the remote computer, type the `ssh` command and enter the TurtleBot password when prompted:

```
$ ssh <username>@<IP address of TurtleBot>
```

(The output is deleted for brevity.)

As described earlier, our TurtleBot IP address is `192.168.11.123`.

The window prompt will change to the window prompt of the TurtleBot netbook. Our netbook prompt is `turtlebot@turtlebot-0428:~$` and has been left in the following command lines to identify where the commands are issued.

After the response, you can send commands to the TurtleBot by typing the following command:

```
turtlebot@turtlebot-0428:~$ roslaunch turtlebot_bringup
minimal.launch
```

The output is as follows:

.

.

Checking log directory for disk usage. This may take awhile

After a long list of parameters and nodes, you will see the ROS Master address. In our system, the output is as follows:

auto-starting new master

process[master]: started with pid [23426]

ROS_MASTER_URI=http://192.168.11.123:11311

This line of the output shows that the TurtleBot is the ROS Master. It is followed by a list of the processes running and other information.

To view the nodes that are active after the minimal launch, use the command:

```
$ rosnode list
```

TurtleBot hardware specifications

Before driving the real TurtleBot around, it would be useful to understand the capabilities of the robot in terms of its possible speed, turning capability, carrying capacity, and other such properties. With this information, you can plan the motion and speed of TurtleBot and design interesting applications. The specifications here are taken from the information provided for the Kobuki base by the Yujin Robot company. Their website for general information can be found at:
`http://kobuki.yujinrobot.com/home-en/documentation/get-started/`

For specific information about the base, check out the specifications at:
`http://kobuki.yujinrobot.com/home-en/about/specifications/`

The base has a rechargeable battery that powers the motors turning the wheels. The netbook has its own battery but is not charged when the TurtleBot is moving on its own. There are a number of sensors in the base.

In the previous examples of teleoperation, the TurtleBot linear speed in the forward or backward direction was 0.2 meters/second or 20 cm/second. That is a bit over 1 foot per second and is probably fast enough for a robot moving in a room with obstacles in its way. The turning rate was 1 radian/second. Since there are 2π *(6.28)* radians in a circle, the TurtleBot will rotate completely around in about 6 seconds or so.

According to the manufacturer, Yujin Robot, the maximum values are as follows:

- The maximum linear speed is 70 cm/second (27.5 inches/second)
- The maximum angular velocity is 180 degrees/second or π radians/second
- The payload is 5 kg (11 pounds) on a hard floor

Review the other functional and hardware specifications to familiarize yourself with TurtleBot and its capabilities and limitations. In our laboratory for safety reasons, we run the TurtleBot at a relatively slow speed compared to its maximum speed.

TurtleBot dashboard

In this section, it is assumed that you have established communication with the TurtleBot and can send commands to start minimal launch. First, view the TurtleBot's dashboard, which indicates its condition, as described on the site at: `http://wiki.ros.org/turtlebot_dashboard`.

In a new terminal window on the remote computer, type the following command to bring up the dashboard:

```
$ roslaunch turtlebot_dashboard turtlebot_dashboard.launch
```

The output is as follows:

... logging to /home/tlharmanphd/.ros/log/2dfad508-6602-11e5-8e83-6c71d9a711bd/roslaunch-D125-43873-4550.log

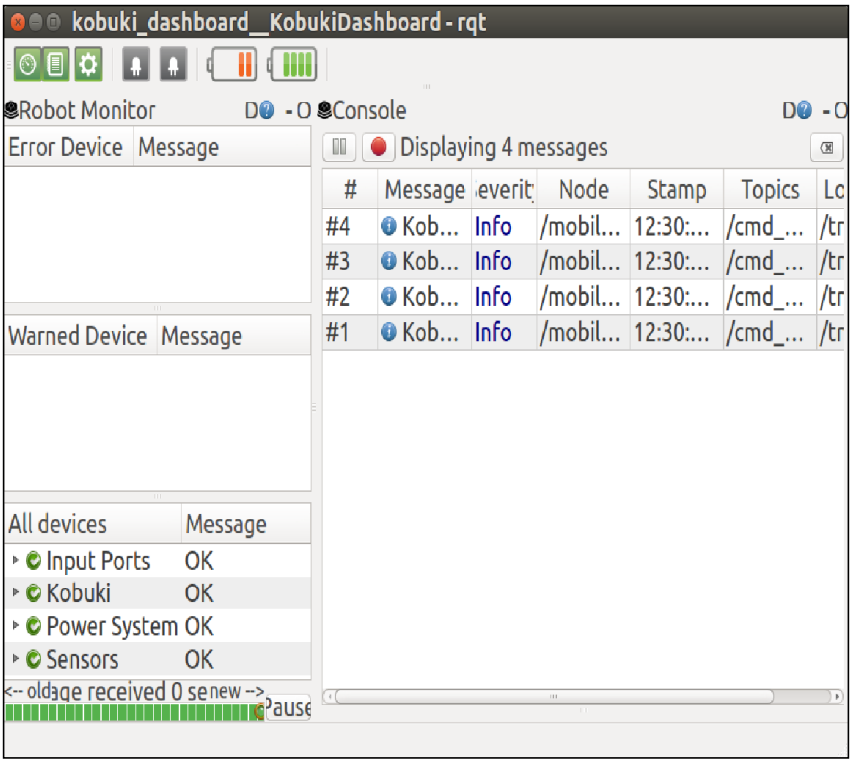
Checking log directory for disk usage. This may take awhile.

Press *Ctrl* + *C* to interrupt.

The dashboard indicates the status of various systems of the TurtleBot:

- A diagnostic indicator, `/rosout` messages, and motor control (OFF/ON) in the upper-left corner of the screen
- Controls for two colored LEDs on the base that can be turned on or off
- Battery monitor indicators for the netbook and the Kobuki base
- Status of the power system, the motors, and the sensors

In the following Turtlebot (Kobuki) dashboard screenshot, we have clicked on the diagnostic icon in the far upper-left corner of the screen to bring up the status messages on the dashboard:



Kobuki dashboard

If your netbook battery monitor does not work, check the directions in the *Setting up to control a real TurtleBot* section to select a proper battery for monitoring.

Move the real TurtleBot

There are a number of ways to move the TurtleBot using ROS. In this section, we present the following three methods:

- Using the keyboard
- Using ROS terminal window commands
- Using a Python script

Using keyboard teleoperation to move TurtleBot

In a new terminal window, launch the TurtleBot keyboard teleop program on the remote computer:

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

The output is as follows:

... logging to /home/turtlebot/.ros/log/b662ab4a-c22e-11e5-b730-6c71d9a711bd/roslaunch-turtlebot-0428-26633.log

Checking log directory for disk usage. This may take awhile.

Press Ctrl-C to interrupt

Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.11.123:58861/

SUMMARY

========

PARAMETERS

*** /rosdistro: indigo**

*** /rosversion: 1.11.16**

*** /turtlebot_teleop_keyboard/scale_angular: 1.5**

*** /turtlebot_teleop_keyboard/scale_linear: 0.5**

NODES

/

turtlebot_teleop_keyboard (turtlebot_teleop/turtlebot_teleop_key)

ROS_MASTER_URI=http://192.168.11.123:11311

core service [/rosout] found

process[turtlebot_teleop_keyboard-1]: started with pid [26653]

Control Your Turtlebot!

Moving around:

u i o

j k l

m , .

q/z : increase/decrease max speeds by 10%

w/x : increase/decrease only linear speed by 10%

e/c : increase/decrease only angular speed by 10%

space key, k : force stop

anything else : stop smoothly

CTRL-C to quit

currently: speed 0.2 turn 1

We have left the entire output from the launch because it is useful to know the parameters and nodes involved when a package is launched. Make sure that the TurtleBot base and the netbook batteries are sufficiently charged so you can move the TurtleBot around to become familiar with its capabilities for straight-line motion and rotation. Now try the `i` key to move the TurtleBot forward or the `,` key to drive backward. The speed is 0.2 meters/second.

Using ROS commands to move TurtleBot around

There are a number of ways to control the TurtleBot movement other than using the keyboard. There are several ROS commands that are useful to move and monitor the TurtleBot in motion:

- `rostopic pub` is used to publish commands to move the TurtleBot
- `rostopic echo` is used to display the messages sent

After the TurtleBot has been brought up with the minimal launch command, the `rostopic pub` command can be used to move and turn the TurtleBot. To move the TurtleBot forward, issue this command from the remote computer:

```
$ rostopic pub -r 10 /mobile_base/commands/velocity
\geometry_msgs/Twist '{linear: {x: 0.2}}'
```

TurtleBot should move forward continuously at 0.2 meters/second until you press *Ctrl* + *C* while the focus is on the active window.

This command publishes (`pub`) the `/mobile_base/commands/velocity` topic at the rate of 10 times per second. The `-r` variable indicates that the rate is repeated. To send the message once, use `-1` instead of `-r`.

To move the TurtleBot backward, issue the following command:

```
$ rostopic pub -r 20 /mobile_base/commands/velocity
\geometry_msgs/Twist '{linear: {x: -0.2}}'
```

Always press *Ctrl* + *C* to stop TurtleBot.

To cause the robot to turn in a circle requires some forward velocity and angular velocity, which the following command shows:

```
$ rostopic pub -r 10 /mobile_base/commands/velocity
\geometry_msgs/Twist '{linear: {x: 0.2}, angular: {x: 0, y: 0, z:
1.0}}'
```

The linear speed is 0.2 meters/second and the rotation is 1.0 radian (about 57 degrees) per second.

To view the messages sent, type the following command in a separate terminal window:

```
$ rostopic echo /mobile_base/commands/velocity
```

The output is as follows:

linear:

x: 0.2

y: 0.0

z: 0.0

angular:

x: 0.0

y: 0.0

z: 1.0

The message repeats the linear velocity and angular rotation values being sent 10 times a second. Use *Ctrl* + *C* to stop the display.

Writing your first Python script to control TurtleBot

We will present a simple Python script to move the TurtleBot in this section. The basic approach to create a script begins with a design. The design should detail the activity to be accomplished. For example, a script can command TurtleBot to move straight ahead, make several turns, and then stop. The next step is to determine the commands for TurtleBot to accomplish the tasks. Finally, a script is written and tested to see whether TurtleBot responds in the expected way. The remote computer will execute the Python script and TurtleBot will move as directed if the script is correctly written.

In terms of the TurtleBot commands that will be used, we can summarize the process as follows:

- Design the program outlining the activities of TurtleBot when the script is executed
- Determine the nodes, topics, and messages to be sent (published) or received (subscribed) from the TurtleBot during the activity
- Study the ROS Python tutorials and examples to determine the way to write Python statements that send or receive messages between the remote computer and the TurtleBot

There is a great deal of documentation describing ROS Python scripts. The statement structure is fixed for many operations. The site `http://wiki.ros.org/rospy` briefly describes `rospy`, which is called the ROS client library for Python. The purpose is to allow statements written in Python language to interface with ROS topics and services.

The site `http://wiki.ros.org/rospy_tutorials` contains a list of tutorials. At the top of the tutorial page, there is a choice of distributions of ROS, and Indigo is chosen for our discussions. A specific tutorial that describes many Python statements that are used in a typical script can be found at: `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)`

To find the nodes that are active after the `keyboard_teleop.launch` file is launched, type this command:

```
$ rosnode list
```

The output is as follows:

/app_manager

/bumper2pointcloud

/capability_server

/capability_server_nodelet_manager

/cmd_vel_mux

/diagnostic_aggregator

/interactions

/master

/mobile_base

/mobile_base_nodelet_manager

/robot_state_publisher

/rosout

/turtlebot_laptop_battery

/turtlebot_teleop_keyboard

/zeroconf/zeroconf

The nodes are described in the Kobuki tutorial that can be found at:
`http://wiki.ros.org/kobuki/Tutorials/Kobuki's%20Control%20System`

According to the site, the `mobile_base` node listens for commands, such as velocity, and publishes sensor information. The `cmd_vel_mux` variable serves to multiplex commands to assure that only one velocity command at a time is relayed to the mobile base.

In the previous example, we used the `rostopic pub` command to publish the linear and angular `geometry_msgs/Twist` data in order to move the TurtleBot. The Python script that follows will accomplish essentially the same thing. The script will send a Twist message on the `cmd_vel_mux/input/navi` topic.

A Python script will be created to move the TurtleBot forward in a simple example. If you are not very familiar with Python, it may be best to study and execute the example script and then refer to the ROS tutorials. The procedure to create an executable script on the remote computer is as follows:

1. Write the script with the required format for a ROS Python script using an ordinary text editor.
2. Give the script a name in the `<name>.py` format and save the script.

We have called our script `ControlTurtleBot.py` and saved it in our home directory.

To make the script executable, execute the Ubuntu command:

```
$ chmod +x ControlTurtleBot.py
```

Make sure that the TurtleBot is ready by running the minimal launch. Then, in a new terminal window, type this command:

```
$ python ControlTurtleBot.py
```

In our example, *Ctrl* + *C* is used to stop the TurtleBot. The comments in the script explain the statements. The tutorials listed previously give further details of Python scripts written using the ROS conventions:

```
#!/usr/bin/env python
# Execute as a python script
# Set linear and angular values of TurtleBot's speed and turning.
import rospy      # Needed to create a ROS node
from geometry_msgs.msg import Twist    # Message that moves base

class ControlTurtleBot():
  def __init__(self):
    # ControlTurtleBot is the name of the node sent to the master
    rospy.init_node('ControlTurtleBot', anonymous=False)

    # Message to screen
    rospy.loginfo(" Press CTRL+c to stop TurtleBot")

    # Keys CNTL + c will stop script
    rospy.on_shutdown(self.shutdown)

    # Publisher will send Twist message on topic
    # cmd_vel_mux/input/navi

    self.cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi',
    Twist, queue_size=10)
```

```
        # TurtleBot will receive the message 10 times per second.
        rate = rospy.Rate(10);
        # 10 Hz is fine as long as the processing does not exceed
        #   1/10 second.

        # Twist is geometry_msgs for linear and angular velocity
        move_cmd = Twist()
        # Linear speed in x in meters/second is + (forward) or
        # - (backwards)
        move_cmd.linear.x = 0.3
        # Modify this value to change speed
        # Turn at 0 radians/s
        move_cmd.angular.z = 0
        # Modify this value to cause rotation rad/s

        # Loop and TurtleBot will move until you type CNTL+c
        while not rospy.is_shutdown():
        # publish Twist values to TurtleBot node /cmd_vel_mux
        self.cmd_vel.publish(move_cmd)
        # wait for 0.1 seconds (10 HZ) and publish again
        rate.sleep()


      def shutdown(self):
        # You can stop turtlebot by publishing an empty Twist message
        rospy.loginfo("Stopping TurtleBot")

        self.cmd_vel.publish(Twist())
        # Give TurtleBot time to stop
        rospy.sleep(1)

    if __name__ == '__main__':
      try:
        ControlTurtleBot()
      except:
        rospy.loginfo("End of the trip for TurtleBot")
```

Introducing rqt tools

The rqt tools (ROS Qt GUI toolkit) that are part of ROS allow graphical representations of ROS nodes, topics, messages, and other information. The ROS wiki lists many of the possible tools that are added to the rqt screen as **plugins**: `http://wiki.ros.org/rqt/Plugins`.

