



Community Experience Distilled

Mastering ROS for Robotics Programming

Design, build, and simulate complex robots using Robot Operating System and master its out-of-the-box functionalities

Lentin Joseph

www.it-ebooks.info

[PACKT] open source*
PUBLISHING
community experience distilled

Mastering ROS for Robotics Programming

Design, build, and simulate complex robots using Robot Operating System and master its out-of-the-box functionalities

Lentin Joseph



BIRMINGHAM - MUMBAI

Mastering ROS for Robotics Programming

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2015

Production reference: 1141215

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

ISBN 978-1-78355-179-8

www.packtpub.com

Credits

Author

Lentin Joseph

Project Coordinator

Harshal Ved

Reviewers

Jonathan Cacace
Ruixiang Du

Proofreader

Safis Editing

Acquisition Editor

Vivek Anantharaman

Indexer

Tejal Daruwale Soni

Content Development Editor

Athira Laji

Production Coordinator

Melwyn D'sa

Technical Editor

Ryan Kochery

Cover Work

Melwyn D'sa

Copy Editor

Merilyn Pereira
Alpha Singh

About the Author

Lentin Joseph is an author, entrepreneur, electronics engineer, robotics enthusiast, machine vision expert, embedded programmer, and the founder and CEO of Qbotics Labs (<http://www.qboticslabs.com>) from India. He completed his bachelor's degree in electronics and communication engineering at the Federal Institute of Science and Technology (FISAT), Kerala. For his final year engineering project, he made a social robot that can interact with people. The project was a huge success and was mentioned in many forms of visual and print media. The main features of this robot were that it can communicate with people and reply intelligently and has some image processing capabilities such as face, motion, and color detection. The entire project was implemented using the Python programming language. His interest in robotics, image processing, and Python started with that project.

After his graduation, for 3 years he worked in a start-up company focusing on robotics and image processing. In the meantime, he learned famous robotic software platforms such as Robot Operating System (ROS), V-REP, Actin (a robotic simulation tool), and image processing libraries such as OpenCV, OpenNI, and PCL. He also knows robot 3D designing and embedded programming on Arduino and Tiva Launchpad.

After 3 years of work experience, he started a new company called Qbotics Labs, which mainly focuses on research to build up some great products in domains such as robotics and machine vision. He maintains a personal website (<http://www.lentinjoseph.com>) and a technology blog called technolabsz (<http://www.technolabsz.com>). He publishes his works on his tech blog. He was also a speaker at PyCon2013, India, on the topic *Learning Robotics using Python*.

Lentin is the author of the book *Learning Robotics Using Python* (refer to <http://learn-robotics.com> to know more) by Packt Publishing. The book was about building an autonomous mobile robot using ROS and OpenCV. The book was launched in ICRA 2015 and was featured in the ROS blog, Robohub, OpenCV, the Python website, and various other such forums.

Lentin was a finalist in the ICRA 2015 challenge, HRATC (<http://www2.isr.uc.pt/~embedded/events/HRATC2015/Welcome.html>).

I dedicate this book to my parents because they gave me the inspiration to write this book. I also convey my regards to my friends who helped and inspired me to write this book.

About the Reviewers

Jonathan Cacace was born in Naples, Italy, on December 13, 1987. He received has a bachelor's and master's degree in computer science from the University of Naples Federico II. Currently, he is attending a PhD Scholar Course in Information and Automation Engineering at the Department of Electrical Engineering and Information Technology (DIETI) in the same institution. His research is focused on autonomous action planning and execution by mobile robots, high-level and low-level control of UAV platforms, and human-robot interaction with humanoid robots in service task execution. He is the author and coauthor of several scientific publications in the robotics field, published at international conferences and scientific journals.

Jonathan is a member of the PRISMA Laboratory (<http://prisma.dieti.unina.it/>) of the University of Naples Federico II. With his research group, he is involved in different EU-funded collaborative research projects focused on several topics, such as the use of unmanned aerial vehicles for search and rescue operations or service task execution (<http://www.sherpa-project.eu/sherpa/> and <http://www.arcas-project.eu/>) and the dynamic manipulation of elastic objects using humanoid robotic platforms (<http://www.rodyman.eu/>).

Ruixiang Du is currently a PhD student studying Robotics at Worcester Polytechnic Institute (WPI). He received his bachelor's degree in Automation from North China Electric Power University in 2011 and a master's degree in Robotics Engineering from WPI in 2013.

Ruixiang has worked on various robotic projects with robot platforms ranging from medical robots, UAV/UGV, to humanoid robots. He was an active member of Team WPI-CMU for the DARPA Robotics Challenge.

Ruixiang has general interests in robotics and in real-time and embedded systems. His research focus is on the control and motion planning of mobile robots in cluttered and dynamic environments.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	xi
Chapter 1: Introduction to ROS and Its Package Management	1
Why should we learn ROS?	2
Why we prefer ROS for robots	2
Why some do not prefer ROS for robots	4
Understanding the ROS file system level	5
ROS packages	7
ROS meta packages	9
ROS messages	10
The ROS services	12
Understanding the ROS computation graph level	12
Understanding ROS nodes	15
ROS messages	16
ROS topics	16
ROS services	17
ROS bags	18
Understanding ROS Master	19
Using the ROS parameter	20
Understanding ROS community level	22
What are the prerequisites to start with ROS?	22
Running ROS Master and ROS parameter server	23
Checking the roscore command output	25
Creating a ROS package	26
Working with ROS topics	28
Creating ROS nodes	28
Building the nodes	32
Adding custom msg and srv files	34
Working with ROS services	37
Working with ROS actionlib	42

Table of Contents

Building the ROS action server and client	46
Creating launch files	48
Applications of topics, services, and actionlib	50
Maintaining the ROS package	51
Releasing your ROS package	51
Preparing the ROS package for the release	52
Releasing our package	53
Creating a Wiki page for your ROS package	55
Questions	57
Summary	58
Chapter 2: Working with 3D Robot Modeling in ROS	59
ROS packages for robot modeling	60
Understanding robot modeling using URDF	61
Creating the ROS package for the robot description	64
Creating our first URDF model	64
Explaining the URDF file	66
Visualizing the robot 3D model in Rviz	68
Interacting with pan and tilt joints	69
Adding physical and collision properties to a URDF model	70
Understanding robot modeling using xacro	71
Using properties	72
Using the math expression	73
Using macros	73
Conversion of xacro to URDF	73
Creating the robot description for a seven DOF robot manipulator	74
Arm specification	75
Type of joints	75
Explaining the xacro model of seven DOF arm	75
Using constants	76
Using macros	76
Including other xacro files	77
Using meshes in the link	77
Working with the robot gripper	78
Viewing the seven DOF arm in Rviz	79
Understanding joint state publisher	80
Understanding the robot state publisher	81
Creating a robot model for the differential drive mobile robot	82
Questions	86
Summary	87

Table of Contents

Chapter 3: Simulating Robots Using ROS and Gazebo	89
Simulating the robotic arm using Gazebo and ROS	90
The Robotic arm simulation model for Gazebo	91
Adding colors and textures to the Gazebo robot model	93
Adding transmission tags to actuate the model	93
Adding the gazebo_ros_control plugin	94
Adding a 3D vision sensor to Gazebo	94
Simulating the robotic arm with Xtion Pro	96
Visualizing the 3D sensor data	97
Moving robot joints using ROS controllers in Gazebo	99
Understanding the ros_control packages	99
Different types of ROS controllers and hardware interfaces	100
How the ROS controller interacts with Gazebo	100
Interfacing joint state controllers and joint position controllers to the arm	102
Launching the ROS controllers with Gazebo	103
Moving the robot joints	105
Simulating a differential wheeled robot in Gazebo	106
Adding the laser scanner to Gazebo	107
Moving the mobile robot in Gazebo	109
Adding joint state publishers in the launch file	110
Adding the ROS teleop node	111
Questions	112
Summary	113
Chapter 4: Using the ROS MoveIt! and Navigation Stack	115
Installing MoveIt!	116
MoveIt! architecture	116
The move_group node	117
Motion planning using MoveIt!	118
Motion planning request adapters	120
MoveIt! planning scene	120
MoveIt! kinematics handling	121
MoveIt! collision checking	121
Generating MoveIt! configuration package using Setup Assistant tool	122
Step 1 – Launching the Setup Assistant tool	123
Step 2 – Generating the Self-Collision matrix	124
Step 3 – Adding virtual joints	125
Step 4 – Adding planning groups	126
Step 5 – Adding the robot poses	127
Step 6 – Setup the robot end effector	128
Step 7 – Adding passive joints	128
Step 8 – Generating configuration files	129

Motion planning of robot in Rviz using MoveIt! configuration package	130
Using the Rviz MotionPlanning plugin	131
Interfacing the MoveIt! configuration package to Gazebo	134
Step 1 – Writing the controller configuration file for MoveIt!	134
Step 2 – Creating the controller launch files	135
Step 3 – Creating the controller configuration file for Gazebo	136
Step 4 – Creating the launch file for Gazebo trajectory controllers	137
Step 5 – Debugging the Gazebo- MoveIt! interface	139
Understanding ROS Navigation stack	140
ROS Navigation hardware requirements	141
Working with Navigation packages	142
Understanding the move_base node	143
Working of Navigation stack	144
Localizing on the map	144
Sending a goal and path planning	145
Collision recovery behavior	145
Sending the command velocity	145
Installing ROS Navigation stack	145
Building a map using SLAM	146
Creating a launch file for gmapping	146
Running SLAM on the differential drive robot	148
Implementing autonomous navigation using AMCL and a static map	151
Creating an AMCL launch file	152
Questions	155
Summary	155
Chapter 5: Working with Pluginlib, Nodelets, and Gazebo Plugins	157
Understanding pluginlib	158
Creating plugins for the calculator application using pluginlib	158
Working with pluginlib_calculator package	159
Understanding ROS nodelets	165
Creating a nodelet	165
Step 1 – Creating a package for nodelet	166
Step 2 – Creating hello_world.cpp nodelet	166
Step 3 – Explanation of hello_world.cpp	166
Step 4 – Creating plugin description file	167
Step 5 – Adding the export tag in package.xml	168
Step 6 – Editing CMakeLists.txt	168
Step 7 – Building and running nodelets	168
Step 8 – Creating launch files for nodelets	170

Understanding the Gazebo plugins	172
Creating a basic world plugin	173
Questions	177
Summary	177
Chapter 6: Writing ROS Controllers and Visualization Plugins	179
 Understanding pr2_mechanism packages	181
pr2_controller_interface package	181
Initialization of the controller	182
Starting the ROS controller	182
Updating ROS controller	183
Stopping the controller	183
pr2_controller_manager	183
 Writing a basic real-time joint controller in ROS	184
Step 1 – Creating controller package	184
Step 2 – Creating controller header file	184
Step 3 – Creating controller source file	185
Step 4 – Explanation of the controller source file	187
Step 5 – Creating plugin description file	188
Step 6 – Updating package.xml	188
Step 7 – Updating CMakeLists.txt	188
Step 8 – Building controller	188
Step 9 – Writing controller configuration file	189
Step 10 – Writing launch file for the controller	189
Step 11 – Running controller along with PR2 simulation in Gazebo	190
 Understanding ros_control packages	191
 Understanding ROS visualization tool (RViz) and its plugins	192
Displays panel	193
RViz toolbar	193
Views	193
Time panel	193
Dockable panels	193
 Writing a RViz plugin for teleoperation	194
Methodology of building Rviz plugin	194
Step 1 – Creating RViz plugin package	195
Step 2 – Creating RViz plugin header file	195
Step 3 – Creating RViz plugin definition	196
Step 4 – Creating plugin description file	198
Step 5 – Adding export tags in package.xml	198
Step 6 – Editing CMakeLists.txt	198
Step 7 – Building and loading plugins	199
Questions	200
Summary	201

Table of Contents

Chapter 7: Interfacing I/O Boards, Sensors, and Actuators to ROS	203
Understanding the Arduino–ROS interface	204
What is the Arduino–ROS interface?	205
Understanding the rosserial package in ROS	206
Installing rosserial packages on Ubuntu 14.04/15.04	208
ROS – Arduino Publisher and Subscriber example	213
Arduino-ROS, example – blink LED and push button	218
Arduino-ROS, example – Accelerometer ADXL 335	221
Arduino-ROS, example – ultrasonic distance sensor	224
Equations to find distance using the ultrasonic range sensor	225
Arduino-ROS, example – Odometry Publisher	228
Interfacing Non-Arduino boards to ROS	230
Setting ROS on Odroid-C1 and Raspberry Pi 2	230
How to install an OS image to Odroid-C1 and Raspberry Pi 2	233
Installation in Windows	233
Installation in Linux	234
Connecting to Odroid-C1 and Raspberry Pi 2 from a PC	235
Configuring an Ethernet hotspot for Odroid-C1 and Raspberry Pi 2	236
Installing Wiring Pi on Odroid-C1	237
Installing Wiring Pi on Raspberry Pi 2	238
Blinking LED using ROS on Odroid-C1 and Raspberry Pi 2	240
Push button + blink LED using ROS on Odroid-C1 and Raspberry Pi 2	242
Interfacing Dynamixel actuators to ROS	246
Questions	247
Summary	247
Chapter 8: Programming Vision Sensors using ROS, Open-CV, and PCL	249
Understanding ROS – OpenCV interfacing packages	250
Understanding ROS – PCL interfacing packages	251
Installing ROS perception	252
Interfacing USB webcams in ROS	254
Working with ROS camera calibration	256
Converting images between ROS and OpenCV using cv_bridge	259
Image processing using ROS and OpenCV	260
Step 1: Creating ROS package for the experiment	260
Step 2: Creating source files	260
Step 3: Explanation of the code	260
Step 4: Editing the CMakeLists.txt file	264
Step 5: Building and running example	264

Interfacing Kinect and Asus Xtion Pro in ROS	265
Interfacing Intel Real Sense camera with ROS	268
Working with point cloud to laser scan package	270
Interfacing Hokuyo Laser in ROS	273
Interfacing Velodyne LIDAR in ROS	275
Working with point cloud data	278
How to publish a point cloud	278
How to subscribe and process the point cloud	280
Writing a point cloud data to a PCD file	282
Read and publish point cloud from a PCD file	282
Streaming webcam from Odroid using ROS	285
Questions	288
Summary	288
Chapter 9: Building and Interfacing Differential Drive	
Mobile Robot Hardware in ROS	289
Introduction to Chefbot- a DIY mobile robot and its hardware configuration	290
Flashing Chefbot firmware using Energia IDE	293
Serial data sending protocol from LaunchPad to PC	294
Serial data sending protocol from PC to Launchpad	295
Discussing Chefbot interface packages on ROS	296
Computing odometry from encoder ticks	302
Computing motor velocities from ROS twist message	305
Running robot stand alone launch file using C++ nodes	306
Configuring the Navigation stack for Chefbot	306
Configuring the gmapping node	307
Configuring the Navigation stack packages	308
Common configuration (local_costmap) and (global_costmap)	310
Configuring global costmap parameters	311
Configuring local costmap parameters	312
Configuring base local planner parameters	312
Configuring DWA local planner parameters	313
Configuring move_base node parameters	314
Understanding AMCL	317
Understanding RViz for working with the Navigation stack	320
2D Pose Estimate button	320
Visualizing the particle cloud	321
The 2D Nav Goal button	322
Displaying the static map	323
Displaying the robot footprint	324
Displaying the global and local cost map	325
Displaying the global plan, local plan, and planner plan	326
The current goal	327

Table of Contents

Obstacle avoidance using the Navigation stack	328
Working with Chefbot simulation	329
Building a room in Gazebo	329
Adding model files to the Gazebo model folder	331
Sending a goal to the Navigation stack from a ROS node	333
Questions	336
Summary	336
Chapter 10: Exploring the Advanced Capabilities of ROS-MoveIt!	337
Motion planning using the move_group C++ interface	338
Motion planning a random path using MoveIt! C++ APIs	338
Motion planning a custom path using MoveIt! C++ APIs	340
Collision checking in robot arm using MoveIt!	342
Adding a collision object in MoveIt!	342
Removing a collision object from the planning scene	346
Checking self collision using MoveIt! APIs	347
Working with perception using MoveIt! and Gazebo	349
Grasping using MoveIt!	355
Working with robot pick and place task using MoveIt!	358
Creating Grasp Table and Grasp Object in MoveIt!	360
Pick and place action in Gazebo and real Robot	363
Understanding Dynamixel ROS Servo controllers for robot hardware interfacing	364
The Dynamixel Servos	364
Dynamixel-ROS interface	366
Interfacing seven DOF Dynamixel based robotic arm to ROS MoveIt!	366
Creating a controller package for COOL arm robot	368
MoveIt! configuration of the COOL Arm	372
Questions	373
Summary	374
Chapter 11: ROS for Industrial Robots	375
Understanding ROS-Industrial packages	376
Goals of ROS-Industrial	376
ROS-Industrial – a brief history	377
Benefits of ROS-Industrial	377
Installing ROS-Industrial packages	377
Block diagram of ROS-Industrial packages	378
Creating URDF for an industrial robot	380
Creating MoveIt! configuration for an industrial robot	382
Updating the MoveIt! configuration files	385

Testing the Moveit! configuration	387
Installing ROS-Industrial packages of universal robotic arm	387
Installing the ROS interface of universal robots	388
Understanding the Moveit! configuration of a universal robotic arm	390
Working with Moveit! configuration of ABB robots	394
Understanding the ROS-Industrial robot support packages	396
Visualizing the ABB robot model in RViz	398
ROS-Industrial robot client package	399
Designing industrial robot client nodes	400
ROS-Industrial robot driver package	401
Understanding Moveit! IKFast plugin	404
Creating the Moveit! IKFast plugin for the ABB-IRB6640 robot	404
Prerequisites for developing the Moveit! IKFast plugin	404
OpenRave and IK Fast Module	405
Moveit! IK Fast	405
Installing Moveit! IKFast package	405
Installing OpenRave on Ubuntu 14.04.3	406
Creating the COLLADA file of a robot to work with OpenRave	408
Generating the IKFast CPP file for the IRB 6640 robot	410
Creating the Moveit! IKFast plugin	411
Questions	413
Summary	414
Chapter 12: Troubleshooting and Best Practices in ROS	415
Setting up Eclipse IDE on Ubuntu 14.04.3	416
Setting ROS development environment in Eclipse IDE	417
Global settings in Eclipse IDE	418
ROS compile script for Eclipse IDE	419
Adding ROS Catkin package to Eclipse	421
Adding run configurations to run ROS nodes in Eclipse	427
Best practices in ROS	429
ROS C++ coding style guide	429
Standard naming conventions used in ROS	429
Code license agreement	430
ROS code formatting	431
ROS code documentation	432
Console output	432
Best practices in the ROS package	432
Important troubleshooting tips in ROS	433
Usage of roswtf	434
Questions	437
Summary	437
Index	439

Preface

Mastering ROS for Robotics Programming is an advanced guide of ROS that is very suitable for readers who already have a basic knowledge in ROS. ROS is widely used in robotics companies, universities, and robotics research institutes for designing, building, and simulating a robot model and interfacing it into real hardware. ROS is now an essential requirement for Robotic engineers; this guide can help you acquire knowledge of ROS and can also help you polish your skills in ROS using interactive examples. Even though it is an advanced guide, you can see the basics of ROS in the first chapter to refresh the concepts. It also helps ROS beginners. The book mainly focuses on the advanced concepts of ROS, such as ROS Navigation stack, ROS MoveIt!, ROS plugins, nodelets, controllers, ROS Industrial, and so on.

You can work with the examples in the book without any special hardware; however, in some sections you can see the interfacing of I/O boards, vision sensors, and actuators to ROS. To work with this hardware, you will need to buy it.

The book starts with an introduction to ROS and then discusses how to build a robot model in ROS for simulating and visualizing. After the simulation of robots using Gazebo, we can see how to connect the robot to Navigation stack and MoveIt!. In addition to this, we can see ROS plugins, controllers, nodelets, and interfacing of I/O boards and vision sensors. Finally, we can see more about ROS Industrial and troubleshooting and best practices in ROS.

What this book covers

Chapter 1, Introduction to ROS and Its Package Management, gives you an understanding of the core underlying concepts of ROS and how to work with ROS packages.

Chapter 2, Working with 3D Robot Modeling in ROS, discusses the design of two robots; one is a seven-DOF (Degree of Freedom) manipulator and the other is a differential drive robot.

Chapter 3, Simulating Robots Using ROS and Gazebo, discusses the simulation of seven-DOF arms, differential wheeled robots, and ROS controllers that help control robot joints in Gazebo.

Chapter 4, Using the ROS MoveIt! and Navigation Stack, interfaces out-of-the-box functionalities such as robot manipulation and autonomous navigation using ROS MoveIt! and Navigation stack.

Chapter 5, Working with Pluginlib, Nodelets, and Gazebo Plugins, shows some of the advanced concepts in ROS, such as ROS pluginlib, nodelets, and Gazebo plugins. We will discuss the functionalities and application of each concept and can practice one example to demonstrate its working.

Chapter 6, Writing ROS Controllers and Visualization Plugins, shows how to write a basic ROS controller for PR2 robots and robots similar to PR2. After creating the controller, we will run the controller using the PR2 simulation in Gazebo. We can also see how to create plugin for RViz.

Chapter 7, Interfacing I/O Boards, Sensors, and Actuators to ROS, discusses interfacing some hardware components, such as sensors and actuators, with ROS. We will see the interfacing of sensors using I/O boards, such as Arduino, Raspberry Pi, and Odroid-C1, with ROS.

Chapter 8, Programming Vision Sensors using ROS, Open-CV, and PCL, discusses how to interface various vision sensors with ROS and program it using libraries such as Open Source Computer Vision (Open-CV) and Point Cloud Library (PCL).

Chapter 9, Building and Interfacing Differential Drive Mobile Robot Hardware in ROS, helps you to build autonomous mobile robot hardware with differential drive configuration and interface it with ROS. This chapter aims at giving you an idea of building a custom mobile robot and interfacing it with ROS.

Chapter 10, Exploring the Advanced Capabilities of ROS-MoveIt!, discusses the capabilities of MoveIt! such as collision avoidance, perception using 3D sensors, grasping, picking, and placing. After that, we can see the interfacing of a robotic manipulator hardware with MoveIt!

Chapter 11, ROS for Industrial Robots, helps you understand and install ROS-Industrial packages in ROS. We can see how to develop an MoveIt! IKFast plugin for an industrial robot.

Chapter 12, Troubleshooting and Best Practices in ROS, discusses how to set the ROS development environment in Eclipse IDE, best practices in ROS, and troubleshooting tips in ROS.

What you need for this book

You should have a good PC running Linux distribution, preferably Ubuntu 14.04.3 or Ubuntu 15.04.

Readers can use a laptop or PC with a graphics card, and a RAM of 4 GB to 8 GB is preferred. This is actually for running high-end simulation in Gazebo and also for processing Point cloud and for computer vision.

The readers should have sensors, actuators, and the I/O board mentioned in the book and should have the provision to connect them all to their PC.

The readers also need a Git tool installed to clone the packages files.

If you are a Windows user, then it will be good to download Virtual box and set up Ubuntu in that. Working with Virtual box can have issues when we try to interface real hardware with ROS, so it would be good if you could work with the real system itself.

Who this book is for

If you are a robotics enthusiast or a researcher who wants to learn more about building robot applications using ROS, this book is for you. In order to learn from this book, you should have a basic knowledge of ROS, GNU/Linux, and C++ programming concepts. The book will also be good for programmers who want to explore the advanced features of ROS.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Create a folder called `launch` and inside this folder create the following launch file called `start_laser.launch`."

A block of code is set as follows:

```
#include <ros/ros.h>
#include <moveit/robot_model_loader/robot_model_loader.h>
#include <moveit/planning_scene/planning_scene.h>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
robot_model_loader::RobotModelLoader robot_model_loader("robot_
description");
robot_model::RobotModelPtr kinematic_model = robot_model_loader.
getModel();
planning_scene::PlanningScene planning_scene(kinematic_model);
```

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

```
$ sudo apt-get update
$ sudo apt-get install ros-indigo-perception
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on **camera | driver** and tick **Color Transformer**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com or qboticslabs@gmail.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. You can also download chapter codes from https://github.com/qboticslabs/mastering_ros.git.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/B04782_ColoredImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Introduction to ROS and Its Package Management

This is an introductory chapter that gives you an understanding of the core underlying concepts of ROS and how to work with ROS packages. We will also go through the ROS concepts such as ROS master, nodes, parameter server, topic, message, service, and actionlib to refresh your memory of the concepts you already know.

The basic building blocks of the ROS software framework are ROS packages. We will see how to create, build, and maintain a ROS package. We will also see how to create a wiki page for our package on the ROS website to contribute to the ROS community.

In this chapter, we will cover the following topics:

- Why should we learn ROS?
- Why we prefer ROS for robot
- Why we do not prefer ROS for robot
- Understanding the ROS file system level
- Understanding the ROS computation graph level
- Understanding ROS nodes, messages, topics, services, bags
- Understanding ROS Master
- Using ROS Parameter
- Understanding ROS community level
- Running ROS Master and ROS Parameter server
- Creating a ROS package
- Working with ROS topics

- Adding custom `msg` and `srv` files
- Working with ROS services
- Working with ROS actionlib
- Creating launch files
- Applications of topics, services, and actionlib
- Maintaining the ROS package
- Releasing your ROS package
- Creating a wiki page for your ROS package

Why should we learn ROS?

Robot Operating System (ROS) is a trending robot application development platform that provides various features such as message passing, distributed computing, code reusing, and so on.

The ROS project was started in 2007 with the name *Switchyard* by Morgan Quigley (<http://wiki.osrfoundation.org/morgan>) as part of the Stanford STAIR robot project. The main development of ROS happened at Willow Garage (<https://www.willowgarage.com/>).

The ROS community is growing very fast and there are many users and developers worldwide. Most of the high-end robotics companies are now porting their software to ROS. This trend is also visible in industrial robotics, in which companies are switching from proprietary robotic application to ROS.

The ROS industrial movement has gained momentum in the past few years owing to the large amount of research done in that field. ROS Industrial can extend the advanced capabilities of ROS to manufacturing. The increasing applications of ROS can generate a lot of job opportunities in this field. So after some years, knowledge in ROS will be an essential requirement for a robotics engineer.

Why we prefer ROS for robots

Imagine that we are going to build an autonomous mobile robot. Here are some of the reasons why people choose ROS over other robotic platforms such as Player, YARP, Orocosp, MRPT, and so on:

- **High-end capabilities:** ROS comes with ready to use capabilities, for example, **SLAM (Simultaneous Localization and Mapping)** and **AMCL (Adaptive Monte Carlo Localization)** packages in ROS can be used for performing autonomous navigation in mobile robots and the **MoveIt** package for motion planning of robot manipulators. These capabilities can directly be used in our robot software without any hassle. These capabilities are its best form of implementation, so writing new code for existing capabilities are like reinventing wheels. Also, these capabilities are highly configurable; we can fine-tune each capability using various parameters.
- **Tons of tools:** ROS is packed with tons of tools for debugging, visualizing, and performing simulation. The tools such as **rqt_gui**, **RViz** and **Gazebo** are some of the strong open source tools for debugging, visualization, and simulation. The software framework that has these many tools is very rare.
- **Support high-end sensors and actuators:** ROS is packed with device drivers and interface packages of various sensors and actuators in robotics. The high-end sensors include Velodyne-LIDAR, Laser scanners, Kinect, and so on and actuators such as Dynamixel servos. We can interface these components to ROS without any hassle.
- **Inter-platform operability:** The ROS message-passing middleware allows communicating between different nodes. These nodes can be programmed in any language that has ROS client libraries. We can write high performance nodes in C++ or C and other nodes in Python or Java. This kind of flexibility is not available in other frameworks.
- **Modularity:** One of the issues that can occur in most of the standalone robotic applications are, if any of the threads of main code crash, the entire robot application can stop. In ROS, the situation is different, we are writing different nodes for each process and if one node crashes, the system can still work. Also, ROS provides robust methods to resume operation even if any sensors or motors are dead.
- **Concurrent resource handling:** Handling a hardware resource by more than two processes is always a headache. Imagine, we want to process an image from a camera for face detection and motion detection, we can either write the code as a single entity that can do both, or we can write a single threaded code for concurrency. If we want to add more than two features in threads, the application behavior will get complex and will be difficult to debug. But in ROS, we can access the devices using ROS topics from the ROS drivers. Any number of ROS nodes can subscribe to the image message from the ROS camera driver and each node can perform different functionalities. It can reduce the complexity in computation and also increase the debug-ability of the entire system.

- **Active community:** When we choose a library or software framework, especially from an open source community, one of the main factors that needs to be checked before using it is its software support and developer community. There is no guarantee of support from an open source tool. Some tools provide good support and some tools don't. In ROS, the support community is active. There is a web portal to handle the support queries from the users too (<http://answers.ros.org>). It seems that the ROS community has a steady growth in developers worldwide.

There are many reasons to choose ROS other than the preceding points.

Next, we can check the various reasons why people don't use ROS. Here are some of the existing reasons.

Why some do not prefer ROS for robots

Here are some of the reasons why people do not prefer ROS for their robotic projects:

- **Difficulty in learning:** It will be difficult to learn ROS from their default wiki pages. Most users depend on books to start with ROS. Even this book covers only the basics; learning ROS is going to be bit difficult.
- **Difficulties in starting with simulation:** The main simulator in ROS is Gazebo. Even though Gazebo works well, to get started with Gazebo is not an easy task. The simulator has no inbuilt features to program. Complete simulation is done only through coding in ROS. When we compare Gazebo with other simulators such as V-REP and Webots, they have inbuilt functionalities to prototype and program the robot. They also have a rich GUI toolset and support a wide variety of robots and have ROS interfaces too. These tools are proprietary, but can deliver a decent job. The toughness of learning simulation using Gazebo and ROS is a reason for not using it in their projects.
- **Difficulties in robot modeling:** The robot modeling in ROS is performed using URDF, which is an XML based robot description. In short, we need to write the robot model as a description using URDF tags. In V-REP, we can directly build the 3D robot model in the GUI itself, or we can import the mesh. In ROS, we should write the robot model definitions using URDF tags. There is a SolidWorks plugin to convert a 3D model from SolidWorks to URDF. But if we use other 3D CAD tools, there are no options at all. Learning to model a robot in ROS will take a lot of time and building using URDF tags is also time consuming compared to other simulators.

- **Need for a computer:** We always need a computer to run ROS. Small robots that work completely on microcontrollers don't require a ROS system. ROS is only required when we want to perform high-level functionalities such as autonomous navigation and motion planning. In basic robots, there is no need to use ROS if you are not planning higher level functionalities on the robot.
- **ROS in commercial robot products:** When we deploy ROS on a commercial product, a lot of things need to be taken care of. One thing is the code quality. ROS codes follow a standard coding style and keep best practices for maintaining the code too. We have to check whether it satisfies the quality level required for our product. We might have to do additional work to improve the quality of code. Most of the code in ROS is contributed by researchers from universities, so if we are not satisfied with the ROS code quality, it is better to write your own code, which is specific to the robot and only use the ROS core functionalities if required.

We now know where we have to use ROS and where we do not. If ROS is really required for your robot, let's start discussing ROS in more detail. First, we can see the underlying core concepts of ROS. There are mainly three levels in ROS: file system level, computation graph level, and community level. We can have a look at each level in short.

Understanding the ROS file system level

Similar to an operating system, ROS files are also organized on the hard disk in a particular fashion. In this level, we can see how these files are organized on the disk. The following graph shows how ROS files and folder are organized on the disk:

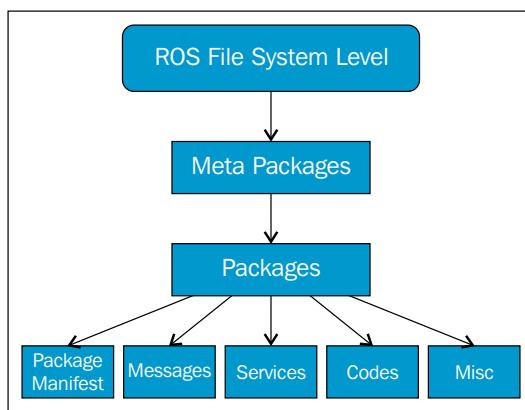


Figure 1 : ROS File system level

Here are the explanations of each block in the file system

- **Packages:** The ROS packages are the most basic unit of the ROS software. It contains the ROS runtime process (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build item and release item in the ROS software.
- **Package manifest:** The package manifest file is inside a package that contains information about the package, author, license, dependencies, compilation flags, and so on. The `package.xml` file inside the ROS package is the manifest file of that package.
- **Meta packages:** The term meta package is used for a group of packages for a special purpose. In an older version of ROS such as Electric and Fuerte, it was called stacks, but later it was removed, as simplicity and meta packages came to existence. One of the examples of a meta package is the ROS navigation stack.
- **Meta packages manifest:** The meta package manifest is similar to the package manifest; differences are that it might include packages inside it as runtime dependencies and declare an export tag.
- **Messages (.msg):** The ROS messages are a type of information that is sent from one ROS process to the other. We can define a custom message inside the `msg` folder inside a package (`my_package/msg/ MyMessageType.msg`). The extension of the message file is `.msg`.
- **Services (.srv):** The ROS service is a kind of request/reply interaction between processes. The reply and request data types can be defined inside the `srv` folder inside the package (`my_package/srv/MyServiceType.srv`).
- **Repositories:** Most of the ROS packages are maintained using a **Version Control System (VCS)** such as Git, subversion (svn), mercurial (hg), and so on. The collection of packages that share a common VCS can be called repositories. The package in the repositories can be released using a catkin release automation tool called `bloom`.

The following screenshot gives you an idea of files and folders of a package that we are going to make in the upcoming sections:

```
mastering_ros_demo_pkg/
|-- action
|   '-- Demo_action.action
-- CMakeLists.txt
|-- include
-- msg
|   '-- demo_msg.msg
-- package.xml
|-- src
|   |-- demo_action_client.cpp
|   |-- demo_action_server.cpp
|   |-- demo_msg_publisher.cpp
|   |-- demo_msg_subscriber.cpp
|   |-- demo_service_client.cpp
|   |-- demo_service_server.cpp
|   |-- demo_topic_publisher.cpp
|   '-- demo_topic_subscriber.cpp
-- srv
`-- demo_srv.srv
```

Figure 2 : List of files inside the exercise package

ROS packages

A typical structure of a ROS package is shown here:

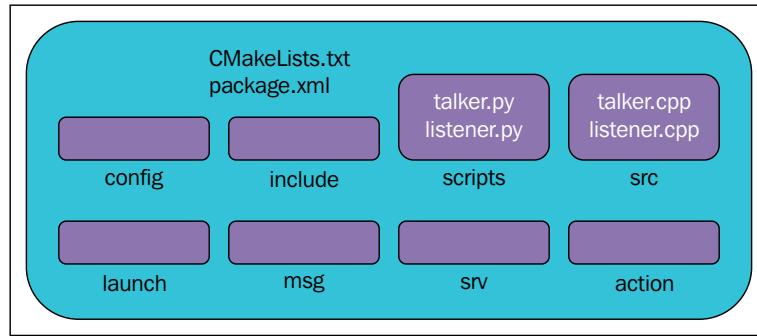


Figure 3 : Structure of a typical ROS package

We can discuss the use of each folder as follows:

- `config`: All configuration files that are used in this ROS package are kept in this folder. This folder is created by the user and is a common practice to name the folder `config` to keep the configuration files in it.
- `include/package_name`: This folder consists of headers and libraries that we need to use inside the package.
- `scripts`: This folder keeps executable Python scripts. In the block diagram, we can see two example scripts.
- `src`: This folder stores the C++ source codes. We can see two examples of the source code in the block diagram.
- `launch`: This folder keeps the launch files that are used to launch one or more ROS nodes.
- `msg`: This folder contains custom message definitions.
- `srv`: This folder contains the service definitions.
- `action`: This folder contains the action definition. We will see more about `actionlib` in the upcoming sections.
- `package.xml`: This is the package manifest file of this package.
- `CMakeLists.txt`: This is the CMake build file of this package.

We need to know some commands to create, modify, and work with the ROS packages. Here are some of the commands used to work with ROS packages:

- `catkin_create_pkg`: This command is used to create a new package
- `rospack`: This command is used to get information about the package in the file system
- `catkin_make`: This command is used to build the packages in the workspace
- `rosdep`: This command will install the system dependencies required for this package

To work with packages, ROS provides a bash-like command called `rosbash` (<http://wiki.ros.org/rosbash>), which can be used to navigate and manipulate the ROS package. Here are some of the `rosbash` commands:

- `roscd`: This command is used to change the package folder. If we give the argument a package name, it will switch to that package folder.
- `roscp`: This command is used to copy a file from a package.
- `rosed`: This command is used to edit a file.
- `rosrun`: This command is used to run an executable inside a package.

The definition of package.xml of a typical package is shown as follows:

```
<?xml version="1.0"?>
<package>
  <name>hello_world</name>
  <version>0.0.0</version>
  <description>The hello_world package</description>

  <maintainer email="qboticslabs@gmail.com">Lentin Joseph</maintainer>
  <license>BSD</license>
  <url type="website">http://wiki.ros.org/hello_world</url>
  <author email="qboticslabs@gmail.com">Lentin Joseph</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <export>
  </export>
</package>
```

Figure 4 : Structure of package.xml

The package.xml file consists of the package name, version of the package, the package description, author details, package build dependencies, and runtime dependencies. The `<build_depend></build_depend>` tag includes the packages that are necessary to build the source code of the package. The packages inside the `<run_depend></run_depend>` tag are necessary during runtime of the package node.

ROS meta packages

Meta packages are specialized packages in ROS that only contain one file, that is, a package.xml file. It doesn't contain folders and files similar to a normal package.

Meta packages simply group a set of multiple packages as a single logical package. In the package.xml file, the meta package contains an export tag, as shown here:

```
<export>
  <metapackage />
</export>
```

Also, in meta packages, there are no `<buildtool_depend>` dependencies for catkin, there are only `<run_depend>` dependencies, which are the packages grouped in the meta package.

The ROS navigation stack is a good example of meta packages. If ROS is installed, we can try the following command, by switching to the navigation meta package folder:

```
$ roscd navigation
```

Open package.xml using gedit text editor

```
$ gedit package.xml
```

This is a lengthy file; here is a stripped down version of it:

```
<package>
  <name>navigation</name>
  <version>1.12.2</version>
  .....
  <buildtool_depend>catkin</buildtool_depend>
  .....
  <run_depend>amcl</run_depend>
  <run_depend>carrot_planner</run_depend>
  .....
  <export>
    <metapackage/>
  </export>
</package>
```

Figure 5 : Structure of meta-package package.xml

ROS messages

The ROS nodes can publish data having a particular type. The types of data are described using a simplified message description language, also called ROS messages. These datatype descriptions can be used to generate source code for the appropriate message type in different target languages.

The data type description of ROS messages are stored in .msg files in the msg subdirectory of a ROS package.

The message definition can consist of two types: fields and constants. The field is split into field types and field name. Field types is the data type of the transmitting message and field name is the name of it. The constants define a constant value in the message file.

Here is an example of message definitions:

```
int32 number
string name
float32 speed
```

Here, the first part is the field type and second is the field name. The field type is the data type and the field name can be used to access the value from the message. For example, we can use `msg.number` for accessing the value of the number from the message.

Here is a table to show some of the built-in field types that we can use in our message:

Primitive type	Serialization	C++	Python
<code>bool(1)</code>	unsigned 8-bit int	<code>uint8_t(2)</code>	<code>bool</code>
<code>int8</code>	signed 8-bit int	<code>int8_t</code>	<code>int</code>
<code>uint8</code>	unsigned 8-bit int	<code>uint8_t</code>	<code>int (3)</code>
<code>int16</code>	signed 16-bit int	<code>int16_t</code>	<code>int</code>
<code>uint16</code>	unsigned 16-bit int	<code>uint16_t</code>	<code>int</code>
<code>int32</code>	signed 32-bit int	<code>int32_t</code>	<code>int</code>
<code>uint32</code>	unsigned 32-bit int	<code>uint32_t</code>	<code>int</code>
<code>int64</code>	signed 64-bit int	<code>int64_t</code>	<code>long</code>
<code>uint64</code>	unsigned 64-bit int	<code>uint64_t</code>	<code>long</code>
<code>float32</code>	32-bit IEEE float	<code>float</code>	<code>float</code>
<code>float64</code>	64-bit IEEE float	<code>double</code>	<code>float</code>
<code>string</code>	ascii string(4)	<code>std::string</code>	<code>string</code>
<code>time</code>	secs/nsecs unsigned 32-bit ints	<code>ros::Time</code>	<code>rospy.Time</code>
<code>duration</code>	secs/nsecs signed 32-bit ints	<code>ros::Duration</code>	<code>rospy.Duration</code>

A special type of ROS message is called message headers. Headers can carry information such as time, frame of reference or `frame_id`, and sequence number. Using headers, we will get numbered messages and more clarity in who is sending the current message. The header information is mainly used to send data such as robot joint transforms (TF). Here is an example of the message header:

```
uint32 seq
time stamp
string frame_id
```

The `rosmsg` command tool can be used to inspect the message header and the field types. The following command helps to view the message header of a particular message:

```
$ rosmsg show std_msgs/Header
```

This will give you an output like the preceding example message header. We can see more about the `rosmsg` command and how to work with custom message definitions in the upcoming sections.

The ROS services

The ROS services are a type request/response communication between ROS nodes. One node will send a request and wait until it gets a response from the other. The request/response communication is also using the ROS message description.

Similar to the message definitions using the `.msg` file, we have to define the service definition in another file called `.srv`, which has to be kept inside the `srv` sub directory of the package. Similar to the message definition, a service description language is used to define the ROS service types.

An example service description format is as follows:

```
#Request message type
string str
---
#Response message type
string str
```

The first section is the message type of request that is separated by `---` and in the next section is the message type of response. In these examples, both Request and Response are strings.

In the upcoming sections, we can see how to work with ROS services.

Understanding the ROS computation graph level

The computation in ROS is done using a network of process called ROS nodes. This computation network can be called the computation graph. The main concepts in the computation graph are **ROS Nodes**, **Master**, **Parameter server**, **Messages**, **Topics**, **Services**, and **Bags**. Each concept in the graph is contributed to this graph in different ways.

The ROS communication related packages including core client libraries such as `roscpp` and `rospython` and the implementation of concepts such as topics, nodes, parameters, and services are included in a stack called `ros_comm` (http://wiki.ros.org/ros_comm).

This stack also consists of tools such as `rostopic`, `rosparam`, `rosservice`, and `rosnode` to introspect the preceding concepts.

The `ros_comm` stack contains the ROS communication middleware packages and these packages are collectively called **ROS Graph layer**.

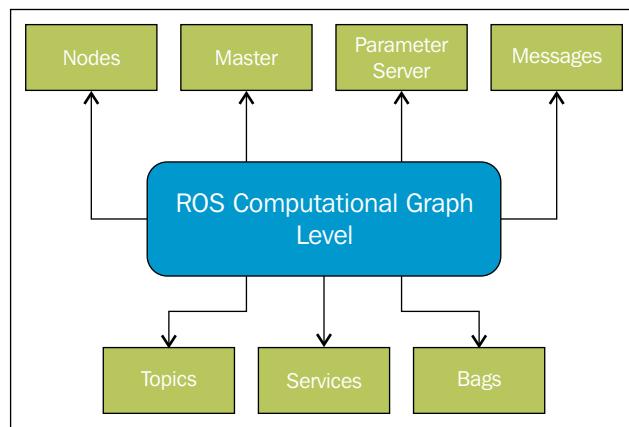


Figure 6 : Structure of the ROS Graph layer

The following are abstracts of each graph's concepts:

- **Nodes:** Nodes are the process that perform computation. Each ROS node is written using ROS client libraries such as `roscpp` and `rospy`. Using client library APIs, we can implement different types of communication methods in ROS nodes. In a robot, there will be many nodes to perform different kinds of tasks. Using the ROS communication methods, it can communicate with each other and exchange data. One of the aims of ROS nodes is to build simple processes rather than a large process with all functionality. Being a simple structure, ROS nodes are easy to debug too.
- **Master:** The ROS Master provides name registration and lookup to the rest of the nodes. Nodes will not be able to find each other, exchange messages, or invoke services without a ROS Master. In a distributed system, we should run the master on one computer, and other remote nodes can find each other by communicating with this master.
- **Parameter Server:** The parameter server allows you to keep the data to be stored in a central location. All nodes can access and modify these values. Parameter server is a part of ROS Master

- **Messages:** Nodes communicate with each other using messages. Messages are simply a data structure containing the typed field, which can hold a set of data and that can be sent to another node. There are standard primitive types (integer, floating point, Boolean, and so on) and these are supported by ROS messages. We can also build our own message types using these standard types.
- **Topics:** Each message in ROS is transported using named buses called topics. When a node sends a message through a topic, then we can say the node is publishing a topic. When a node receives a message through a topic, then we can say that the node is subscribing to a topic. The publishing node and subscribing node are not aware of each other's existence. We can even subscribe a topic that might not have any publisher. In short, the production of information and consumption of it are decoupled. Each topic has a unique name, and any node can access this topic and send data through it as long as they have the right message type.
- **Services:** In some robot applications, a publish/subscribe model will not be enough if it needs a request/response interaction. The publish/subscribe model is a kind of one-way transport system and when we work with a distributed system, we might need a request/response kind of interaction. ROS Services are used in these cases. We can define a service definition that contains two parts; one is for requests and the other is for responses. Using ROS Services, we can write a server node and client node. The server node provides the service under a name, and when the client node sends a request message to this server, it will respond and send the result to the client. The client might need to wait until the server responds. The ROS service interaction is like a remote procedure call.
- **Bags:** Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, which can be difficult to collect but is necessary for developing and testing robot algorithms. Bags are very useful features when we work with complex robot mechanisms.

The following graph shows how the nodes communicate with each other using topics. The topics are mentioned in a rectangle and nodes are represented in ellipses. The messages and parameters are not included in this graph. These kinds of graphs can be generated using a tool called `rqt_graph` (http://wiki.ros.org/rqt_graph).

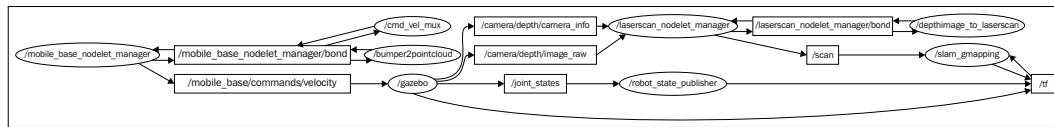


Figure 7 : Graph of communication between nodes using topics

Understanding ROS nodes

ROS nodes are a process that perform computation using ROS client libraries such as `roscpp` and `rospy`. One node can communicate with other nodes using ROS Topics, Services, and Parameters.

A robot might contain many nodes, for example, one node processes camera images, one node handles serial data from the robot, one node can be used to compute odometry, and so on.

Using nodes can make the system fault tolerant. Even if a node crashes, an entire robot system can still work. Nodes also reduce the complexity and increase debug-ability compared to monolithic codes because each node is handling only a single function.

All running nodes should have a name assigned to identify them from the rest of the system. For example, `/camera_node` could be a name of a node that is broadcasting camera images.

There is a `rosbash` tool to introspect ROS nodes. The `rosnode` command can be used to get information about a ROS node. Here are the usages of `rosnode`:

- `$ rosnode info [node_name]`: This will print the information about the node
- `$ rosnode kill [node_name]`: This will kill a running node
- `$ rosnode list`: This will list the running nodes
- `$ rosnode machine [machine_name]`: This will list the nodes running on a particular machine or a list of machines
- `$ rosnode ping`: This will check the connectivity of a node
- `$ rosnode cleanup`: This will purge the registration of unreachable nodes

We will see example nodes using the `roscpp` client and will discuss the working of ROS nodes that use functionalities such ROS Topics, Service, Messages, and actionlib.

ROS messages

ROS nodes communicate with each other by publishing messages to a topic. As we discussed earlier, messages are a simple data structure containing field types. The ROS message supports standard primitive datatypes and arrays of primitive types.

Nodes can also exchange information using service calls. Services are also messages, the service message definitions are defined inside the `.srv` file.

We can access the message definition using the following method. For example, to access `std_msgs/msg/String.msg`, we can use `std_msgs/String`. If we are using the `roscpp` client, we have to include `std_msgs/String.h` for the string message definition.

In addition to message data type, ROS uses an MD5 checksum comparison to confirm whether the publisher and subscriber exchange the same message data types.

ROS has inbuilt tools called `rosmsg` to get information about ROS messages. Here are some parameters used along with `rosmsg`:

- `$ rosmsg show [message]`: This shows the message description
- `$ rosmsg list`: This lists all messages
- `$ rosmsg md5 [message]`: This displays md5sum of a message
- `$ rosmsg package [package_name]`: This lists messages in a package
- `$ rosmsg packages [package_1] [package_2]`: This lists packages that contain messages

ROS topics

ROS topics are named buses in which ROS nodes exchange messages. Topics can anonymously publish and subscribe, which means that the production of messages is decoupled from the consumption. The ROS nodes are not interested to know which node is publishing the topic or subscribing topics, it only looks for the topic name and whether the message types of publisher and subscriber are matching.

The communication using topics are unidirectional, if we want to implement request/response such as communication, we have to switch to ROS services.

The ROS nodes communicate with topics using TCP/IP-based transport known as **TCPROS**. This method is the default transport method used in ROS. Another type of communication is **UDPROS**, which has low-latency, loose transport, and is only suited for teleoperation.

The ROS topic tool can be used to get information about ROS topics. Here is the syntax of this command:

- `$ rostopic bw /topic`: This command will display the bandwidth used by the given topic
- `$ rostopic echo /topic`: This command will print the content of the given topic
- `$ rostopic find /message_type`: This command will find topics using the given message type
- `$ rostopic hz /topic`: This command will display the publishing rate of the given topic
- `$ rostopic info /topic`: This command will print information about an active topic
- `$ rostopic list`: This command will list all active topics in the ROS system
- `$ rostopic pub /topic message_type args`: This command can be used to publish a value to a topic with a message type
- `$ rostopic type /topic`: This will display the message type of the given topic

ROS services

When we need a request/response kind of communication in ROS, we have to use the ROS services. ROS topics can't do this kind of communication because it is unidirectional. ROS services are mainly used in a distributed system.

The ROS services is defined using a pair of messages. We have to define a request datatype and a response datatype in a `srv` file. The `srv` files are kept in a `srv` folder inside a package.

In ROS services, one node acts as a ROS server in which the service client can request the service from the server. If the server completes the service routine, it will send the results to the service client.

The ROS service definition can be accessed by the following method, for example, if `my_package/srv/Image.srv` can be accessed by `my_package/Image`.

In ROS services also, there is an MD5 checksum that checks in the nodes. If the sum is equal, then only the server responds to the client.

There are two ROS tools to get information about the ROS service. The first tool is `rossrv`, which is similar to `rosmg`, and is used to get information about service types. The next command is `rosservice`, which is used to list and query about the running ROS services.

The following explain how to use the `rosservice` tool to get information about the running services:

- `$ rosservice call /service args`: This tool will call the service using the given arguments
- `$ rosservice find service_type`: This command will find services in the given service type
- `$ rosservice info /services`: This will print information about the given service
- `$ rosservice list`: This command will list the active services running on the system
- `$ rosservice type /service`: This command will print the service type of a given service
- `$ rosservice uri /service`: This tool will print the service ROSRPC URI

ROS bags

A bag file in ROS is for storing ROS message data from topics and services. The `.bag` extension is used to represent a bag file.

Bag files are created using the `rosbag` command, which will subscribe one or more topics and store the message's data in a file as it's received. This file can play the same topics as they are recorded from or it can remap the existing topics too.

The main application of `rosbag` is data logging. The robot data can be logged and can visualize and process offline.

The `rosbag` command is used to work with `rosbag` files. Here are the commands to record and playback a bag file:

- `$ rosbag record [topic_1] [topic_2] -o [bag_name]`: This command will record the given topics into a bag file that is given in the command. We can also record all topics using the `-a` argument.
- `$ rosbag play [bag_name]`: This will playback the existing bag file.

Here are more details about this command:

<http://wiki.ros.org/rosbag/Commandline>

There is a GUI tool to handle record and playback of bag files called `rqt_bag`. Go to the following link to know more about `rqt_bag`:

http://wiki.ros.org/rqt_bag

Understanding ROS Master

ROS Master is much like a DNS server. When any node starts in the ROS system, it will start looking for ROS Master and register the name of the node in it. So ROS Master has the details of all nodes currently running on the ROS system. When any details of the nodes change, it will generate a call-back and update with the latest details. These node details are useful for connecting with each node.

When a node starts publishing a topic, the node will give the details of the topic such as name and data type to ROS Master. ROS Master will check whether any other nodes are subscribed to the same topic. If any nodes are subscribed to the same topic, ROS Master will share the node details of the publisher to the subscriber node. After getting the node details, these two nodes will interconnect using the TCPROS protocol, which is based on TCP/IP sockets. After connecting to the two nodes, ROS Master has no role in controlling them. We might be able to stop either the publisher node or the subscriber node according to our wish. If we stop any nodes, it will check with ROS Master once again. This same method is for the ROS services.

The nodes are written using the ROS client libraries such as `roscpp` and `rospy`. These clients interact with ROS Master using **XMLRPC (XML Remote Procedure Call)** based APIs, which act as the backend of the ROS system APIs.

The `ROS_MASTER_URI` environment variable contains the IP and port of ROS Master. Using this variable, ROS nodes can locate ROS Master. If this variable is wrong, the communication between nodes will not take place. When we use ROS in a single system, we can use the IP of localhost or the name `localhost` itself. But in a distributed network, in which computation is on different physical computers, we should define `ROS_MASTER_URI` properly, only then, the remote node could find each other and communicate with each other. We need only one Master, in a distributed system, and it should run on a computer in which all other computers can ping it properly to ensure that remote ROS nodes can access the Master.

The following diagram shows an illustration of how ROS Master interacts with a publishing and subscribing node, the publisher node publishing a string type topic with a "Hello World" message and the subscriber node subscribes to this topic.

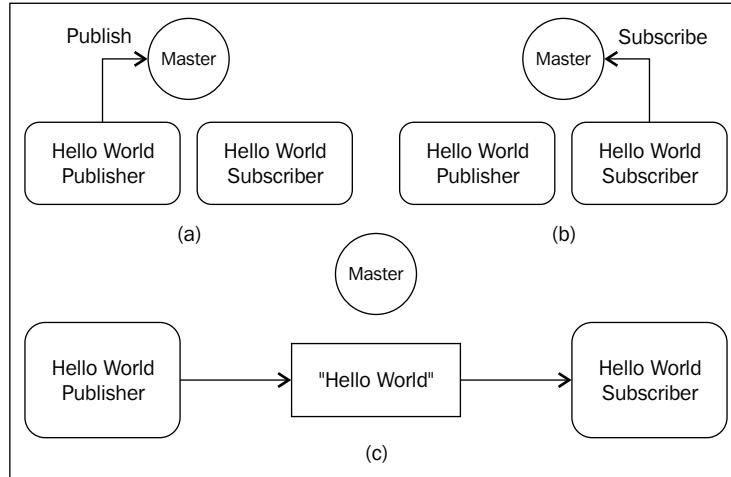


Figure 8: Communication between ROS Master and Hello World publisher and subscriber

When the publisher node starts publishing the "Hello World" message in a particular topic, ROS Master gets the details of the topic and details of the node. It will search whether any node is subscribing the same topic. If there are no nodes subscribing the same topic at that time, both nodes remain unconnected. If the publisher and subscriber nodes run at the same time, ROS Master exchanges the details of the publisher to the subscriber and they will connect and can exchange data through ROS messages.

Using the ROS parameter

While programming a robot, we might have to define robot parameters such as robot controller gain such as P, I, and D. When the number of parameters increases, we might need to store it as files. In some situation, these parameters have to share between two or more programs too. In this case, ROS provides a parameter server, which is a shared server in which all ROS nodes can access parameters from this server. A node can read, write, modify and delete parameter values from the parameter server.

We can store these parameters in a file and load them into the server. The server can store a wide variety of data types and can even store dictionaries. The programmer can also set the scope of the parameter, that is, whether it can be accessed by only this node or all the nodes.

The parameter server supports the following XMLRPC datatypes, which include:

- 32-bit integers
- Booleans
- strings
- doubles
- iso8601 dates
- lists
- base64-encoded binary data

We can also store dictionaries on the parameter server. If the number of parameters is high, we can use a YAML file to save it. Here is an example of the YAML file parameter definitions:

```
/camera/name : 'nikon'      #string type
/camera/fps : 30              #integer
/camera/exposure : 1.2        #float
/camera/active : true         #boolean
```

The `rosparam` tool used to get and set the ROS parameter from the command line. The following are the commands to work with ROS parameters:

- `$ rosparam set [parameter_name] [value]`: This command will set a value in the given parameter
- `$ rosparam get [parameter_name]`: This command will retrieve a value from the given parameter
- `$ rosparam load [YAML file]`: The ROS parameters can be saved into a YAML file and it can load to the parameter server using this command
- `$ rosparam dump [YAML file]`: This command will dump the existing ROS parameters to a YAML file
- `$ rosparam delete [parameter_name]`: This command will delete the given parameter
- `$ rosparam list`: This command will list existing parameter names

The parameters can be changed dynamically during the execution of the node that uses these parameters, using the `dynamic_reconfigure` package (http://wiki.ros.org/dynamic_reconfigure).

Understanding ROS community level

These are ROS resources that enable a new community for ROS to exchange software and knowledge. The various resources in these communities are as follows:

- **Distributions:** Similar to the Linux distribution, ROS distributions are a collection of versioned meta packages that we can install. The ROS distribution enables easier installation and collection of the ROS software. The ROS distributions maintain consistent versions across a set of software.
- **Repositories:** ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.
- **The ROS Wiki:** The ROS community Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.
- **Bug ticket system:** If we find a bug in the existing software or need to add a new feature, we can use this resource.
- **Mailing lists:** The ROS-users mailing list is the primary communication channel about new updates to ROS, as well as a forum to ask questions about the ROS software.
- **ROS Answers:** This website resource helps to ask questions related to ROS. If we post our doubts on this site, other ROS users can see this and give solutions.
- **Blog:** The ROS blog updates with news, photos, and videos related to the ROS community (<http://www.ros.org/news>).

What are the prerequisites to start with ROS?

Before getting started with ROS and trying the code in this book, the following prerequisites should be met:

- **Ubuntu 14.04.2 LTS / Ubuntu 15.04:** We have to use Ubuntu as the operating system for installing ROS. We prefer to stick on to the L.T.S version of Ubuntu, that is, Ubuntu 14.04/14.04.3, or if you want to explore new ROS distribution you can use Ubuntu 15.04.
- **ROS Jade/Indigo desktop full installation:** Install the full desktop installation of ROS. The version we prefer is ROS Indigo, the latest version, Jade, is also supported. The following link gives you the installation instruction of the latest ROS distribution: <http://wiki.ros.org/indigo/Installation/Ubuntu>.

Running ROS Master and ROS parameter server

Before running any ROS nodes, we should start ROS Master and the ROS parameter server. We can start ROS Master and the ROS parameter server using a single command called `roscore`, which will start the following programs:

- ROS Master
- ROS parameter server
- rosout logging nodes

The `rosout` node will collect log messages from other ROS nodes and store them in a log file, and will also rebroadcast the collected log message to another topic. The topic `/rosout` is published by ROS nodes working using ROS client libraries such as `roscpp` and `rospy` and this topic is subscribed by the `rosout` node which rebroadcasts the message in another topic called `/rosout_agg`. This topic has an aggregate stream of log messages. The command `roscore` is a prerequisite before running any ROS node. The following screenshot shows the messages printing when we run the `roscore` command in a terminal.

The following is a command to run `roscore` on a Linux terminal:

```
$ roscore
```

robot@robot-VirtualBox:~\$ roscore
... logging to /home/robot/.ros/log/a3a8e160-e1ae-11e4-b7be-0800273c354c/roslaunch-robot-Virtu
alBox-2138.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.
1
started roslaunch server http://robot-VirtualBox:42377/
ros_comm version 1.11.10
2

SUMMARY
=====

PARAMETERS
* /rosdistro: indigo
* /rosversion: 1.11.10
3

NODES
auto-starting new master
process[master]: started with pid [2183]
ROS_MASTER_URI=http://robot-VirtualBox:11311/
4
setting /run_id to a3a8e160-e1ae-11e4-b7be-0800273c354c
process[rosout-1]: started with pid [2196]
5
started core service [/rosout]

Figure 9 : Terminal messages while running the `roscore` command

The following are explanations of each section when executing `roscore` on the terminal:

- In the first section, we can see a log file is creating inside the `~/ros/log` folder for collecting logs from ROS nodes. This file can be used for debugging purposes.
- In the second section, the command starts a ROS launch file called `roscore.xml`. When a launch file starts, it automatically starts the `rosmaster` and ROS parameter server. The `roslaunch` command is a Python script, which can start `rosmaster` and the ROS parameter server whenever it tries to execute a launch file. This section shows the address of the ROS parameter server within the port.
- In the third section, we can see the parameters such as `rosdistro` and `rosversion` displayed on the terminal. These parameters are displayed when it executes `roscore.xml`. We can see more on `roscore.xml` and its details in the next section.
- In the fourth section, we can see the `rosmaster` node is started using `ROS_MASTER_URI`, which we defined earlier as an environment variable.
- In the fifth section, we can see the `rosout` node is started, which will start subscribing the `/rosout` topic and rebroadcasting into `/rosout_agg`.

The following is the content of `roscore.xml`:

```
<launch>
  <group ns="/">
    <param name="rosversion" command="rosversion roslaunch" />
    <param name="rosdistro" command="rosversion -d" />
    <node pkg="rosout" type="rosout" name="rosout" respawn="true"/>
  </group>
</launch>
```

When the `roscore` command is executed, initially, the command checks the command line argument for a new port number for the `rosmaster`. If it gets the port number, it will start listening to the new port number, otherwise it will use the default port. This port number and the `roscore.xml` launch file will pass to the `roslaunch` system. The `roslaunch` system is implemented in a Python module, it will parse the port number and launch the `roscore.xml` file.

In the `roscore.xml` file, we can see the ROS parameters and nodes are encapsulated in a group XML tag with a "/" namespace. The group XML tag indicates that all the nodes inside this tag have the same settings.

The two parameters called `rosversion` and `rosdistro` store the output of the `rosversion` `roslaunch` and `rosversion -d` commands using the `command` tag, which is a part of the `ROS param` tag. The command tag will execute the command mentioned on it and store the output of the command in these two parameters.

The `rosmaster` and parameter server are executed inside `roslaunch` modules by using the `ROS_MASTER_URI` address. This is happening inside the `roslaunch` Python module. The `ROS_MASTER_URI` is a combination of the IP address and port in which `rosmaster` is going to listen. The port number can be changed according to the given port number in the `roscore` command.

Checking the `roscore` command output

Let's check the ROS topics and ROS parameters created after running `roscore`. The following command will list the active topics on the terminal:

```
$ rostopic list
```

The list of topics are as follows, as per our discussion on the `rosout` node subscribe `/rosout` topic, which have all log messages from the ROS nodes and `/rosout_agg` rebroadcast the log messages:

```
/rosout  
/rosout_agg
```

The following command lists out the parameters available when running `roscore`. The following is the command to list the active ROS parameter:

```
$ rosparam list
```

The parameters are mentioned here; they have the ROS distribution name, version, address of `roslaunch` server and `run_id`, where `run_id` is a unique ID associated with a particular run of `roscore`:

```
/rosdistro  
/roslaunch/uris/host_robot_virtualbox_51189  
/rosversion  
/run_id
```

The list of the ROS service generated during the running `roscore` can be checked using the following command:

```
$ rosservice list
```

The list of services running is as follows:

```
/rosout/get_loggers  
/rosout/set_logger_level
```

These ROS services are generated for each ROS node for setting the logging levels.

After understanding the basics of ROS Master, Parameter server, and `roscore` we can go to the procedure to build a ROS package. Along with working with the ROS package, we can refresh the concepts of ROS nodes, topics, messages, services, and actionlib.

Creating a ROS package

The ROS packages are the basic unit of the ROS system. We can create the ROS package, build it and release it to the public. The current distribution of ROS we are using is Jade/Indigo. We are using the `catkin` build system to build ROS packages. A build system is responsible for generating 'targets' (executable/libraries) from a raw source code that can be used by an end user. In older distributions, such as Electric and Fuerte, `rosbuild` was the build system. Because of the various flaws of `rosbuild`, `catkin` came into existence, which is basically based on **CMake (Cross Platform Make)**. This has lot of advantages such as porting the package into other operating system, such as Windows. If an OS supports CMake and Python, `catkin` based packages can be easily ported into it.

The first requirement in creating ROS packages is to create a ROS `catkin` workspace. Here is the procedure to build a `catkin` workspace.

Build a workspace folder in the home directory and create a `src` folder inside the workspace folder:

```
$ mkdir ~/catkin_ws/src
```

Switch to the source folder. The packages are created inside this package:

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

Initialize a new `catkin` workspace:

```
$ catkin_init_workspace
```

We can build the workspace even if there are no packages. We can use the following command to switch to the workspace folder:

```
$ cd ~/catkin_ws
```

The `catkin_make` command will build the following workspace:

```
$ catkin_make
```

After building the empty workspace, we should set the environment of the current workspace to be visible by the ROS system. This process is called overlaying a workspace. We should add the package environment using the following command:

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

This command will source a bash script called `setup.bash` inside the `devel` workspace folder. To set the environment in all bash sessions, we need to add a `source` command in the `.bashrc` file, which will source this script whenever a bash session starts.

This is the link of the procedure http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

1. After setting the `catkin` workspace, we can create our own package that has sample nodes to demonstrate the working of ROS topics, messages, services, and actionlib.
2. The `catkin_create_pkg` command is used to create a ROS package. This command is used to create our package in which we are going to create demos of various ROS concepts.
3. Switch to the `catkin` workspace `src` folder and create the package using the following command:

Syntax of `catkin_create_pkg` : `catkin_create_pkg [package_name] [dependency1] [dependency2]`

4. Here is the command to create the sample ROS package:

```
$ catkin_create_pkg mastering_ros_demo_pkg roscpp std_msgs
actionlib actionlib_msgs
```

The dependencies in the packages are as follows:

- **roscpp**: This is the C++ implementation of ROS. It is a ROS client library which provides APIs to C++ developers to make ROS nodes with ROS topics, services, parameters, and so on. We are including this dependency because we are going to write a ROS C++ node. Any ROS package which uses the C++ node must add this dependency.
- **std_msgs**: This package contains basic ROS primitive data types such as integer, float, string, array, and so on. We can directly use these data types in our nodes without defining a new ROS message.

- **actionlib**: The actionlib meta-package provides interfaces to create preemptable tasks in ROS nodes. We are creating actionlib based nodes in this package. So we should include this package to build the ROS nodes.
 - **actionlib_msgs**: This package contains standard message definitions needed to interact with the action server and action client.

We will get the following message if the package is successfully created:

Figure 10 : Terminal messages while creating a ROS package

- After creating this package, build the package without adding any nodes using the `catkin_make` command. This command must be executed from the `catkin workspace path`. The following command shows you how to build our empty ROS package:

```
~/catkin_ws$ catkin_make
```

- After a successful build, we can start adding nodes to the `src` folder of this package.

The build folder in the CMake build files mainly contains executables of the nodes that are placed inside the `catkin workspace src` folder. The `devel` folder contains bash script, header files, and executables in different folders generated during the build process. We can see how to make ROS nodes and build using `catkin make`.

Working with ROS topics

Topics are the basic way of communicating between two nodes. In this section, we can see how the topics works. We are going to create two ROS nodes for publishing a topic and subscribing the same. Navigate to the `chapter_1_codes/mastering_ros_demo_package/src` folder for the codes. `demo_topic_publisher.cpp` and `demo_topic_subscriber.cpp` are the two sets of code that we are going to discuss.

Creating ROS nodes

The first node we are going to discuss is `demo_topic_publisher.cpp`. This node will publish an integer value on a topic called `/numbers`. Copy the current code into a new package or use this existing file from the code repository.

Here is the complete code:

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>
int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_topic_publisher");
    ros::NodeHandle node_obj;
    ros::Publisher number_publisher =
    node_obj.advertise<std_msgs::Int32>("/numbers", 10);
    ros::Rate loop_rate(10);
    int number_count = 0;
    while (ros::ok())
    {
        std_msgs::Int32 msg;
        msg.data = number_count;
        ROS_INFO("%d", msg.data);
        number_publisher.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++number_count;
    }
    return 0;
}
```

Here is the detailed explanation of the preceding code:

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>
```

The `ros/ros.h` is the main header of ROS. If we want to use the `roscpp` client APIs in our code, we should include this header. The `std_msgs/Int32.h` is the standard message definition of integer datatype.

Here, we are sending an integer value through a topic. So we should need a message type for handling the integer data. `std_msgs` contains standard message definition of primitive datatypes. `std_msgs/Int32.h` contains integer message definition:

```
ros::init(argc, argv, "demo_topic_publisher");
```

This code will initialize a ROS node with a name. It should be noted that the ROS node should be unique. This line is mandatory for all ROS C++ nodes:

```
ros::NodeHandle node_obj;
```

This will create a `Nodehandle` object, which is used to communicate with the ROS system:

```
ros::Publisher number_publisher =  
node_obj.advertise<std_msgs::Int32>("/numbers", 10);
```

This will create a topic publisher and name the topic `/numbers` with a message type `std_msgs::Int32`. The second argument is the buffer size. It indicates that how many messages need to be put in a buffer before sending. It should be set to high if the data sending rate is high:

```
ros::Rate loop_rate(10);
```

This is used to set the frequency of sending data:

```
while (ros::ok())  
{
```

This is an infinite while loop, and it quits when we press `Ctrl+C`. The `ros::ok()` function returns zero when there is an interrupt; this can terminate this while loop:

```
std_msgs::Int32 msg;  
msg.data = number_count;
```

The first line creates an integer ROS message and the second line assigns an integer value to the message. Here, `data` is the field name of the `msg` object:

```
ROS_INFO("%d", msg.data);
```

This will print the message data. This line is used to log the ROS information:

```
number_publisher.publish(msg);
```

This will publish the message to the topics `/numbers`:

```
ros::spinOnce();
```

This command will read and update all ROS topics. The node will not publish without a `spin()` or `spinOnce()` function:

```
loop_rate.sleep();
```

This line will provide the necessary delay to achieve a frequency of 10Hz.

After discussing the publisher node, we can discuss the subscriber node, which is `demo_topic_subscriber.cpp`. Copy the code to a new file or use the existing file.

Here is the definition of the subscriber node:

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>
void number_callback(const std_msgs::Int32::ConstPtr& msg)
{
    ROS_INFO("Received [%d]", msg->data);
}
int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_topic_subscriber");
    ros::NodeHandle node_obj;
    ros::Subscriber number_subscriber = node_obj.subscribe("/numbers", 10, number_callback);
    ros::spin();
    return 0;
}
```

Here is the code explanation:

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>
```

This is the header needed for the subscribers:

```
void number_callback(const std_msgs::Int32::ConstPtr& msg)
{
    ROS_INFO("Received [%d]", msg->data);
}
```

This is a callback function that will execute whenever a data comes to the /numbers topic. Whenever a data reaches this topic, the function will call and extract the value and print it on the console:

```
ros::Subscriber number_subscriber =
node_obj.subscribe("/numbers", 10, number_callback);
```

This is the subscriber and here, we are giving the topic name needed to subscribe, buffer size, and the callback function. We are subscribing /numbers topic and we have already seen the callback function in the preceding section:

```
ros::spin();
```

This is an infinite loop in which the node will wait in this step. This code will fasten the callbacks whenever a data reaches the topic. The node will quit only when we press the *Ctrl+C* key.

Building the nodes

We have to edit the `CMakeLists.txt` file in the package to compile and build the source code. Navigate to `chapter_1_codes/mastering_ros_demo_package/CMakeLists.txt` to view the existing `CMakeLists.txt` file. The following code snippet in this file is responsible for building these two nodes:

```
include_directories(  
    include  
    ${catkin_INCLUDE_DIRS}  
    ${Boost_INCLUDE_DIRS}  
)  
#This will create executables of the nodes  
add_executable(demo_topic_publisher src/demo_topic_publisher.cpp)  
add_executable(demo_topic_subscriber src/demo_topic_subscriber.cpp)  
  
#This will generate message header file before building the target  
add_dependencies(demo_topic_publisher mastering_ros_demo_pkg_generate_messages_cpp)  
add_dependencies(demo_topic_subscriber mastering_ros_demo_pkg_generate_messages_cpp)  
  
#This will link executables to the appropriate libraries  
target_link_libraries(demo_topic_publisher ${catkin_LIBRARIES})  
target_link_libraries(demo_topic_subscriber ${catkin_LIBRARIES})
```

We can add the preceding snippet to create a new a `CMakeLists.txt` file for compiling the two codes.

The `catkin_make` command is used to build the package.

We can first switch to workspace:

```
$ cd ~/catkin_ws
```

Build `mastering_ros_demo_package` as follows:

```
$ catkin_make mastering_ros_demo_package
```

We can either use the preceding command to build a specific package or just `catkin_make` to build the entire workspace.

This will create executables in `~/catkin_ws/devel/lib/<package name>`.

If the building is done, we can execute the nodes.

First start `roscore`:

```
$ roscore
```

Now run both commands in two shells.

In the running publisher:

```
$ rosrun mastering_ros_demo_package demo_topic_publisher
```

In the running subscriber:

```
$ rosrun mastering_ros_demo_package demo_topic_subscriber
```

We can see the output as shown here:

```
roscore http://robot-VirtualBox:11311/      x | robot@robot-VirtualBox: ~      x | robot@robot-VirtualBox:~$ rosrun mastering_ros_demo_pkg demo_topic_publisher
[ INFO] [1429195851.035054959]: 0
[ INFO] [1429195851.135337732]: 1
[ INFO] [1429195851.235186036]: 2
[ INFO] [1429195851.335841095]: 3
[ INFO] [1429195851.435267706]: 4
[ INFO] [1429195851.535966447]: 5
[ INFO] [1429195851.635125303]: 6
[ INFO] [1429195851.810513189]: 7
[ INFO] [1429195851.835139308]: 8
[ INFO] [1429195851.935245007]: 9
[ INFO] [1429195852.035188269]: 10
[ INFO] [1429195852.135188558]: 11
[ INFO] [1429195852.235172453]: 12
[ INFO] [1429195852.336646534]: 13
[ INFO] [1429195852.435268877]: 14
[ INFO] [1429195852.535263274]: 15
[ INFO] [1429195852.636214524]: 16
[ INFO] [1429195852.735282604]: 17
[ INFO] [1429195852.836172657]: 18
[ INFO] [1429195851.337429267]: Received [3]
[ INFO] [1429195851.435783179]: Received [4]
[ INFO] [1429195851.536240701]: Received [5]
[ INFO] [1429195851.635804953]: Received [6]
[ INFO] [1429195851.816521812]: Received [7]
[ INFO] [1429195851.835736951]: Received [8]
[ INFO] [1429195851.939650759]: Received [9]
[ INFO] [1429195852.035614896]: Received [10]
[ INFO] [1429195852.135903902]: Received [11]
[ INFO] [1429195852.235513913]: Received [12]
[ INFO] [1429195852.337660217]: Received [13]
[ INFO] [1429195852.435941239]: Received [14]
[ INFO] [1429195852.535806815]: Received [15]
[ INFO] [1429195852.636739531]: Received [16]
[ INFO] [1429195852.735823562]: Received [17]
[ INFO] [1429195852.837438784]: Received [18]
[ INFO] [1429195852.935985331]: Received [19]
[ INFO] [1429195853.035816398]: Received [20]
[ INFO] [1429195853.135980807]: Received [21]
[ INFO] [1429195853.236516729]: Received [22]
```

Figure 11 : Running topic publisher and subscriber

The following diagram shows how the nodes communicate with each other. We can see the `demo_topic_publisher` node publish the `/numbers` topic and subscribe by then `demo_topic_subscriber` node.

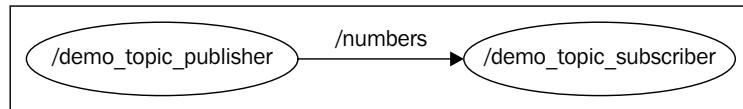


Figure 12 : Graph of the communication between publisher and subscriber nodes.

We can use the `rosnode` and `rostopic` tools to debug and understand the working of two nodes:

- `$ rosnode list`: This will list the active nodes
- `$ rosnode info demo_topic_publisher`: This will get the info of the publisher node
- `$ rostopic echo /numbers`: This will display the value sending through the `/numbers` topic
- `$ rostopic type /numbers`: This will print the message type of the `/numbers` topic

Adding custom msg and srv files

In this section, we can see how to create custom messages and services definitions in the current package. The message definitions are stored in a `.msg` file and service definition are stored in a `.srv` file. These definitions inform ROS about the type of data and name of data to be transmitted from a ROS node. When a custom message is added, ROS will convert the definitions into equivalent C++ codes, which we can include in our nodes.

We can start with message definitions.

Message definitions have to be written in the `.msg` file and have to be kept in the `msg` folder, which is inside the package.

We are going to create a message file called `demo_msg.msg` with the following definition:

```
string greeting
int32 number
```

Until now, we have worked only with standard message definitions. Now, we have created our own definitions and can see how to use them in our code.

The first step is to edit the `package.xml` file of the current package and uncomment the lines `<build_depend>message_generation</build_depend>` and `<run_depend>message_runtime</run_depend>`.

Edit the current `CMakeLists.txt` and add the `message_generation` line as follows:

```
find_package(catkin REQUIRED COMPONENTS
    roscpp
    rospy
    std_msgs
    actionlib
```

```
    actionlib_msgs
    message_generation
)
```

Uncomment the following line and add the custom message file:

```
add_message_files(
    FILES
    demo_msg.msg
)
## Generate added messages and services with any dependencies listed
here
generate_messages(
    DEPENDENCIES
    std_msgs
    actionlib_msgs
)
```

After these steps, we can compile and build the package:

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

To check whether the message is built properly, we can use the `rosmsg` command:

```
$ rosmsg show mastering_ros_demo_pkg/demo_msg
```

If the content shown by the command and the definition are the same, the procedure is correct.

If we want to test the custom message, we can build a publisher and subscriber using the custom message type named `demo_msg_publisher.cpp` and `demo_msg_subscriber.cpp`. Navigate to the `chapter_1_codes/mastering_ros_demo_pkg/src` folder for these codes.

We can test the message by adding the following lines of code in `CMakeLists.txt`:

```
add_executable(demo_msg_publisher src/demo_msg_publisher.cpp)
add_executable(demo_msg_subscriber src/demo_msg_subscriber.cpp)

add_dependencies(demo_msg_publisher mastering_ros_demo_pkg_generate_
messages_cpp)
add_dependencies(demo_msg_subscriber mastering_ros_demo_pkg_generate_
messages_cpp)

target_link_libraries(demo_msg_publisher ${catkin_LIBRARIES})
target_link_libraries(demo_msg_subscriber ${catkin_LIBRARIES})
```

Build the package using `catkin_make` and test the node using the following commands.

- Run `roscore`:

```
$ roscore
```

- Start the custom message publisher node:

```
$ rosrun mastering_ros_demo_pkg demo_msg_publisher
```

- Start the custom message subscriber node:

```
$ rosrun mastering_ros_demo_pkg demo_msg_subscriber
```

The publisher node publishes a string along with an integer, and the subscriber node subscribes the topic and prints the values. The output and graph are shown as follows:

The screenshot shows two terminal windows side-by-side. The left window is titled "roscore http://robot-VirtualBox:11311/" and contains the command "robot@robot-VirtualBox:~/catkin_ws\$ rosrun mastering_ros_demo_pkg demo_msg_publisher". The right window is titled "robot@robot-VirtualBox:~/catkin_ws\$" and contains the command "rosrun mastering_ros_demo_pkg demo_msg_subscriber". Both windows show a series of INFO log messages from the nodes, indicating they are publishing and receiving messages over the "/demo_msg_topic" topic.

Figure 13 : Running publisher and subscriber using custom message definitions.

The topic in which the nodes are communicating is called `/demo_msg_topic`. Here is the graph view of two nodes:

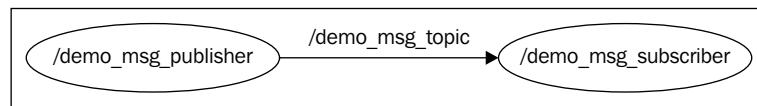


Figure 14 : Graph of the communication between message publisher and subscriber

Next, we can add `srv` files to the package. Create a new folder called `srv` in the current package folder and add a `srv` file called `demo_srv.srv`. The definition of this file is as follows:

```
string in
---
string out
```

Here, both the Request and Response are strings.

In the next step, we need to uncomment the following lines in package.xml as we did for the ROS messages:

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Take CMakeLists.txt and add message_runtime in catkin_package():

```
catkin_package(
    CATKIN_DEPENDS roscpp rospy std_msgs actionlib actionlib_msgs
    message_runtime
)
```

We need to follow the same procedure in generating services as we did for the ROS message. Apart from that, we need additional sections to be uncommented as shown here:

```
## Generate services in the 'srv' folder
add_service_files(
    FILES
    demo_srv.srv
)
```

After making these changes, we can build the package using catkin_make and using the following command we can verify the procedure:

```
$ rossrv show mastering_ros_demo_pkg/demo_srv
```

If we see the same content as we defined in the file, we can confirm it's working.

Working with ROS services

In this section, we are going to create ROS nodes, which can use the services definition that we defined already. The service nodes we are going to create can send a string message as a request to the server and the server node will send another message as a response.

Navigate to chapter_1_codes/mastering_ros_demo_pkg/src and find nodes with the names demo_service_server.cpp and demo_service_client.cpp.

The demo_service_server.cpp is the server and its definition is as follows:

```
#include "ros/ros.h"
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
```

```
#include <sstream>
using namespace std;

bool demo_service_callback(mastering_ros_demo_pkg::demo_srv::Request
&req,
                           mastering_ros_demo_pkg::demo_srv::Response &res)
{
    std::stringstream ss;
    ss << "Received Here";
    res.out = ss.str();
    ROS_INFO("From Client [%s], Server says [%s]", req.in.c_str(), res.
out.c_str());
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_service_server");
    ros::NodeHandle n;
    ros::ServiceServer service = n.advertiseService("demo_service",
demo_service_callback);
    ROS_INFO("Ready to receive from client.");
    ros::spin();
    return 0;
}
```

Let's see the explanation of the code:

```
#include "ros/ros.h"
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
#include <sstream>
```

Here, we included `ros/ros.h`, which is a mandatory header for a ROS CPP node. The `mastering_ros_demo_pkg/demo_srv.h` header is a generated header, which contains our service definition and can use this in our code. The `sstream.h` is for getting string streaming classes:

```
bool demo_service_callback(mastering_ros_demo_pkg::demo_srv::Request
&req,
                           mastering_ros_demo_pkg::demo_srv::Response &res)
{
```

This is the server callback function executed when a request is received on the server. The server can receive the request from clients having a message type of `mastering_ros_demo_pkg::demo_srv::Request` and sends the response in the `mastering_ros_demo_pkg::demo_srv::Response` type:

```
std::stringstream ss;
ss << "Received Here";
res.out = ss.str();
```

In this code, the string data "Received Here" is passing to the service Response instance. Here, `out` is the field name of the response that we have given in the `demo_srv.srv`. This response will go to the service client node:

```
ros::ServiceServer service = n.advertiseService("demo_service",
                                               demo_service_callback);
```

This creates a service having a name as `demo_service` and a callback function is executed when a request comes to this service. The callback function is `demo_service_callback`, which we saw in the preceding section.

Next, we can see how the `demo_service_client.cpp` is working.

Here is the definition of this code:

```
#include "ros/ros.h"
#include <iostream>
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
#include <sstream>
using namespace std;

int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_service_client");
    ros::NodeHandle n;
    ros::Rate loop_rate(10);
    ros::ServiceClient client = n.serviceClient<mastering_ros_demo_
pkg::demo_srv>("demo_service");
    while (ros::ok())
    {
        mastering_ros_demo_pkg::demo_srv srv;
        std::stringstream ss;
        ss << "Sending from Here";
        srv.request.in = ss.str();
        if (client.call(srv))
        {
```

```
ROS_INFO("From Client [%s], Server says [%s]", srv.request.in.c_
str(), srv.response.out.c_str());

}

else
{
    ROS_ERROR("Failed to call service");
    return 1;
}

ros::spinOnce();
loop_rate.sleep();

}

return 0;
}
```

Let's explain the code:

```
ros::ServiceClient client =
n.serviceClient<mastering_ros_demo_pkg::demo_srv>("demo_service");
```

This line creates a service client that has message type `mastering_ros_demo_pkg::demo_srv` and communicates to a ROS service named `demo_service`:

```
mastering_ros_demo_pkg::demo_srv srv;
```

This line will create a new service object instance:

```
std::stringstream ss;
ss << "Sending from Here";
srv.request.in = ss.str();
```

Fill the request instance with a string called "Sending from Here":

```
if (client.call(srv))
{
```

This will send the service call to the server. If it is sent successfully, it will print the response and request, if it failed, it do nothing:

```
ROS_INFO("From Client [%s], Server says [%s]", srv.request.in.c_
str(), srv.response.out.c_str());
```

If the response is received, then it will print the request and the response.

After discussing the two nodes, we can discuss how to build these two nodes. The following code is added to CMakeLists.txt to compile and build the two nodes:

```
add_executable(demo_service_server src/demo_service_server.cpp)
add_executable(demo_service_client src/demo_service_client.cpp)

add_dependencies(demo_service_server mastering_ros_demo_pkg_generate_
messages_cpp)
add_dependencies(demo_service_client mastering_ros_demo_pkg_generate_
messages_cpp)

target_link_libraries(demo_service_server ${catkin_LIBRARIES})
target_link_libraries(demo_service_client ${catkin_LIBRARIES})
```

We can execute the following commands to build the code:

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

To start nodes, first execute roscore and use the following commands:

```
$ rosrun mastering_ros_demo_pkg demo_service_server
$ rosrun mastering_ros_demo_pkg demo_service_client
```

```
roscore http://robot-VirtualBox:11311/ x | robot@robot-VirtualBox: ~/catkin_ws
robot@robot-VirtualBox:~/catkin_ws$ rosrun mastering_ros_demo_pkg demo_service_server
[ INFO] [1429209451.849630707]: Ready to receive from client.
[ INFO] [1429209493.680367116]: From Client [Sending from Here], Server says [Received Here]
[ INFO] [1429209493.779223905]: From Client [Sending from Here], Server says [Received Here]
[ INFO] [1429209493.879772425]: From Client [Sending from Here], Server says [Received Here]
[ INFO] [1429209493.984153977]: From Client [Sending from Here], Server says [Received Here]

robot@robot-VirtualBox:~/catkin_ws$ rosrun mastering_ros_demo_pkg demo_service_client
INFO] [1429209493.684084766]: From Client [Sending from Here], Server says [Received Here]
INFO] [1429209493.779956469]: From Client [Sending from Here], Server says [Received Here]
INFO] [1429209493.880286580]: From Client [Sending from Here], Server says [Received Here]
INFO] [1429209493.984721087]: From Client [Sending from Here], Server says [Received Here]
```

Figure 15 : Running ROS service client and server nodes.

We can work with `rosservice` using the `rosservice` command:

- `$ rosservice list`: This will list the current ROS services
- `$ rosservice type /demo_service`: This will print the message type of `/demo_service`
- `$ rosservice info /demo_service`: This will print the information of `/demo_service`

Working with ROS actionlib

In ROS services, the user implements a request/reply interaction between two nodes, but consider if the reply takes too much time or the server is not finished with the given work, we have to wait until it completes.

There is another method in ROS in which we can preempt the running request and start sending another one if the request is not finished on time as we expected. Actionlib packages provide a standard way to implement these kinds of preemptive tasks. Actionlib is highly used in robot arm navigation and mobile robot navigation. We can see how to implement an action server and action client implementation.

Like ROS services, in actionlib, we have to specify the action specification. The action specification is stored inside the action file having an extension of `.action`. This file must be kept inside the `action` folder, which is inside the ROS package. The action file has the following parts:

- **Goal:** The action client can send a goal that has to be executed by the action server. This is similar to the request in the ROS service. For example, if a robot arm joint wants to move from 45 degrees to 90 degrees, the goal here is 90 degrees.
- **Feedback:** When an action client sends a goal to the action server, it will start executing a call-back function. Feedback is simply giving the progress of the current operation inside the callback function. Using the feedback definition, we can get the current progress. In the preceding case, the robot arm joint has to move to 90 degrees; in this case, the feedback can be the intermediate value between 45 and 90 degrees in which the arm is moving.
- **Result:** After completing the goal, the action server will send a final result of completion, it can be the computational result or an acknowledgement. In the preceding example, if the joint reaches 90 degrees it achieves the goal and the result can be anything indicating it finished the goal.

We can discuss a demo action server and action client here. The demo action client will send a number as the goal. When an action server receives the goal, it will count from 0 to the goal number with a step size of 1 and with a one second delay. If it completes before the given time, it will send the result, otherwise, the task will be preempted by the client. The feedback here is the progress of counting. The action file of this task is as follows. The action file is named `Demo_action.action`:

```
#goal definition
int32 count
---
#result definition
int32 final_count
---
#feedback
int32 current_number
```

Here, the `count` value is the goal in which the server has to count from zero to this number. `final_count` is the result, in which the final value after completion of a task and `current_number` is the feedback value. It will specify how much the progress is.

Navigate to `chapter_1_codes/mastering_ros_demo_pkg/src` and you can find the action server node as `demo_action_server.cpp` and action client node as `demo_action_client.cpp`.

Creating the ROS action server

In this section, we will discuss `demo_action_server.cpp`. The action server receives a goal value that is a number. When the server gets this goal value, it will start counting from zero to this number. If the counting is complete, it will successfully finish the action, if it is preempted before finishing, the action server will look for another goal value.

This code is a bit lengthy, so we can discuss the important code snippet of this code.

Let's start from the header files:

```
#include <actionlib/server/simple_action_server.h>
#include "mastering_ros_demo_pkg/Demo_actionAction.h"
```

The first header is the standard action library to implement an action server node. The second header is generated from the stored action files. It should include for accessing our action definition:

```
class Demo_actionAction
{
```

This class contains the action server definition:

```
actionlib::SimpleActionServer<mastering_ros_demo_pkg::Demo_
actionAction> as;
```

Create a simple action server instance with our custom action message type:

```
mastering_ros_demo_pkg::Demo_actionFeedback feedback;
```

Create a feedback instance for sending feedback during the operation:

```
mastering_ros_demo_pkg::DemoActionResult result;
```

Create a result instance for sending the final result:

```
Demo_actionAction(std::string name) :
    as(nh_, name, boost::bind(&Demo_actionAction::executeCB, this,
_1), false),
    action_name(name)
```

This is an action constructor, and an action server is created here by taking an argument such as Nodehandle, action_name, and executeCB, where executeCB is the action callback where all the processing is done:

```
as.registerPreemptCallback(boost::bind(&Demo_actionAction::preemptCB,
this));
```

This line registers a callback when the action is preempted. The preemptCB is the callback name executed when there is a preempt request from the action client:

```
void executeCB(const mastering_ros_demo_pkg::Demo_actionGoalConstPtr
&goal)
{
    if(!as.isActive() || as.isPreemptRequested()) return;
```

This is the callback definition which is executed when the action server receives a goal value. It will execute callback functions only after checking whether the action server is currently active or it is preempted already:

```
for(progress = 0 ; progress < goal->count; progress++) {
    //Check for ros
    if(!ros::ok()) {
```

This loop will execute until the goal value is reached. It will continuously send the current progress as feedback:

```
        if(!as.isActive() || as.isPreemptRequested()) {
            return;
        }
```

Inside this loop, it will check whether the action server is active or it is preempted. If it occurs, the function will return:

```
if(goal->count == progress) {
    result.final_count = progress;
    as.setSucceeded(result);
}
```

If the current value reaches the goal value, then it publishes the final result:

```
Demo_actionAction demo_action_obj(ros::this_node::getName());
```

In `main()`, we create an instance of `Demo_actionAction`, which will start the action server.

Creating the ROS action client

In this section, we will discuss the working of an action client. `demo_action_client.cpp` is the action client node that will send the goal value consisting of a number which is the goal. The client is getting the goal value from the command line arguments. The first command line argument of the client is the goal value and the second is the time of completion for this task.

The goal value will be sent to the server and the client will wait until the given time, in seconds. After waiting, the client will check whether it completed or not; if it is not complete, the client will preempt the action.

The client code is a bit lengthy, so we will discuss the important sections of the code:

```
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include "mastering_ros_demo_pkg/Demo_actionAction.h"
```

In action client, we need to include `actionlib/client/simple_action_client.h` to get the action client APIs which are used to implement action clients:

```
actionlib::SimpleActionClient<mastering_ros_demo_pkg::Demo_
actionAction> ac("demo_action", true);
```

This will create an action client instance:

```
ac.waitForServer();
```

This line will wait for an infinite time if there is no action server running on the system. It will exit only when there is an action server running on the system:

```
mastering_ros_demo_pkg::Demo_actionGoal goal;
goal.count = atoi(argv[1]);
ac.sendGoal(goal);
```

Create an instance of a goal and send the goal value from the first command line argument:

```
bool finished_before_timeout =
ac.waitForResult(ros::Duration(atoi(argv[2])));
```

This line will wait for the result from the server until the given seconds:

```
ac.cancelGoal();
```

If it is not finished, it will preempt the action.

Building the ROS action server and client

After creating these two files in the `src` folder, we have to edit the `package.xml` and `CMakeLists.txt` to build the nodes.

The `package.xml` file should contain message generation and runtime packages as we did for ROS service and messages.

We have to include the **Boost** library in `CMakeLists.txt` to build these nodes. Also, we have to add the action files that we wrote for this example:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  actionlib
  actionlib_msgs
  message_generation
)
```

We should pass `actionlib`, `actionlib_msgs`, and `message_generation` in `find_package()`:

```
## System dependencies are found with CMake's conventions
find_package(Boost REQUIRED COMPONENTS system)
```

We should add `Boost` as a system dependency:

```
## Generate actions in the 'action' folder
add_action_files(
  FILES
  Demo_action.action
)
```

We need to add our action file in `add_action_files()`:

```
## Generate added messages and services with any dependencies listed
here
generate_messages(
    DEPENDENCIES
    std_msgs
    actionlib_msgs
)
```

We have to add `actionlib_msgs` in `generate_messages()`:

```
catkin_package(
    CATKIN_DEPENDS roscpp rospy std_msgs actionlib actionlib_msgs
    message_runtime
)

include_directories(
    include
    ${catkin_INCLUDE_DIRS}
    ${Boost_INCLUDE_DIRS}
)
```

We have to add Boost to include the directory:

```
##Building action server and action client

add_executable(demo_action_server src/demo_action_server.cpp)
add_executable(demo_action_client src/demo_action_client.cpp)

add_dependencies(demo_action_server mastering_ros_demo_pkg_generate_
messages_cpp)
add_dependencies(demo_action_client mastering_ros_demo_pkg_generate_
messages_cpp)

target_link_libraries(demo_action_server ${catkin_LIBRARIES} )
target_link_libraries(demo_action_client ${catkin_LIBRARIES})
```

After `catkin_make`, we can run these nodes using the following commands:

- Run `roscore`:

```
$ roscore
```

- Launch the action server node:

```
$rosrun mastering_ros_demo_pkg demo_action_server
```

- Launch the action client node:

```
$ rosrun mastering_ros_demo_pkg demo_action_client 50 4
```

The output of these process is shown as follows:

The image shows two terminal windows side-by-side. The top window is titled 'roscore http://robot-VirtualBox:11311/' and displays a series of INFO log messages from a demo action server. These messages show the server setting goals at intervals of 1 unit (from 5 to 50) and then preempting them. The bottom window is titled 'robot@robot-VirtualBox: ~/catkin_ws\$' and shows the command being run to start the demo action client. It waits for the action server to start, sends a goal, and then receives a response indicating the action did not finish before the time out.

```
[ INFO] [1429250341.000239113]: Setting to goal 5 / 50
[ INFO] [1429250341.200344553]: Setting to goal 6 / 50
[ INFO] [1429250341.400197928]: Setting to goal 7 / 50
[ INFO] [1429250341.599506755]: Setting to goal 8 / 50
[ INFO] [1429250341.799517562]: Setting to goal 9 / 50
[ INFO] [1429250342.000124882]: Setting to goal 10 / 50
[ INFO] [1429250342.199469649]: Setting to goal 11 / 50
[ INFO] [1429250342.400600719]: Setting to goal 12 / 50
[ INFO] [1429250342.600010870]: Setting to goal 13 / 50
[ INFO] [1429250342.799993908]: Setting to goal 14 / 50
[ INFO] [1429250343.000609981]: Setting to goal 15 / 50
[ INFO] [1429250343.199882194]: Setting to goal 16 / 50
[ INFO] [1429250343.399496625]: Setting to goal 17 / 50
[ INFO] [1429250343.600134928]: Setting to goal 18 / 50
[ INFO] [1429250343.800066610]: Setting to goal 19 / 50
[ WARN] [1429250343.800445746]: /demo_action got preempted!
```



```
robot@robot-VirtualBox:~/catkin_ws$ rosrun mastering_ros_demo_pkg demo_action_client 50 4
[ INFO] [1429250339.515053322]: Waiting for action server to start.
[ INFO] [1429250339.798580290]: Action server started, sending goal.
[ INFO] [1429250339.798645128]: Sending Goal [50] and Preempt time of [4]
[ INFO] [1429250343.800161096]: Action did not finish before the time out.
robot@robot-VirtualBox:~/catkin_ws$
```

Figure 16 : Running ROS actionlib server and client

Creating launch files

The launch files in ROS are a very useful feature for launching more than one node. In the preceding examples, we have seen a maximum of two ROS nodes, but imagine a scenario in which we have to launch 10 or 20 nodes for a robot. It will be difficult if we run each node in a terminal one by one. Instead of that, we can write all nodes inside a XML based file called launch files and using a command called **roslaunch**, we can parse this file and launch the nodes.

The `roslaunch` command will automatically start ROS Master and the parameter server. So in essence, there is no need to start the `roscore` command and individual node; if we launch the file, all operations will be done in a single command.

Let's start creating launch files. Switch to the package folder and create a new launch file called `demo_topic.launch` to launch two ROS nodes that are publishing and subscribing an integer value. We keep the launch files in a `launch` folder, which is inside the package:

```
$ roscd mastering_ros_demo_pkg  
$ mkdir launch  
$ cd launch  
$ gedit demo_topic.launch
```

Paste the following content into the file:

```
<launch>  
  <node name="publisher_node" pkg="mastering_ros_demo_pkg" type="demo_  
topic_publisher" output="screen"/>  
  
  <node name="subscriber_node" pkg="mastering_ros_demo_pkg"  
type="demo_topic_subscriber" output="screen"/>  
</launch>
```

Let's discuss what is in the code. The `<launch></launch>` tags are the root element in a launch file. All definitions will be inside these tags.

The `<node>` tag specifies the desired node to launch:

```
<node name="publisher_node" pkg="mastering_ros_demo_pkg"  
type="demo_topic_publisher" output="screen"/>
```

The name tag inside `<node>` indicates the name of the node, `pkg` is the name of the package, and `type` is the name of executable we are going to launch.

After creating the launch file `demo_topic.launch`, we can launch it using the following command:

```
$ roslaunch mastering_ros_demo_pkg demo_topic.launch
```

Here is the output we get if the launch is successful:

```
started rosrun server http://robot-VirtualBox:56174/  
  
SUMMARY  
=====  
  
PARAMETERS  
* /rosdistro: indigo  
* /rosversion: 1.11.10  
  
NODES  
/  
    publisher_node (mastering_ros_demo_pkg/demo_topic_publisher)  
    subscriber_node (mastering_ros_demo_pkg/demo_topic_subscriber)  
  
. auto-starting new master  
process[master]: started with pid [2713]  
ROS_MASTER_URI=http://localhost:11311  
  
setting /run_id to 3f5f5bca-7499-11e5-8e21-0800273c354c
```

Figure 17 : Terminal messages while launching the demo_topic.launch file

We can check the list of nodes using:

```
$ rosnode list
```

We can also view the log messages and debug the nodes using a GUI tool called rqt_console:

```
$ rqt_console
```

We can see the logs generated by two nodes in this tool as shown here:

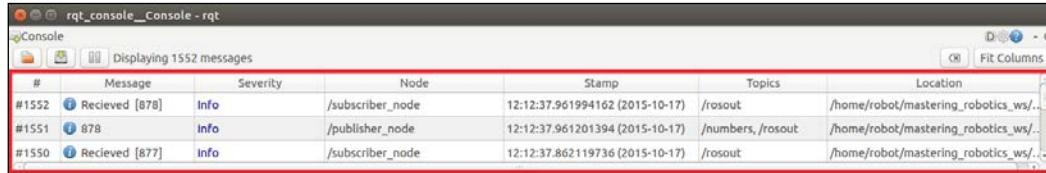


Figure 18 : Logging using the rqt_console tool

Applications of topics, services, and actionlib

Topics, services, and actionlib are used in different scenarios. We know topics are a unidirectional communication method, services are a bidirectional request/reply kind of communication, and actionlib is a modified form of ROS services in which we can cancel the executing process running on the server whenever required.

Here are some of areas where we use these methods:

- **Topics:** Robot teleoperation, publishing odometry, sending robot transform (TF), and sending robot joint states
- **Services:** This saves camera calibration parameters to a file, saves a map of the robot after SLAM, and loads a parameter file
- **Actionlib:** This is used in motion planners and ROS navigation stacks



The complete source code of this project can be cloned from the following Git repository. The following command will clone the project repo:

```
$ git clone
https://github.com/qboticslabs/mastering_ros_demo_
pkg.git
```

Maintaining the ROS package

Most of the ROS packages are released as open source with the BSD license. There are active developers around the globe who are contributing to the ROS platform. Maintaining packages are an important constraint in all software especially open source application. Open source software is maintained and supported by a community of developers. Creating a version control system for our package is essential if we want to maintain and accept a contribution from other developers. The preceding package is already updated in GitHub and you can view the source code of the project at https://github.com/qboticslabs/mastering_ros_demo_pkg

After uploading the code in GitHub, we can see what the procedures are to release our current package to ROS.

Releasing your ROS package

After creating a ROS package in GitHub, we can officially release our package. ROS provides detailed steps to release the ROS package using a tool called bloom (<http://ros-infrastructure.github.io/bloom/>). Bloom is a release automation tool, designed to make platform-specific releases from the source projects. Bloom is designed to work best with the catkin project.

The prerequisites before releasing the package are as follows:

- Install the bloom tool
- Create a Git repository for the current package
- Create an empty Git repository for the release

The following command will install bloom in Ubuntu:

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

Create a Git repository for the current package. The repository that has the package is called the upstream repository. Here, we already created a repository at https://github.com/qboticslabs/mastering_ros_demo_pkg.

Create an empty repository in Git for the release package. This repository is called the release repository. We have created a package called `demo_pkg-release`. This package is at https://github.com/qboticslabs/demo_pkg-release.

After meeting these prerequisites, we can start to create the release of the package. Navigate to the `mastering_ros_demo_pkg` local repository where we push our package code to Git. Open a terminal inside this local repository and execute the following command:

```
$ catkin_generate_changelog
```

The purpose of this command is, it will create a `CHANGELOG.rst` file inside the local repository. After executing this command it will show this option:

Continue without -all option [y/N]. Give y here

It will create a `CHANGELOG.rst` in the local repository.

After the creation of the log file, we can update the Git repository by committing the changes:

```
$ git add -A  
$ git commit -m 'Updated CHANGELOG.rst'  
$ git push -u origin master
```

Preparing the ROS package for the release

In this step, we are checking whether the package contains change logs, versions, and so on. The following command makes our package consistent and recommended for a release.

This command should execute from the local repository of the package:

```
$ catkin_prepare_release
```

The command will set a version tag if there is no current version and commit the changes in the upstream repository.

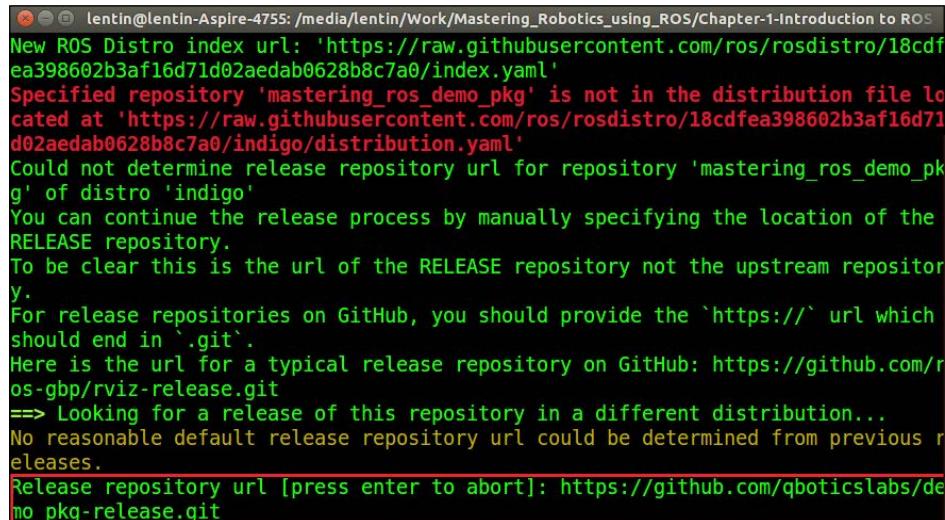
Releasing our package

The following command starts the release. The syntax of this command is as follows:

```
bloom-release --rosdistro <ros_distro> --track <ros_distro> repository_name
$ bloom-release --rosdistro indigo --track indigo mastering_ros_demo_pkg
```

When this command is executed, it will go to the rosdistro (<https://github.com/ros/rosdistro>) package repository to get the package details. The rosdistro package in ROS contains an index file, which contains a list of all the packages in ROS. Currently, there is no index for our package because this is our first release, but we can add our package details to this index file called `distributions.yaml`.