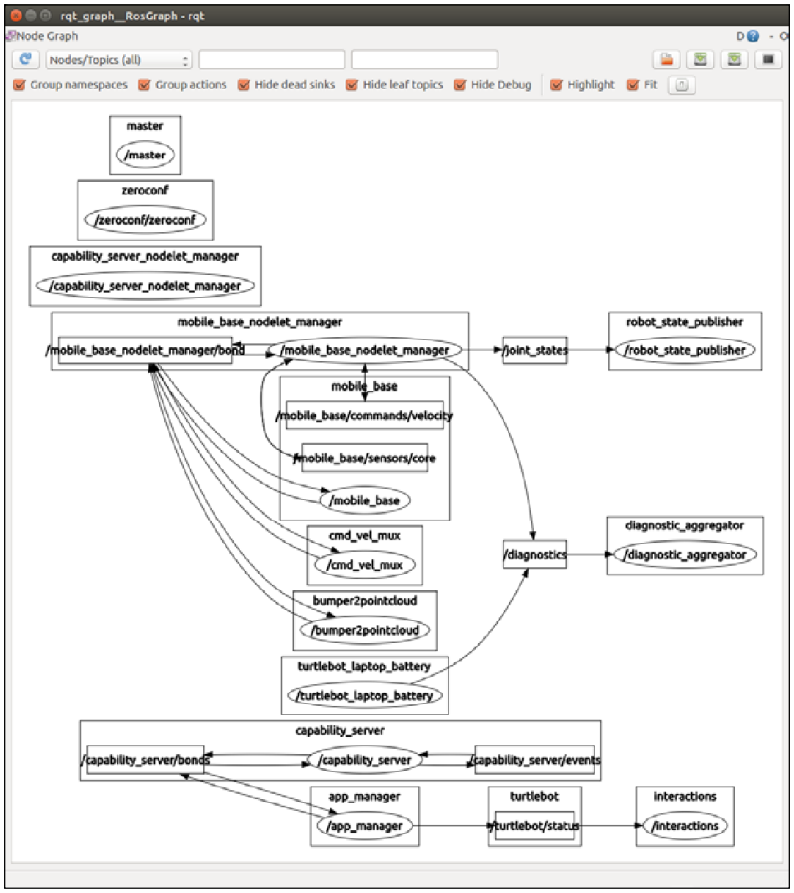
The ROS tutorial on topics also describes some of the features of the rqt tools at:
`http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics`

rqt_graph

One of the common uses of rqt is to view the nodes and topics that are active. Bring the TurtleBot up with minimal launch as previously described. Then, on the remote computer, issue the following command:

```
$ rqt_graph
```

Select the top-left box, **Nodes/Topics (all)**. The following screenshot of `rqt_graph` shows the nodes that are active and the connections between the publishers and subscribers that deal with moving the base of the TurtleBot. Pass the cursor over the various items to see the nodes and topics and see how they communicate:



rqt_graph after minimal launch of TurtleBot

On the menu bar at the top of the screen, keep the dead sinks, leaf topics, and debug topics hidden to simplify the graph. Take a look at the preceding graph; the names, such as **master** and **mobile_base** are called namespaces to identify the items. Ellipses (ovals) represent nodes while arrows represent connections through topics. The names in the rectangles represent topics.

For example, the `/mobile_base_nodelet_manager` node publishes on the `/joint_states` topic.

After the minimal launch, the `keyboard_teleop.launch` command is issued in a separate terminal window, as described earlier. When the following command is issued:

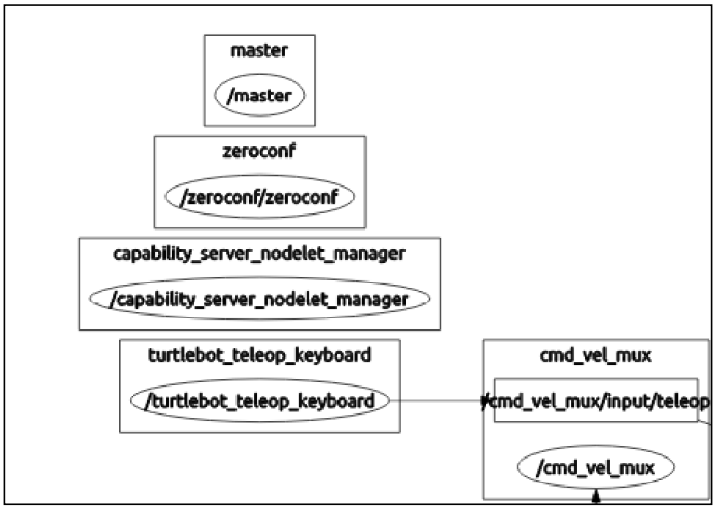
```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

One of the screen outputs shows the name of the node that the `turtlebot_teleop` package is using as follows:

NODES

/ turtlebot_teleop_keyboard (turtlebot_teleop/turtlebot_teleop_key)

The following screenshot portion shows selected nodes and topics after the launch of `keyboard_teleop.launch`. As shown in the following screenshot, a new `turtlebot_teleop_keyboard` node has appeared, publishing on the `/cmd_vel_mux/input/teleop` topic:



rqt_graph after TurtleBot teleoperation

To list all the active nodes, the `rosnode list` command, on the remote computer, can be issued. For details on a particular node, such as `/turtlebot_teleop_keyboard`, type the following command:

```
$ rosnode info /turtlebot_teleop_keyboard
```

The output is as follows:

```
--------------------------------------------------------------------------------
Node [/turtlebot_teleop_keyboard]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /cmd_vel_mux/input/teleop [geometry_msgs/Twist]

Subscriptions: None

Services:
 * /turtlebot_teleop_keyboard/set_logger_level
 * /turtlebot_teleop_keyboard/get_loggers


contacting node http://192.168.11.120:46784/ ...
Pid: 12745
Connections:
 * topic: /cmd_vel_mux/input/teleop
   * to: /mobile_base_nodelet_manager
   * direction: outbound
   * transport: TCPROS
 * topic: /rosout
   * to: /rosout
   * direction: outbound
   * transport: TCPROS
```

From the keyboard, the messages of the `geometry_msgs/Twist` type are sent when you press a key that moves TurtleBot as indicated by the node's publications.

rqt message publisher and topic monitor

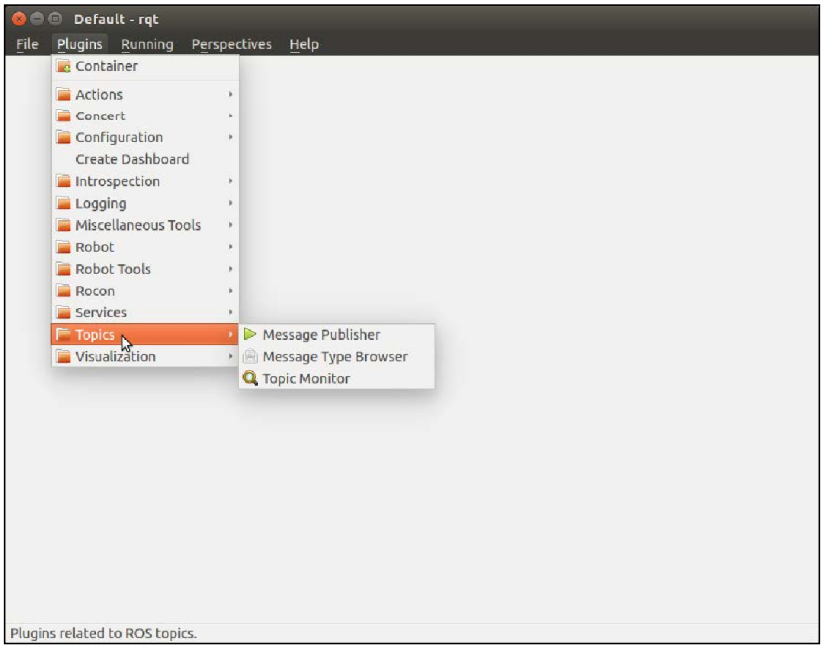
There are a number of variations of the `rqt` command with options. The simplest command is as follows:

```
$ rqt
```

This brings up a display screen, as shown in the following screenshot. In the menu bar, there are drop-down menu items that allow you to make choices to perform the following steps:

1. Select the **Plugins** tab that will be displayed; in our screenshot, the **Message Publisher** and the **Topic Monitor** options were chosen from the **Plugins** tab.
2. Select the topics or other information for your plugins.
3. Rearrange the screen to suit your preferences if you choose more than one plugin.

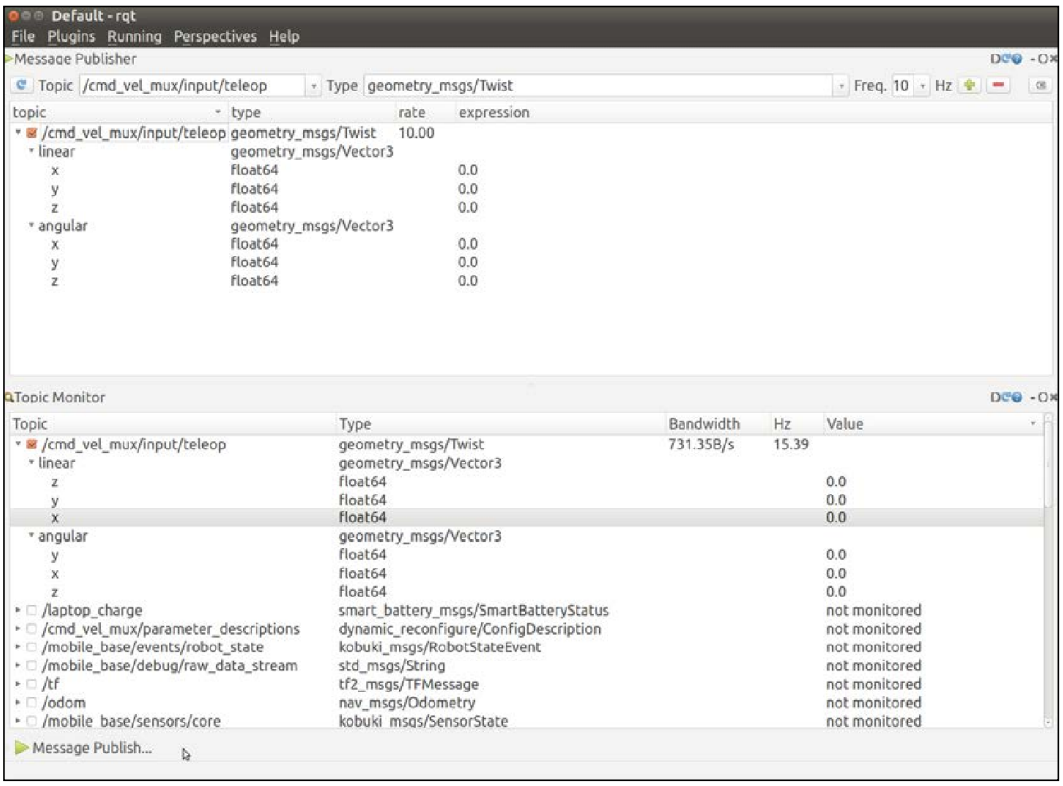
The `rqt` command and the drop-down menu selections are shown in the following screenshot:



rqt command initial screen with plugin selections

For the following screenshot of `rqt`, the selections are made in the following order:

1. Issue the `rqt` command.
2. From the **Plugins** tab, select **Message Publisher** under the **Topics** tab.
3. From the **Plugins** tab, select **Topic Monitor** under the **Topics** tab.
4. Choose to publish the `Twist` message to `/cmd_vel_mux` and see the message monitored.
5. Rearrange the plugins on the screen for convenient viewing:



Two rqt plugins to publish and monitor messages

After you make the selections, there will be two plugins on the screen. You can rearrange them by clicking on the undocking symbol **()** in the upper-right corner of the plugins screen and dragging the window of the **Topic Monitor** below that of the **Message Publisher**.

Specifically, in the **Topic** entry box, type `/cmd_vel_mux/input/teleop` and click on the **+** button to add the topic. Left-click to check the topic's checkbox and right-click to expand the parameters of the message to see the angular and linear parameters of the `Twist` message. Note that the **rate** variable has been set to **10.00** from its original value of 1.00 so that the TurtleBot can move smoothly. When you click on the **linear x** or **angular z** variable, you can change the parameter under the column titled **expression**. Changing the values from 0.0 will cause the TurtleBot to move because the message will be published.

The result shows the **Message Publisher** and the **Topic Monitor** with the `/cmd_vel_mux/input/teleop` topic selected. From the `geometry_msgs` package, the `Twist` message will be sent to the TurtleBot to move the robot.

From the screenshot showing the drop-down menu, it is clear that there are many options associated with the rqt tools. View the tutorials and try various options to experience the power of the rqt tools to allow you to control and monitor your robot's activities. One use is for **debugging** your scripts if the TurtleBot does not respond as expected, because you can monitor the messages sent to the TurtleBot.

TurtleBot's odometry

In this section, we explore the TurtleBot's odometry. The general definition of odometry is the use of data from motion sensors to estimate change in position over time. Odometry is used by the TurtleBot to estimate its position and orientation relative to a starting location given in terms of an x and y position and an orientation around the z (upward) axis. The topic is `/odom` and the command to view the form of the `/odom` message is as follows:

```
$ rostopic echo /odom
```

When you execute this `echo` command, the output will be updated continuously on the screen. However, we wish to display TurtleBot's motion using rviz. When the odometry option is chosen in rviz, the TurtleBot's position and orientation will be displayed with arrows that are generated as TurtleBot moves.

Odom for the simulated TurtleBot

The simulated TurtleBot will be used to demonstrate the odometry display possible in rviz.

To run Gazebo on your remote computer, you must reassign the ROS Master if it is assigned to the TurtleBot in the `.bashrc` file. In each terminal window you open, type the following command:

```
$ export ROS_MASTER_URI=http://localhost:11311
```

The commands executed on the remote computer to start Gazebo for simulation and rviz for visualization are as follows:

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

In another terminal window, run this command:

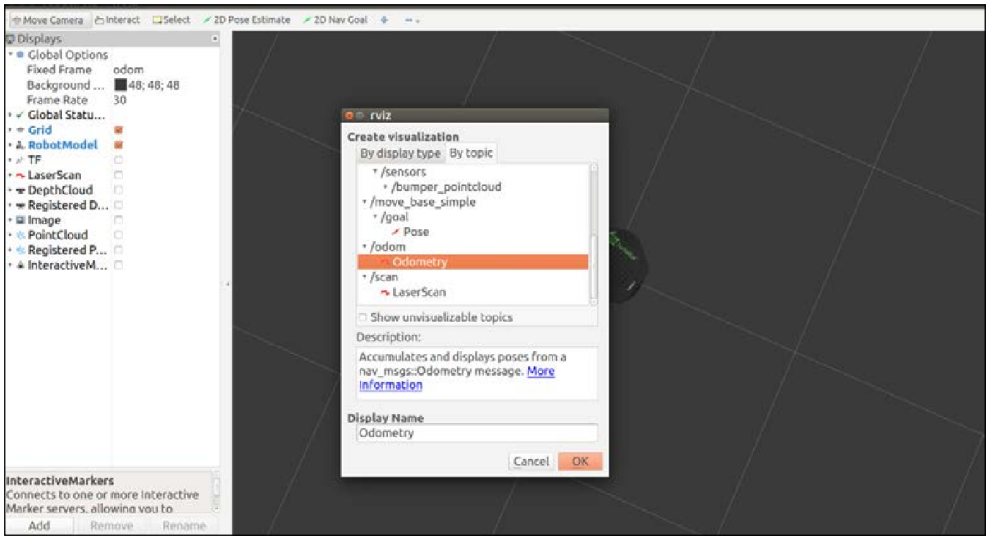
```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```

Gazebo includes the physics of the robot and rviz allows a variety of visualization options. In particular, it is useful to show the pose of the robot as indicated by arrows that point in the direction of motion of the TurtleBot on the screen.

In rviz, it is necessary to choose several options to show the TurtleBot's odometry arrows on the screen. As shown in the following screenshot, we choose the following:

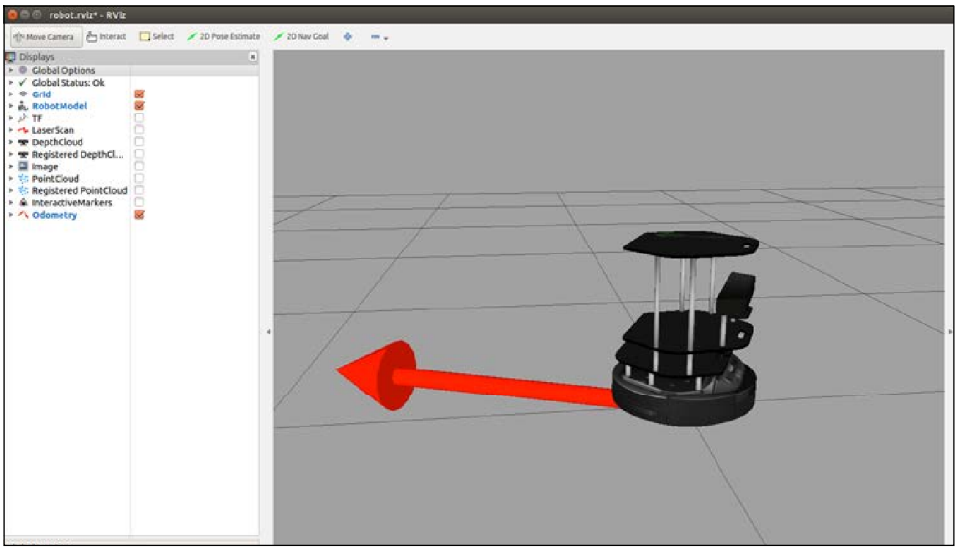
1. Under **Global Options** on the left side panel for **Fixed Frame**, change **base_link** or **base_footprint** to **odom**.
2. Click on **Add**, and select the **By topic** tab shown.
3. Choose **Odometry** and click on **OK**.

4. On the left side panel, click on the small arrow to the left of **Odometry** to show the various options. The topic is **odom** and the screen will keep 100 arrows that point to the direction of the simulated TurtleBot as it moves:



Selection of the odom topic in rviz showing a list of topics

5. Once these selections are made, the simulated TurtleBot will appear on the screen with an arrow pointing in its forward direction, as shown in the following screenshot:



Rviz showing odom arrow for initial position of simulated TurtleBot

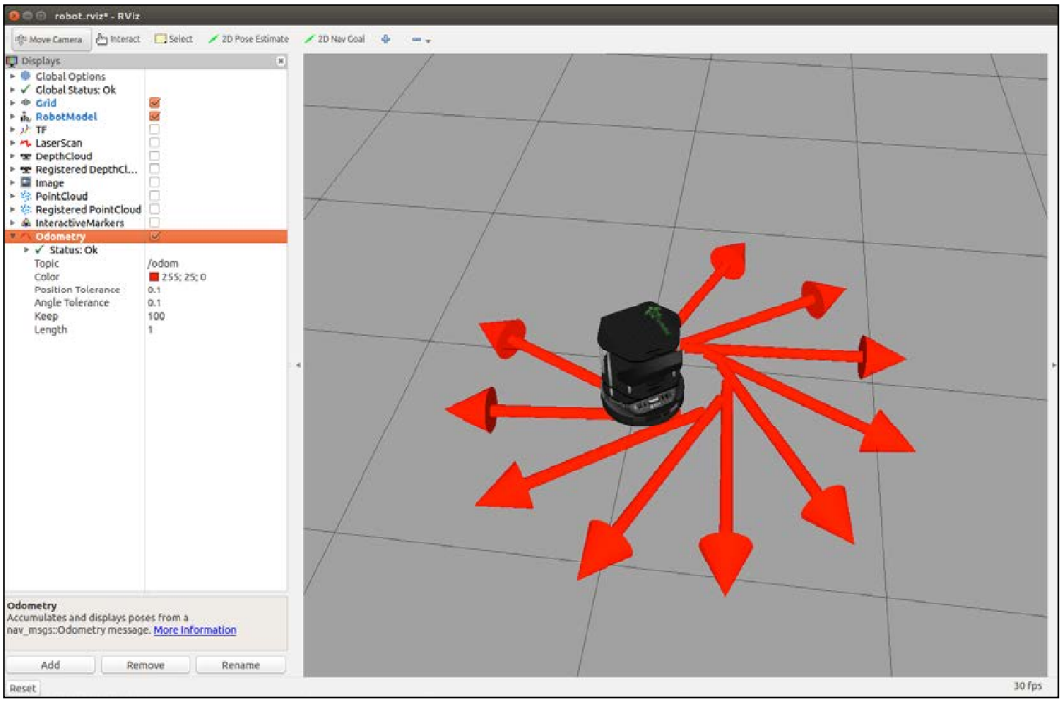
To track the motion of the simulated TurtleBot on the screen and display the arrows, we issue a movement command. Once the two screens are up for Gazebo and rviz, any commands to move the robot are possible, including the execution of a Python script. For example, in a third terminal window, issue one of the following commands to make the TurtleBot move in a circle on the screen:

```
$ rostopic pub -r 10 /cmd_vel_mux/input/teleop \geometry_msgs/Twist
'{linear: {x: 0.1, y: 0, z: 0}, angular: {x: 0, y: 0, z: -0.5}}'
```

```
$ rostopic pub -r 10 /mobile_base/commands/velocity
\geometry_msgs/Twist '{linear: {x: 0.1, y: 0, z: 0}, angular: {x: 0,
y: 0, z: -0.5}}'
```

The result is the same in terms of the movement of the robot in our examples, but the `/mobile_base/commands/velocity` topic is used to control the mobile base as explained in the Kobuki tutorial at: `http://wiki.ros.org/kobuki/Tutorials/Kobuki's%20Control%20System`.

The `/cmd_vel_mux` variable is used to multiplex velocity commands from different sources, such as the keyboard or a Python script. Either commands make the TurtleBot move in a circle with the result shown in the following screenshot:



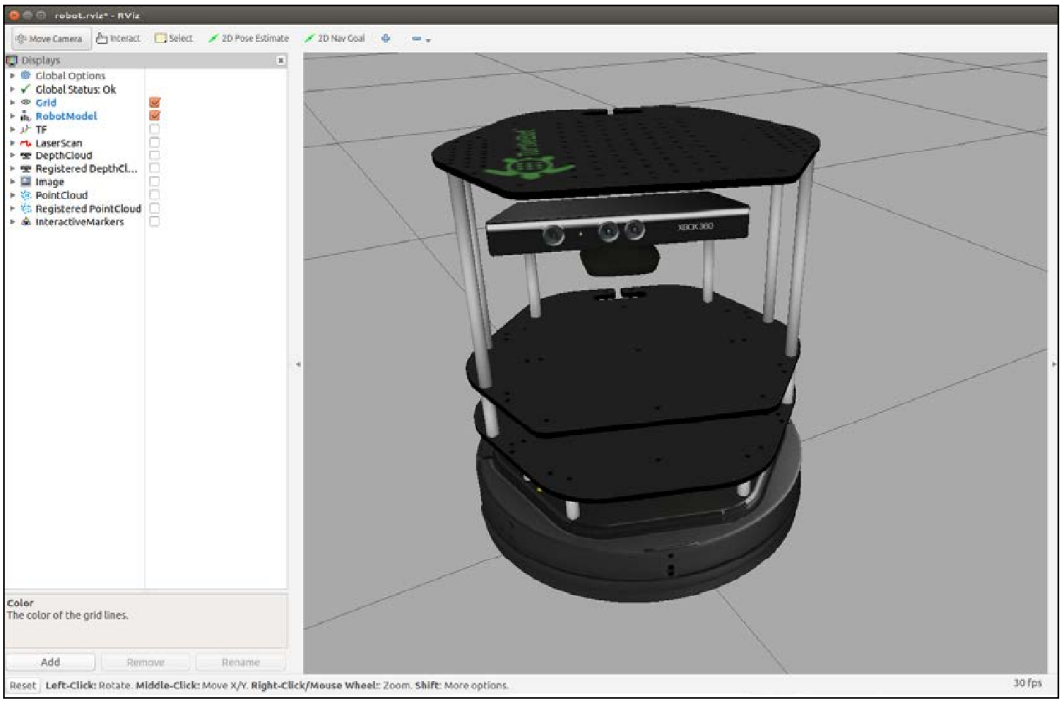
Simulated TurtleBot moving in a circle with the direction shown in rviz

Real TurtleBot's odometry display in rviz

The commands used in simulation can be used with the physical TurtleBot. After bringing up the real TurtleBot with the minimal launch, start rviz on the remote computer:

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```

TurtleBot will appear in rviz, as shown in the following screenshot:



TurtleBot on rviz bringup

Then, set up rviz with odom for **Fixed Frame** and navigate to **Add** | **By topic** | **Odometry**, as was done with the simulated TurtleBot.

Run the following command to move TurtleBot in a circle:

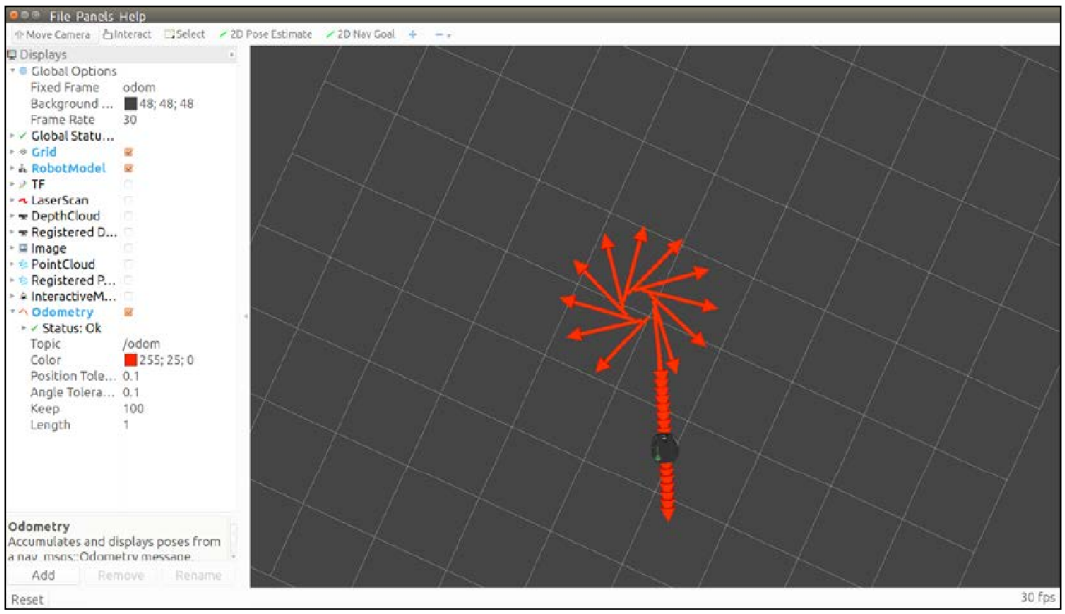
```
$ rostopic pub -r 10 /mobile_base/commands/velocity
\geometry_msgs/Twist '{linear: {x: 0.1, y: 0, z: 0}, angular: {x: 0,
y: 0, z: -0.5}}'
```

Stop the TurtleBot by pressing *Ctrl* + *C* with focus on the window in which you executed the command to move the robot.

In the following screenshot, TurtleBot's turning was stopped by pressing *Ctrl* + *C*, and the Python script was executed that drives TurtleBot straight forward until the *Ctrl* + *C* keys are pressed again.

The command is as follows:

```
$ python ControlTurtleBot.py
```



TurtleBot's path after the Twist message and running of the Python script

TurtleBot automatic docking

The TurtleBot has the capability of finding its docking station and moving to that station for recharging as described in the tutorial available at: `http://wiki.ros.org/kobuki/Tutorials/Testing%20Automatic%20Docking`.

According to the tutorial, the TurtleBot must be placed in line-of-sight of the docking station since the robot homes on the station using an infrared beam. The docking station will show a solid red light when it is powered up. If the TurtleBot finds the station and docks properly, the red light will turn to blinking green when charging and solid green when TurtleBot's battery is fully charged.

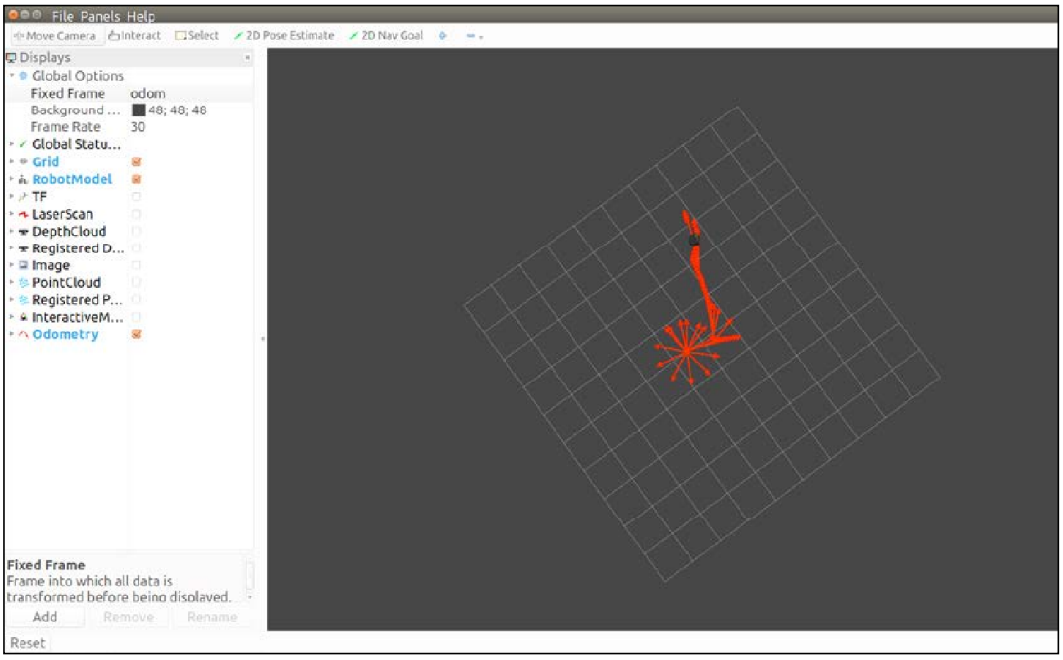
Make sure that the minimal launch is active and the TurtleBot is within the line-of-sight to the docking station. On the remote computer, type the following command:

```
$ roslaunch kobuki_auto_docking minimal.launch
```

Then, in another terminal window, type the following command to cause the TurtleBot to start the search for the docking station:

```
$roslaunch kobuki_auto_docking activate.launch
```

The following screenshot shows the TurtleBot rotating to find the IR signal and then heading toward the dock:



TurtleBot docking

In the preceding screenshot, the distance that the TurtleBot moved was about 2 meters (about 6 feet) to find the docking station and start recharging the base battery. The screen output as TurtleBot was completing the docking is as follows:

Feedback: [DockDrive: DOCKED_IN]:

Feedback: [DockDrive: DOCKED_IN]:

Result - [ActionServer: SUCCEEDED]: Arrived on docking station successfully.

[dock_drive_client_py-1] process has finished cleanly

log file: /home/tlharmanphd/.ros/log/789b3ecc-6bca-11e5-b156-6c71d9a711bd/dock_drive_client_py-1*.log

all processes on machine have died, roslaunch will exit

shutting down processing monitor...

... shutting down processing monitor complete

done

Summary

This chapter introduced the TurtleBot robot and described how to load the necessary software for TurtleBot. The Gazebo simulator was used to show the capability of ROS to control TurtleBot.

To control a real TurtleBot and allow it to roam autonomously, it is desirable to set up a wireless communication between a remote computer and the TurtleBot's netbook. The explanation given in this chapter will allow you to set up the network and remotely control TurtleBot.

The various methods to control TurtleBot were presented. Teleoperation from the remote computer is one of the common methods used to control the robot's motion. A Python script was shown, which, when executed, will make the TurtleBot move in a straight line. This chapter also covered the use of rqt tools to send commands to TurtleBot and monitor them.

An important aspect of this chapter is that the TurtleBot can be controlled in simulation or in a real environment with the same commands and scripts. This use of a simulator can save much time in planning, testing, and debugging the applications for TurtleBot before the real robot is turned loose.

Odometry for the TurtleBot was described for the simulated TurtleBot and the real TurtleBot using rviz for visualizing the robot's motion. Finally, the auto-docking feature of TurtleBot was demonstrated.

The next chapter explains TurtleBot's use of the vision sensor. The chapter shows in detail how to create a map for TurtleBot and enable it to autonomously navigate around its environment.

4
Navigating the World with TurtleBot

In the previous chapter, the TurtleBot robot was described as a two-wheeled differential drive robot developed by Willow Garage. The setup of the TurtleBot hardware, netbook, network system, and remote computer were explained, so the user could set up and operate his own TurtleBot. Then, the TurtleBot was driven around using the keyboard control, command-line control, and a Python script.

In this chapter, we will expand TurtleBot's capabilities by giving the robot vision. The chapter begins by describing 3D vision systems and how they are used to map obstacles within the camera's field of view. The three types of 3D sensors typically used for TurtleBot are shown and described, detailing their specifications.

Setting up the 3D sensor for use on TurtleBot is described and the configuration is tested in a standalone mode. To visualize the sensor data coming from TurtleBot, two ROS tools are utilized: Image Viewer and rviz. Then, an important aspect of TurtleBot is described and realized: **navigation**. TurtleBot will be driven around and the vision system will be used to build a map of the environment. The map is loaded into rviz and used to give the user point and click control of TurtleBot so that it can autonomously navigate to a location selected on the map.

In this chapter, you will learn the following topics:

- How 3D vision sensors work
- The difference between the three primary 3D sensors for TurtleBot
- Information on TurtleBot environmental variables and the ROS software required for the sensors
- ROS tools for the rgb and depth camera output

- How to use TurtleBot to map a room using **Simultaneous Localization and Mapping** (**SLAM**)
- How to operate TurtleBot in autonomous navigation mode by **adaptive monte carlo localization** (**amcl**)

3D vision systems for TurtleBot

TurtleBot's capability is greatly enhanced by the addition of a 3D vision sensor. The function of 3D sensors is to map the environment around the robot by discovering nearby objects that are either stationary or moving. The mapping function must be accomplished in real time so that the robot can move around the environment, evaluate its path choices, and avoid obstacles. For autonomous vehicles, such as Google's self-driving cars, 3D mapping is accomplished by a high-cost LIDAR system that uses laser radar to illuminate its environment and analyze the reflected light. For our TurtleBot, we will present a number of low cost but effective options. These standard 3D sensors for TurtleBot include Kinect sensors, ASUS Xtion sensors, and Carmine sensors.

How these 3D vision sensors work

The 3D vision systems that we describe for TurtleBot have a common infrared technology to sense depth. This technology was developed by PrimeSense, an Israeli 3D sensing company and originally licensed to Microsoft in 2010 for the Kinect motion sensor used in the Xbox 360 gaming system. The **depth camera** uses an infrared projector to transmit beams that are reflected back to a monochrome **Complementary Metal–Oxide–Semiconductor** (**CMOS**) sensor that continuously captures image data. This data is converted into depth information, indicating the distance that each infrared beam has traveled. Data in x, y, and z distance is captured for each point measured from the sensor axis reference frame.

For a quick explanation of how 3D sensors work, the video *How the Kinect Depth Sensor Works in 2 Minutes* is worth watching at `https://www.youtube.com/watch?v=uq9SEJxZiUg`.

This 3D sensor technology is primarily for use indoors and does not typically work well outdoors. Infrared from the sunlight has a negative effect on the quality of readings from the depth camera. Objects that are shiny or curved also present a challenge for the depth camera.

Comparison of 3D sensors

Currently, three manufacturers produce 3D vision sensors that have been integrated with the TurtleBot. Microsoft Kinect, ASUS Xtion, and PrimeSense Carmine have all been integrated with camera drivers that provide a ROS interface. The ROS packages that handle the processing for these 3D sensors will be described in an upcoming section, but first, a comparison of the three products is provided.

Microsoft Kinect

Kinect was developed by Microsoft as a motion sensing device for video games, but it works well as a mapping tool for TurtleBot. Kinect is equipped with a rgb camera, a depth camera, an array of microphones, and a tilt motor.

The rgb camera acquires 2D color images in the same way in which our smart phones or webcams acquire color video images. The Kinect microphones can be used to capture sound data and a 3-axis accelerometer can be used to find the orientation of the Kinect. These features hold the promise of exciting applications for TurtleBot, but unfortunately, this book will not delve into the use of these Kinect sensor capabilities.

Kinect is connected to the TurtleBot netbook through a USB 2.0 port (USB 3.0 for Kinect v2). Software development on Kinect can be done using the Kinect **Software Develoment Kit** (**SDK**), freenect, and **OpenSource Computer Vision** (**OpenCV**). The Kinect SDK was created by Microsoft to develop Kinect apps, but unfortunately, it only runs on Windows. OpenCV is an open source library of hundreds of computer vision algorithms and provides support for mostly 2D image processing. 3D depth sensors, such as the Kinect, ASUS, and PrimeSense are supported in the `VideoCapture` class of OpenCV. **Freenect** packages and libraries are open source ROS software that provides support for Microsoft Kinect. More details on freenect will be provided in an upcoming section titled *Configuring TurtleBot and installing 3D sensor software*.

Microsoft has developed three versions of Kinect to date: Kinect for Xbox 360, Kinect for Xbox One, and Kinect for Windows v2. The following table presents images of their variations and the subsequent table shows their specifications:

Microsoft Kinect versions

Microsoft Kinect version specifications:

Spec	**Kinect 360**	**Kinect One**	**Kinect for Windows v2**
Release date	November 2010	November 2013	July 2014
Horizontal field of view (degrees)	57	57	70
Vertical field of view (degrees)	43	43	60
Color camera data	640 x 480 32-bit @ 30 fps	640 x 480 @ 30 fps	1920 x 1080 @ 30 fps

Spec	Kinect 360	Kinect One	Kinect for Windows v2
Depth camera data	320 x 240 16-bit @ 30 fps	320 x 240 @ 30 fps	512 x 424 @ 30 fps
Depth range (meters)	1.2 – 3.5	0.5 – 4.5	0.5 – 4.5
Audio	16-bit @ 16 kHz	4 microphones	
Dimensions	28 x 6.5 x 6.5 cm	25 x 6.5 x 6.5 cm	25 x 6.5.x 7.5 cm
Additional information	Motorized tilt base range ± 27 degrees; USB 2.0	Manual tilt base; USB 2.0	No tilt base; USB 3.0 only
	Requires external power		

fps: **frames per second**

ASUS

ASUS Xtion, Xtion PRO, and PRO LIVE are also 3D vision sensors designed for motion sensing applications. The technology is similar to the Kinect, using an infrared projector and a monochrome CMOS receptor to capture the depth information.

The ASUS sensor is connected to the TurtleBot netbook through a USB 2.0 port and no other external power is required. Applications for the ASUS Xtion PRO can be developed using the ASUS development solution software, OpenNi, and OpenCV for the PRO LIVE rgb camera. **OpenNi** packages and libraries are open source software that provides support for ASUS and PrimeSense 3D sensors. More details on OpenNi will be provided in the following *Configuring TurtleBot and installing 3D sensor software* section.

The following figure presents images of the ASUS sensor variations and the subsequent table shows their specifications:

ASUS Xtion and PRO versions

ASUS Xtion and PRO version specifications:

Spec	**Xtion**	**Xtion PRO**	**Xtion PRO LIVE**
Horizontal field of view (degrees)	58	58	58
Vertical field of view (degrees)	45	45	45
Color camera data	none	none	1280 x 1024
Depth camera data	unspecified	640 x 480 @ 30 fps 320 x 240 @ 60 fps	640 x 480 @ 30 fps 320 x 240 @ 60 fps
Depth range (meters)	0.8 – 3.5	0.8 – 3.5	0.8 – 3.5
Audio	none	none	2 microphones
Dimensions	18 x 3.5 x 5 cm	18 x 3.5 x 5 cm	18 x 3.5 x 5 cm

Spec	Xtion	Xtion PRO	Xtion PRO LIVE
Additional information	USB 2.0	USB 2.0	USB 2.0/ 3.0
	No additional power required—powered through USB		

PrimeSense Carmine

PrimeSense was the original developer of the 3D vision sensing technology using near-infrared light. They also developed the NiTE software that allows developers to analyze people, track their motions, and develop user interfaces based on gesture control. PrimeSense offered its own sensors, Carmine 1.08 and 1.09, to the market before the company was bought by Apple in November 2013. The Carmine sensor is shown in the following image. ROS OpenNi packages and libraries also support the PrimeSense Carmine sensors. More details on OpenNi will be provided in an upcoming section titled *Configuring TurtleBot and installing 3D sensor software*:

PrimeSense Carmine

PrimeSense has two versions of the Carmine sensor: 1.08 and the short range 1.09. The preceding image shows how the sensors look and the subsequent table shows their specifications:

Spec	Carmine 1.08	Carmine 1.09
Horizontal field of view (degrees)	57.5	57.5
Vertical field of view (degrees)	45	45
Color camera data	640 x 480 @ 30 Hz	640 x 480 @ 60 Hz
Depth camera data	640 x 480 @ 60 Hz	640 x 480 @ 60 Hz
Depth range (meters)	0.8 – 3.5	0.35 – 1.4
Audio	2 microphones	2 microphones
Dimensions	18 x 2.5 x 3.5 cm	18 x 3.5 x 5 cm
Additional information	USB 2.0 / 3.0	USB 2.0 / 3.0
	No additional power required—powered through USB	

TurtleBot uses 3D sensing for autonomous navigation and obstacle avoidance, as described later in this chapter. Other applications that these 3D sensors are used in include 3D motion capture, skeleton tracking, face recognition, and voice recognition.