The following message will be displayed when there is no reference of the package in rosdistro:



```
lentin@lentin-Aspire-4755: /media/lentin/Work/Mastering_Robotics_using_ROS/Chapter-1-Introduction to ROS
New ROS Distro index url: 'https://raw.githubusercontent.com/ros/rosdistro/18cdfe
ea398602b3af16d71d02aedab0628b8c7a0/index.yaml'
Specified repository 'mastering_ros_demo_pkg' is not in the distribution file lo
cated at 'https://raw.githubusercontent.com/ros/rosdistro/18cdfe
a398602b3af16d71d02aedab0628b8c7a0/indigo/distribution.yaml'
Could not determine release repository url for repository 'mastering_ros_demo_pk
g' of distro 'indigo'
You can continue the release process by manually specifying the location of the
RELEASE repository.
To be clear this is the url of the RELEASE repository not the upstream repositor
y.
For release repositories on GitHub, you should provide the `https://` url which
should end in '.git'.
Here is the url for a typical release repository on GitHub: https://github.com/r
os-gbp/rviz-release.git
==> Looking for a release of this repository in a different distribution...
No reasonable default release repository url could be determined from previous r
eleases.
Release repository url [press enter to abort]: https://github.com/qboticslabs/de
mo pkg-release.git
```

Figure 19 : Terminal messages when there is no reference of the package in rosdistro

We should give the release repository in the terminal that is marked in red in the preceding screenshot. In this case, the URL was https://github.com/qboticslabs/demo_pkg-release.

```
Create a new track called 'indigo' now [Y/n]? Y
Creating track 'indigo'...
Repository Name:
  upstream
    Default value, leave this as upstream if you are unsure
  <name>
    Name of the repository (used in the archive name)
  ['upstream']: mastering_ros_demo_pkg
Upstream Repository URI:
  <uri>
    Any valid URI. This variable can be templated, for example an svn url
    can be templated as such: "https://svn.foo.com/foo/tags/foo-:{version}"
    where the :{version} token will be replaced with the version for this release.
  [None]: https://github.com/qboticslabs/mastering_ros_demo_pkg.git
```

Figure 20 : Inputting the release repository URL

In the upcoming steps, the wizard will ask for the repository name, upstream, URL, and so on. We can give these options and finally, a pull request to rosdistro will be submitted, which is shown in the following screenshot:

```
==> Pulling latest rosdistro branch
remote: Counting objects: 68220, done.
remote: Compressing objects: 100% (51/51), done.
remote: Total 68220 (delta 28), reused 0 (delta 0), pack-reused 68169
Receiving objects: 100% (68220/68220), 20.31 MiB | 94.00 KiB/s, done.
Resolving deltas: 100% (42818/42818), done.
From https://github.com/ros/rosdistro
 * branch      master    -> FETCH HEAD
==> git reset --hard 18cdfea398602b3af16d71d02aedab0628b8c7a0
HEAD is now at 18cdfea Merge pull request #9661 from mikepurvis/bloom-roboteq-1
==> Writing new distribution file: indigo/distribution.yaml
==> git add indigo/distribution.yaml
==> git commit -m "mastering_ros_demo_pkg: 0.0.2-0 in 'indigo/distribution.yaml' [bloom]"
[bloom-mastering_ros_demo_pkg-0 9f5f7bf] mastering_ros_demo_pkg: 0.0.2-0 in 'indigo/distribution.yaml' [bloom]
 1 file changed, 6 insertions(+)
==> Pushing changes to fork
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 487 bytes | 0 bytes/s, done.
Total 4 (delta 3), reused 0 (delta 0)
To https://5b3a85ef89117532b0b62deba8d5dd14d3070d63:x-oauth-basic@github.com/qboticslabs/rosdistro.git
 * [new branch] bloom-mastering_ros_demo_pkg-0 -> bloom-mastering_ros_demo_pkg-0
<== Pull request opened at: https://github.com/ros/rosdistro/pull/9662
lentin@lentin-Aspire-4755:/media/lentin/Work/Mastering_Robotics_using_ROS/Chapter-1-Introduction to ROS Package management/codes/mastering_ros_demo_pkgs$
```

Figure 21 : Sending a pull request to rosdistro

The pull request for this package can be viewed at <https://github.com/ros/rosdistro/pull/9662>.

If it is accepted, it will merge to `indigo/distribution.yaml`, which contains the index of all packages in ROS.

The following screenshot displays the package as an index in `indigo/distribution.yaml`:



```

6 indigo/distribution.yaml
View

@@ -4651,6 +4651,12 @@ repositories:
4651   4651     url: https://github.com/swri-robotics/marti_messages.git
4652   4652     version: indigo-devel
4653   4653     status: developed
4654 +   master_ros_demo_pkg:
4655 +     release:
4656 +       tags:
4657 +         release: release/indigo/{package}/{version}
4658 +       url: https://github.com/qboticslabs/demo_pkg-release.git
4659 +       version: 0.0.2-0
4654   4660     may_comm:
4655   4661     doc:
4656   4662       type: git

```

Figure 22 : The distribution.yaml file of ROS Indigo

After this step, we can confirm that the package is released and officially added to the ROS index.

Creating a Wiki page for your ROS package

ROS wiki allows users to create their own home pages to showcase their package, robot, or sensors. The official wiki page of ROS is wiki.ros.org. Now, we are going to create a wiki page for our package.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. You can also download chapter codes from https://github.com/qboticslabs/mastering_ros.git.

The first step is to register in wiki using your mail address. Go to wiki.ros.org, and click on the login button as shown in the screenshot:



Figure 23 : Locating the login option from ROS wiki

After clicking on **Login**, you can register or directly login with your details if you are already registered. After login, press the user name link on the right side of the wiki page as shown in the screenshot:

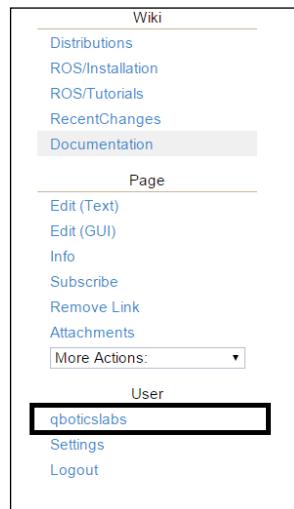


Figure 24 : Locating the user account button from ROS wiki

After clicking on this link, you will get a chance to create a home page for your package; you will get a text editor with GUI to enter data into. The following screenshot shows you the page we created for this demo package:

The screenshot shows a web-based interface for managing a ROS package. At the top, there's a header with the logo 'qboticslabs'. Below it, a message box says 'Thank you for your changes. Your attention to detail is appreciated.' with a 'Clear message' button. The main content area has a title 'Mastering Robotics using ROS Package Summary' and a subtitle 'A demo package which has example codes demonstrating topic, service, custom messages and actionlib'. It lists maintainer, author, license, and source information. A section titled '1. Installation' with the sub-instruction 'You can use git clone to install package.' is shown. On the right side, there are several vertical menus: 'Wiki' (Distributions, ROS/Installation, ROS/Tutorials, RecentChanges, Documentation, qboticslabs), 'Page' (Edit (Text), Edit (GUI), Info, Subscribe, Add Link, Attachments, More Actions), and 'User' (qboticslabs, Settings, Logout).

Figure 25 : Creating a new wiki page

The wiki page of this package can be viewed at <http://wiki.ros.org/qboticslabs>.

Questions

- Why should we learn ROS?
- How does ROS differ from other robotic software platforms?
- What is the internal working of roscore?
- How do ROS topic and service differ in their operations?
- How do ROS service and actionlib differ in their operations?

Summary

ROS is now a trending software framework among roboticists. Gaining knowledge in ROS is essential in the upcoming years if you are planning to build your career as a robotics engineer. In this chapter, we have gone through the basics of ROS mainly to refresh the concepts if you have already learned ROS. We discussed the necessity of learning ROS and how it excels among the current robotics software platforms. We went through the basic concepts such as ROS Master, Parameter server, and `roscore` and saw the explanation of the working of `roscore`. After discussing the internal working of `roscore`, we discussed each ROS concept, such as ROS topics, services, messages, and `actionlib` by illustrating examples. After demonstrating the working of each concept, we uploaded the package to GitHub and created a wiki page for the package. In the next chapter, we will discuss ROS robot modeling using URDF and xacro and will design some robot models.

2

Working with 3D Robot Modeling in ROS

The first phase of robot manufacturing is its design and modeling. We can design and model the robot using CAD tools such as AutoCAD, Solid Works, Blender, and so on. One of the main purposes of modeling robot is simulation.

The robotic simulation tool can check the critical flaws in the robot design and can confirm the working of the robot before it goes to the manufacturing phase.

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

In this chapter, we are going to discuss the designing of two robots. One is a **seven DOF (Degrees of Freedom)** manipulator and the other is a differential drive robot. In the upcoming chapters, we can see its simulation and how to build the real hardware and finally, it's interfacing to ROS.

If we are planning to create the 3D model of the robot and simulate using ROS, you need to learn about some ROS packages which helps in robot designing. ROS has a standard meta package for designing, and creating robot models called `robot_model`, which consists of a set of packages called `urdf`, `kdl_parser`, `robot_state_publisher`, `collada_urdf`, and so on. These packages help us create the 3D robot model description with the exact characteristics of the real hardware.

In this chapter, we will cover the following topics:

- ROS packages for robot modeling
- Understanding robot modeling using URDF
- Creating the ROS package for the robot description

- Creating our first URDF model
- Explaining the URDF code
- Understanding robot modeling using xacro
- Creating our first Xacro model
- Explanation first Xacro model
- Conversion of xacro to URDF
- Creating a robot description for a seven DOF robot manipulator
- Working with the joint state publisher and robot state publisher
- Creating Robot description for a differential wheeled robot

ROS packages for robot modeling

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

- `robot_model`: ROS has a meta package called `robot_model`, which contains important packages that help build the 3D robot models. We can see all the important packages inside this meta-package:
 - `urdf`: One of the important packages inside the `robot_model` meta package is `urdf`. The URDF package contains a C++ parser for the **Unified Robot Description Format (URDF)**, which is an XML file to represent a robot model.
- We can define a robot model, sensors, and a working environment using URDF and can parse it using URDF parsers. We can only describe a robot in URDF that has a tree-like structure in its links, that is, the robot will have rigid links and will be connected using joints. Flexible links can't be represented using URDF. The URDF is composed using special XML tags and we can parse these XML tags using parser programs for further processing. We can work on URDF modeling in the upcoming sections.
 - `joint_state_publisher`: This tool is very useful while designing robot models using URDF. This package contains a node called `joint_state_publisher`, which reads the robot model description, finds all joints, and publishes joint values to all nonfixed joints using GUI sliders. The user can interact with each robot joint using this tool and can visualize using RViz. While designing URDF, the user can verify the rotation and translation of each joint using this tool. We can discuss more about the `joint_state_publisher` node and its usage in the upcoming chapter.

- `kdl_parser`: **Kinematic and Dynamics Library (KDL)** is an ROS package that contains parser tools to build a KDL tree from the URDF representation. The kinematic tree can be used to publish the joint states and also to forward and inverse kinematics of the robot.
- `robot_state_publisher`: This package reads the current robot joint states and publishes the 3D poses of each robot link using the kinematics tree build from the URDF. The 3D pose of the robot is published as ROS `tf` (transform). ROS `tf` publishes the relationship between coordinates frames of a robot.
- `xacro`: Xacro stands for (XML Macros) and we can define how `xacro` is equal to URDF plus add-ons. It contains some add-ons to make URDF shorter, readable, and can be used for building complex robot descriptions. We can convert `xacro` to URDF at any time using some ROS tools. We will see more about `xacro` and its usage in the upcoming sections.

Understanding robot modeling using URDF

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

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

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

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

The syntax is as follows:

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

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

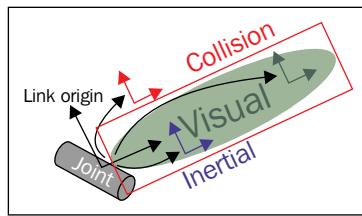


Figure 1 : Visualization of a URDF link

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

The syntax is as follows:

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

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

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

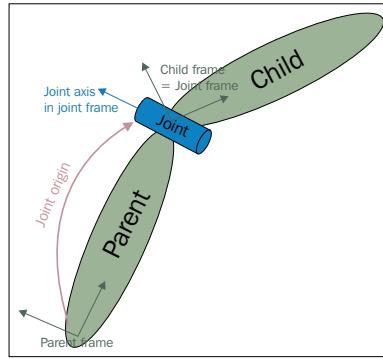


Figure 2 : Visualization of a URDF joint

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

The syntax is as follows:

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

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

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

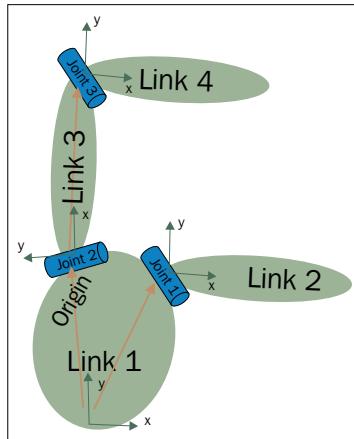


Figure 3 : Visualization of a robot model having joints and links

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

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

We can find more URDF tags at <http://wiki.ros.org/urdf/XML>.

Creating the ROS package for the robot description

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

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

The package mainly depends on the `urdf` and `xacro` packages, and we can create the `urdf` file of the robot inside this package and create launch files to display the created `urdf` in RViz. The full package is available on the following Git repository, you can clone the repository for a reference to implement this package or you can get the package from the book's source code:

```
$ git clone
https://github.com/qboticslabs/mastering_ros_robot_description_pkg.git
```

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

Creating our first URDF model

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

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

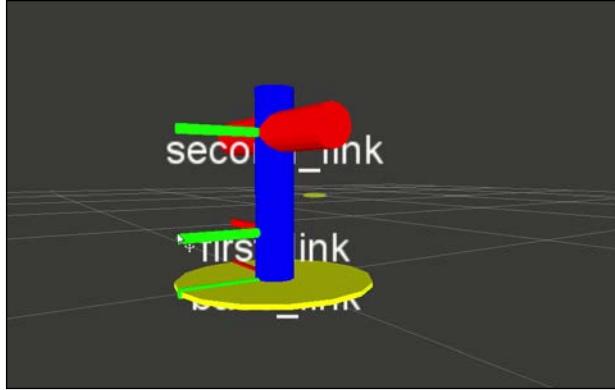


Figure 4 : Visualization of a pan and tilt mechanism in RViz

Let's see the URDF code of this mechanism. Navigate to `chapter_2_code/mastering_ros_robot_description_pkg/urdf` and open `pan_tilt.urdf`. The code indentation in URDF is not mandatory for URDF but it keeping indentation can improve code readability:

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

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

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

```
</joint>

<link name="pan_link">
  <visual>
    <geometry>
      <cylinder length="0.4" radius="0.04"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0.09"/>
    <material name="red">
      <color rgba="0 0 1 1"/>
    </material>
  </visual>
</link>

<joint name="tilt_joint" type="revolute">
  <parent link="pan_link"/>
  <child link="tilt_link"/>
  <origin xyz="0 0 0.2"/>
  <axis xyz="0 1 0" />
</joint>

<link name="tilt_link">
  <visual>
    <geometry>
      <cylinder length="0.4" radius="0.04"/>
    </geometry>
    <origin rpy="0 1.5 0" xyz="0 0 0"/>
    <material name="green">
      <color rgba="1 0 0 1"/>
    </material>
  </visual>
</link>
</robot>
```

Explaining the URDF file

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

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

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

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

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

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

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

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

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

```
$ check_urdf pan_tilt.urdf
```

The `check_urdf` command will parse `urdf` and show an error, if any. If everything is OK, it will show an output as follows:

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

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

```
$ urdf_to_graphviz pan_tilt.urdf
```

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

```
$ evince pan_tilt.pdf
```

We will get the following output:

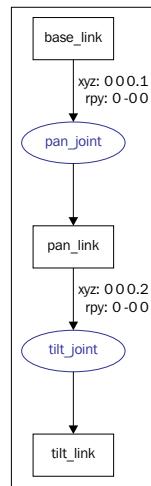


Figure 5 : Graph of joint and links in pan and tilt mechanism

Visualizing the robot 3D model in RViz

After designing URDF, we can view it on RViz. We can create a `view_demo.launch` launch file and put the following code into the launch folder. Navigate to `chapter_2_code/mastering_ros_robot_description_pkg/launch` for the same code:

```
<launch>
  <arg name="model" />
  <param name="robot_description" textfile="$(find mastering_ros_
robot_description_pkg)/urdf/pan_tilt.urdf" />
  <param name="use_gui" value="true"/>

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

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

</launch>
```

We can launch the model using the following command:

```
$ rosrun mastering_ros_robot_description_pkg view_demo.launch
```

If everything works fine, we will get a pan and tilt mechanism in RViz.

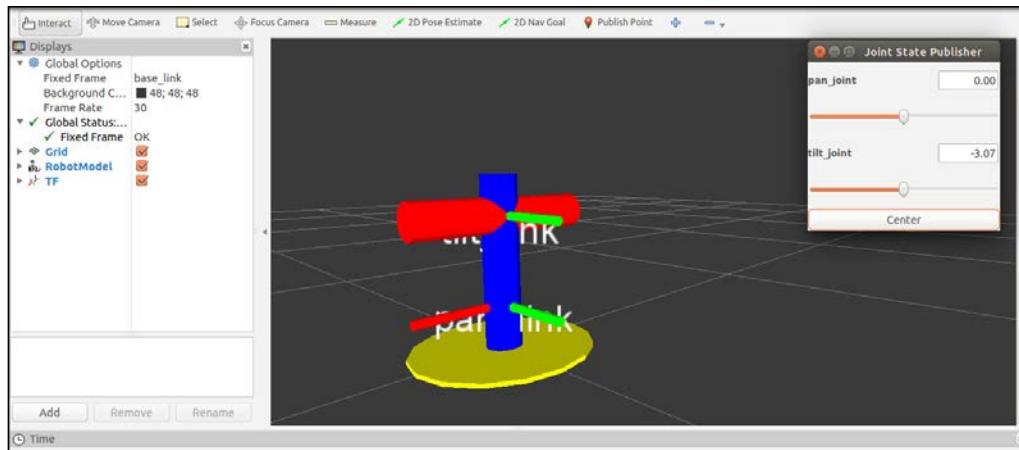


Figure 6 : Joint level of pan and tilt mechanism

Interacting with pan and tilt joints

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

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

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

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

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



Figure 6 : Joint level of pan and tilt mechanism

The preceding screenshot shows the GUI of **Joint State Publisher** with sliders and current joint values shown in the box.

Adding physical and collision properties to a URDF model

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

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

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

```
</collision>

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

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

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

Understanding robot modeling using xacro

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

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

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

Also, if there is some programmability, such as adding variable, constants, mathematical expressions, conditional statement, and so on, in the description language, it will be more user friendly.

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

- **Simplify URDF:** Xacro is the cleaned up version of URDF. What it does is, it creates macros inside the robot description and reuses the macros. This can reduce the code length. Also, it can include macros from other files and make the code more readable, simpler, and modular.

- **Programmability:** The xacro language support a simple programming statement in its description. There are variables, constants, mathematical expressions, conditional statements, and so on that make the description more intelligent and efficient.

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

We can discuss the same description of pan and tilt using xacro. Navigate to chapter_2_code/mastering_ros_robot_description_pkg/urdf, and the file name is pan_tilt.xacro. Instead of .urdf, we need to use .xacro for the xacro file definition. Here is the explanation of the xacro code:

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

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

Using properties

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

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

```
<xacro:property name="base_link_length" value="0.01" />
<xacro:property name="base_link_radius" value="0.2" />

<xacro:property name="pan_link_length" value="0.4" />
<xacro:property name="pan_link_radius" value="0.04" />
```

We can use the value of the variable by replacing the hard coded value by the following definition as given here:

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

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

Using the math expression

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

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

Using macros

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

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

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

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

The xacro definition improved the code readability and reduced the number of lines compared to urdf. Next, we can see how to convert xacro to the urdf file.

Conversion of xacro to URDF

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

```
$ rosrun xacro xacro.py pan_tilt.xacro > pan_tilt_generated.urdf
```

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

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

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

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

Creating the robot description for a seven DOF robot manipulator

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

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

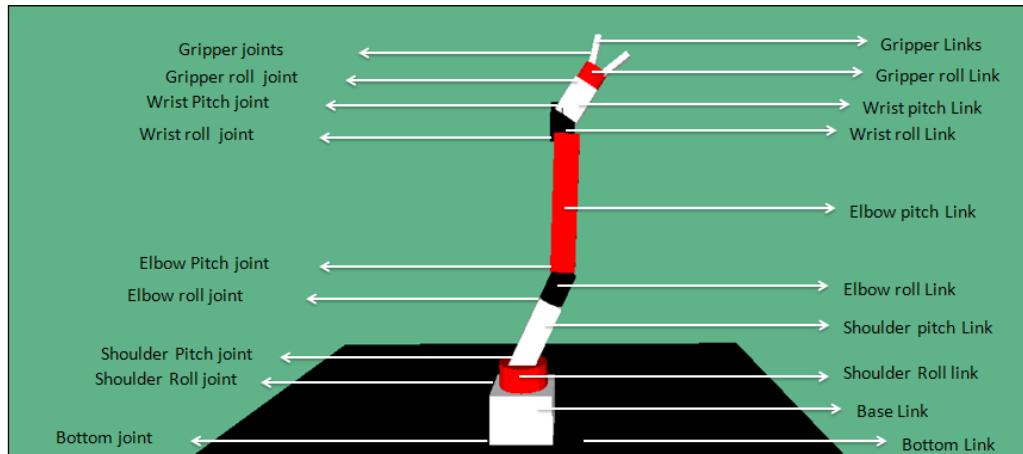


Figure 8 : Joints and Links of seven dof arm robot

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

Arm specification

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

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

Type of joints

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

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

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

Explaining the xacro model of seven DOF arm

We will define 10 links and 9 joints on this robot and 2 links and 2 joints in the robot gripper.

Let's start by discussing the `xacro` definition:

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

<robot name="seven_dof_arm"
  xmlns:xacro="http://www.ros.org/wiki/xacro">
```

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

Using constants

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

```
<property name="deg_to_rad" value="0.01745329251994329577"/>
<property name="M_PI" value="3.14159"/>

<property name="elbow_pitch_len" value="0.22" />
<property name="elbow_pitch_width" value="0.04" />
<property name="elbow_pitch_height" value="0.04" />
```

Using macros

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

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

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

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

```
<hardwareInterface>PositionJointInterface</hardwareInterface>
  <mechanicalReduction>1</mechanicalReduction>
  </actuator>
  </transmission>
</xacro:macro>
```

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

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

Including other xacro files

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

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

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

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

We will discuss more on vision processing in *Chapter 9, Building and Interfacing Differential Drive Mobile Robot Hardware in ROS*.

Using meshes in the link

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

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

Working with the robot gripper

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

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

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

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

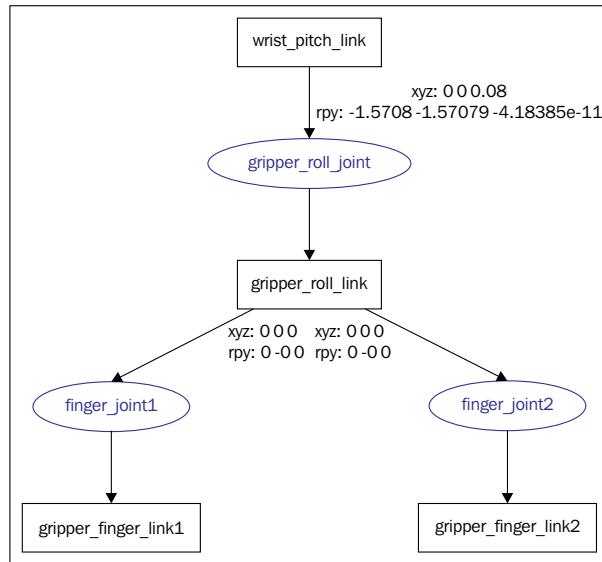


Figure 9 : Graph of the end effector section of seven dof arm robot

Viewing the seven DOF arm in RViz

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

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

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

  <!-- Parsing xacro and loading robot_description parameter -->

  <param name="robot_description" command="$(find xacro)/xacro.py
$(find mastering_ros_robot_description_pkg)/urdf/ seven_dof_arm.xacro
" />

  <!-- Setting gui parameter to true for display joint slider, for
getting joint control -->
  <param name="use_gui" value="true"/>

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

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

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

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

```
$ rosrun master Ros_robot_description_pkg view_arm.launch
```

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

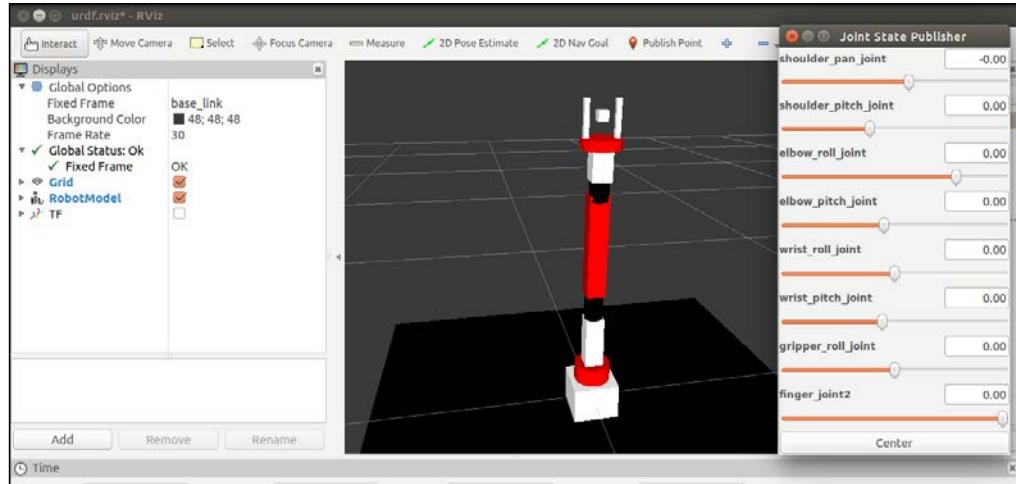


Figure 10 : Seven dof arm in RViz with joint_state_publisher

We can interact with the joint slider and move the joints of the robot. We can next discuss what the joint state publisher is.

Understanding joint state publisher

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

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

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

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

If we set `use_gui` to true, the `joint_state_publisher` node displays a slider based control window to control each joint. The lower and upper value of a joint will be taken from the lower and upper values associated with the `limit` tag used inside the `joint` tag. The preceding screenshot shows RViz along with GUI to publish joint states with the `use_gui` parameter set to true.

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

Understanding the robot state publisher

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

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

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

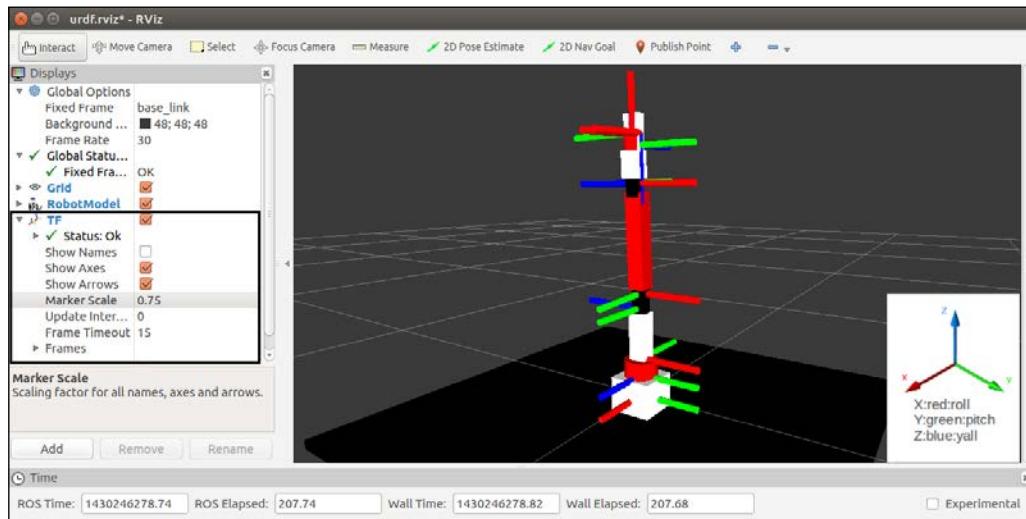


Figure 11 : TF view of seven dof arm in RViz

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

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

Creating a robot model for the differential drive mobile robot

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

There are two supporting wheels called caster wheels that will support the robot and freely rotate according to the movement of the main wheels.

The UDRF model of this robot is present in the cloned ROS package. The final robot model is shown as follows:

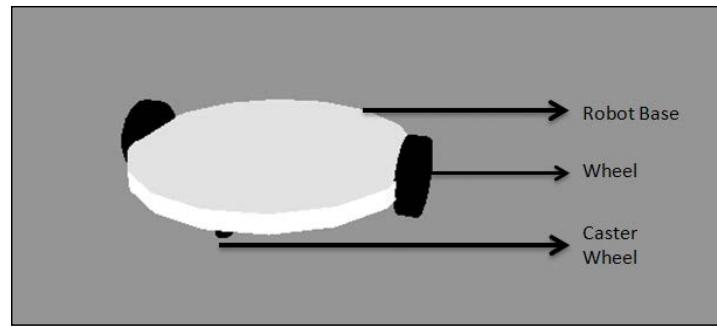


Figure 12 : 3D model of differential drive mobile robot

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

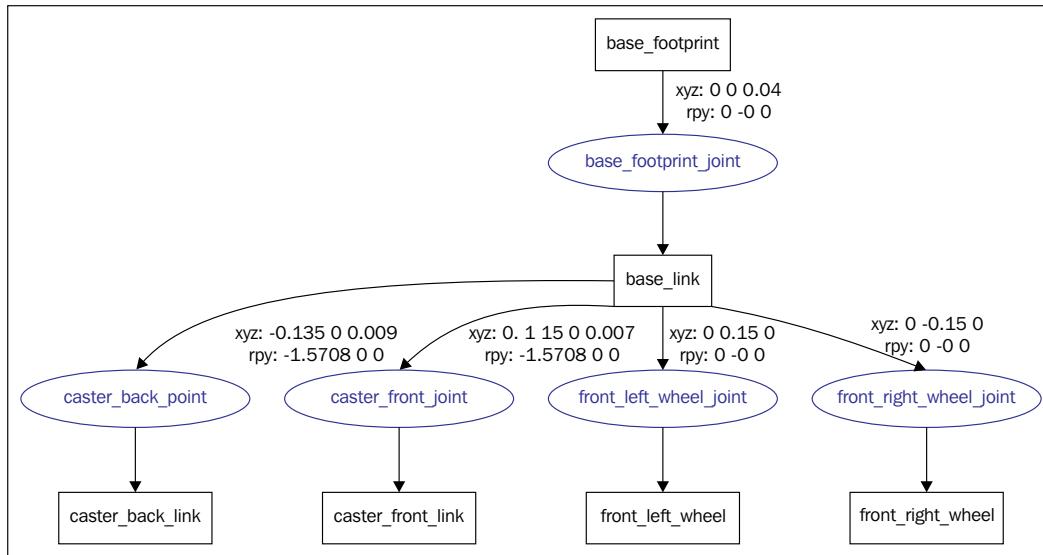


Figure 13 : Graphical representation of the links and joints in mobile robot

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

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

```

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

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

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

```

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

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

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

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

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

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

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

</xacro:macro>
</robot>
```

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

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

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

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

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

```
<launch>
    <arg name="model" />
    <!-- Parsing xacro and setting robot_description parameter -->
    <param name="robot_description" command="$(find xacro)/xacro.py
$(find mastering_ros_robot_description_pkg)/urdf/diff_wheeled_robot.
xacro" />
    <!-- Setting gui parameter to true for display joint slider -->
    <param name="use_gui" value="true"/>
    <!-- Starting Joint state publisher node which will publish the
joint values -->
    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
    <!-- Starting robot state publish which will publish tf -->
    <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
    <!-- Launch visualization in rviz -->
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find mastering_
ros_robot_description_pkg)/urdf.rviz" required="true" />
</launch>
```

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

We can view the mobile robot using the following command:

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

The screenshot of the robot in RViz is as follows:

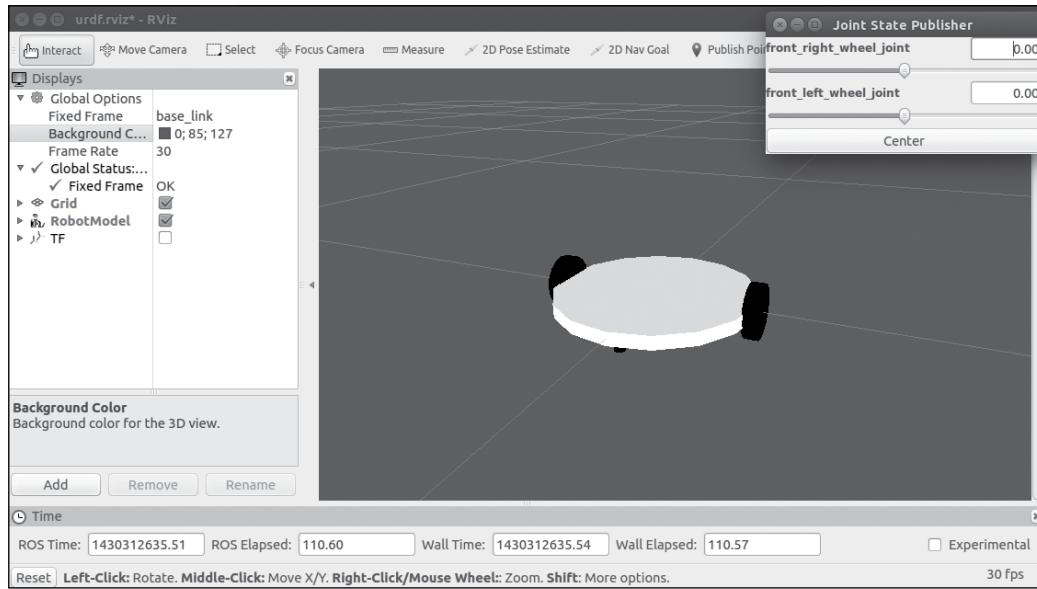


Figure 14 : Visualizing mobile robot in RViz with joint state publisher.

Questions

1. What are the packages used for robot modeling in ROS?
2. What are the important URDF tags used for robot modeling?
3. What are the reasons for using xacro over URDF?
4. What is the use of the joint state publisher and robot state publisher packages?
5. What is the use of the transmission tag in URDF?

Summary

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

3

Simulating Robots Using ROS and Gazebo

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

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

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

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

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

We will cover the following list of topics in this chapter:

- Simulating robotic arms in Gazebo
- Adding sensors to the robotic arm simulation
- Interfacing Gazebo to ROS
- Adding ROS controllers to robots

- Working with the robotic arm joint control
- Simulating the mobile robot in Gazebo
- Adding sensors to mobile robot simulation
- Moving the mobile robot in Gazebo using a keyboard teleop

Simulating the robotic arm using Gazebo and ROS

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

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

- In ROS Jade:

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

- In ROS Indigo:

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

The use of each package is as follows:

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

After installation, check whether the Gazebo is properly installed in Ubuntu using the following command:

```
$ gazebo
```

We can check the ROS interface of Gazebo using the following command:

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

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

The Robotic arm simulation model for Gazebo

We can create the simulation model for a robotic arm by updating the existing robot description by adding simulation parameters. You can see the complete simulation model of the robot in the `chapter_3_code/ mastering_ros_robot_description_pkg/urdf/ seven_dof_arm.xacro` file.

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

To launch the existing simulation model, we can use the `chapter_3_code/seven_dof_arm_gazebo` package, which has a launch file called `seven_dof_arm_world.launch`. The file definition is as follows:

```
<launch>

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

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

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

    <!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
```

```
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
  respawn="false" output="screen"
  args="--urdf -model seven_dof_arm -param robot_description"/>
</launch>
```

Build the package called `seven_dof_arm_gazebo` from `chapter_3_code` in your `catkin` workspace. This is the package we used for the robot arm simulation.

Launch the following command and check what you get:

```
$ rosrun seven_dof_arm_gazebo seven_dof_arm_world.launch
```

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

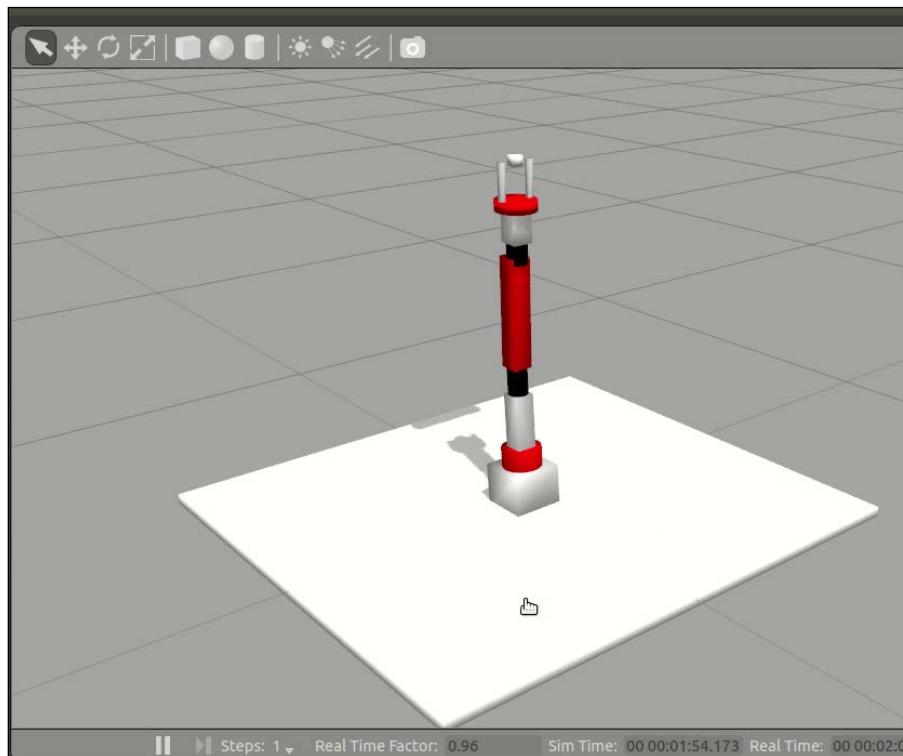


Figure 1 : Simulation of Seven DOF arm in Gazebo

Let's discuss the `seven_dof_arm.xacro` simulation model in detail.

Adding colors and textures to the Gazebo robot model

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

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

Adding transmission tags to actuate the model

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

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

Here, the `<joint name = "">` is the joint in which we link the actuators. The `<type>` element is the type of transmission. Currently, `transmission_interface/SimpleTransmission` is only supported. The `<hardwareInterface>` element is the type of hardware interface to load (position, velocity, or effort interfaces). The hardware interface is loaded by the `gazebo_ros_control` plugin; we can see more about this plugin in the next section.

Adding the `gazebo_ros_control` plugin

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

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

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

Adding a 3D vision sensor to Gazebo

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

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

Here, we are adding a 3D vision sensor called the Asus Xtion Pro model in Gazebo. The sensor model is already implemented in the `gazebo_ros_pkgs/gazebo_plugins` ROS package, which we already installed in our ROS system.

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

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

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

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

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

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

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

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

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

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

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

```
<cameraName>${name}</cameraName>
<imageTopicName>rgb/image_raw</imageTopicName>

</plugin>
```

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

Simulating the robotic arm with Xtion Pro

After discussing the camera plugin definition in Gazebo, we can launch the complete simulation using the following command:

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

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

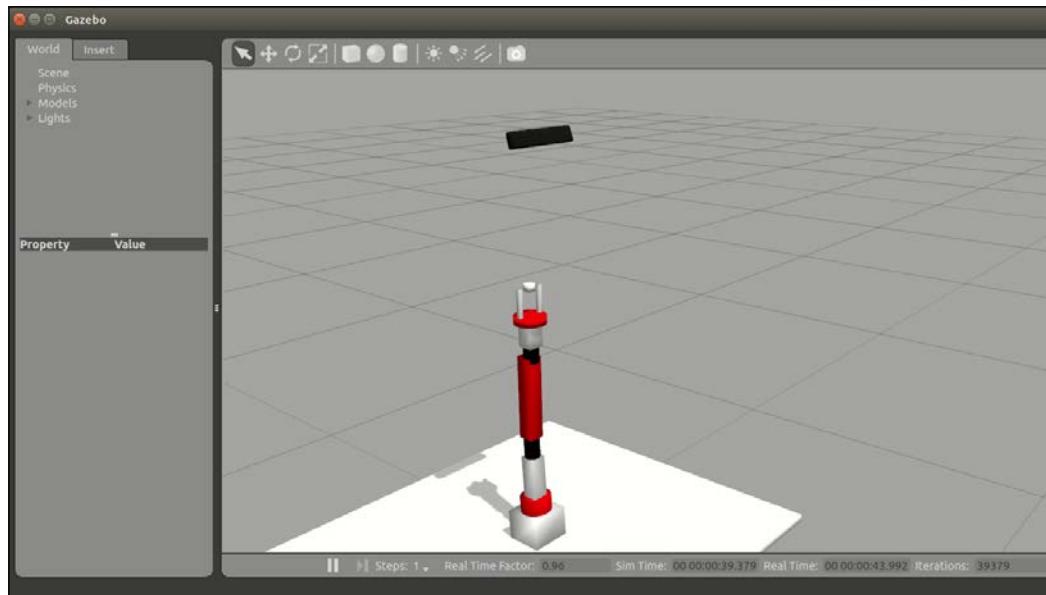


Figure 2 : Simulation of seven DOF arm with Asus Xtion Pro in Gazebo

We can work with the Xtion Pro data from Gazebo and check whether it provides the correct image output.

Visualizing the 3D sensor data

We can just list out the topics generated while performing simulation, and here are the topics generated by the sensor plugin:

```
$ rostopic list
```

```
/rgbd_camera/depth/camera_info
/rgbd_camera/depth/image_raw
/rgbd_camera/depth/points
/rgbd_camera/ir/camera_info
/rgbd_camera/ir/image_raw
/rgbd_camera/ir/image_raw/compressed
/rgbd_camera/ir/image_raw/compressed/parameter_descriptions
/rgbd_camera/ir/image_raw/compressed/parameter_updates
/rgbd_camera/ir/image_raw/compressedDepth
/rgbd_camera/ir/image_raw/compressedDepth/parameter_descriptions
/rgbd_camera/ir/image_raw/compressedDepth/parameter_updates
/rgbd_camera/ir/image_raw/theora
/rgbd_camera/ir/image_raw/theora/parameter_descriptions
/rgbd_camera/ir/image_raw/theora/parameter_updates
/rgbd_camera/parameter_descriptions
/rgbd_camera/parameter_updates
/rgbd_camera/rgb/camera_info
/rgbd_camera/rgb/image_raw
/rgbd_camera/rgb/image_raw/compressed
```

Figure 3 : ROS topics generated by 3D sensor in Gazebo

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

- View the RGB raw image:

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

- View the IR raw image:

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

- View the depth image:

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

Here is the screenshot with all these images:

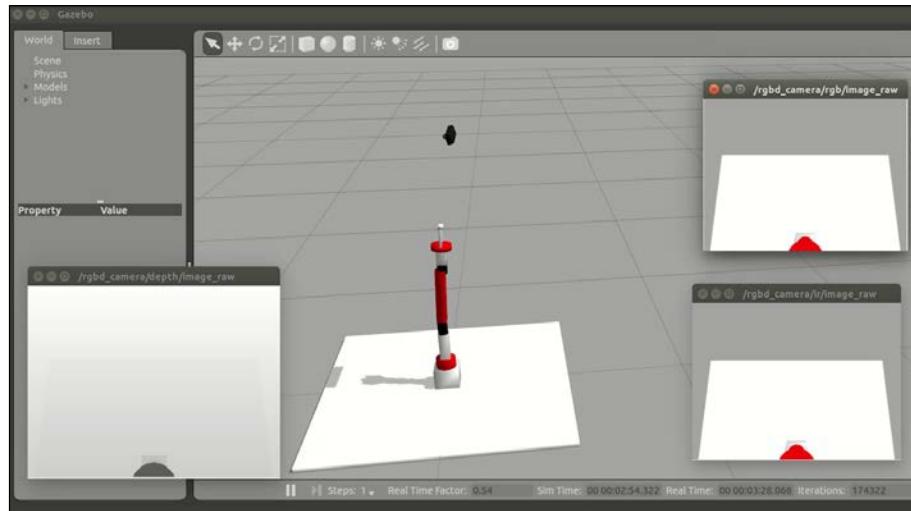


Figure 4 : Viewing images of Xtion Pro in Gazebo

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

Launch RViz using the following command:

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

Add a **PointCloud2** display type and **Topic** as `/rgbd_camera/depth/points`. Set the **Color Transformer** option as `RGB8`. We will get a point cloud view as follows:

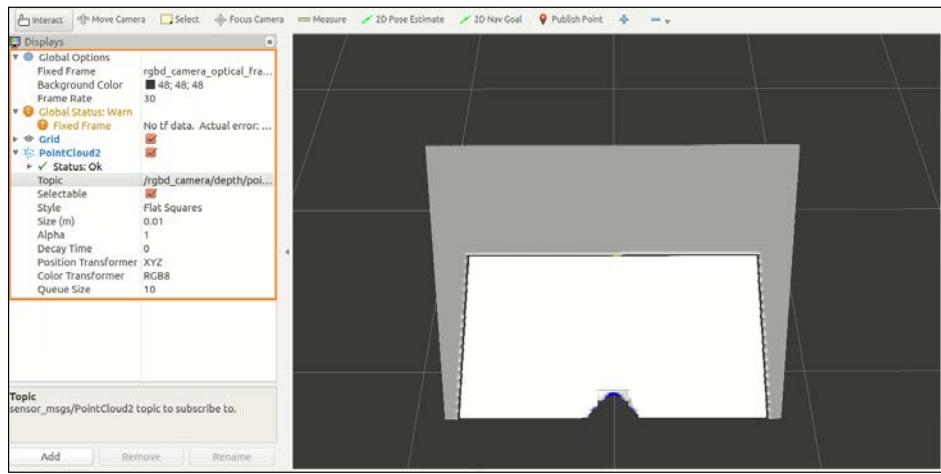


Figure 5 : Viewing point cloud data from Xtion Pro in RViz

Moving robot joints using ROS controllers in Gazebo

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

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

A ROS controller mainly consists of a feedback mechanism, most probably a PID loop, which can receive a set point, and control the output using the feedback from the actuators.

The ROS controller will not directly communicate with the hardware, instead of that, the robot hardware interface can talk with hardware. The main function of the hardware interface is that it will act as a mediator between ROS controllers and the Real Hardware/Simulator and allocate the necessary resources for the controllers and check the resource conflicts too.

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

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

Understanding the `ros_control` packages

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

- `control_toolbox`: This package contains common modules (P.I.D and Sine) that can be used by all controllers
- `controller_interface`: This package contains the interface base class for controllers
- `controller_manager`: This package provides the infrastructure to load, unload, start, and stop controllers
- `controller_manager_msgs`: This package provides the message and service definition for the controller manager

- `hardware_interface`: This contains the base class for the hardware interfaces
- `transmission_interface`: This package contains the interface classes for the transmission interface (differential, four bar linkage, joint state, position, and velocity)

Different types of ROS controllers and hardware interfaces

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

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

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

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

How the ROS controller interacts with Gazebo

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

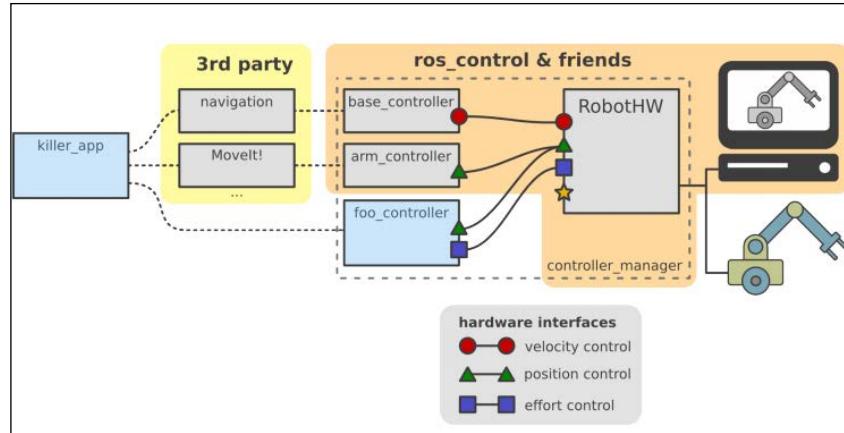


Figure 6 : Interacting ROS controllers with Gazebo

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

The hardware interface allocates each resource for the controllers and sends values to each resource. The detailed diagram of communications between the robot controllers and robot hardware interfaces are shown as follows:

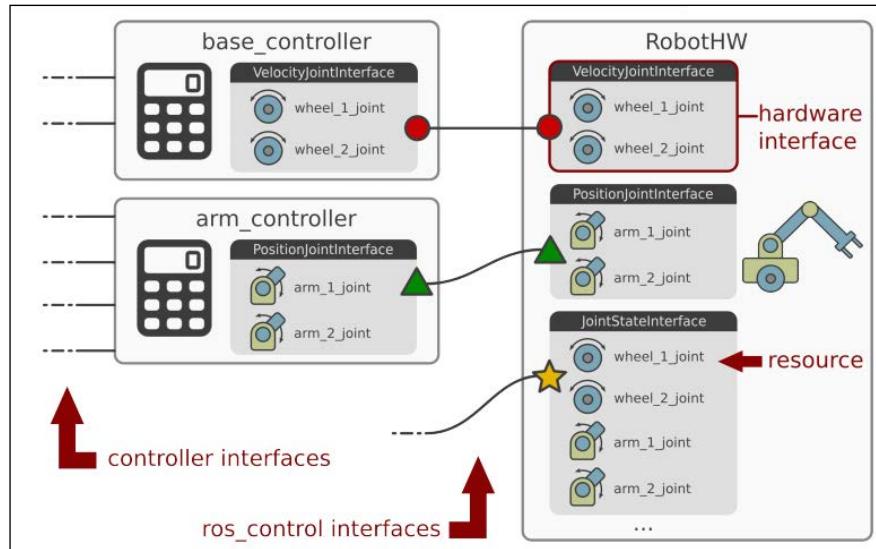


Figure 7 : Illustration of ROS controllers and hardware interfaces

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

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

Interfacing joint state controllers and joint position controllers to the arm

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

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

We will find the configuration file for the controller at `seven_dof_arm_gazebo_control.yaml` in `chapter_3_code/seven_dof_arm_gazebo/config`.

Here is the configuration file definition:

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

```

pid: {p: 100.0, i: 0.01, d: 10.0}
joint5_position_controller:
  type: position_controllers/JointPositionController
  joint: wrist_roll_joint
  pid: {p: 100.0, i: 0.01, d: 10.0}
joint6_position_controller:
  type: position_controllers/JointPositionController
  joint: wrist_pitch_joint
  pid: {p: 100.0, i: 0.01, d: 10.0}
joint7_position_controller:
  type: position_controllers/JointPositionController
  joint: gripper_roll_joint
  pid: {p: 100.0, i: 0.01, d: 10.0}

```

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

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

Launching the ROS controllers with Gazebo

If the controller configuration is ready, we can build a launch file that starts all the controllers along with the Gazebo simulation. Navigate to `chapter_3_code/seven_dof_arm_gazebo/launch` and open the `seven_dof_arm_gazebo_control.launch` file:

```

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

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

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

```

```
joint2_position_controller  
joint3_position_controller  
joint4_position_controller  
joint5_position_controller  
joint6_position_controller  
joint7_position_controller"/>  
  
<!-- convert joint states to TF transforms for rviz, etc -->  
<node name="robot_state_publisher" pkg="robot_state_publisher"  
type="robot_state_publisher"  
respawn="false" output="screen">  
  <remap from="/joint_states" to="/seven_dof_arm/joint_states" />  
</node>  
  
</launch>
```

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

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

```
$ roslaunch seven_dof_gazebo seven_dof_gazebo_control.launch
```

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

```
[INFO] [WallTime: 1445626906.221147] [0.223000] Loading controller: joint_state_controller  
[INFO] [WallTime: 1445626906.447525] [0.378000] Loading controller: joint1_position_controller  
[INFO] [WallTime: 1445626906.886213] [0.762000] Loading controller: joint2_position_controller  
[INFO] [WallTime: 1445626906.921447] [0.778000] Loading controller: joint3_position_controller  
[INFO] [WallTime: 1445626906.956767] [0.804000] Loading controller: joint4_position_controller  
[INFO] [WallTime: 1445626907.001207] [0.840000] Loading controller: joint5_position_controller  
[INFO] [WallTime: 1445626907.029617] [0.869000] Loading controller: joint6_position_controller  
[INFO] [WallTime: 1445626907.062851] [0.899000] Loading controller: joint7_position_controller  
[INFO] [WallTime: 1445626907.089303] [0.925000] Controller Spawner: Loaded controllers: joint_state_controller, joint1_position_controller, joint2_position_controller, joint3_position_controller, joint4_position_controller, joint5_position_controller, joint6_position_controller, joint7_position_controller  
[INFO] [WallTime: 1445626907.095819] [0.932000] Started controllers: joint_state_controller, joint1_position_controller, joint2_position_controller, joint3_posi
```

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

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

```
$ rostopic list
```

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

Figure 9 : Position controller command topics of seven DOF arm

Moving the robot joints

After getting done with the preceding topics, we can start commanding positions to each joint.

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

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

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

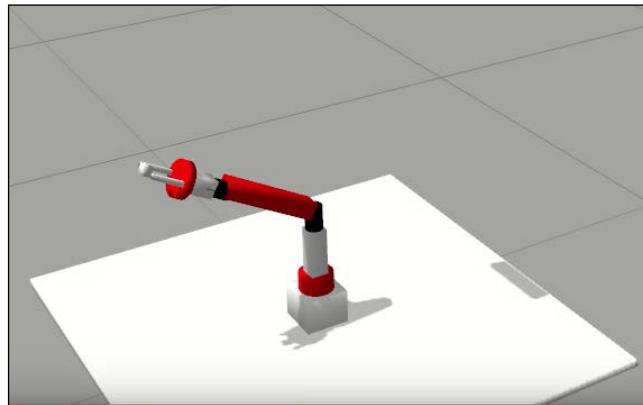


Figure 10 : Moving a joint of the arm in Gazebo

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

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

Simulating a differential wheeled robot in Gazebo

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

You will get the `diff_wheeled_robot.xacro` mobile robot description at `chapter_3_code/mastering_ros_robot_description_pkg/urdf`.

Let's create a launch file to spawn the simulation model in Gazebo.

Navigate to `chapter_3_code/diff_wheeled_robot_gazebo/launch` and take the `diff_wheeled_gazebo.launch` file. Here is the definition of this launch:

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

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

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

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

</launch>
```

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

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

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

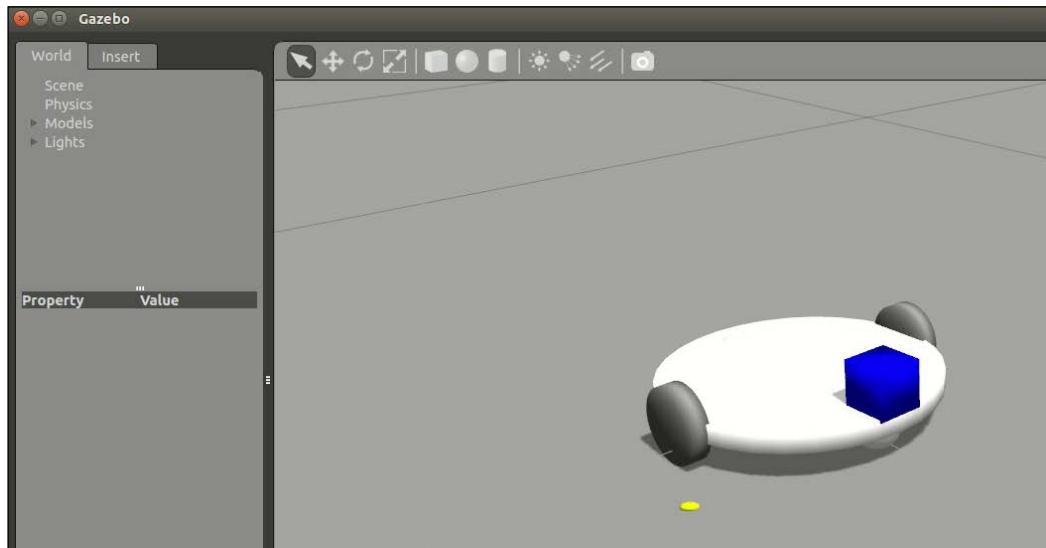


Figure 11 : Differential wheeled robot in Gazebo

After successful simulation, let's add the laser scanner to the robot. In the preceding figure, we can see a box on the top of the robot, which is the sensor we added to the URDF, and here is how we do it.

Adding the laser scanner to Gazebo

We add the laser scanner on the top of Gazebo in order to perform high-end operations such as autonomous navigation using this robot. Here, we can see that an extra code section needed to be added in `diff_wheeled_robot.xacro` to have the laser scanner on the robot:

```
<link name="hokuyo_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="${hokuyo_size} ${hokuyo_size} ${hokuyo_size}" />
    </geometry>
    <material name="Blue" />
  </visual>
</link>
```

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

In this section, we use the Gazebo ROS plugin file called `libgazebo_ros_laser.so` to simulate the laser scanner.

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

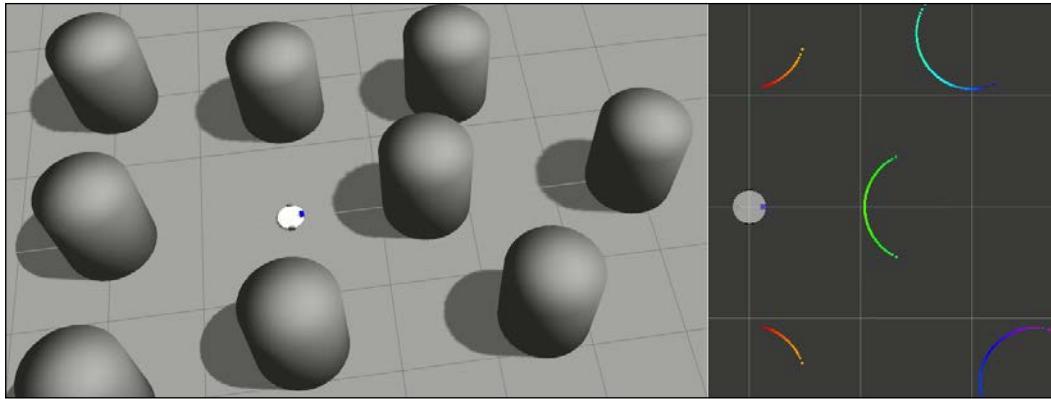


Figure 12 : Differential drive robot in random object in Gazebo

The laser scanner plugin publishes laser data to scan a topic; we can just echo the topic to get the laser scan data array:

```
$ rostopic echo /scan
```

Moving the mobile robot in Gazebo

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

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

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

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

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

        <leftJoint>front_left_wheel_joint</leftJoint>
```

```
<rightJoint>front_right_wheel_joint</rightJoint>

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

</plugin>
</gazebo>
```

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

An important parameter that we need to move the robot is

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

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

Adding joint state publishers in the launch file

After adding the differential drive plugin, we need to joint state publishers to the existing launch file, or we can build a new one. You can see the new final launch file: `diff_wheeled_gazebo_full.launch` from `chapter_3_code/diff_wheeled_robot_gazebo/launch`.

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

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

Adding the ROS teleop node

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

The teleop implemented in `chapter_3_code/diff_wheeled_robot_control` package. The script folder contains the `diff_wheeled_robot_key` node, which is the teleop node.

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

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

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

Let's start moving the robot.

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

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

Start the teleop node:

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

Start RViz to visualize the robot state and laser data:

```
$ rosrun rviz rviz
```

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

In the teleop terminal, we can use some keys (*U, I, O, J, K, L, M, , , .*) for direction adjustment and other keys (*Q, Z, W, X, E, C, K, space key*) for speed adjustments. Here is the screenshot showing the robot moving in Gazebo using teleop and its visualization in RViz.

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

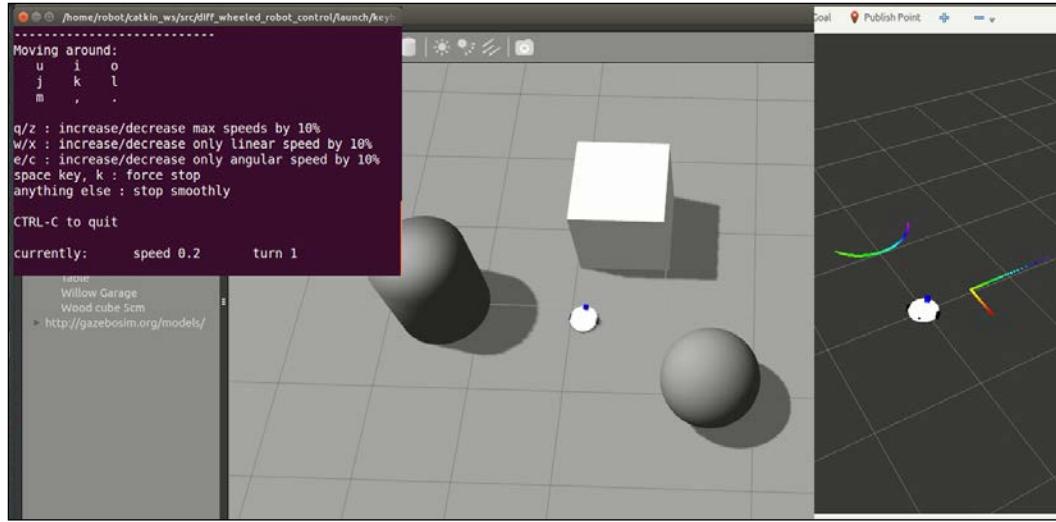


Figure 13 : Moving differential drive robot in Gazebo using teleoperation

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

Questions

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

Summary

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

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

Similar to the robotic arm, we created the URDF for Gazebo simulation and added the necessary Gazebo ROS plugin for the laser scanner and differential drive mechanism. After completing the simulation model, we launched the simulation using a custom launch file. At last, we have seen how to move the robot using the teleop node.

We will get to know more about the robotic arm and mobile robots, which are supported by ROS, from the following link <http://wiki.ros.org/Robots>.

In the next chapter, we can see how to interface the robotic arm with the ROS MoveIt package and the mobile robot with the Navigation stack.

4

Using the ROS MoveIt! and Navigation Stack

In the previous chapters, we have been discussing about the designing and simulation of a robotic arm and mobile robot. We controlled each joint of the robotic arm in Gazebo using the ROS controller and moved the mobile robot inside Gazebo using the `teleop` node.

In this chapter, we are going to interface out of the box functionalities, such as robot manipulation and autonomous navigation using the ROS **MoveIt!** and **Navigation stack**.

MoveIt! is a set of packages and tools for doing mobile manipulation in ROS. The official web page (<http://moveit.ros.org/>) contains the documentations, the list of robots using MoveIt!, and various examples to demonstrate pick and place, grasping, simple motion planning using inverse kinematics, and so on.

MoveIt! contains state of the art software for motion planning, manipulation, 3D perception, kinematics, collision checking, control, and navigation. Apart from the command line interface, MoveIt! has some good GUI to interface a new robot to MoveIt!. Also, there is a RViz plugin which enables motion planning from RViz itself. We will also see how to motion plan our robot using MoveIt! C++ APIs.

Next is the Navigation stack, and of course this is another set of powerful tools and libraries to work mainly for mobile robot navigation. The Navigation stack contains ready-to-use navigation algorithms which can be used in mobile robots, especially for differential wheeled robots. Using these stacks, we can make the robot autonomous and that is the final concept that we are going to see in the Navigation stack.

The first section of this chapter will discuss more on the MoveIt! package, installation, and architecture. After discussing the main concepts of MoveIt!, we will see how to create a MoveIt! package for our robotic arm, which can provide collision-aware path planning to our robot. Using this package, we can perform motion planning (inverse kinematics) in RViz, and can interface to Gazebo or the real robot for executing the paths.

After discussing the interfacing, we will discuss more about the Navigation stack and see how to perform autonomous navigation using **Simultaneous Localization And Mapping (SLAM)** and **Adaptive Monte Carlo Localization (AMCL)**.

Installing MoveIt!

Let's start with installing MoveIt!. The installation procedure is very simple and is just a single command.

Installing MoveIt! on ROS Indigo can be done using the following command. Here we are installing MoveIt! binary packages.

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

In ROS Jade, we can install MoveIt! using the following command:

```
$ sudo apt-get install ros-jade-moveit-ros ros-jade-moveit-plugins  
ros-jade-moveit-planners
```

MoveIt! architecture

Let's start with MoveIt! and its architecture. Understanding the architecture of MoveIt! helps to program and interface the robot to MoveIt!. We will quickly go through the architecture and the important concepts of MoveIt!, and start interfacing and programming our robots.

Here is the MoveIt! architecture, included in their official web page at <http://moveit.ros.org/documentation/concepts>:

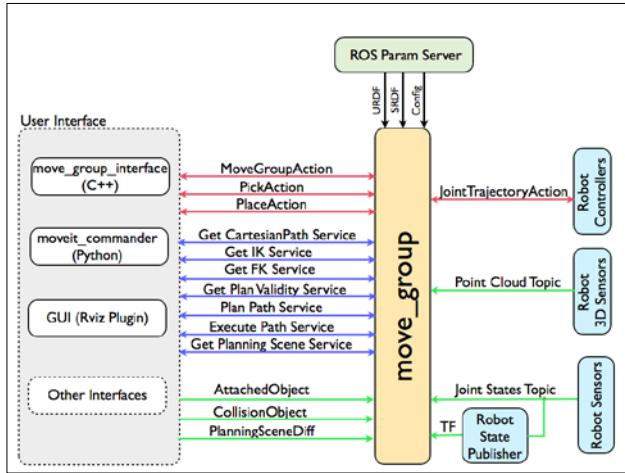


Figure 1: MoveIt! architecture diagram

The move_group node

We can say that `move_group` is the heart of MoveIt! as this node acts as an integrator of the various components of the robot and delivers actions/services according to the user's needs.

From the architecture, it's clear that the `move_group` node collects robot information such as point cloud, joint state of the robot, and transform (T.F) of the robot in the form of topics and services.

From the parameter server, it collects the robot kinematics data, such as `robot_description` (URDF), **SRDF (Semantic Robot Description Format)**, and the configuration files. The SRDF file and the configuration files are generated while we generate a MoveIt! package for our robot. The configuration files contain the parameter file for setting joint limits, perception, kinematics, end effector, and so on. We will see the files when we discuss generating the MoveIt! package for our robot.

When MoveIt! gets all this information about the robot and its configuration, we can say it is properly configured and we can start commanding the robot from the user interfaces. We can either use C++ or Python MoveIt! APIs to command the `move_group` node to perform actions such as pick/place, IK, FK, among others. Using the RViz motion planning plugin, we can command the robot from the RViz GUI itself.

As we already discussed, the `move_group` node is an integrator; it does not run any kind of motion planning algorithms but instead connects all the functionalities as plugins. There are plugins for kinematics solvers, motion planning, and so on. We can extend the capabilities through these plugins.

After motion planning, the generated trajectory talks to the controllers in the robot using the `FollowJointTrajectoryAction` interface. This is an action interface in which an action server is run on the robot, and `move_node` initiates an action client which talks to this server and executes the trajectory on the real robot/Gazebo simulator.

At the end of the MoveIt! discussion, we will see how to talk from MoveIt! RViz GUI to Gazebo. Following is a screenshot showing a robotic arm that is controlling from RViz and the trajectory is executed inside Gazebo:

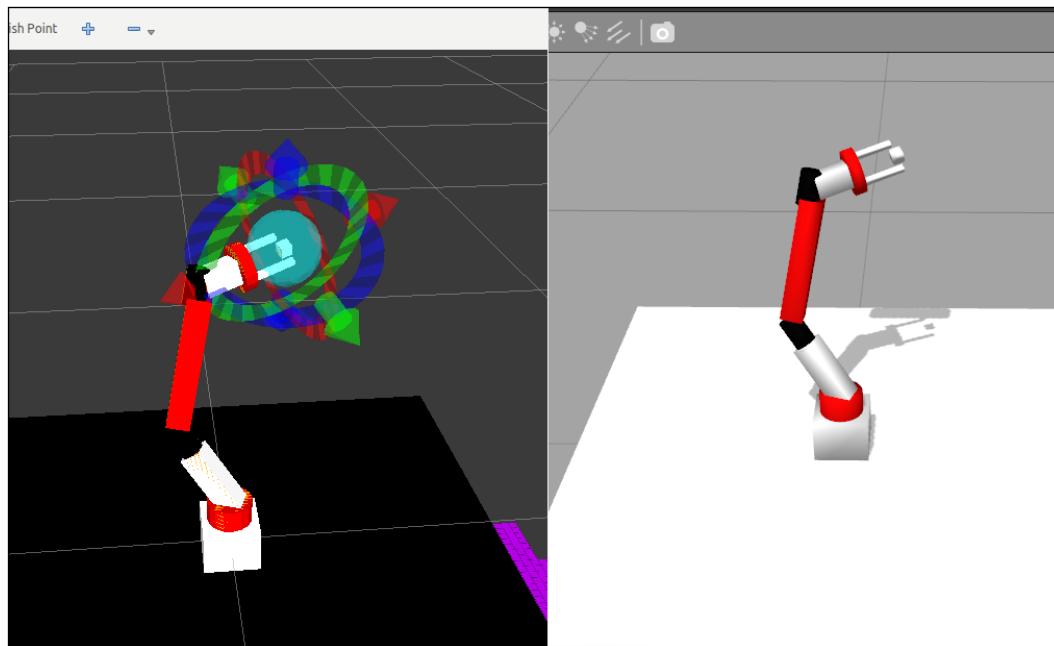


Figure 2 : Trajectory from RViz GUI is executing in Gazebo

Motion planning using MoveIt!

Assume that we know the starting pose of the robot, a desired goal pose of the robot, the geometrical description of the robot, and geometrical description of the world, then motion planning is the technique to find an optimum path that moves the robot gradually from the start pose to the goal pose, while never touching any obstacles in the world and without colliding with the robot links.

Here the geometrical description of the robot is our URDF file and the geometrical description of the world can also be included in URDF and using laser scanner/3D vision sensor we can generate the world in 3D, which can help to avoid dynamic obstacles rather than static objects defined using URDF.

In the case of the robotic arm, the motion planner should find a trajectory (consisting of joint spaces of each joint) in which the links of the robot should never collide with the environment, avoid self-collision (collision between two robot links), and also not violate the joint limits.

MoveIt! can talk to the motion planners through the plugin interface. We can use **any motion planner** by simply changing the plugin. This method is highly extensible so we can try our **own custom motion** planners using this interface. The `move_group` node talks to the motion planner plugin via the ROS action/services. The default planner for the `move_group` node is OMPL (<http://ompl.kavrakilab.org/>).

To start motion planning, we should **send** a motion planning **request to the motion planner** which specified our planning requirements. The planning requirement may be setting a new goal pose of the end-effector, for example, for a pick and place operation.

We can set additional kinematic constraints for the motion planners. Given next are some inbuilt constraints in MoveIt!:

- **Position constraints:** These restrict the position of a link
- **Orientation constraints:** These restrict the orientation of a link
- **Visibility constraints:** These restrict a point on the link to be visible in a particular area (view of a sensor)
- **Joint constraints:** These restrict a joint within its joint limits
- **User-specified constraints:** Using these constraints, the user can define his own constraints using the callback functions

Using these constraints, we can send a motion planning request and the planner will generate a suitable trajectory according to the request. The **move_group node will generate** the suitable trajectory from the motion planner which **obeys all the constraints**. This can be sent to robot joint trajectory controllers.

Motion planning request adapters

The planning request adapters help to pre-process the motion planning request and post process the motion planning response. One of the uses of pre-processing requests is that it helps to correct if there is a violation in the joints states and, for the post processing, it can convert the path generated by the planner to a time-parameterized trajectory. Following are some of the default planning request adapters in MoveIt!:

- **FixStartStateBounds**: If a joint state is slightly outside the joint limits, then this adapter can fix the initial joint limits within the limits.
- **FixWorkspaceBounds**: This specifies a workspace for planning with a cube size of 10m x 10m x 10m.
- **FixStartStateCollision**: This adapter samples a new collision free configuration if the existing joint configuration is in collision. It makes a new configuration by changing the current configuration by a small factor called `jiggle_factor`.
- **FixStartStatePathConstraints**: This adapter is used when the initial pose of the robot does not obey the path constraints. In this, it finds a near pose which satisfies the path constraints and uses that pose as the initial state.
- **AddTimeParameterization**: This adapter parameterizes the motion plan by applying the velocity and acceleration constraints.

MoveIt! planning scene

The term *planning scene* is used to represent the world around the robot and also store the state of the robot itself. The planning scene monitor inside `move_group` maintains the planning scene representation. The `move_group` node consists of another section called the **world geometry monitor**, which builds the world geometry from the sensors of the robot and from the user input.

The planning scene monitor reads the `joint_states` topic from the robot, and the `sensor information` and world geometry from the world geometry monitor. The world scene monitor reads from the occupancy map monitor, which uses 3D perception to build a 3D representation of the environment, called **Octomap**. The Octomap can be generated from point clouds which are handled by a `point cloud occupancy map` update plugin and depth images handled by a `depth image occupancy map` updater. The following image shows the representation of the planning scene from the MoveIt! official wiki (<http://moveit.ros.org/documentation/concepts/>):

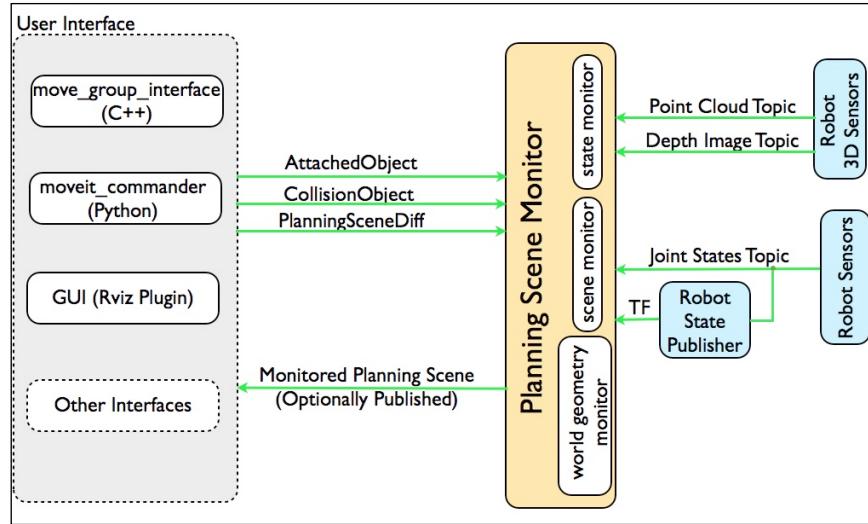


Figure 3 : MoveIt! planning scene overview diagram

MoveIt! kinematics handling

MoveIt! provides a great flexibility to **switch the inverse kinematics algorithms** using the robot plugins. Users can write their own IK solver as a MoveIt! plugin and switch from the default solver plugin whenever required. The default IK solver in MoveIt! is a numerical **jacobian-based solver**.

Compared to the analytic solvers, the numerical solver can take time to solve IK. The package called **IKFast** can be used to generate a C++ code for solving IK using analytical methods, which can be used for different kinds of robot manipulator and perform **better if the DOF is less than 6**. This C++ code can also be converted into the MoveIt! plugin by using some ROS tool. We will look at this procedure in the upcoming chapters.

Forward kinematics and finding jacobians are already integrated to the MoveIt! RobotState class, so we don't need to use plugins for solving FK.

MoveIt! collision checking

The CollisionWorld object inside MoveIt! is used to find collisions inside a *planning scene* which is using the **FCL (Flexible Collision Library)** package as a backend. MoveIt! supports collision checking for different types of objects, such as meshes, primitive shapes such as boxes, cylinders, cones, spheres, and such, and Octomap.

The collision checking is one of the computationally expensive tasks during motion planning. To reduce this computation, MoveIt! provides a matrix called **ACM** (**Allowed Collision Matrix**). It contains a binary value corresponding to the need to check for collision between two pairs of bodies. If the value of matrix is 1, it means collision of the corresponding pair is not needed. We can set the value as 1 where the bodies are always so far that they would never collide with each other. Optimizing ACM can reduce the total computation needed for collision avoidance.

After discussing the basic concepts in MoveIt!, we can now discuss how to interface a robotic arm into MoveIt!. For interfacing a robot arm in MoveIt!, we need to satisfy the components that we saw in Figure 1. The `move_group` node essentially **requires** parameters such as **URDF, SRDF, config files, and joint states topics** along with **TF** from a robot to start with motion planning.

MoveIt! provides a GUI based tool called `Setup Assistant` to generate all these elements. Following is the procedure to generate a MoveIt! configuration from the `Setup Assistant` tool.

Generating MoveIt! configuration package using Setup Assistant tool

The MoveIt! `Setup Assistant` is a graphical user interface for configuring any robot to MoveIt!. Basically, this tool generates SRDF, configuration files, launch files, and scripts generating from the robot URDF model, which is required to configure the `move_group` node.

The SRDF file contains details about the arm joints, end effector joints, virtual joints, and also the collision link pairs which are configured during the MoveIt! configuration process using the `Setup Assistant` tool.

The configuration file contains details about the kinematic solvers, joint limits, controllers, and so on, which are also configured and saved during the configuration process.

Using the generated configuration package of the robot, we can work with motion planning in RViz without the presence of a real robot or simulation interface.

Let's start the configuration wizard, and we can see the step by step procedure to build the configuration package of our robotic arm.

Step 1 – Launching the Setup Assistant tool

- To start the MoveIt! Setup Assistant tool, we can use the following command:


```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```
- This will bring up a window with two choices: **Create New MoveIt! Configuration Package** or **Edit Existing MoveIt! Configuration Package**. Here we are creating a new package, so we need that option. If we have a MoveIt! package already, then we can select the second option.
- Click on the button **Create New MoveIt! Configuration Package**, which will bring a new screen, as shown next:



Figure 4 : MoveIt Setup Assistant

- In this step, the wizard asks for the URDF model of the new robot. To give the URDF file, click on the **Browse** button and navigate to `mastering_ros_robot_description_pkg/urdf/ seven_dof_arm.xacro`. Choose this file and press the **Load** button to load the URDF. We can either give the robot model as pure URDF or xacro, if we give xacro, the tool will convert to URDF internally.

- If the robot model is successfully parsed, we can see the robot model on the window, as shown next:

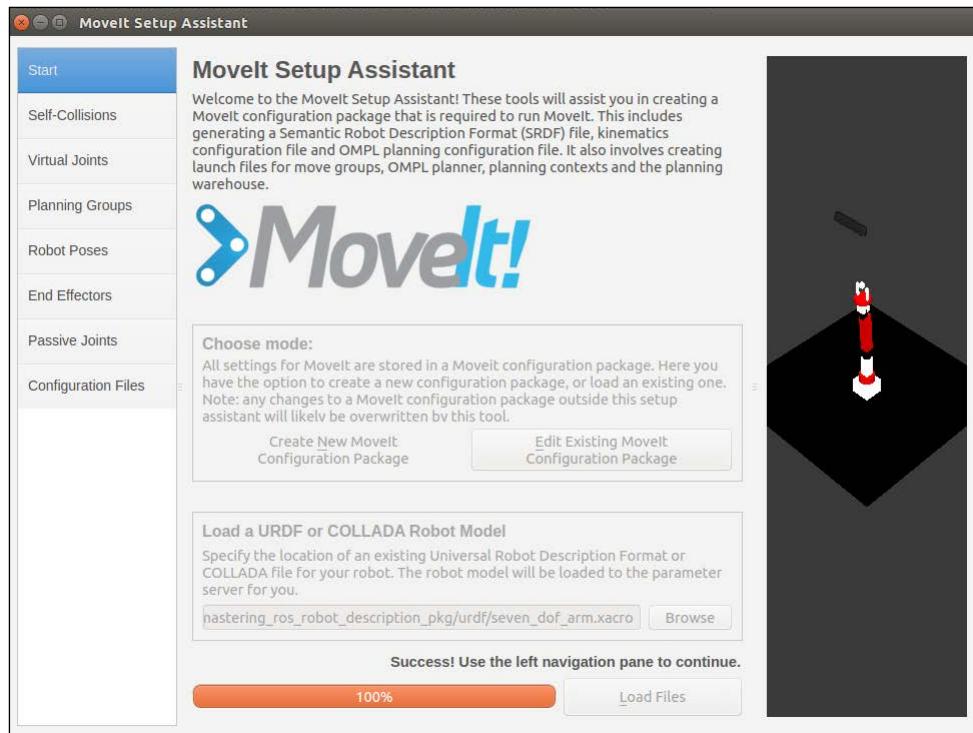


Figure 5 : Successfully parsing the robot model in the Setup Assistant tool

Step 2 – Generating the Self-Collision matrix

- In this step, MoveIt! searches for a pair of links on the robot which can be safely disabled from the collision checking. These can reduce the processing time. This tool analyses each link pair and categorizes the links as always in collision, never in collision, default in collision, adjacent links disabled, and sometimes in collision, and it disables the pair of links which makes any kind of collision. The following image shows the **Self-Collisions** window:

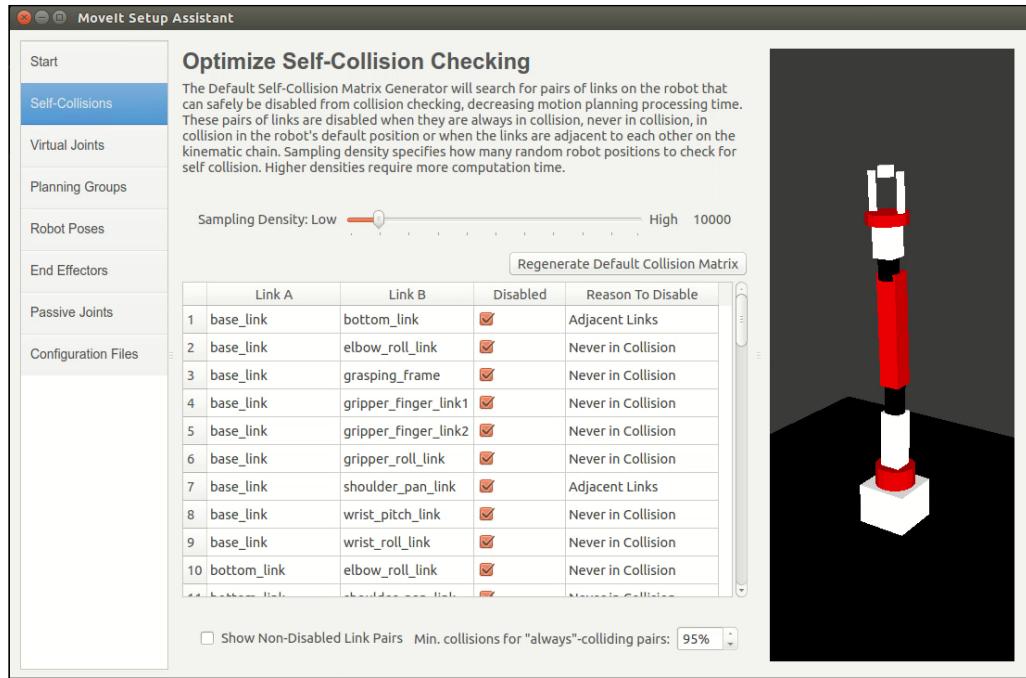


Figure 6 : Regenerating the Self-Collision matrix

- The sampling density is the number of random positions to check for self-collision. If the density is large, computation will be high but self-collision will be less. The default value is 10,000. We can see the disabled pair of links by pressing the **Regenerate Default Collision Matrix** button; it will take a few seconds to list out the disabled pair of links.

Step 3 – Adding virtual joints

- Virtual joints attach the robot to the world. It is not mandatory for a static robot which does not move. We need virtual joints when the base position of the arm is not fixed. For example, if a robot arm is fixed on a mobile robot, we should define a virtual joint with respect to the `odom` frame.
- In the case of our robot, we are not creating virtual joints.

Step 4 – Adding planning groups

- A planning group is basically a group of joints/links in a robotic arm which plans together in order to achieve a goal position of a link or the end effector. We have to create two planning groups, one for the arm and one for the gripper.
- Click on the **Planning Groups** tab on the left side and click on the **Add Group** button. You will see the following screen, which has the settings of the `arm` group:

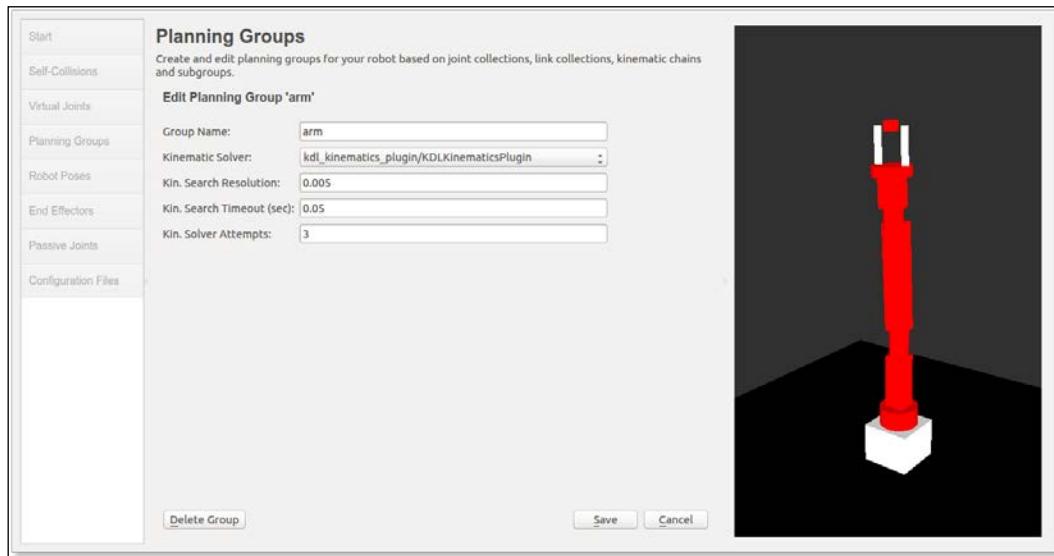


Figure 7 : Adding the planning group of the arm

- Here we are giving **Group Name** as `arm`, and **Kinematic Solver** as `kdl_kinematics_plugin/KDLKinematicsPlugin`, which is the default numerical IK solver with MoveIt!. We can keep the other parameters as the default values.
- Inside the `arm` group, first we have to add a kinematic Chain. We have to add `base_link` as the first link to `grasping_frame`.

- Add a group called `gripper` and we don't need to have a kinematic solver for the `gripper` group. Inside this group, we can add the joints and links of the gripper. These settings are shown next:

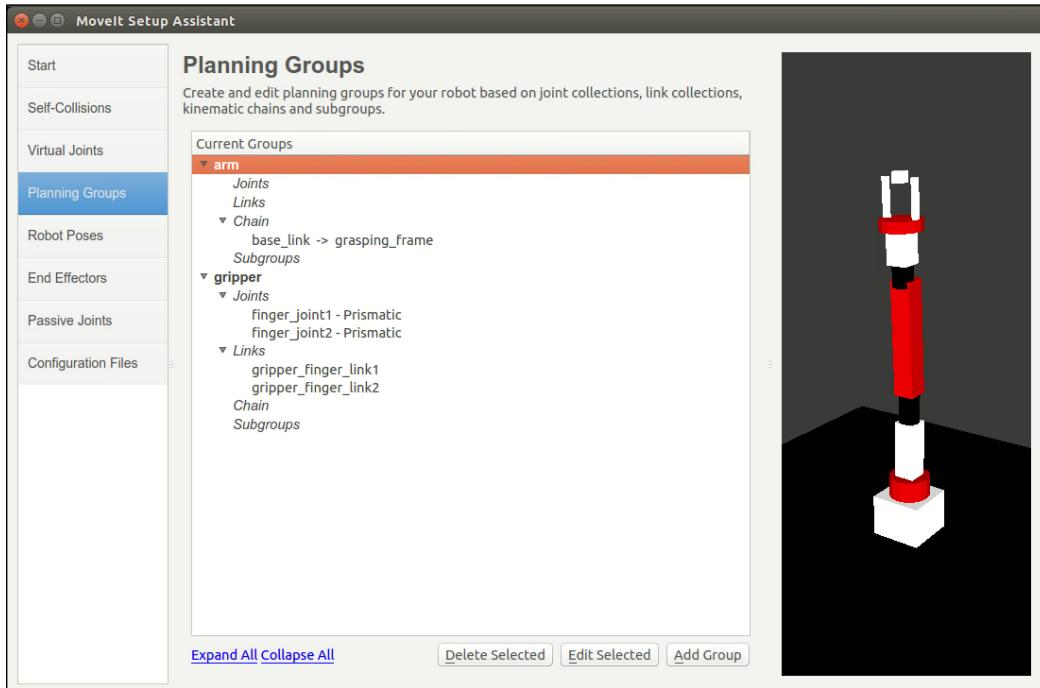


Figure 8 : Adding the planning group of the arm and gripper

Step 5 – Adding the robot poses

- In this step, we can add certain fixed poses in the robot configuration. For example, we can assign a home position or a pick/place position in this step. The advantage is that while programming with MoveIt! APIs, we can directly call these poses, which are also called group states. This has many applications in the pick/place and grasping operation. The robot can switch to the fixed poses without any hassle.

Step 6 – Setup the robot end effector

- In this step, we name the robot end effector and assign the end effector group, the parent link, and the parent group.
- We can add any number of end effectors to this robot. In our case, it's a gripper designed for pick and place operation.
- Click on the **Add End Effector** button and name the end effector as `robot_eef`, planning group as `gripper` which we have already created, parent link as `grasping_frame`, and parent group as `arm`.

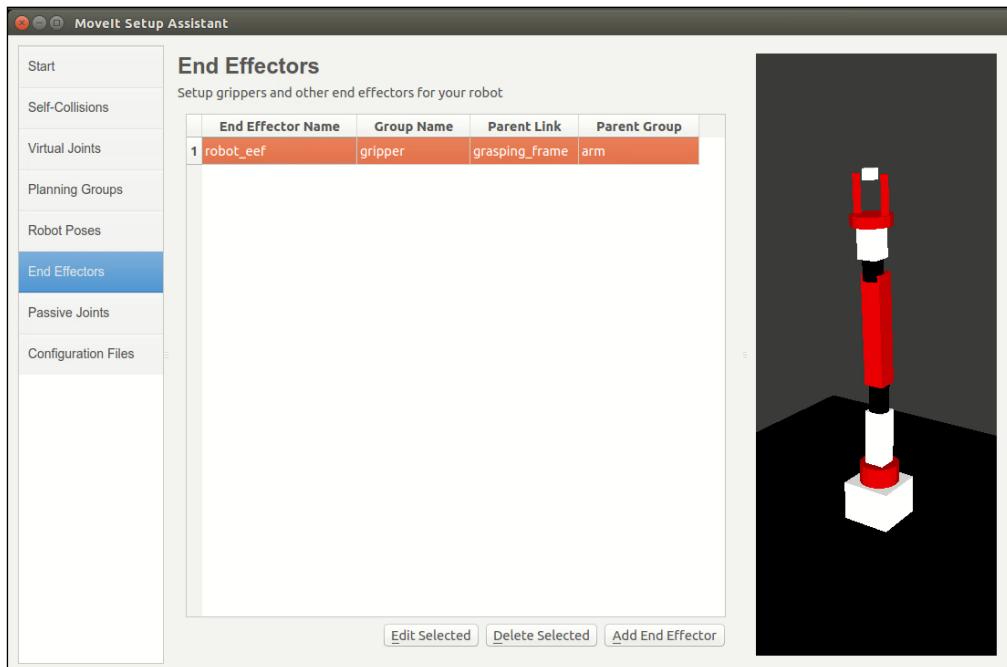


Figure 9 : Adding end effectors

Step 7 – Adding passive joints

- In this step, we can specify the passive joints in the robot. Passive joints mean that the joints do not have any actuators. Caster wheels are one of the examples of passive joints. The planner will ignore these kind of joints during motion planning.

Step 8 – Generating configuration files

- We are almost done!! We are in the final stage, that is, generating the configuration files. In this step, the tool will generate a configuration package which contains the file needed to interface MoveIt!.
- Click on the **Browse** button to locate a folder to save the configuration file that is going to be generated by the **Setup Assistant** tool. Here we can see the files are generating inside a folder called `seven_dof_config`. You can add `_config` or `_generated` along with the robot name for the configuration package.
- Click on the **Generate Package** button, and it will generate the files to the given folder.
- If the process is successful, we can click on **Exit Setup Assistant**, which will exit us from the tool.
- Following is the screenshot of the generation process:

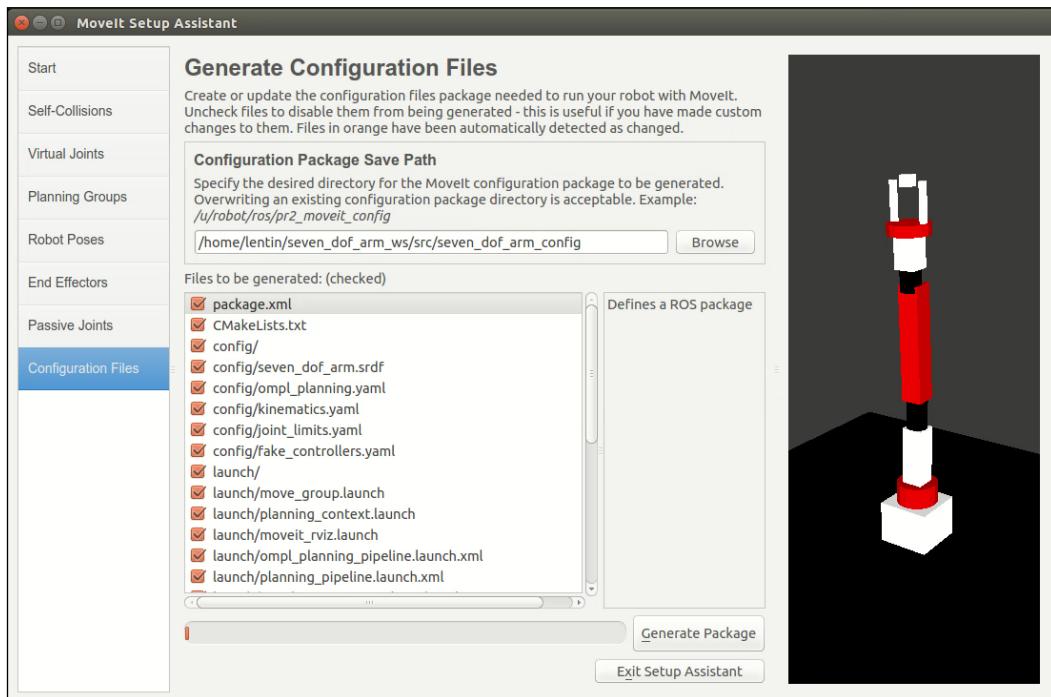


Figure 10 : Generating the MoveIt! configuration package

After generating the MoveIt! configuration package, we can copy it into our `catkin` workspace and build it using the `catkin_make` command. In the following section, we are going to work with this package.

Motion planning of robot in RViz using MoveIt! configuration package

MoveIt! provides a plugin for RViz which allows it to create new planning scenes where **robot works, generate motion plans, add new objects**, visualize the planning output and can directly interact with the visualized robot.

The MoveIt! configuration package consists of configuration files and launch files to start motion planning in RViz. There is a demo launch file in the package to explore all the functionalities of this package.

Following is the command to invoke the demo launch file:

```
$ rosrun seven_dof_arm_config demo.launch
```

If everything works fine, we will get the following screen of RViz being loaded with the MotionPlanning plugin provided by MoveIt!:

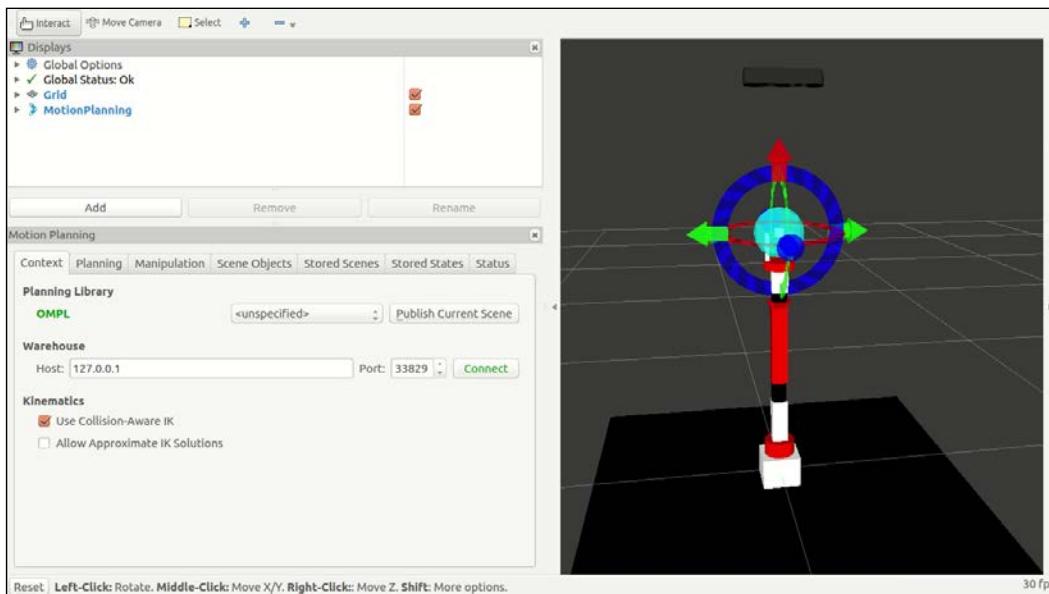


Figure 11 : MoveIt! - RViz motion planning interface

Using the RViz MotionPlanning plugin

From the preceding Figure 11, we can see that the RViz-Motion Planning plugin is loaded on the left side of the screen. There are several tabs on the **Motion Planning** window, such as **Context**, **Planning**, and so on. The default tab is the **Context** tab and we can see the default **Planning Library** as OMPL, which is shown in green. It indicates that MoveIt! successfully loaded the motion planning library. If it is not loaded, we can't perform motion planning.

Next is the **Planning** tab. This is one of the frequently used tabs used to assign the **Start State**, **Goal State Plan** a path, and execute the path. Shown next is the GUI of the **Planning** tab:

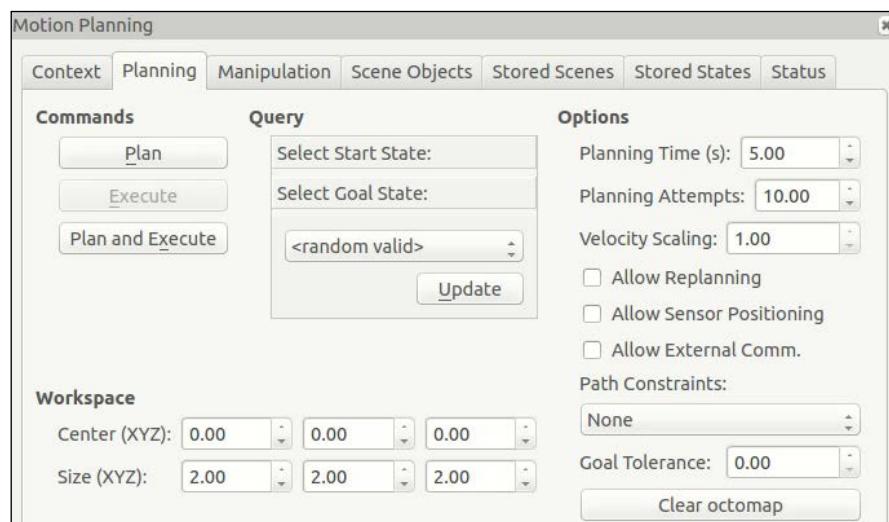


Figure 12 : MoveIt! -RViz Planning tab

We can assign the start state and the goal state of the robot under the **Query** panel. Using the **Plan** button, we can plan the path from the start to the goal state, and if the planning is successful, we can execute it. By default, execution is done on fake controllers. We can change these controllers into trajectory controllers for executing the planned trajectory in Gazebo or the real robot.

We can set the starting and the goal position of the robot end effector by using the interactive marker attached on the arm gripper. We can translate and rotate the marker pose, and if there is a planning solution, we can see an arm in orange color. In some situations, the arm will not move even the end effector marker pose moves, and if the arm does not come to the marker position, we can assume that there is no IK solution in that pose. We may need more DOF to reach there or there might be some collision between the links.

Following are the screenshots of a valid goal pose and an invalid goal pose:

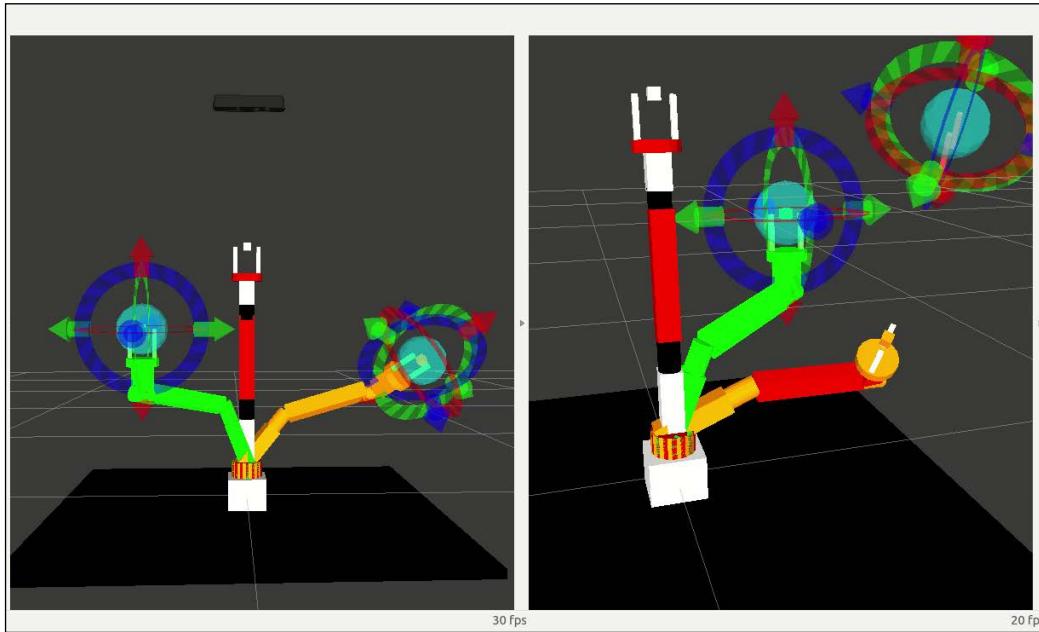


Figure 13 : A valid pose and an invalid pose of the robot in RViz

The green color arm represents the starting position of the arm, and the orange color represents the goal position. In the first figure, if we press the **Plan** button, MoveIt! plans a path from start to goal. In the second image, we can observe two things. First, one of the links of the orange arm is red which means that the goal pose is in a self-collided state. Secondly, look at the end effector marker; it is far from the actual end effector and it has also turned red.

We can also work with some quick motion planning using random valid options in the **Start State** and the **Goal State**. If we select the goal state as random valid and press the **Update** button, it will generate a random valid goal pose. Click on the **Plan** button and we can see the motion planning.