Obstacle avoidance drawbacks

There are a few drawbacks that you need to know about when using the infrared 3D sensor technology for obstacle avoidance. These sensors have a narrow imaging area of about 58 degrees horizontal and 43 degrees vertical. It can also not detect anything within the first 0.5 meters (~20 inches). Highly reflective surfaces, such as metals, glass, or mirrors cannot be detected by the 3D vision sensors.

Configuring TurtleBot and installing the 3D sensor software

There are minor but important environmental variables and software that are needed for the TurtleBot based on your selection of 3D sensors. We have attached a Kinect Xbox 360 sensor to our TurtleBot, but we will provide instructions to configure each of the 3D sensors mentioned in this chapter. These environmental variables are used by the ROS launch files to launch the correct camera drivers. In ROS Indigo, the Kinect and ASUS sensors are supported by different camera drivers, as described in the following sections.

Kinect

The environmental variables for the Kinect sensors are as follows:

```
export KINECT_DRIVER=freenect
export TURTLEBOT_3D_SENSOR=kinect
```

These variables should be added to the `~/.bashrc` files of both the TurtleBot and the remote computer.

Kinect also requires a special driver for the camera to be downloaded from GitHub. Type the following commands in a terminal window on the TurtleBot netbook:

```
$ mkdir ~/kinectdriver
$ cd ~/kinectdriver
$ git clone https://github.com/avin2/SensorKinect
$ cd SensorKinect/Bin/
$ tar xvjf SensorKinect093-Bin-Linux-x64-v5.1.2.1.tar.bz2
$ cd Sensor-Bin-Linux-x64-v5.1.2.1/
$ sudo ./install.sh
```

If your netbook has an 86-bit processor, change the fifth and sixth commands, as follows:

```
$ tar xvjf SensorKinect093-Bin-Linux-x86-v5.1.2.1.tar.bz2
$ cd Sensor-Bin-Linux-x86-v5.1.2.1/
```

Libfreenect is an open source library that provides an interface for Microsoft Kinect to be used with Linux, Windows, and Mac. ROS packages for Kinect 360 and Kinect One can be installed with the following commands:

```
$ sudo apt-get install libfreenect-dev
$ sudo apt-get install ros-indigo-freenect*
```

This command installs the following packages:

- `freeglut3-dev`
- `ros-indigo-depth-image-proc`
- `ros-indigo-eigen-conversions`
- `ros-indigo-freenect-camera`
- `ros-indigo-freenect-launch`
- `ros-indigo-freenect-stack`
- `ros-indigo-image-proc`
- `ros-indigo-libfreenect`
- `ros-indigo-rgbd-launch`

Kinect for Windows v2 requires a different camera driver named **libfreenect2** and the `iai_kinect2` software toolkit. The installation of this software is described in *Chapter 9, Flying a Mission with Crazyflie*.

For the latest information on the ROS freenect software, check the ROS wiki at `http://wiki.ros.org/freenect_launch`. Maintainers of the freenect software utilize as much of the OpenNi software as possible to preserve compatibility.

ASUS and PrimeSense

The TurtleBot software for ROS Indigo is configured to working with the ASUS Xtion PRO as the default configuration. It is possible to add the following environmental variable:

```
export TURTLEBOT_3D_SENSOR=asus_xtion_pro
```

although (at this time) it is not necessary.

The `openni2_camera` ROS package supports the ASUS Xtion, Xtion PRO, and the PrimeSense 1.08 and 1.09 cameras. The `openni2_camera` package does not support any Kinect devices. This package provides drivers for the cameras to publish raw rgb, depth, and infrared image streams.

ROS packages for OpenNi2 can be installed with the following command:

```
$ sudo apt-get install ros-indigo-openni2*
```

For the latest information on the ROS OpenNi2 software, check the ROS wiki at `http://wiki.ros.org/openni2_launch`.

Camera software structure

The `freenect_camera` and `openni_camera` packages are ROS nodelet packages used to streamline the processing of the enormous quantity of image data. Initially, a nodelet manager is launched and then nodelets are added to the manager. The default 3D sensor data type for the `freenect/openni` nodelet processing is `depth_image`. The camera driver software publishes the `depth_image` message streams. These messages can be converted to point cloud data types to make them more usable for **Point Cloud Library** (**PCL**) algorithms. Basic navigation operations on TurtleBot use `depth_images` for faster processing. Launching nodelets to handle the conversion of raw depth, rgb, and IR data streams to the `depth_image`, `disparity_image`, and `registered_point_cloud` messages is the method of handling all the conversions in one process. Nodelets allow multiple algorithms to be running in a single process without creating multiple copies of the data when messages are passed between processes.

The `depthimage_to_laserscan` package uses the `depth_image` data to create `sensor_msgs/LaserScan` in order to utilize more processing power to generate maps. For more complex applications, converting `depth_images` to the point cloud format offers the advantage of using the PCL algorithms.

Defining terms

The important terms that are used in configuring TurtleBot are as follows:

Depth cloud: Depth cloud is another name for the `depth_image` produced by the 3D sensor, such as the Kinect, ASUS, and PrimeSense depth cameras.

Point cloud: A point cloud is a set of points with x, y, and z coordinates that represent the surface of an object.

Registered DepthCloud and **Registered PointCloud**: These terms are used by ROS for special DepthCloud or PointCloud data colored by the rgb image data. These data streams are available when the `depth_registration` option is selected (set to `true`).

Testing the 3D sensor in standalone mode

Before we make an attempt to control the TurtleBot from a remote computer, it is wise to test the TurtleBot in standalone mode. TurtleBot will be powered on and we will use its netbook to check whether the robot is operational on its own.

To prepare the TurtleBot, perform the following steps:

1. Plug in the power to the 3D sensor via the TurtleBot base connection (Kinect only).
2. Plug in the power to the netbook via the TurtleBot base connection.
3. Power on the netbook and establish the network connection on the netbook. This should be the network used for TurtleBot's `ROS_MASTER_URI` IP address.
4. Power on the TurtleBot base.
5. Plug in the 3D sensor to the netbook through a USB 2.0 port (Kinect for Windows v2 uses the USB 3.0 port).

Ensure that ROS environment variables are configured correctly on the netbook. Refer to the *Netbook network setup* section in *Chapter 3*, *Driving Around with TurtleBot*, and the *Configuring TurtleBot and installing 3D sensor software* section.

To test the operation of the TurtleBot 3D sensor in standalone mode, perform the following steps **on the netbook**:

1. On the TurtleBot netbook, bring up a terminal window and run the TurtleBot minimal launch:

   ```
   $ roslaunch turtlebot_bringup minimal.launch
   ```

2. Open another terminal window and start the camera nodelets for Kinect:

   ```
   $ roslaunch freenect_launch freenect.launch
   ```

   If you are using an ASUS or Carmine sensor, start the camera nodelets using the following command:

   ```
   $ roslaunch openni2_launch openni2.launch
   ```

If these commands run on TurtleBot with no errors, you are ready to proceed with running 3D visualizations from the remote computer. If you receive errors, such as **No devices connected…**, make sure that the correct camera drivers are installed, as described in the *Configuring TurtleBot and installing 3D sensor software* section. Also, make sure that the TurtleBot base is powered on.

Running ROS nodes for visualization

Viewing images on the remote computer is the next step to setting up the TurtleBot. Two ROS tools can be used to visualize the rgb and depth camera images. Image Viewer and rviz are used in the following sections to view the image streams published by the Kinect sensor.

Visual data using Image Viewer

A ROS node can allow us to view images that come from the rgb camera on Kinect. The `camera_nodelet_manager` node implements a basic camera capture program using OpenCV to handle publishing ROS image messages as a topic. This node publishes the camera images in the `/camera` namespace.

Three terminal windows will be required to launch the base and camera nodes on TurtleBot and launch the Image Viewer node on the remote computer. The steps are as follows:

1. Terminal Window 1: Minimal launch of TurtleBot:

   ```
   $ ssh <username>@<TurtleBot's IP Address>
   $ roslaunch turtlebot_bringup minimal.launch
   ```

2. Terminal Window 2: Launch freenect camera:

```
$ ssh <username>@<TurtleBot's IP Address>
$ roslaunch freenect_launch freenect.launch
```

   This `freenect.launch` file starts the `camera_nodelet_manager` node, which prepares to publish both the rgb and depth stream data. When the node is running, we can check the topics by executing the `rostopic list` command. The topic list shows the `/camera` namespace with multiple depth, `depth_registered`, `ir`, `rectify_color`, `rectify_mono`, and rgb topics.

3. To view the image messages, open a third terminal window and type the following command to bring up the Image Viewer:

```
$ rosrun image_view image_view image:=/camera/rgb/image_color
```

   This command creates the `/image_view` node that opens a window, subscribes to the `/camera/rgb/image_color` topic, and displays the image messages. These image messages are published over the network from the TurtleBot to the remote computer (a workstation or laptop). If you want to save an image frame, you can right-click on the window and save the current image to your directory.

If you are using an ASUS sensor and `openni2_launch`, the `/camera/rgb/image_color` topic does not exist. Instead, use the `/camera/rgb/image_raw` topic.

An image view of a rgb image

4. To view depth camera images, press *Ctrl* + *C* keys to end the previous `image_view` process. Then, type the following command in the third terminal window:

   ```
   $ rosrun image_view image_view image:=/camera/depth/image
   ```

   A pop-up window for Image Viewer will appear on your screen:

An image view of a depth image

To close the Image Viewer and other windows, press *Ctrl* + *C* keys in each terminal window.

Visual data using rviz

To visualize the 3D sensor data from the TurtleBot using rviz, begin by launching the TurtleBot minimal launch software. Next, a second terminal window will be opened to start the launch software for the 3D sensor.

1. Terminal Window 1: Minimal launch of TurtleBot:

   ```
   $ ssh <username>@<TurtleBot's IP Address>
   $ roslaunch turtlebot_bringup minimal.launch
   ```

2. Terminal Window 2: Launch 3D sensor software:

   ```
   $ ssh <username>@<TurtleBot's IP Address>
   $ roslaunch turtlebot_bringup 3dsensor.launch
   ```

The `3dsensor.launch` file within the `turtlebot_bringup` package configures itself based on the `TURTLEBOT_3D_SENSOR` environment variable set by the user. Using this variable, it includes a custom Kinect or ASUS Xtion PRO `launch.xml` file that contains all of the unique camera and processing parameters set for that particular 3D sensor. The `3dsensor.launch` file turns on all the sensor processing modules as the default. These modules include the following:

- `rgb_processing`
- `ir_processing`
- `depth_processing`
- `depth_registered_processing`
- `disparity_processing`
- `disparity_registered_processing`
- `scan_processing`

It is typically not desirable to generate so much sensor data for an application. The `3dsensor.launch` file allows users to set arguments to minimize the amount of sensor data generated. Typically, TurtleBot applications only turn on the sensor data needed in order to minimize the amount of processing performed. This is done by setting these `roslaunch` arguments to `false` when data is not needed.

When the `3dsensor.launch` file is executed, the `turtlebot_bringup` package launches a `/camera_nodelet_manager` node with multiple nodelets. Nodelets were described in the *Camera software structure* section. The following is a list of nodelets that are started:

NODES

/camera/

camera_nodelet_manager (nodelet/nodelet)

debayer (nodelet/nodelet)

depth_metric (nodelet/nodelet)

depth_metric_rect (nodelet/nodelet)

depth_points (nodelet/nodelet)

depth_rectify_depth (nodelet/nodelet)

depth_registered_rectify_depth (nodelet/nodelet)

disparity_depth (nodelet/nodelet)

disparity_registered_hw (nodelet/nodelet)

disparity_registered_sw (nodelet/nodelet)

driver (nodelet/nodelet)

points_xyzrgb_hw_registered (nodelet/nodelet)

points_xyzrgb_sw_registered (nodelet/nodelet)

rectify_color (nodelet/nodelet)

rectify_ir (nodelet/nodelet)

rectify_mono (nodelet/nodelet)

register_depth_rgb (nodelet/nodelet)

/

depthimage_to_laserscan (nodelet/nodelet)

Next, rviz is launched to allow us to see the various forms of visualization data. A third terminal window will be opened for the following command:

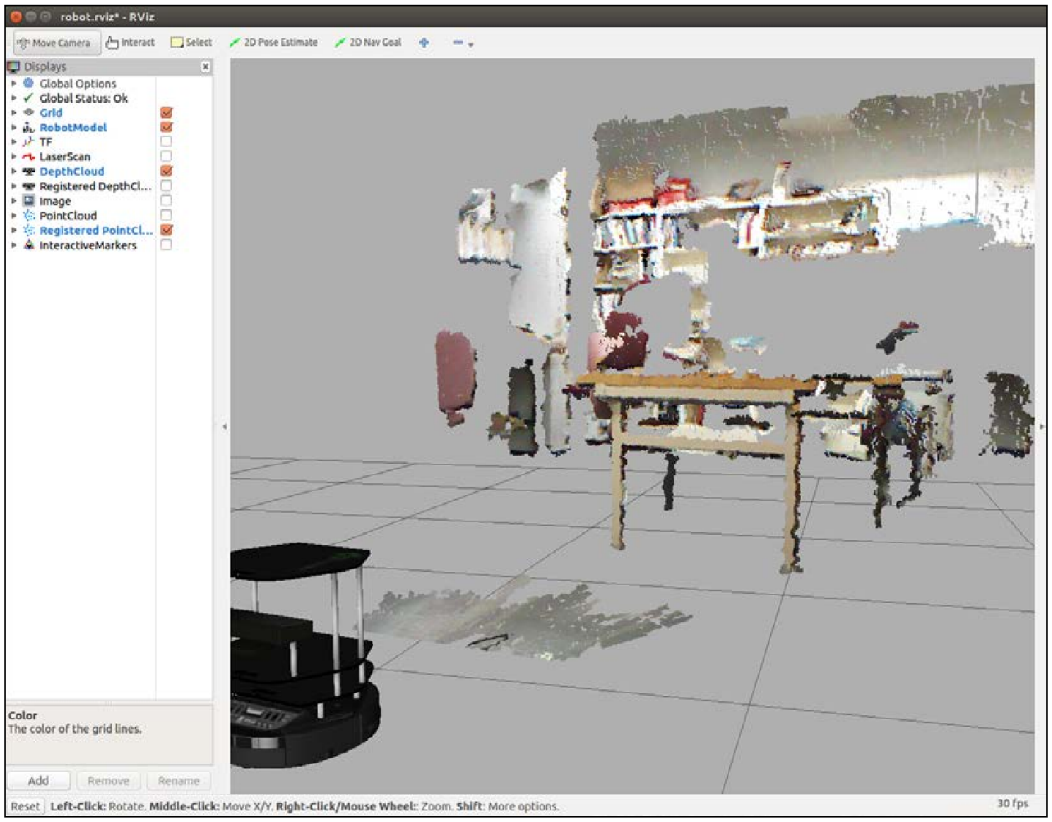
3. Terminal Window 3: View sensor data on rviz:

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```

The `turtlebot_rviz_launchers` package provides the `view_robot.launch` file for bringing up rviz and is configured to visualize the TurtleBot and its sensor output.