We can customize the RViz visualization using the various options in the MotionPlanning plugin. Shown next are some of the options of this plugin:

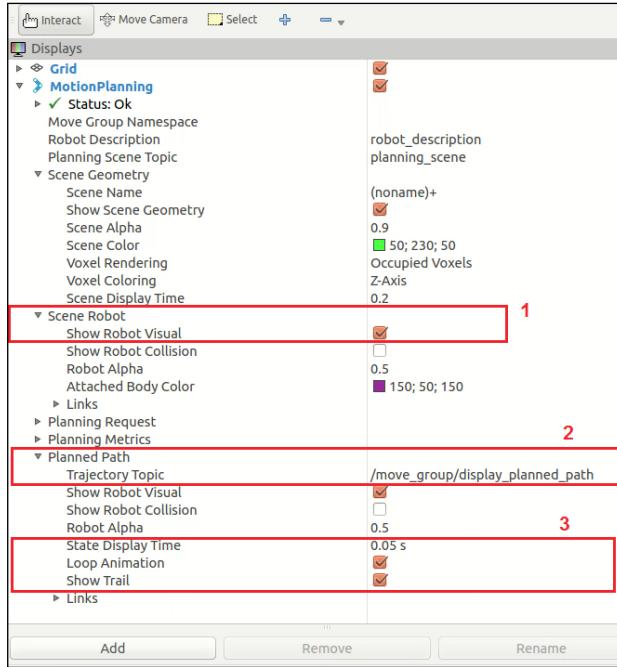


Figure 14 : Settings of the MotionPlanning plugin on RViz

- The first marked area is **Scene Robot** which will show the robot model; if it is unchecked, we won't see any robot model
- The second marked area is the **Trajectory Topic**, in which RViz gets the visualization trajectory
- If we want to animate the motion planning and want to display the motion trails, we should enable this option

One of the other sections in the plugin settings is shown in the following image:



Figure 15 : Planning Request setting in MotionPlanning plugin

In the preceding figure, we can see the **Query Start State** and the **Query Goal State** options. These options can visualize the start pose and the goal pose of the arm which we saw in Figure 13. **Show Workspace** visualizes the cubic workspace (world geometry) around the robot. The visualization can help to debug our motion planning algorithm and understand the robot motion behavior in detail.

In the next section, we will see how to interface the MoveIt! configuration package to Gazebo. This will execute the trajectory generated by MoveIt! in Gazebo.

Interfacing the MoveIt! configuration package to Gazebo

We have already worked with the Gazebo simulation of this arm and attached controllers to it. For interfacing the arm in MoveIt! to Gazebo, we need a trajectory controller which has the `FollowJointTrajectoryAction` interface, as we mentioned in the MoveIt! architecture.

Following is the procedure to interface MoveIt! to Gazebo:

Step 1 – Writing the controller configuration file for MoveIt!

The first step is to create a configuration file for talking with the trajectory controllers in Gazebo from MoveIt!. The controller configuration file called `controllers.yaml` has to be created inside the `config` folder of the `seven_dof_arm_config` package.

Locate the file `controllers.yaml` from the `chapter_4_codes/seven_dof_arm_config/config` folder for getting the controller definition.

Given next is the `controllers.yaml` definition:

```
controller_manager_ns: controller_manager
controller_list:
  - name: seven_dof_arm/seven_dof_arm_joint_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: true
    joints:
      - shoulder_pan_joint
      - shoulder_pitch_joint
      - elbow_roll_joint
      - elbow_pitch_joint
      - wrist_roll_joint
```

```
- wrist_pitch_joint  
- gripper_roll_joint  
  
- name: seven_dof_arm/gripper_controller  
  action_ns: follow_joint_trajectory  
  type: FollowJointTrajectory  
  default: true  
  joints:  
    - finger_joint1  
    - finger_joint2
```

The controller configuration file contains the definition of the two controller interfaces; one is for `arm` and the other is for `gripper`. The type of action used in the controllers is `FollowJointTrajectory`, and the action namespace is `follow_joint_trajectory`. We have to list out the joints under each group. The `default: true` indicates that it will use the default controller, which is the primary controller in MoveIt! for communicating with the set of joints.

Step 2 – Creating the controller launch files

Next, we have to create a new launch file called `seven_dof_arm_moveit_controller_manager.launch` which can start the trajectory controllers. The name of the file starts with the robot name, which is added with `_moveit_controller_manager`.

Locate the `chapter_4_codes/seven_dof_arm_config/launch` folder for getting this file.

Following is the launch file definition:

```
<launch>  
  <!-- Set the param that trajectory_execution_manager needs to find  
      the controller plugin -->  
  <arg name="moveit_controller_manager" default="moveit_simple_  
        controller_manager/MoveItSimpleControllerManager" />  
  <param name="moveit_controller_manager" value="$(arg moveit_  
        controller_manager)"/>  
  
  <!-- load controller_list -->  
  <arg name="use_controller_manager" default="true" />  
  <param name="use_controller_manager" value="$(arg use_controller_  
        manager)" />
```

```
<!-- Load joint controller configurations from YAML file to
parameter server -->
<rosparam file="$(find seven_dof_arm_config)/config/controllers.
yaml"/>
</launch>
```

This launch file starts the `MoveItSimpleControllerManager` and loads the joint trajectory controllers defined inside `controllers.yaml`.

Step 3 – Creating the controller configuration file for Gazebo

After creating the MoveIt! files, we have to create the Gazebo controller configuration file and the launch file.

Create a new file called `trajectory_control.yaml` which contains the list of the Gazebo ROS controllers that need to be loaded along with Gazebo.

You will get this file from the `chapter_4_codes/seven_dof_arm_gazebo/config` folder.

Following is the definition of this file:

```
seven_dof_arm:
  seven_dof_arm_joint_controller:
    type: "position_controllers/JointTrajectoryController"
    joints:
      - shoulder_pan_joint
      - shoulder_pitch_joint
      - elbow_roll_joint
      - elbow_pitch_joint
      - wrist_roll_joint
      - wrist_pitch_joint
      - gripper_roll_joint

    gains:
      shoulder_pan_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      shoulder_pitch_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      elbow_roll_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      elbow_pitch_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      wrist_roll_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      wrist_pitch_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
```

```

gripper_roll_joint:      {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}

gripper_controller:
  type: "position_controllers/JointTrajectoryController"
  joints:
    - finger_joint1
    - finger_joint2
  gains:
    finger_joint1: {p: 50.0, d: 1.0, i: 0.01, i_clamp: 1.0}
    finger_joint2: {p: 50.0, d: 1.0, i: 0.01, i_clamp: 1.0}

```

Here we created a `position_controllers/JointTrajectoryController` which has an action interface of `FollowJointTrajectory` for both the arm and the gripper. We also defined the PID gain associated with each joint which can provide smooth motion.

Step 4 – Creating the launch file for Gazebo trajectory controllers

After creating the configuration file, we can load the controllers along with Gazebo. We have to create a launch file which launches Gazebo, the trajectory controllers, and the MoveIt! interface in a single command.

The launch file `seven_dof_arm_bringup_moveit.launch` contains the definition to launch all these commands:

```

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

  <!-- ros_control seven dof arm launch file -->
  <include file="$(find seven_dof_arm_gazebo)/launch/seven_dof_arm_
gazebo_states.launch" />

  <!-- ros_control position control dof arm launch file -->
  <!--<include file="$(find seven_dof_arm_gazebo)/launch/seven_dof_
arm_gazebo_position.launch" /> -->

  <!-- ros_control trajectory control dof arm launch file -->
  <include file="$(find seven_dof_arm_gazebo)/launch/seven_dof_arm_
trajectory_controller.launch" />

```

Using the ROS MoveIt! and Navigation Stack

```
<!-- moveit launch file -->
<include file="$(find seven_dof_arm_config)/launch/moveit_planning_
execution.launch" />

</launch>
```

This launch file spawns the robot model in Gazebo, publishes the joint states, attaches the position controller, attaches the trajectory controller, and at last launches `moveit_planning_execution.launch` inside the MoveIt! package for starting the MoveIt! nodes along with RViz. We may need to load the MotionPlanning plugin in RViz if it is not loaded by default.

We can start motion planning inside RViz and execute in Gazebo using the following single command:

```
$ rosrun seven_dof_arm_config seven_dof_arm_bringup_moveit.launch
```

This will launch RViz and Gazebo, and we can do motion planning inside RViz. After motion planning, click on the **Execute** button to send the trajectory to the Gazebo controllers.

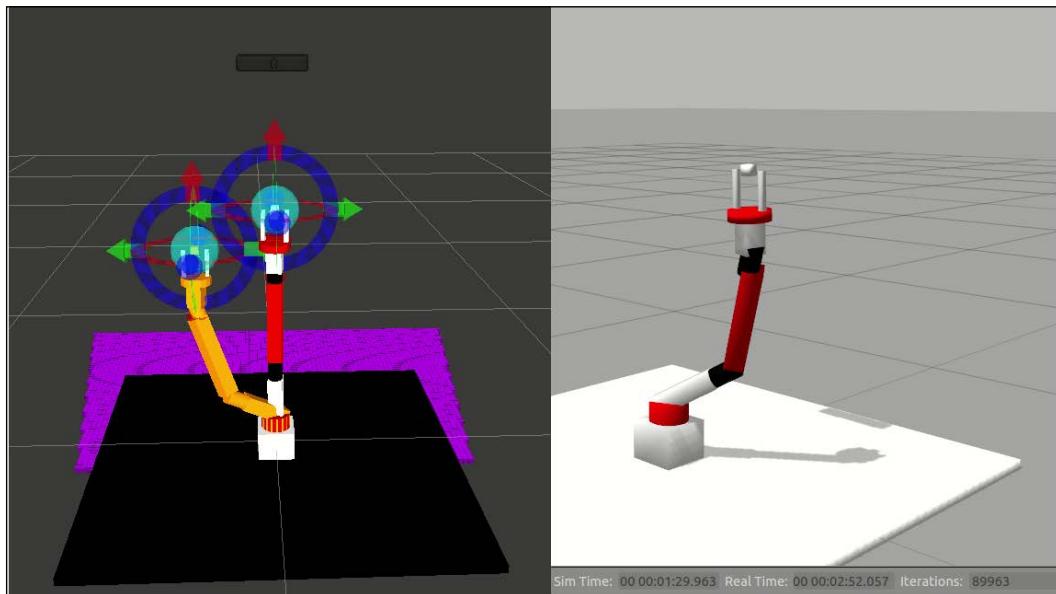


Figure 16 : Gazebo trajectory controllers executing the trajectory from MoveIt!

Step 5 – Debugging the Gazebo- MoveIt! interface

In this section, we will discuss some of the common issues and debugging techniques in this interface.

If the trajectory is not executing on Gazebo first list the topics.

```
$ rostopic list
```

If the Gazebo controllers are started properly, we will get the following joint trajectory topics in the list:

```
/seven_dof_arm/gripper_controller/command
/seven_dof_arm/gripper_controller/follow_joint_trajectory/cancel
/seven_dof_arm/gripper_controller/follow_joint_trajectory/feedback
/seven_dof_arm/gripper_controller/follow_joint_trajectory/goal
/seven_dof_arm/gripper_controller/follow_joint_trajectory/result
/seven_dof_arm/gripper_controller/follow_joint_trajectory/status
/seven_dof_arm/gripper_controller/state
/seven_dof_arm/joint_states
/seven_dof_arm/seven_dof_arm_joint_controller/command
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/cancel
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/feedback
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/goal
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/result
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/status
/seven_dof_arm/seven_dof_arm_joint_controller/state
/tf
/tf_static
/trajectory_execution_event
```

Figure 17 : Topics from the Gazebo-ROS trajectory controllers

We can see `follow_joint_trajectory` for the gripper and the `arm` group. If the controllers are not ready, the trajectory will not execute in Gazebo.

Also check the terminal message while starting the launch file.

```
[ INFO] [1436443220.700537882, 16.009000000]: Using planning request adapter 'Fi  
x Start State Bounds'  
[ INFO] [1436443220.704012811, 16.009000000]: Using planning request adapter 'Fi  
x Start State In Collision'  
[ INFO] [1436443220.705014564, 16.009000000]: Using planning request adapter 'Fi  
x Start State Path Constraints'  
[ INFO] [1436443221.637662714, 16.250000000]: MoveitSimpleControllerManager: Add  
d FollowJointTrajectory controller for seven_dof_arm/seven_dof_arm_joint_controller  
[ INFO] [1436443221.637879744, 16.250000000]: Returned 1 controllers in list  
[ INFO] [1436443221.735805220, 16.265000000]: Trajectory execution is managing c  
ntrollers  
Loading 'move_group/MoveGroupCartesianPathService'...  
Loading 'move_group/MoveGroupExecuteService'...  
Loading 'move_group/MoveGroupKinematicsService'...  
Loading 'move_group/MoveGroupMoveAction'...  
Loading 'move_group/MoveGroupPickPlaceAction'...  
Loading 'move_group/MoveGroupPlanService'...  
Loading 'move_group/MoveGroupQueryPlannersService'...  
Loading 'move_group/MoveGroupStateValidationService'...  
Loading 'move_group/MoveGroupGetPlanningSceneService'...  
Loading 'move_group/ClearOctomapService'...  
[ INFO] [1436443223.371378749, 16.710000000]:  
[ INFO] [1445951681.529080293, 32.872000000]: LBKPIECE1: Created 148 (28 start +  
120 goal) states in 132 cells (27 start (27 on boundary) + 105 goal (105 on bou  
ndary)  
[ INFO] [1445951681.686798180, 32.934000000]: LBKPIECE1: Created 173 (77 start +  
96 goal) states in 156 cells (75 start (75 on boundary) + 81 goal (81 on bounda  
ry))  
[ INFO] [1445951683.584298242, 34.156000000]: ParallelPlan::solve(): Solution fo  
und by one or more threads in 2.387348 seconds
```

1

2

Figure 18 : The terminal message shows successful trajectory execution

In the preceding image, the first section shows that the MoveItSimpleControllerManager was able to connect with the Gazebo controller and if it couldn't connect to controller, it shows that it can't connect to the controller. The second section shows a successful motion planning. If the motion planning is not successful, MoveIt! will not send the trajectory to Gazebo.

In the next section, we will discuss the ROS Navigation stack and look at the requirements needed to interface the Navigation stack to the Gazebo simulation.

Understanding ROS Navigation stack

The main aim of the ROS navigation package is to move a robot from the start position to the goal position, without making any collision with the environment. The ROS Navigation package comes with an implementation of several navigation related algorithms which can easily help implement autonomous navigation in the mobile robots.

The user only needs to feed the goal position of the robot and the robot odometry data from sensors such as wheel encoders, IMU, and GPS, along with other sensor data streams such as laser scanner data or 3D point cloud from sensors like **Kinect**. The output of the Navigation package will be the velocity commands which will drive the robot to the given goal position.

The Navigation stack contains implementation of the standard algorithms, such as SLAM, A *(star), Dijkstra, AMCL, and so on, which can directly be used in our application.

ROS Navigation hardware requirements

The ROS navigation stack is designed as generic. There are some hardware requirements that should be satisfied by the robot. Following are the requirements:

- The Navigation package will work better in differential drive and holonomic (total DOF of robot equals to controllable DOF of robots). Also, the mobile robot should be controlled by sending velocity commands in the form : x: velocity, y: velocity (linear velocity), and theta :velocity (angular velocity).
- The robot should mount a planar laser somewhere around the robot. It is used to build the map of the environment.
- The Navigation stack will perform better for square and circular shaped mobile bases. It will work on an arbitrary shape but performance is not guaranteed.

Following are the basic building blocks of the Navigational stack taken from ROS website (<http://wiki.ros.org/navigation/Tutorials/RobotSetup>). We can see what are the purposes of each block and how to configure the Navigation stack for a custom robot.

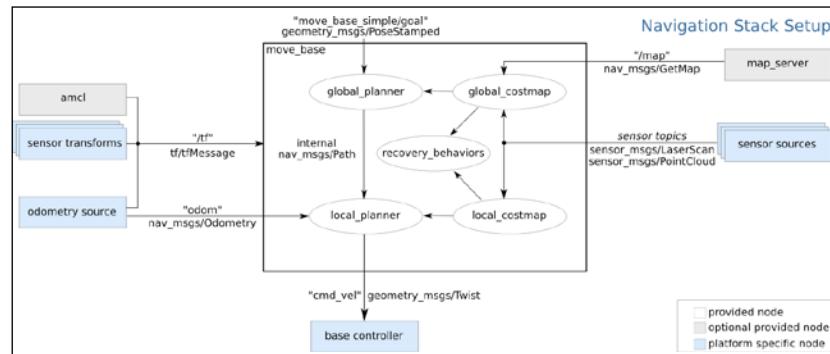


Figure 19 : Navigation stack setup diagram

According to the Navigation setup diagram, for configuring the Navigation package for a custom robot, we must provide functional blocks which are interface to the Navigation stack. Following are the explanations of all the blocks which are provided as input to the Navigational stack:

- **Odometry source:** Odometry data of a robot gives the **robot position with respect to its starting position**. Main odometry sources are wheel encoders, IMU, and 2D/3D cameras (visual odometry). The odom value should **publish to the Navigation stack**, which has a message type of `nav_msgs/Odometry`. The `odom` message can hold the position and the velocity of the robot. Odometry data is a mandatory input to the Navigational stack.
- **Sensor source:** We have to provide laser scan data or **point cloud data** to the Navigation stack for mapping the robot environment. This data, along with odometry, combines to **build the global and local cost map** of the robot. The main sensors used here are Laser Range finders or Kinect 3D sensors. The data should be of type `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud`.
- **sensor transforms/tf:** The robot should publish the relationship between the **robot coordinate frame** using ROS tf.
- **base_controller:** The main function of the base controller is to **convert the output of the Navigation stack**, which is a twist (`geometry_msgs/Twist`) message, and convert it into corresponding motor velocities of the robot.

The optional nodes of the Navigation stack are `amcl` and map server, which allow localization of the robot and help to save/load the robot map.

Working with Navigation packages

Before working with the Navigation stack, we were discussing MoveIt! and the `move_group` node. In the Navigation stack also, there is a node similar to the `move_group` node, called the **move_base node**. From Figure 19, it is clear that the `move_base` node takes input from sensors, joint states, tf, and odometry, which is very similar to the `move_group` node that we saw in MoveIt!.

Let's see more about the `move_base` node.

Understanding the move_base node

The `move_base` node is from a package called `move_base`. The main function of this package is to move a robot from its current position to a goal position with the help of other Navigation nodes. The `move_base` node inside this package links the global-planner and the local-planner for the path planning, connecting to the rotate-recovery package if the robot is stuck in some obstacle and connecting global costmap and local costmap for getting the map.

The `move_base` node basically is an implementation of `SimpleActionServer` which takes a goal pose with message type (`geometry_msgs/PoseStamped`). We can send a goal position to this node using a `SimpleActionClient` node.

The `move_base` node subscribes the goal from a topic called `move_base_simple/goal`, which is the input of the Navigation stack, as shown in the previous diagram.

When this node receives a goal pose, it links to components such as `global_planner`, `local_planner`, `recovery_behavior`, `global_costmap`, and `local_costmap`, generates the output which is the command velocity (`geometry_msgs/Twist`), and sends to the base controller for moving the robot for achieving the goal pose.

Following is the list of all the packages which are linked by the `move_base` node:

- **global-planner:** This package provides libraries and nodes for planning the optimum path from the current position of the robot to the goal position, with respect to the robot map. This package has implementation of path finding algorithms such as A*, Dijkstra, and so on for finding the shortest path from the current robot position to the goal position.
- **local-planner:** The main function of this package is to navigate the robot in a section of the global path planned using the global planner. The local planner will take the odometry and sensor reading, and send an appropriate velocity command to the robot controller for completing a segment of the global path plan. The base local planner package is the implementation of the trajectory rollout and dynamic window algorithms.
- **rotate-recovery:** This package helps the robot to recover from a local obstacle by performing a 360 degree rotation.
- **clear-costmap-recovery:** This package is also for recovering from a local obstacle by clearing the costmap by reverting the current costmap used by the Navigation stack to the static map.

- **costmap-2D:** The main use of this package is to map the robot environment. Robot can only plan a path with respect to a map. In ROS, we create 2D or 3D occupancy grid maps, which is a representation of the environment in a grid of cells. Each cell has a probability value which indicates whether the cell is occupied or not. The costmap-2D package can build the grid map of the environment by subscribing sensor values of the laser scan or point cloud and also the odometry values. There are global cost maps for global navigation and local cost maps for local navigations.

Following are the other packages which are interfaced to the move_base node:

- **map-server:** Map server package allows us to save and load the map generated by the costmap-2D package.
- **AMCL:** AMCL is a method to localize the robot in map. This approach uses particle filter to track the pose of the robot with respect to the map, with the help of probability theory. In the ROS system, AMCL can only work with maps which were built using laser scans.
- **gmapping:** The gmapping package is an implementation of an algorithm called Fast SLAM which takes the laser scan data and odometry to build a 2D occupancy grid map.

After discussing each functional block of the Navigation stack, let's see how it really works.

Working of Navigation stack

In the previous section, we saw the functionalities of each block in the ROS Navigation stack. Let's check how the entire system works. The robot should publish proper odometry value, τ_f information, and sensor data from the laser, and have a base controller and map of the surrounding.

If all these requirements are satisfied, we can start working with the Navigation package.

Localizing on the map

The first step the robot is going to perform is localizing itself on the map. The AMCL package will help to localize the robot on the map.

Sending a goal and path planning

After getting the current position of the robot, we can send a goal position to the move_base node. The move_base node will send this goal position to a global planner which will plan a path from the current robot position to the goal position.

This plan is with respect to the global costmap which is feeding from the map server. The global planner will send this path to the local planner, which executes each segment of the global plan.

The local planner gets the odometry and the sensor value from the move_base node and finds a collision free local plan for the robot. The local planner is associated with the local costmap, which can monitor the obstacle(s) around the robot.

Collision recovery behavior

The global and local costmap are tied with the laser scan data. If the robot is stuck somewhere, the Navigation package will trigger the recovery behavior nodes, such as the clear costmap recovery or rotate recovery nodes.

Sending the command velocity

The local planner generates command velocity in the form of a twist message which contains linear and angular velocity (geometry_msgs/Twist), to the robot base controller. The robot base controller converts the twist message to the equivalent motor speed.

Installing ROS Navigation stack

Installing ROS desktop full installation will not install the ROS Navigation stack. We have to install the Navigation stack separately, using the following commands:

- In ROS Jade

```
$ sudo apt-get install ros-jade-navigation
```

- In ROS Indigo

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

After installing the Navigation package, let's start learning how to build a map of the robot environment. The robot we are using here is the differential wheeled robot that we discussed in the previous chapter. This robot satisfies all the three requirements of the Navigation stack.

Building a map using SLAM

The ROS Gmapping package is a wrapper of open source implementation of SLAM called OpenSLAM (<https://www.openslam.org/gmapping.html>). The package contains a node called `slam_gmapping`, which is the implementation of SLAM which helps to create a 2D occupancy grid map from the laser scan data and the mobile robot pose.

The basic hardware requirement for doing SLAM is a laser scanner which is horizontally mounted on the top of the robot, and the robot odometry data. In this robot, we have already satisfied these requirements. We can generate the 2D map of the environment using the `gmapping` package through the following procedure.

Creating a launch file for gmapping

The main task while creating a launch file for the `gmapping` process is to set the parameters for the `slam_gmapping` node and the `move_base` node. The `slam_gmapping` node is the core node inside the ROS `gmapping` package. The `slam_gmapping` node subscribes the laser data (`sensor_msgs/LaserScan`) and the `tf` data, and publishes the occupancy grid map data as output (`nav_msgs/OccupancyGrid`). This node is highly configurable and we can fine tune the parameters to improve the mapping accuracy. The parameters are mentioned at <http://wiki.ros.org/gmapping>.

The next node we have to configure is the `move_base` node. The main parameters needed to configure are the global and local costmap parameters, the local planner, and the `move_base` parameters. The parameters list is very lengthy. We are representing these parameters in several YAML files. Each parameter is included in the `param` folder inside the `diff_wheeled_robot_gazebo` package.

Following is the `gmapping.launch` file used in this robot. The launch file is placed in the `diff_wheeled_robot_gazebo/launch` folder.

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

  <!-- Defining parameters for slam_gmapping node -->

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
        output="screen">
    <param name="base_frame" value="base_footprint"/>
    <param name="odom_frame" value="odom"/>
    <param name="map_update_interval" value="5.0"/>
    <param name="maxUrange" value="6.0"/>
    <param name="maxRange" value="8.0"/>
```

```

<param name="sigma" value="0.05"/>
<param name="kernelSize" value="1"/>
<param name="lstep" value="0.05"/>
<param name="astep" value="0.05"/>
<param name="iterations" value="5"/>
<param name="lsigma" value="0.075"/>
<param name="ogain" value="3.0"/>
<param name="lskip" value="0"/>
<param name="minimumScore" value="100"/>
<param name="srr" value="0.01"/>
<param name="srt" value="0.02"/>
<param name="str" value="0.01"/>
<param name="stt" value="0.02"/>
<param name="linearUpdate" value="0.5"/>
<param name="angularUpdate" value="0.436"/>
<param name="temporalUpdate" value="-1.0"/>
<param name="resampleThreshold" value="0.5"/>
<param name="particles" value="80"/>

<param name="xmin" value="-1.0"/>
<param name="ymin" value="-1.0"/>
<param name="xmax" value="1.0"/>
<param name="ymax" value="1.0"/>

<param name="delta" value="0.05"/>
<param name="llsamplerange" value="0.01"/>
<param name="llsamplestep" value="0.01"/>
<param name="lasamplerange" value="0.005"/>
<param name="lasamplestep" value="0.005"/>
<remap from="scan" to="$(arg scan_topic)"/>
</node>

<!-- Defining parameters for move_base node -->

<node pkg="move_base" type="move_base" respawn="false" name="move_
base" output="screen">
    <rosparam file="$(find diff_wheeled_robot_gazebo)/param/costmap_
common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find diff_wheeled_robot_gazebo)/param/costmap_
common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find diff_wheeled_robot_gazebo)/param/local_
costmap_params.yaml" command="load" />
    <rosparam file="$(find diff_wheeled_robot_gazebo)/param/global_
costmap_params.yaml" command="load" />

```

```
<rosparam file="$(find diff_wheeled_robot_gazebo)/param/base_
local_planner_params.yaml" command="load" />
<rosparam file="$(find diff_wheeled_robot_gazebo)/param/dwa_local_
planner_params.yaml" command="load" />
<rosparam file="$(find diff_wheeled_robot_gazebo)/param/move_base_
params.yaml" command="load" />

</node>

</launch>
```

Running SLAM on the differential drive robot

We can build the ROS package called `diff_wheeled_robot_gazebo` and can run the `gmapping.launch` file for building the map. Following are the commands to start with the mapping procedure.

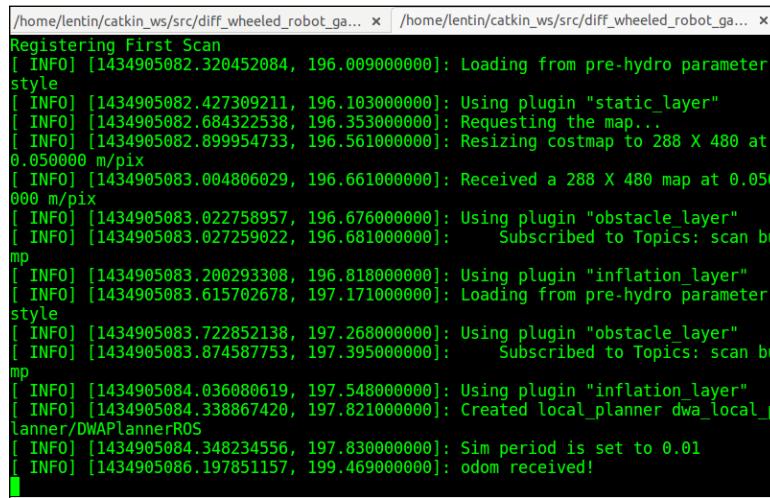
Start the robot simulation by using Willow Garage world:

```
$ rosrun diff_wheeled_robot_gazebo diff_wheeled_gazebo_full.launch
```

Start the gmapping launch file by using the following command:

```
$ rosrun diff_wheeled_robot_gazebo gmapping.launch
```

If the gmapping launch file is working fine, we will get the following kind of output on the terminal:



The image shows a terminal window with a black background and white text. It displays log messages from the gmapping process. The messages include information about registering the first scan, loading parameters, requesting the map, resizing costmaps, receiving a map, and subscribing to topics like scan. It also shows the creation of local planners and DWAPlanners, setting the sim period to 0.01, and receiving odometry data. The text is in a monospaced font and is mostly in green and white colors.

```
/home/lentin/catkin_ws/src/diff_wheeled_robot_ga... x | /home/lentin/catkin_ws/src/diff_wheeled_robot_ga... x
Registering First Scan
[ INFO] [1434905082.320452084, 196.009000000]: Loading from pre-hydro parameter
style
[ INFO] [1434905082.427309211, 196.103000000]: Using plugin "static_layer"
[ INFO] [1434905082.684322538, 196.353000000]: Requesting the map...
[ INFO] [1434905082.899954733, 196.561000000]: Resizing costmap to 288 X 480 at
0.050000 m/pix
[ INFO] [1434905083.004806029, 196.661000000]: Received a 288 X 480 map at 0.050
000 m/pix
[ INFO] [1434905083.022758957, 196.676000000]: Using plugin "obstacle_layer"
[ INFO] [1434905083.027259022, 196.681000000]: Subscribed to Topics: scan bu
mp
[ INFO] [1434905083.200293308, 196.818000000]: Using plugin "inflation_layer"
[ INFO] [1434905083.615702678, 197.171000000]: Loading from pre-hydro parameter
style
[ INFO] [1434905083.722852138, 197.268000000]: Using plugin "obstacle_layer"
[ INFO] [1434905083.874587753, 197.395000000]: Subscribed to Topics: scan bu
mp
[ INFO] [1434905084.036080619, 197.548000000]: Using plugin "inflation_layer"
[ INFO] [1434905084.338867420, 197.821000000]: Created local_planner dwa_local_p
lanner/DWAPlannerROS
[ INFO] [1434905084.348234556, 197.830000000]: Sim period is set to 0.01
[ INFO] [1434905086.197851157, 199.469000000]: odom received!
```

Figure 21 : Terminal messages during gmapping

Start the keyboard teleoperation for manually navigating the robot around the environment. The robot can map its environment only if it covers the entire area.

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

The current Gazebo view of the robot and the robot environment is shown next. The environment is with obstacle around the robot.

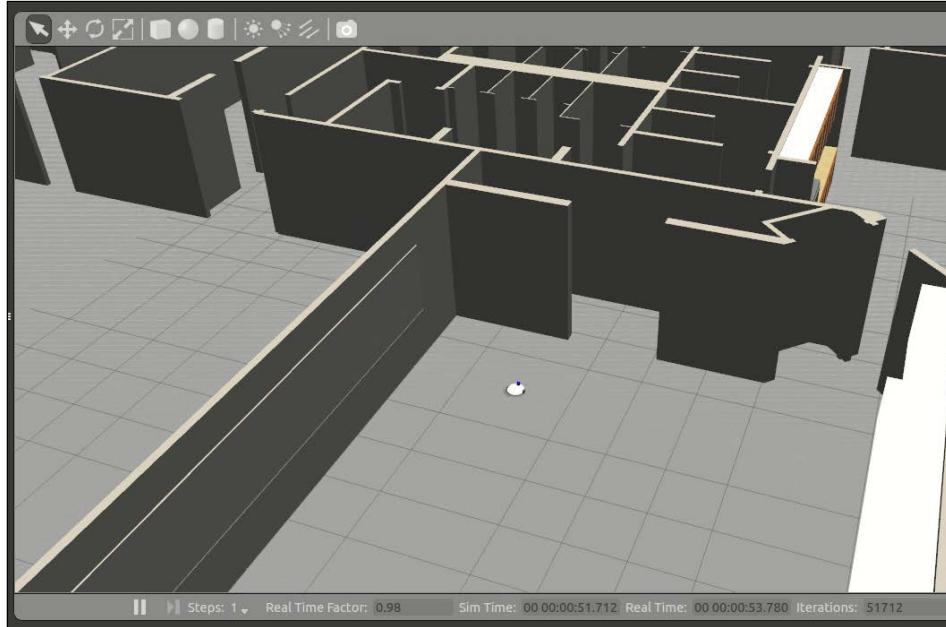


Figure 20 : Simulation of the robot using Willow Garage world

We can launch RViz and add a display type called **Map** and the topic name as /map.

We can start moving the robot inside the world by using key board teleoperation, and we can see a map building according to the environment. The following image shows the completed map of the environment shown in RViz:

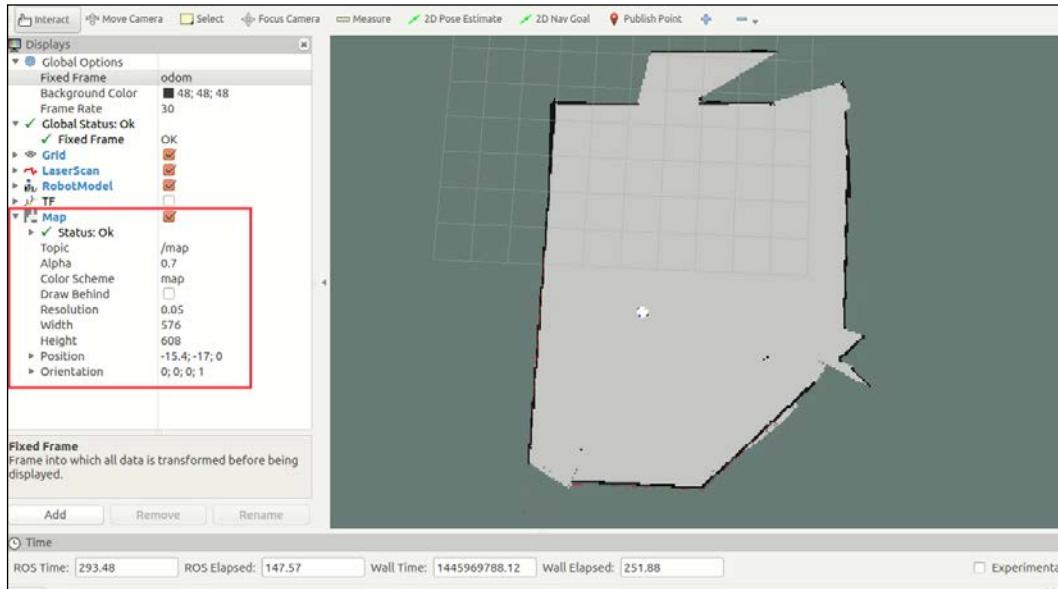


Figure 22 : Completed map of the room in RViz

We can save the built map using the following command. This command will listen to the map topic and save into the image. The map server package does this operation.