Within rviz, the 3D sensor data can be displayed in many formats. If images are not visible in the environment window, set the **Fixed Frame** (under **Global Options**) on the **Displays** panel to `/camera_link`. Try checking the box for the **Registered PointCloud** and rotating the TurtleBot's screen environment in order to see what the Kinect is sensing. Then wait. Patience is required because displaying a point cloud involves a lot of processing power.

The following screenshot shows the rviz display of a **Registered PointCloud** image in our lab:



A Registered PointCloud image

On the rviz **Displays** panel, the following sensors can be added and checked for display in the environment window:

- **DepthCloud**
- **Registered DepthCloud**
- **Image**
- **LaserScan**
- **PointCloud**
- **Registered PointCloud**

The following table describes the different types of image sensor displays available in rviz and the message types that they display:

Sensor name	Description	Messages used
Camera	This creates a new rendering window from the perspective of a camera and overlays the image on top of it.	`sensor_msgs/Image`, `sensor_msgs/CameraInfo`
DepthCloud, **Registered DepthCloud**	This displays point clouds based on depth maps.	`sensor_msgs/Image`
Image	This creates a new rendering window with an image. Unlike the camera display, this display does not use camera information.	`sensor_msgs/Image`
LaserScan	This shows data from a laser scan with different options for rendering modes, accumulation, and so on.	`sensor_msgs/LaserScan`
Map	This displays an occupancy grid on the ground plane.	`nav_msgs/OccupancyGrid`
PointCloud, **PointCloud2**, and **Registered PointCloud**	This shows data from a point cloud with different options for rendering modes, accumulation, and so on.	`sensor_msgs/PointCloud`, `sensor_msgs/PointCloud2`

Navigating with TurtleBot

Launch files for TurtleBot will create ROS nodes either remotely on the TurtleBot netbook (via SSH to TurtleBot) or locally on the remote computer. As a general rule, the launch files (and nodes) that handle the GUI and visualization processing should run on the remote computer while the minimal launch and camera drivers should run on the TurtleBot netbook. Note that we will specify when to SSH to TurtleBot for a ROS command or omit the SSH for using a ROS command on the remote computer.

Mapping a room with TurtleBot

TurtleBot can autonomously drive around its environment if a map is made of the environment. The 3D sensor is used to create a 2D map of the room as the TurtleBot is driven around either by a joystick, keyboard, or any other method of teleoperation.

Since we are using the Kobuki base, calibration of the gyro inside the base is not necessary. If you are using the Create base, make sure that you perform the gyro calibration procedure in the TurtleBot ROS wiki at `http://wiki.ros.org/turtlebot_calibration/Tutorials/Calibrate%20Odometry%20and%20Gyro` before you begin with the mapping operation.

Defining terms

The core terms that are used in TurtleBot navigation are as follows:

Odometry: Data gathered from moving sensors is used to estimate the change in a robot's position over time. This data is used to estimate the current position of the robot relative to its starting location.

Map: For TurtleBot, a map is a 2D representation of an environment encoded with occupancy data.

Occupancy Grid Map (**OGM**): An OGM is a map generated from the 3D sensor measurement data and the known pose of the robot. The environment is divided into an evenly-spaced grid in which the presence of obstacles is identified as a probabilistic value in each cell on the grid.

Localization: Localization determines the present position of the robot with respect to a known map. The robot uses features in the map to determine where its current position is on the map.

Building a map

The following steps are fairly complex and will require the use of four or five terminal windows. Be conscious of which commands are on TurtleBot (requiring ssh from the remote computer) and those that are on the remote computer (not requiring ssh). In each terminal window, enter the commands following the `$` prompt:

1. Terminal window 1: Minimal launch of TurtleBot

   ```
   $ ssh <username>@<TurtleBot's IP Address>
   $ roslaunch turtlebot_bringup minimal.launch
   ```

   These commands are the now familiar process of setting the many arguments and parameters and launching nodes for the TurtleBot mobile base functionality.

2. Terminal window 2: Launch the gmapping operation as follows:

   ```
   $ ssh <username>@<TurtleBot's IP Address>
   $ roslaunch turtlebot_navigation gmapping_demo.launch
   ```

   Look for the following text on your window:

   odom received!

   The `gmapping_demo` launch file launches the `3dsensor.launch` file, specifying turning off the `rgb_processing`, `depth_registration`, and `depth_processing` modules. This leaves the modules for `ir_processing`, `disparity_processing`, `disparity_registered_processing`, and `scan_processing`. The `.xml` files for `gmapping.launch` and `move_base.launch` are also invoked. `gmapping.launch.xml` launches the `slam_gmapping` node and sets multiple parameters in the `.xml` file. `move_base.launch.xml` launches the `move_base` node and also starts the nodes for `velocity_smoother` and `safety_controller`. A more complete description of this processing is provided in the following *How does TurtleBot accomplish this mapping task?* section.

3. Terminal window 3: View navigation on rviz by running the following command:

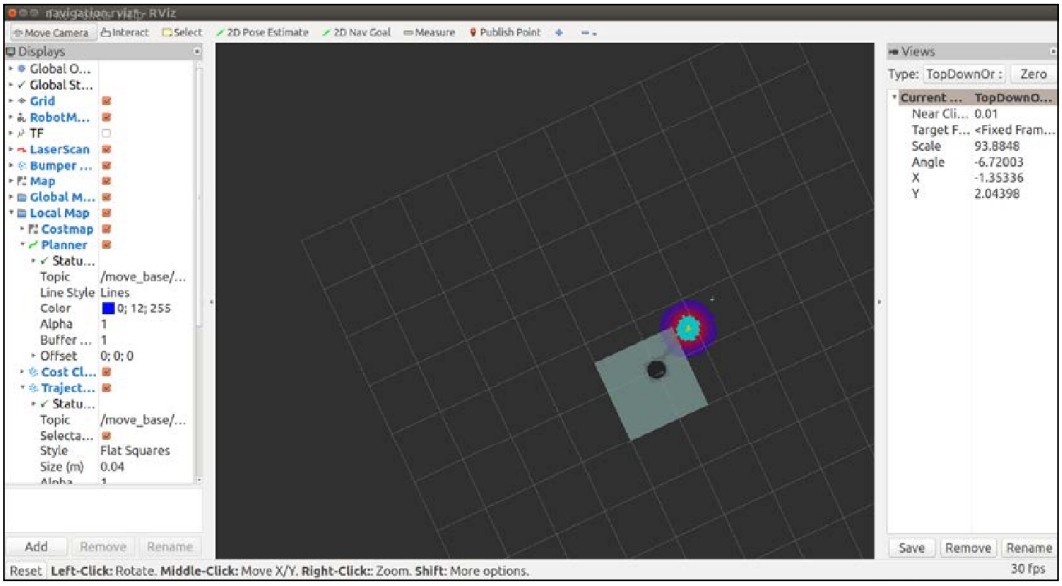
   ```
   $ roslaunch turtlebot_rviz_launchers view_navigation.launch
   ```

   Rviz should come up in the **TopDownOrtho** view identified in the **Views** panel on the right side of the screen. This environment shows a map that is the initial OGM, which shows occupied space, free space, and unknown space.

If a map is not displayed, make sure that the following display checkboxes have been selected on the **Displays** panel (on the left side):

- **Grid**
- **RobotModel**
- **LaserScan**
- **Bumper Hit**
- **Map**
- **Global Map**
- **Local Map**
- **Amcl Particle Swarm**
- **Full Plan**

Your rviz screen should display results similar to the following screenshot:



An initial gmapping screen in rviz

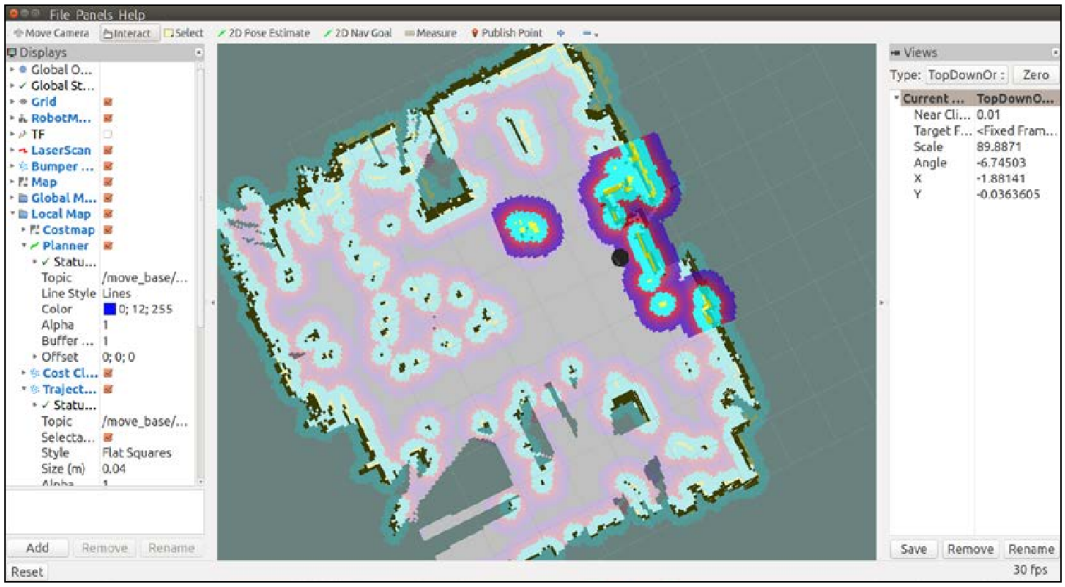
4. Terminal window 4: Keyboard control of TurtleBot:

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

Here, the keyboard navigation command is used, but the joystick teleop or interactive marker navigation can be used instead.

At this point, the operator should use keyboard commands to navigate TurtleBot completely around the environment. A representation of the map is built and can be viewed in rviz as TurtleBot's 3D sensor detects objects within its range.

The following screenshot shows a map of our lab that TurtleBot produced on rviz:



TurtleBot mapping a room

Notice that light gray areas are clear, unoccupied space, dark gray areas are unexplored areas, black indicates a solid border, such as a wall, and colored spots are obstacles in the room. The area of the brightest color is TurtleBot's local map (the area the sensor is currently detecting).

When a complete map of the environment appears on rviz, the map should be saved. Without killing any of the prior processes, open another terminal window and type the following commands:

```
$ ssh <username>@<Turtlbot's IP Address>
$ rosrun map_server map_saver -f  /home/<TurtleBot's username>/my_map
```

If you do not know the TurtleBot's username, after ssh'ing to TurtleBot, use the `pwd` command to find it.

The process creates two files: `my_map.yaml` and `my_map.pgm` and places them in your TurtleBot netbook home directory. The path and filename can be changed as you desire, but files should be saved on the TurtleBot.

The `.yaml` file contains configuration information of the map and the path and name of the `.pgm` image file. The `.pgm` file is in portable gray map format and contains the image of the OGM.

The map configuration information includes the following:

- The absolute pathname to the `.pgm` image file
- The map resolution in meters per pixel
- Coordinates (x, y, and yaw) of the origin on the lower-left corner of the grid
- A flag to reverse the `white pixel=free` and `black pixel=occupied` semantics of the map color space
- The lowest threshold value at which pixels will be considered completely occupied
- The highest threshold value at which pixels will be considered completely free

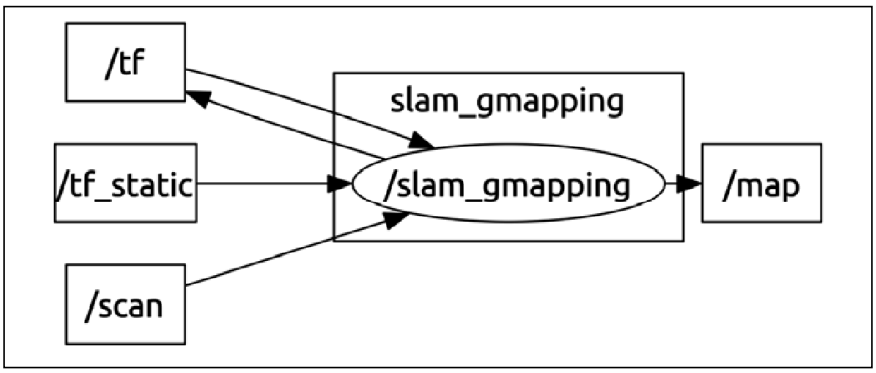
In the next section, we will examine TurtleBot's mapping process from a more in-depth ROS perspective.

How does TurtleBot accomplish this mapping task?

TurtleBot builds maps using the ROS `gmapping` package. The `gmapping` package is based on OpenSlam's Gmapping (`http://openslam.org/gmapping.html`), which is a highly efficient Rao-Blackwellized particle filter algorithm. This approach is based on a laser scan-based SLAM implementation. Although a laser scanner would work the best for SLAM, the Kinect will provide a simulated laser scan for the TurtleBot. The ROS `gmapping` package contains the `slam_gmapping` node that takes the incoming laser scan stream and transforms it to the odometry tf reference frame.

The `gmapping` process is implemented by a set of parameters within the `gmapping_demo.launch` file in the `turtlebot_navigation` package. This launch file initiates the `3dsensor.launch` file from the `turtlebot_bringup` package to handle the processing of the 3D sensor. Some of the sensor processing modules are turned off to minimize processing for this task.

The `slam_gmapping` node subscribes to the `sensor_msgs/LaserScan` messages from the `camera_nodelet_manager` node and the `tf/tfMessage` messages containing the odometry frames. The following diagram from `rqt_graph` shows the `/tf` and `/tf_static` topics (with `tf/tfMessage` messages) and the `/scan` topic (with `sensor_msgs/LaserScan` messages) being subscribed to by the `slam_gmapping` node. The `slam_gmapping` node combines this data to create an OGM of the environment. As the robot is driven around the room, the `slam_gmapping` node publishes the `/map` topic to update the OGM with an estimate of TurtleBot's location and the surrounding environment based on data from the laser scan.



slam_gmapping node

When the operator issues the command to save the map, the `map_saver` node of the `map_server` package gets activated. The `map_saver` node provides a ROS service to take the OGM data and saves it to a pair of files (the `.pgm` and `.yaml` files described in the previous section). Each cell of the OGM records its occupancy state as a color for the corresponding pixel. Free space is identified as white with a value of 0 and occupied space is identified as black with a value of 100. A special value of -1 is used for unknown (unmapped) space. The threshold values within the `.yaml` file make the pixel values between 0 and 100 categorized as occupied, free, or in-between.

Autonomous navigation with TurtleBot

ROS has implemented the concept of a **Navigation Stack**. ROS **stacks** are a collection of packages that provide a useful functionality, in this case navigation. Packages in the Navigation Stack handle the processing of odometry data and sensor streams into velocity commands for the robot base. As a differential drive base, TurtleBot takes advantage of the ROS Navigation Stack to perform tasks, such as autonomous navigation and obstacle avoidance. Therefore, understanding TurtleBot's navigation processes will provide the knowledge base for many other ROS mobile robots as well as a basic understanding of navigation for aerial and underwater robots.

In this section, we will use the map that we created in the *Mapping a room with TurtleBot* section. As an alternative, you can use a bitmap image of a map of the environment, but you will need to build the `.yaml` file by hand. Values for map resolution, coordinates of the origin, and the threshold values will need to be selected. With the environment map loaded, we will command TurtleBot to move from its present location to a given location on the map defined as its goal.

At this point, understand that:

- TurtleBot is publishing odometry data and accepting velocity commands
- Kinect is publishing 3D sensor data (fake laser scan data)
- The tf library is maintaining the transformations between `base_link`, odom frame, and the depth sensor frame of Kinect
- Our map (`my_map`) will identify the environment locations that have obstacles

Defining terms

The following are the core terms used for autonomous navigation with TurtleBot:

Amcl: The amcl algorithm works to figure out where the robot would need to be on the map in order for its laser scans to make sense. Each possible location is represented by a particle. Particles with laser scans that do not match well are removed, resulting in a group of particles representing the location of the robot in the map. The amcl node uses the particle positions to compute and publish the transform from map to `base_link`.

Global navigation: These processes perform path planning for a robot to reach a goal on the map.

Local navigation: These processes perform path planning for a robot to create paths to nearby locations on a map and avoid obstacles.

Global costmap: This costmap keeps information for global navigation. Global costmap parameters control the global navigation behavior. These parameters are stored in `global_costmap_params.yaml`. Parameters common to global and local costmaps are stored in `costmap_common_params.yaml`.

Local costmap: This costmap keeps information for local navigation. Local costmap parameters control the local navigation behavior and are stored in `local_costmap_params.yaml`.

Driving without steering TurtleBot

To navigate the environment, TurtleBot needs a map, a localization module, and a path planning module. TurtleBot can safely and autonomously navigate the environment if the map completely and accurately defines the environment.

Before we begin with the steps for autonomous navigation, check the location of your `.yaml` and `.pgm` map files created in the previous section.

As in the previous section, be conscious of which commands are on TurtleBot (requiring `ssh` from the remote computer) and those that are on the remote computer (not requiring `ssh`). In each terminal window, enter the commands following the `$` prompt:

1. Terminal Window 1: Minimal launch of TurtleBot:

   ```
   $ ssh <username>@<TurtleBot's IP Address>
   $ roslaunch turtlebot_bringup minimal.launch
   ```

2. Terminal Window 2: Launch amcl operation:

   ```
   $ ssh <username>@<TurtleBot's IP Address>
   $ roslaunch turtlebot_navigation amcl_demo.launch map_file:=/
   home/<TurtleBot's username>/my_map.yaml
   ```

   Look for the following text on your window:

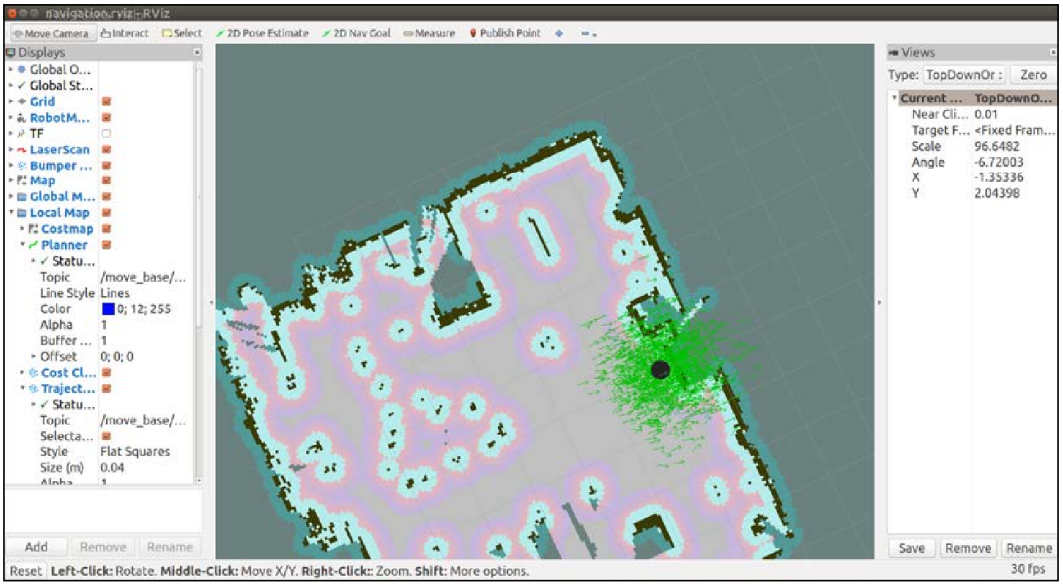
   odom received!

   The `amcl_demo` launch file launches the `3dsensor.launch` file, specifying to turning off the `rgb_processing`, `depth_registration`, and `depth_processing` modules. This leaves the modules for `ir_processing`, `disparity_processing`, `disparity_registered_processing`, and `scan_processing`. The `map_server` node is launched to read the map data from the file. The `.xml` files for `amcl.launch` and `move_base.launch` are also invoked. `amcl.launch.xml` launches the amcl node and processing sets multiple parameters in the `.xml` file. `move_base.launch.xml` launches the `move_base` node and also starts the nodes for `velocity_smoother` and `safety_controller`. A more complete description of this processing is provided in the following *How does TurtleBot accomplish this navigation task?* section.

3. Terminal Window 3: View navigation on rviz:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

   This command launches the rviz node and rviz will come up in the **TopDownOrtho** view. Your rviz screen should display results similar to the following screenshot:



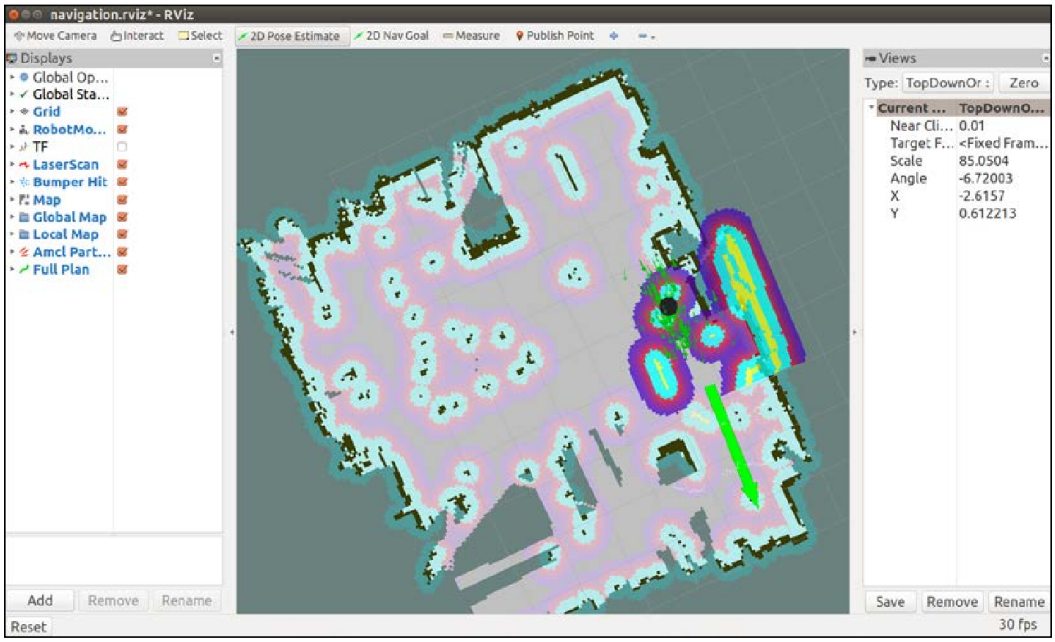
An initial amcl screen in rviz

rviz control

When `amcl_demo` loads the map of the environment, TurtleBot does not know its current location on the map. It needs a little help. Locate TurtleBot's position in the rviz environment and let TurtleBot know this location by performing the following steps:

1. Click on the **2D Pose Estimate** button on the tool toolbar at the top of the main screen.
2. Click on the location on the map where TurtleBot is located **and** drag the mouse in the direction TurtleBot is facing.

A giant green arrow will appear to help you align the direction of TurtleBot's orientation, as shown in the following screenshot:



TurtleBot 2D Pose Estimate

When the mouse button is released, a collection of small arrows will appear around TurtleBot to show the direction. If the location and/or orientation are not correct, these steps can be repeated.

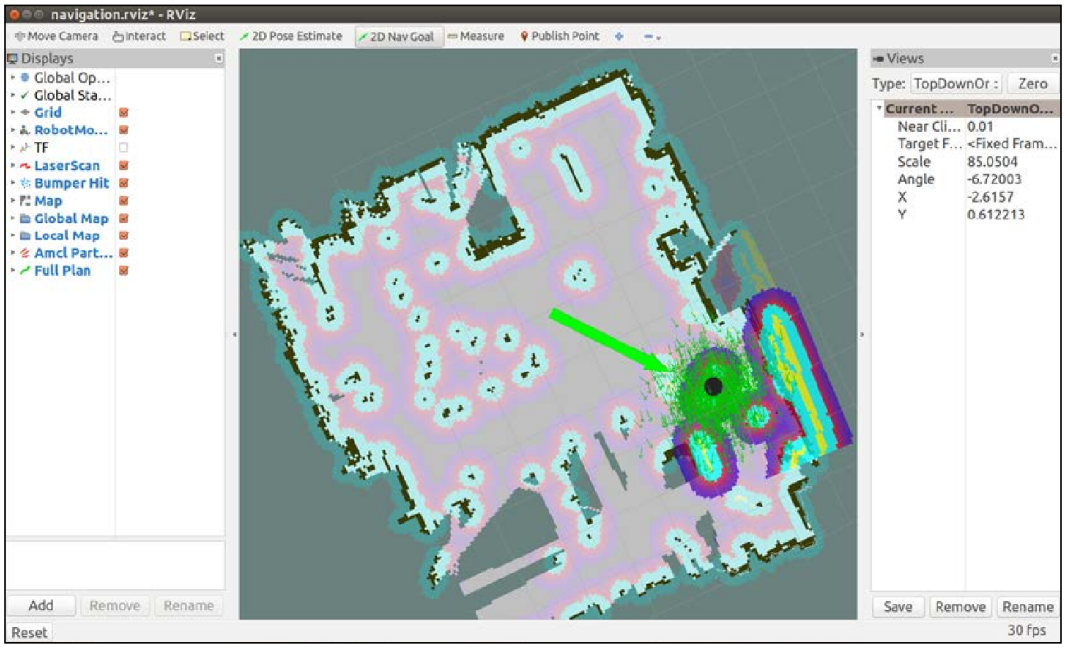
The previous steps seed TurtleBot's localization, so it has some idea where it is on the environment map. To improve the accuracy of the localization, it is best to drive TurtleBot around a bit so that the estimate of its current position converges when comparing data from the map with TurtleBot's current sensor streams. Use one of the teleoperation methods previously discussed. Be careful driving around the environment because there is no obstacle avoidance software running at this point. TurtleBot can be driven into obstacles even though they appear on its map.

Next, we can command TurtleBot to a new location and orientation in the room by identifying a goal:

1. Click on the **2D Nav Goal** button on the tool toolbar at the top of the main screen.
2. Click on the location on the map where you want TurtleBot to go and drag the mouse in the direction TurtleBot should be facing when it is finished.

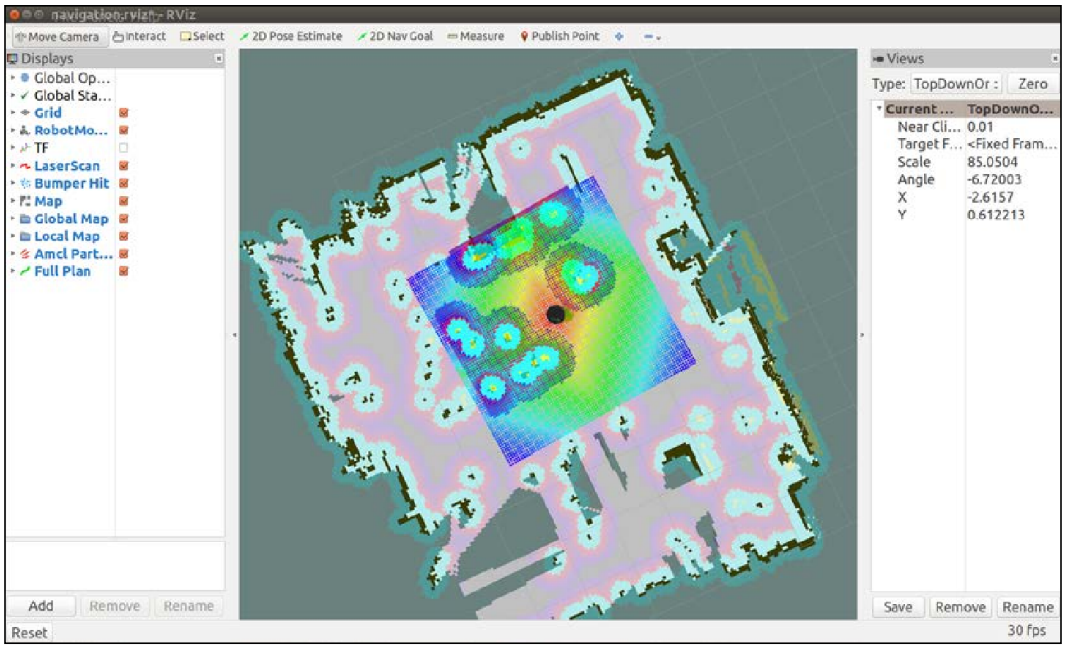
Warning: Try to avoid navigating near obstacles that have low protrusions that will not be detected by the 3D sensor. In our lab, the extensions at the base of the Baxter robot cannot be seen by the TurtleBot.

The following screenshot shows setting the navigation goal for our TurtleBot:



TurtleBot 2D Nav Goal

The following screenshot shows our TurtleBot accomplishing the goal:



TurtleBot reaches its goal

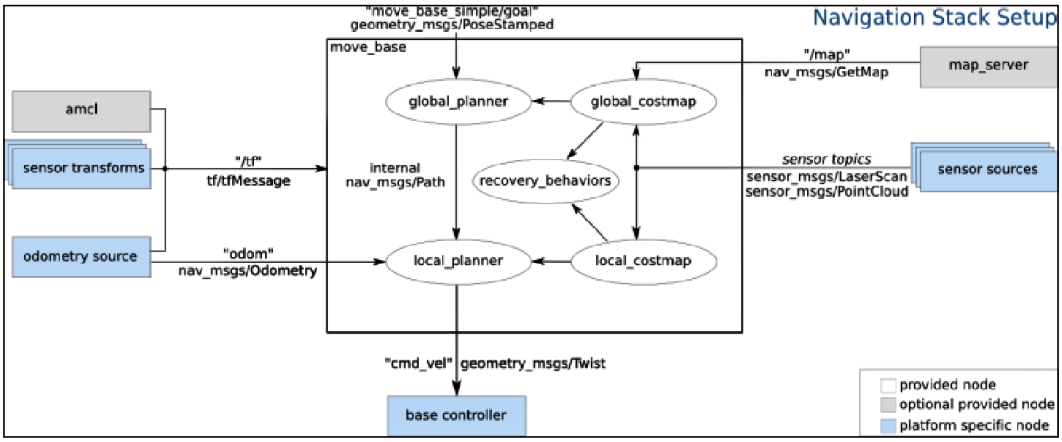
TurtleBot can also perform obstacle avoidance during autonomous navigation. While TurtleBot is on its way to a goal, step in front of it (at least 0.5 meters (1.6 feet) in front of the Kinect) and see that TurtleBot will move around you. Objects can be moved around or doors can be opened or closed to alter the environment. TurtleBot can also respond to the teleoperation control during this autonomous navigation.

In the next section, we will examine TurtleBot's autonomous navigation process from a more in-depth ROS perspective.

How does TurtleBot accomplish this navigation task?

At the highest level of processing, ROS navigation acquires odometry data from the robot base, 3D sensor data, and a goal robot pose. To accomplish the autonomous navigation task, safe velocity commands are sent to the robot to move it to the goal location.

TurtleBot's navigation package, `turtlebot_navigation`, contains a collection of launch and YAML configuration files to launch nodes with the flexibility of modifying process parameters on-the-fly. The following diagram shows an overview of the navigation process:



The ROS navigation process

When the **amcl** node is launched, it begins providing localization information on the robot based on the current 3D sensor scans (**sensor_msgs/LaserScan**), tf transforms (**tf/tfMessage**), and the OGM (`nav_msgs/OccupancyGrid`). When a **2D Pose Estimate** is input by the operator, an `initialpose` message (`geometry_msgs/PoseWithCovaianceStamped`) resets the localization parameter and reinitializes the amcl particle filter. As laser scans are read, amcl resolves the data to the odometry frame. The **amcl** node provides TurtleBot's estimated position in the map (`geometry_msgs/PoseWithCovarianceStamped`), a particle cloud (`geometry_msgs/PoseArray`), and the tf transforms for odom (`tf/tfMessage`).

The main component of the TurtleBot navigation is the **move_base** node. This node performs the task of commanding the TurtleBot to make an attempt to reach the goal location. This task is set as a preemptable action based on its implementation as a ROS action and TurtleBot's progress toward the goal is provided as feedback. The **move_base** node uses a global and a local planner to accomplish the task. Two costmaps, **global_costmap** and **local_costmap**, are also maintained for the planners by the **move_base** node.

The behavior of the `move_base` node relies on the following YAML files:

- `costmap_common_params.yaml`
- `local_costmap_params.yaml`
- `global_costmap_params.yaml`
- `dwa_local_planner_params.yaml`
- `move_base_params.yaml`
- `global_planner_params.yaml`
- `navfn_global_planner_params.yaml`

The global planner and costmap are used to create long-term plans over the entire environment, such as path planning for the robot to get to its goal. The local planner and costmap are primarily used for interim goals and obstacle avoidance.

The `move_base` node receives the goal information as a pose with position and orientation of the robot in relation to its reference frame. A `move_base_msg/MoveBaseActionGoal` message is used to specify the goal. The global planner will calculate a route from the robot's starting location to the goal taking into account data from the map. The 3D sensor will publish `sensor_msgs/LaserScan` with information on obstacles in the world to be avoided. The local planner will send navigation commands for TurtleBot to steer around objects even if they are not on the map. Navigation velocity commands are generated by the `move_base` node as `geometry_msgs/Twist` messages. TurtleBot's base will use the `cmd_vel.linear.x`, `cmd_vel.linear.y`, and `cmd_vel.angular.z` velocities for the base motors.

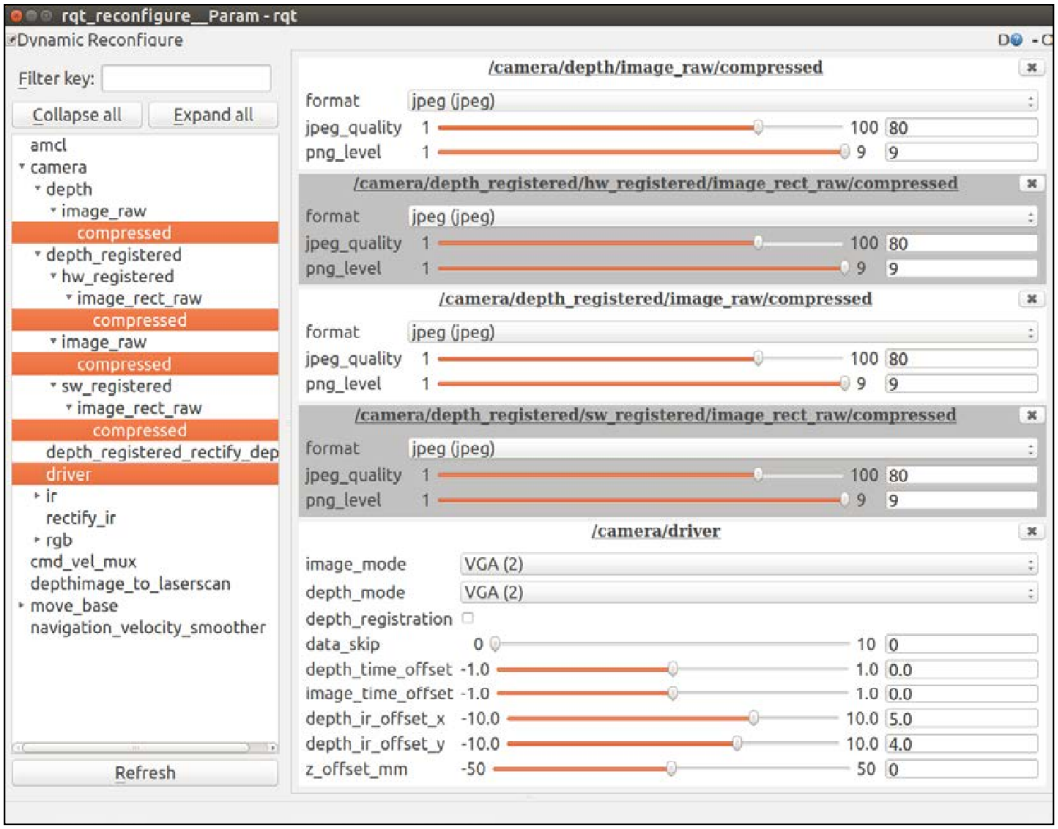
Goal tolerance is a parameter set by the user to specify the acceptable limit for achieving the goal pose. The `move_base` node will attempt certain recovery behaviors if TurtleBot is stuck and cannot proceed. These recovery behaviors include clearing out the supplied map and using sensor data by rotating in place.

rqt_reconfigure

The many parameters involved in TurtleBot navigation can be tweaked on-the-fly by using the `rqt_reconfigure` tool. This tool was previously named Dynamic Reconfigure and this name still appears on the screen. To activate this rqt plugin, use the following command:

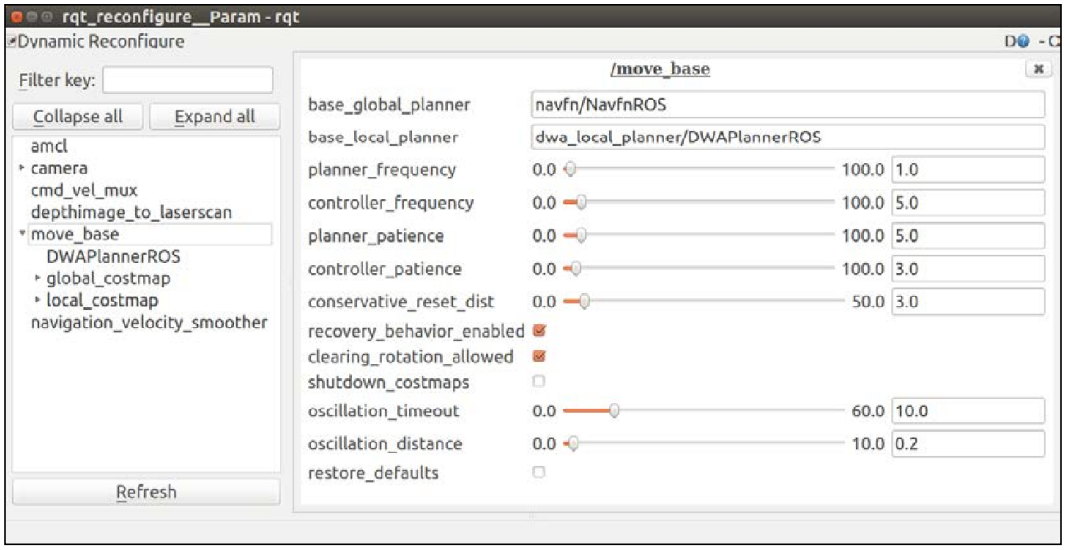
```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Nodes that have been programmed using the `rqt_reconfigure` API will be visible on the `rqt_reconfigure` GUI. On the GUI, nodes can be selected and a window with the nodes' parameters will appear with the current values and range limits. Sliders and input boxes allow the user to enter new values that will dynamically overwrite the current values. The following screenshot shows configuration parameters that can be changed for the `/camera/depth`, `/camera/depth_registered`, and `/camera/driver`:



rqt_reconfigure camera parameters

The parameters for the `move_base` node control can be accessed through `rqt_reconfigure`. These parameters are set by the `move_base_params.yaml` file mentioned in the previous section. This screen identifies the `base_global_planner` and the `base_local_planner` as well as how often to update the planning process (`planner_frequency`) and so on. These parameters allow the operator to tweak the performance of the software during an operation.



rqt_reconfigure move_base parameters

Exploring ROS navigation further

The ROS wiki provides extensive information on all aspects of setting up and configuring the navigation parameters. The following links are provided to enhance your understanding:

- `http://wiki.ros.org/navigation`
- `http://wiki.ros.org/navigation/Tutorials/RobotSetup`
- `http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide`

The following book is worth reading to gain an understanding on amcl and robotic navigation:

Probabilistic Robotics by *Thrum, Burgard,* and *Fox* by *MIT Press*

Summary

TurtleBot comes with its own 3D vision system that is a low-cost laser scanner. The Kinect, ASUS, or PrimeSense devices can be mounted on the TurtleBot base and provide a 3D depth view of the environment. This chapter provides a comparison of these three types of sensors and identifies the software that is needed to operate them as ROS components. We check their operation by testing the sensor on TurtleBot in standalone mode. To use the devices, we can utilize Image Viewer or rviz to view image streams from the rgb or depth cameras.

The primary objective is for TurtleBot to see its surroundings and be able to autonomously navigate through them. First, TurtleBot is driven around in teleoperation mode to create a map of the environment. The map provides the room boundaries and obstacles so that TurtleBot's navigation algorithm, amcl, can plan a path through the environment from its start location to a user-defined goal.

In the next chapter, we will return to the ROS simulation world and create a robot arm. The development of a URDF for a robotic arm and control of it in simulation will then prepare us to examine the robotic arms of Baxter in *Chapter 6*, *Wobbling Robot Arms Using Joint Control*. Using Baxter's robot arms, we will explore the complexities of multiple joint control and the mathematics of kinematic solutions for positioning multiple joints.