```
$ rosrun map_server map_saver -f willo
```

Here `willo` is the name of the map file. The map file is stored as two files: one is the YAML file which contains the map metadata and the image name, and second is the image which has the encoded data of the occupancy grid map. Following is the screenshot of the preceding command, running without any errors:

```
lentin@lentin-Aspire-4755:~$ rosrun map_server map_saver -f test1
[ INFO] [1434905213.468430940]: Waiting for the map
[ INFO] [1434905213.732655598]: Received a 288 X 480 map @ 0.050 m/pix
[ INFO] [1434905213.732744137]: Writing map occupancy data to test1.pgm
[ INFO] [1434905213.770990814, 314.112000000]: Writing map occupancy data to test1.yaml
[ INFO] [1434905213.771252850, 314.112000000]: Done
```

Figure 23 : Terminal screenshot while saving a map

The saved encoded image of the map is shown next. If the robot gives accurate robot odometry data, we will get this kind of precise map similar to the environment. The accurate map improves the navigation accuracy through efficient path planning.

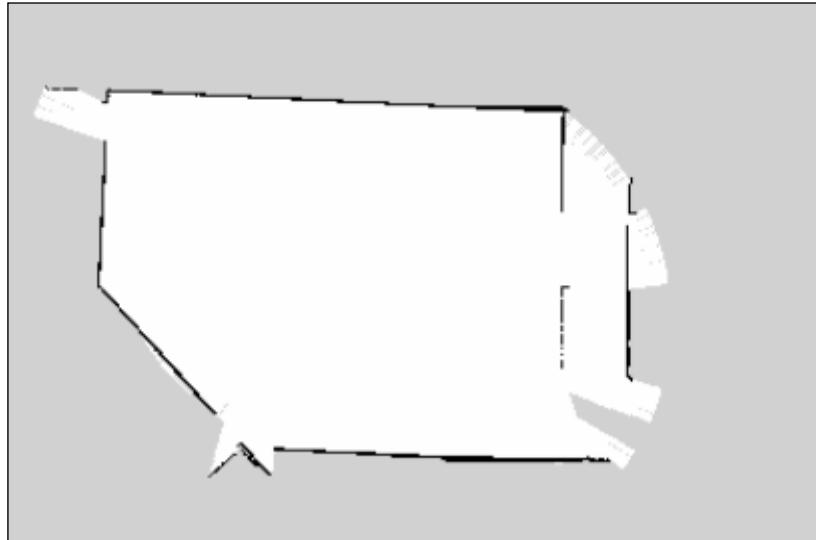


Figure 24 : The saved map

The next procedure is to localize and navigate in this static map.

Implementing autonomous navigation using AMCL and a static map

The ROS AMCL package provide nodes for localizing the robot on a static map. The `amcl` node subscribes the laser scan data, laser scan based maps, and the `tf` information from the robot. The `amcl` node estimates the pose of the robot on the map and publishes its estimated position with respect to the map.

If we create a static map from the laser scan data, the robot can autonomously navigate from any pose of the map using AMCL and the `move_base` nodes. The first step is to create a launch file for starting the `amcl` node. The `amcl` node is highly customizable; we can configure it with a lot of parameters. The list of parameters are available in the ROS package site (<http://wiki.ros.org/amcl>).

Creating an AMCL launch file

A typical `amcl` launch file is given next. The AMCL node is configured inside the `amcl.launch.xml` file which is in the `diff_wheeled_robot_gazebo/launch/include` package. The `move_base` node is also configured separately in the `move_base.launch.xml` file. The map file we created in the gmapping process is loaded here using the `map_server` node.

```
<launch>

    <!-- Map server -->
    <arg name="map_file" default="$(find diff_wheeled_robot_gazebo)/maps/test1.yaml"/>
    <node name="map_server" pkg="map_server" type="map_server"
args="$(arg map_file)" />

    <include file="$(find diff_wheeled_robot_gazebo)/launch/includes/
amcl.launch.xml">

        <arg name="initial_pose_x" value="0"/>
        <arg name="initial_pose_y" value="0"/>
        <arg name="initial_pose_a" value="0"/>

    </include>

    <include file="$(find diff_wheeled_robot_gazebo)/launch/includes/
move_base.launch.xml"/>

</launch>
```

Following is the code snippet of `amcl.launch.xml`. This file is a bit lengthy as we have to configure a lot of parameters for the `amcl` node.

```
<launch>
    <arg name="use_map_topic" default="false"/>
    <arg name="scan_topic"      default="scan"/>
    <arg name="initial_pose_x" default="0.0"/>
    <arg name="initial_pose_y" default="0.0"/>
    <arg name="initial_pose_a" default="0.0"/>

    <node pkg="amcl" type="amcl" name="amcl">
        <param name="use_map_topic"           value="$(arg use_map_
topic)"/>
        <!-- Publish scans from best pose at a max of 10 Hz -->
        <param name="odom_model_type"        value="diff"/>
```

```

<param name="odom_alpha5" value="0.1" />
<param name="gui_publish_rate" value="10.0" />
<param name="laser_max_beams" value="60" />
<param name="laser_max_range" value="12.0" />

```

After creating this launch file, we can start the `amcl` node using the following procedure:

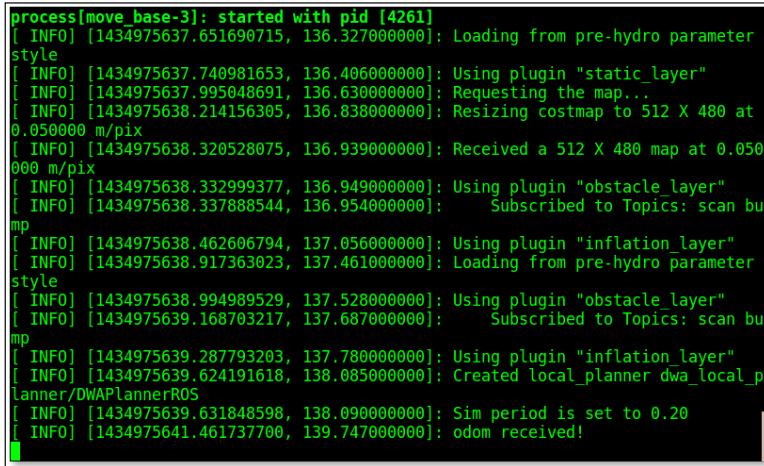
- Start the simulation of robot in Gazebo:

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

- Start the `amcl` launch file using the following command:

```
$ roslaunch diff_wheeled_robot_gazebo amcl.launch
```

If the `amcl` launch file is loaded well, the terminal shows the following message:



```

process[move_base-3]: started with pid [4261]
[ INFO] [1434975637.651690715, 136.327000000]: Loading from pre-hydro parameter
style
[ INFO] [1434975637.740981653, 136.406000000]: Using plugin "static_layer"
[ INFO] [1434975637.995048691, 136.630000000]: Requesting the map...
[ INFO] [1434975638.214156305, 136.838000000]: Resizing costmap to 512 X 480 at
0.050000 m/pix
[ INFO] [1434975638.320528075, 136.939000000]: Received a 512 X 480 map at 0.050
000 m/pix
[ INFO] [1434975638.332999377, 136.949000000]: Using plugin "obstacle_layer"
[ INFO] [1434975638.337888544, 136.954000000]: Subscribed to Topics: scan bu
mp
[ INFO] [1434975638.462606794, 137.056000000]: Using plugin "inflation_layer"
[ INFO] [1434975638.917363023, 137.461000000]: Loading from pre-hydro parameter
style
[ INFO] [1434975638.994989529, 137.528000000]: Using plugin "obstacle_layer"
[ INFO] [1434975639.168703217, 137.687000000]: Subscribed to Topics: scan bu
mp
[ INFO] [1434975639.287793203, 137.780000000]: Using plugin "inflation_layer"
[ INFO] [1434975639.624191618, 138.085000000]: Created local_planner dwa_local_p
lanner/DWAPlannerROS
[ INFO] [1434975639.631848598, 138.090000000]: Sim period is set to 0.20
[ INFO] [1434975641.461737700, 139.747000000]: odom received!

```

Figure 25 : Terminal screenshot while executing AMCL

If `amcl` is working fine, we can start commanding the robot to go into a particular position on the map using RViz, as shown in the following figure. In the figure, the arrow indicates the goal position. We have to enable `LaserScan`, `Map`, and `Path` visualizing plugins in RViz for viewing the laser scan, the global/local costmap, and the global/local paths. Using the **2D NavGoal** button in RViz, we can command the robot to go to a particular position.

The robot will plan a path to that point and give velocity commands to the robot controller to reach that point.

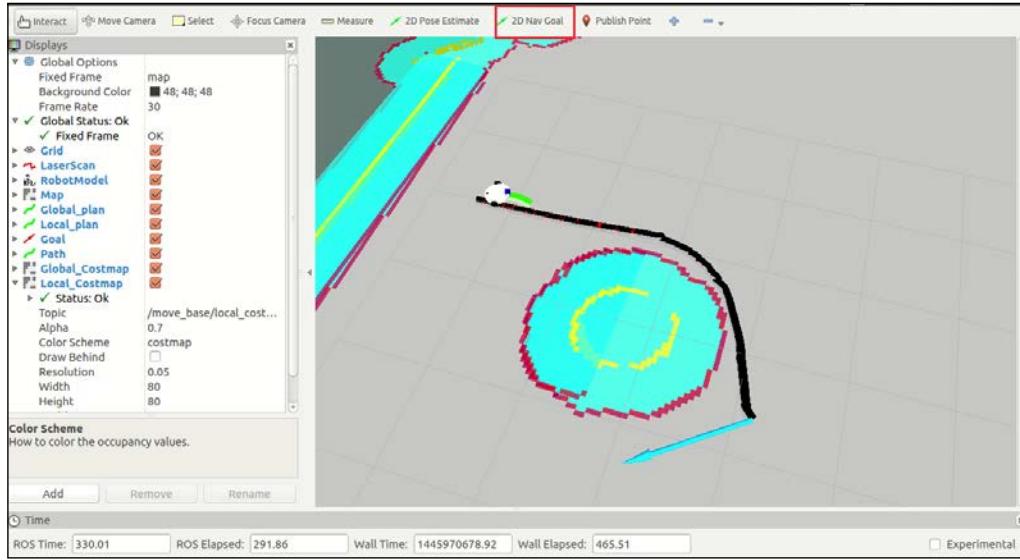


Figure 26 : Autonomous navigation using AMCL and the map

In the preceding image, we can see that we have placed a random obstacle in the robot path and that the robot has planned a path to avoid the obstacle.

We can view the AMCL particle cloud around the robot by adding a **Pose Array** on RViz and the topic is `/particle_cloud`. The following image shows the AMCL particle around the robot:

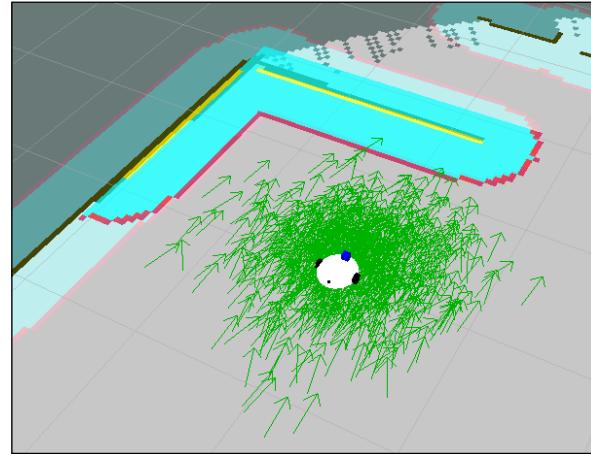


Figure 27 : The AMCL particle cloud

Questions

1. What is the main purpose of MoveIt! packages?
2. What is the importance of the `move_group` node in MoveIt!?
3. What is the purpose of the `move_base` node in the Navigation stack?
4. What are the functions of the SLAM and AMCL packages?

Summary

This chapter offered a brief overview of MoveIt! and the Navigation stack of ROS and demonstrated its capabilities using Gazebo simulation of a robotic arm mobile base. The chapter started with a MoveIt! overview and discussed detailed concepts about MoveIt!. After discussing MoveIt!, we interfaced MoveIt! and Gazebo. After interfacing, we executed the trajectory from MoveIt! on Gazebo.

The next section was about the ROS Navigation stack. We discussed its concepts and workings as well. After discussing the concepts, we tried to interface our robot in Gazebo to the Navigation stack and build a map using SLAM. After doing SLAM, we performed autonomous navigation using AMCL and the static map.

In the next chapter, we will discuss pluginlib, nodelets, and controllers.

5

Working with Pluginlib, Nodelets, and Gazebo Plugins

In the previous chapter, we have discussed about the interfacing and simulation of the robotic arm mobile robot to the ROS MoveIt! and Navigation stack. In this chapter, we will see some of the advanced concepts in ROS such as the ROS pluginlib, nodelets, and Gazebo plugins. We will discuss the functionalities and applications of each concept and will look at an example to demonstrate it's working. We have used Gazebo plugins in the previous chapters to get the sensor and robot behavior inside the Gazebo simulator. In this chapter, we are going to see how to create it. We will also discuss an modified form of ROS nodes called ROS nodelets. These features in ROS are implemented using a plugin architecture called `pluginlib`.

In this chapter, we will discuss the following topics:

- Understanding `pluginlib`
- Implementing a sample plugin using `pluginlib`
- Understanding ROS nodelets
- Implementing a sample nodelet
- Understanding and creating a Gazebo plugin

Understanding pluginlib

Plugins are a commonly used term in the computer world. Plugins are modular piece of software which can add a new feature to the existing software application. The advantage of plugins are that we don't need to write all the features in a main software; instead, we can make an infrastructure on the main software to accept new plugins to it. Using this method, we can extend the capabilities of software to any level.

We need plugins for our robotics application too. When we are going to build a complex ROS based application for a robot, plugins will be a good choice to extend the capabilities of the application.

The ROS system provides a plugin framework called **pluginlib** to dynamically load/unload plugins, which can be a library or class. **pluginlib** is a set of a C++ library, which helps to write plugins and load/unload whenever we need.

Plugin files are runtime libraries such as **shared objects (.so)** or **dynamic link libraries (.DLL)**, which is built without linking to the main application code. Plugins are separate entities which do not have any dependencies with the main software.

The main advantage of plugins is that we can expand the application capabilities without making many changes in the main application code.

We can create a simple plugin using **pluginlib** and can see all the procedures involved in creating a plugin using ROS **pluginlib**.

Here, we are going to create a simple calculator application using **pluginlib**. We are adding each functionality of the calculator using plugins.

Creating plugins for the calculator application using pluginlib

Creating a calculator application using plugins is a slightly tedious task compared to writing a single code for The aim of this example is to show how to add new features to calculator without modifying main application code.

In this example, we will see a calculator application that loads plugins to perform each operation. Here, we only implement the main operations such as addition, subtraction, multiplication, and division. We can expand to any level by writing individual plugins for each operation.

Before going on to create the plugin definition, we can access the `calculator` code from the `chapter_5_codes/pluginlib_calculator` folder for reference.

We are going to create a ROS package called `pluginlib_calculator` to build these plugins and the main calculator application.

The following diagram shows how the calculator plugins and application are organized inside the `pluginlib_calculator` ROS package:

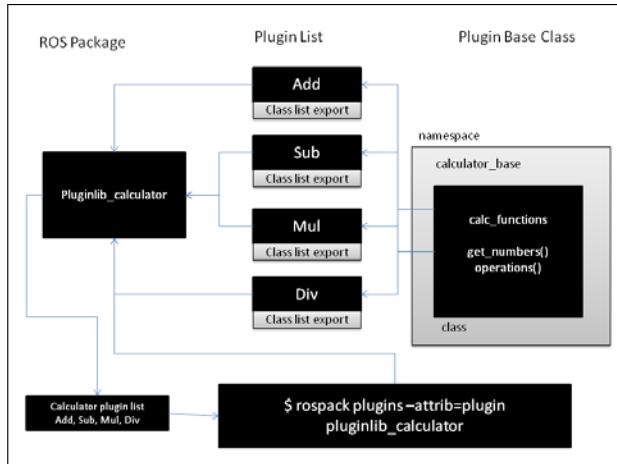


Figure 1: Organization of plugins in the calculator application

We can see the list of plugins of the calculator and a plugin base class called `calc_functions`. The plugin base class implements the common functionalities that are required by all of these plugins.

Here is how we can create the ROS package and start developing plugins for the main calculator application.

Working with `pluginlib_calculator` package

For a quick start, we can use the existing ROS plugin package (`chapter_5_codes/pluginlib_calculator`).

If we want to create this package from scratch, you can use the following command:

```
$ catkin_create_pkg pluginlib_calculator pluginlib roscpp std_msgs
```

The main dependency of this package is `pluginlib`. We can discuss the main source files in this package to build plugins.

Step 1 – Creating calculator_base header file

The `calculator_base.h` file is present in the `chapter_5_codes/pluginlib_calculator/include/pluginlib_calculator` folder and the main purpose of this file is to declare functions/methods that are commonly used by the plugins:

```
namespace calculator_base
{
    class calc_functions
    {
```

Inside this code, we declare a class called `calc_functions` that encapsulate methods used by the plugins. This class is included in a namespace called `calculator_base`. We can add more classes inside this namespace to expand the functionalities of this base class:

```
    virtual void get_numbers(double number1, double number2) = 0;
    virtual double operation() = 0;
```

These are the main methods implemented inside the `calc_function` class. The `get_number()` function can retrieve two numbers as input to the calculator, and the `operation()` function defines the mathematical operation we want to perform.

Step 2 – Creating calculator_plugins header file

The `calculator_plugins.h` file is present in the `chapter_5_codes/pluginlib_calculator/include/pluginlib_calculator` folder and the main purpose of this file is to define complete functions of the calculator plugins, which are named as `Add`, `Sub`, `Mul`, and `Div`. Here is the explanation of this code:

```
#include <pluginlib_calculator/calculator_base.h>
#include <cmath>

namespace calculator_plugins
{
    class Add : public calculator_base::calc_functions
    {
```

This header file includes the `calculator_base.h` for accessing the basic functionalities of a calculator. Each plugin is defined as a class and it inherits the `calc_functions` class from the `calculator_base.h` class:

```
    public:
    Add()
    {
        number1_ = 0;
        number2_ = 0;
```

```
}

void get_numbers(double number1, double number2)
{
    try{

        number1_ = number1;
        number2_ = number2;
    }

    catch(int e)
    {
        std::cerr<<"Exception while inputting numbers"<<std::endl;
    }

}

double operation()
{
    return(number1_+number2_);
}

private:
    double number1_;
    double number2_;
};
```

In this code, we can see definitions of inherited `get_numbers()` and `operations()` functions. The `get_number()` retrieves two number inputs and `operations()` performs the desired operation. In this case, it performs additional operations. We can see all other plugin definitions inside this header file.

Step 3 – Exporting plugins using calculator_plugins.cpp

In order to load the class of plugins dynamically, we have to export each class using a special macro called `PLUGINLIB_EXPORT_CLASS`. This macro has to put in any CPP file that consists of plugin classes. We have already defined the plugin class, and in this file we are going to define the macro statement only.

Locate the `calculator_plugins.cpp` file from the `chapter_5_codes/pluginlib_calculator/src` folder, and here is how we export each plugin:

```
#include <pluginlib/class_list_macros.h>
#include <pluginlib_calculator/calculator_base.h>
#include <pluginlib_calculator/calculator_plugins.h>
```

```
PLUGINLIB_EXPORT_CLASS(calculator_plugins::Add, calculator_base::calc_
functions);
```

Inside `PLUGINLIB_EXPORT_CLASS`, we need to provide the class name of the plugin and the base class.

Step 4 – Implementing plugin loader using calculator_loader.cpp

This plugin loader node loads each plugin and inputs the number to each plugin and fetch's the result from the plugin. We can locate the `calculator_loader.cpp` file from the `chapter_5_codes/pluginlib_calculator/src` folder.

Here is the explanation of this code:

```
#include <boost/shared_ptr.hpp>
#include <pluginlib/class_loader.h>
#include <pluginlib_calculator/calculator_base.h>
```

These are the necessary header files to load the plugins:

```
pluginlib::ClassLoader<calculator_base::calc_functions>
calc_loader("pluginlib_calculator",
"calculator_base::calc_functions");
```

The `pluginlib` provides the `ClassLoader` class, which is inside `class_loader.h`, to load classes in runtime. We need to provide a name for the loader and the calculator base class as arguments:

```
boost::shared_ptr<calculator_base::calc_functions> add =
calc_loader.createInstance("pluginlib_calculator/Add");
```

This will create an instance of the `Add` class using the `ClassLoader` object:

```
add->get_numbers(10.0,10.0);
double result = add->operation();
```

These lines give an input and perform the operations in the plugin instance.

Step 5 – Creating plugin description file: calculator_plugins.xml

After creating the calculator loader code, next we have to describe the list of plugins inside this package in an XML file called the `Plugin Description File`. The plugin description file contains all the information about the plugins inside a package such as the name of the classes, types of classes and base class, and so on.

The plugin description is an important file for plugin based packages, because it helps the ROS system to automatically discover, load, and reason about the plugin. It also holds information such as the description of the plugin.

The following code shows the plugin description file of our package called `calculator_plugins.xml`, which is stored along with the `CMakeLists.txt` and `package.xml` files. You can get this file from the `chapter_5_codes/pluginlib_calculator` folder itself.

Here is the explanation of this file:

```
<library path="lib/libpluginlib_calculator">
    <class name="pluginlib_calculator/Add" type="calculator_
plugins::Add" base_class_type="calculator_base::calc_functions">
        <description>This is a add plugin.</description>
    </class>
```

This code is for the Add plugin and it defines the library path of the plugin, the class name, the class type, the base class, and the description.

Step 6 – Registering plugin with the ROS package system

For `pluginlib` to find all plugins based packages in the ROS system, we should export the plugin description file inside `package.xml`. If we do not include this plugin, the ROS system won't find the plugins inside the package.

Here, we add the `export` tag to `package.xml` as follows:

```
<export>
    <pluginlib_calculator plugin="${prefix}/calculator_plugins.xml"
/>
</export>
```

In order to work this export command properly, we should insert the following lines in `package.xml`:

```
<build_depend>pluginlib_calculator</build_depend>
<run_depend>pluginlib_calculator</run_depend>
```

The current package should directly *depend* on itself, both at the time of building and also at runtime.

Step 7 – Editing the CMakeLists.txt file

In order to build the calculator plugins and loader nodes, we should add the following lines in `CMakeLists.txt`:

```
## pluginlib_tutorials library
add_library(pluginlib_calculator src/calculator_plugins.cpp)
target_link_libraries(pluginlib_calculator ${catkin_LIBRARIES})

## calculator_loader executable
add_executable(calculator_loader src/calculator_loader.cpp)
target_link_libraries(calculator_loader ${catkin_LIBRARIES})
```

You can get the complete `CMakeLists.txt` from the package itself.

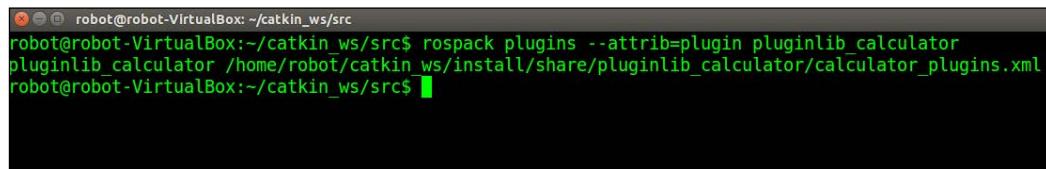
We are almost done with all the settings and now it's time to build the package using the `catkin_make` command.

Step 8: Querying the list of plugins in a package

If the package is built properly, we can execute the loader. The following command will query the plugins inside a package:

```
$ rospack plugins --attrib=plugin pluginlib_calculator
```

We will get the following result if everything is built properly:

A terminal window titled "robot@robot-VirtualBox: ~/catkin_ws/src". It shows the command "rospack plugins --attrib=plugin pluginlib_calculator" being run, followed by the output: "pluginlib_calculator /home/robot/catkin_ws/install/share/pluginlib_calculator/calculator_plugins.xml".

```
robot@robot-VirtualBox:~/catkin_ws/src$ rospack plugins --attrib=plugin pluginlib_calculator
pluginlib_calculator /home/robot/catkin_ws/install/share/pluginlib_calculator/calculator_plugins.xml
robot@robot-VirtualBox:~/catkin_ws/src$
```

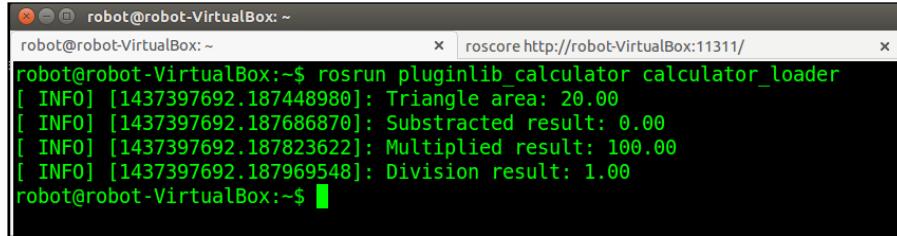
Figure 2: The result of the plugin query

Step 9 – Running the plugin loader

We can run the `calculator_loader` using the following command:

- Run the `roscore`, as follows:
`$ roscore`
- Run the `calculator_loader` using the following command:
`$ rosrun pluginlib_calculator calculator_loader`

The following screenshot shows the output of this command, to check whether everything is working fine. The loader gives both inputs as 10.0 and we are getting the proper result as shown using plugins in the screenshot:



```
robot@robot-VirtualBox:~$ rosrun pluginlib calculator calculator_loader
[ INFO] [1437397692.187448980]: Triangle area: 20.00
[ INFO] [1437397692.187686870]: Subtracted result: 0.00
[ INFO] [1437397692.187823622]: Multiplied result: 100.00
[ INFO] [1437397692.187969548]: Division result: 1.00
robot@robot-VirtualBox:~$
```

Figure 3: Result of the plugin loader node

In the next section, we will look at a new concept called **nodelets** and discuss how to implement it.

Understanding ROS nodelets

Nodelets are a type of ROS node that are designed to run multiple nodes in a single process, with each node running as a thread. The threaded nodes can communicate with each other efficiently without overloading the network having, zero copy transport between two nodes. These threaded nodes can communicate with external nodes too.

As we did using pluginlib, in nodelets also, we can dynamically load each class as a plugin, which has a separate namespace. Each loaded class can act as separate nodes, which are on a single process called nodelet.

Nodelets are used when the volume of data transferred between nodes are very high, for example, in transferring data from 3D sensors or cameras.

Next, we look at how to create a nodelet.

Creating a nodelet

In this section, we are going to create a basic nodelet that can subscribe a string topic called /msg_in and publish the same string (`std_msgs/String`) on the topic /msg_out.

Step 1 – Creating a package for nodelet

We can create a package called `nodelet_hello_world` using the following command to create our nodelet:

```
$ catkin_create_pkg nodelet_hello_world nodelet roscpp std_msgs
```

Otherwise, we can use the existing package from `chapter_5_codes/nodelet_hello_world`.

Here, the main dependency of this package is the `nodelet` package, which provides APIs to build a ROS nodelet.

Step 2 – Creating hello_world.cpp nodelet

Now, we are going to create the nodelet code. Create a folder called `src` inside the package and create a file called `hello_world.cpp`.

You will get the existing code from the `chapter_5_codes/nodelet_hello_world/src` folder.

Step 3 – Explanation of hello_world.cpp

Here is the explanation of the code:

```
#include <pluginlib/class_list_macros.h>
#include <nodelet/nodelet.h>
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <stdio.h>
```

These are the header files of this code. We should include `class_list_macros.h` and `nodelet.h` to access `pluginlib` APIs and `nodelets` APIs:

```
namespace nodelet_hello_world
{
    class Hello : public nodelet::Nodelet
    {
```

Here, we create a nodelet class called `Hello`, which inherits a standard nodelet base class. All nodelet classes should inherit from the nodelet base class and be dynamically loadable using `pluginlib`. Here, the `Hello` class is going to be used for dynamic loading:

```
    virtual void onInit()
    {
        ros::NodeHandle& private_nh = getPrivateNodeHandle();
```

```
NODELET_DEBUG("Initialized the Nodelet");
pub = private_nh.advertise<std_msgs::String>("msg_out",5);
sub = private_nh.subscribe("msg_in",5, &Hello::callback, this);
}
```

This is the initialization function of a nodelet. This function should not block or do significant work. Inside the function, we are creating a node handle object, topic publisher, and subscriber on the topic `msg_out` and `msg_in` respectively. There are macros to print debug messages while executing a nodelet. Here, we use `NODELET_DEBUG` to print debug messages in the console. The subscriber is tied up with a callback function called `callback()`, which is inside the `Hello` class:

```
void callback(const std_msgs::StringConstPtr input)
{
    std_msgs::String output;
    output.data = input->data;
    NODELET_DEBUG("Message data = %s",output.data.c_str());
    ROS_INFO("Message data = %s",output.data.c_str());
    pub.publish(output);
}
```

In the `callback()` function, it will print the messages from the `/msg_in` topic and publish to the `/msg_out` topic:

```
PLUGINLIB_EXPORT_CLASS(nodelet_hello_world::Hello,nodelet::Nodelet);
```

Here, we are exporting the `Hello` as a plugin for the dynamic loading.

Step 4 – Creating plugin description file

Similar to the `pluginlib` example, we have to create a plugin description file inside the `nodelet_hello_world` package. The plugin description file `hello_world.xml` is as follows:

```
<library path="libnodelet_hello_world">
    <class name="nodelet_hello_world/Hello" type="nodelet_hello_
world::Hello" base_class_type="nodelet::Nodelet">
        <description>
            A node to republish a message
        </description>
    </class>
</library>
```

Step 5 – Adding the export tag in package.xml

We need to add the export tag in `package.xml` and also add build and run dependencies:

```
<export>
  <nodelet plugin="${prefix}/hello_world.xml"/>
</export>

<build_depend>nodelet_hello_world</build_depend>
<run_depend>nodelet_hello_world</run_depend>
```

Step 6 – Editing CMakeLists.txt

We need to add additional lines of code in `CMakeLists.txt` to build a nodelet package. Here are the extra lines. You will get the complete `CMakeLists.txt` file from the existing package itself:

```
## Declare a cpp library
add_library(nodelet_hello_world
  src/hello_world.cpp
)

## Specify libraries to link a library or executable target against
target_link_libraries(nodelet_hello_world
  ${catkin_LIBRARIES}
)
```

Step 7 – Building and running nodelets

After following this procedure, we can build the package using `catkin_make` and if the build is successful, we can generate the shared object `libnodelet_hello_world.so` file, which is actually a plugin.

The first step in running nodelets is to start the **nodelet manager**. A nodelet manager is a C++ executable program, which will listen to the ROS services and dynamically load nodelets. We can run a standalone manager or can embed it within a running node.

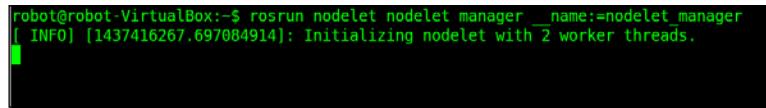
The following commands can start the nodelet manager:

- Start `roscore`
`$ roscore`

- Start the nodelet manager using the following command

```
$ rosrun nodelet nodelet manager __name:=nodelet_manager
```

If the nodelet manager runs successfully, we will get a message as shown here:



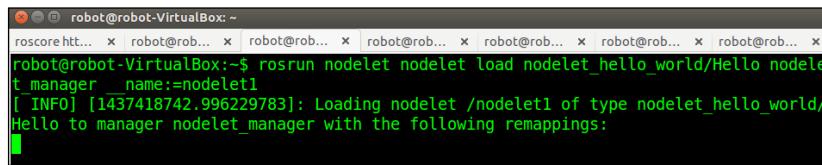
```
robot@robot-VirtualBox:~$ rosrun nodelet nodelet manager __name:=nodelet_manager
[ INFO] [1437416267.697084914]: Initializing nodelet with 2 worker threads.
```

Figure 4: Running the nodelet manager

After launching the nodelet manager, we can start the nodelet by using the following command:

```
$ rosrun nodelet nodelet load nodelet_hello_world/Hello
nodelet_manager __name:=nodelet1
```

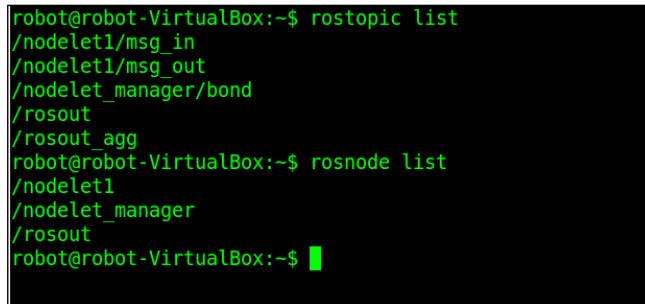
When we execute the preceding command, the nodelet contacts the nodelet manager to instantiate an instance of the nodelet_hello_world/Hello nodelet with a name of nodelet1. The following screenshot shows the message when we load the nodelet:



```
robot@robot-VirtualBox:~$ rosrun nodelet nodelet load nodelet_hello_world/Hello
nodelet_manager __name:=nodelet1
[ INFO] [1437418742.996229783]: Loading nodelet /nodelet1 of type nodelet_hello_world/
Hello to manager nodelet_manager with the following remappings:
```

Figure 5: Running nodelet

The topics generated after running this nodelet and the list of nodes are shown here:



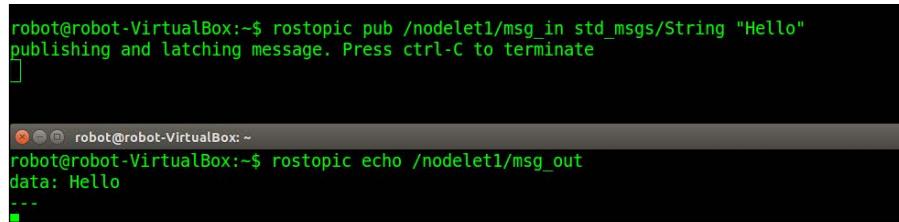
```
robot@robot-VirtualBox:~$ rostopic list
/nodelet1/msg_in
/nodelet1/msg_out
/nodelet_manager/bond
/rosout
/rosout_agg
robot@robot-VirtualBox:~$ rosnode list
/nodelet1
/nodelet_manager
/rosout
robot@robot-VirtualBox:~$
```

Figure 6: The list of topics of the nodelet

We can test the node by publishing a string to the /nodelet1/msg_in topic and check whether we receive the same message in nodelet1/msg_out.

The following command publishes a string to /nodelet1/msg_in:

```
$ rostopic pub /nodelet1/msg_in std_msgs/String "Hello"
```



A screenshot of a terminal window titled 'robot@robot-VirtualBox:~'. It shows two commands being run: 'rostopic pub /nodelet1/msg_in std_msgs/String "Hello"' and 'rostopic echo /nodelet1/msg_out'. The first command outputs 'publishing and latching message. Press ctrl-C to terminate'. The second command outputs 'data: Hello' followed by three ellipses.

Figure 7: Publishing and subscribing using the Nodelet

We can echo the msg_out topic and can confirm whether the code is working good.

Here, we have seen that a single instance of the Hello() class is created as a node. We can create multiple instances of the Hello() class with different node names inside this nodelet.

Step 8 – Creating launch files for nodelets

We can also write launch files to load more than one instance of the nodelet class. The following launch file will load two nodelets with the names test1 and test2, and we can save it with a name hello_world.launch:

```
<launch>

<!-- Started nodelet manager -->

<node pkg="nodelet" type="nodelet" name="standalone_nodelet"
args="manager" output="screen"/>

<!-- Starting first nodelet -->

<node pkg="nodelet" type="nodelet" name="test1" args="load nodelet_
hello_world>Hello standalone_nodelet" output="screen">
</node>

<!-- Starting second nodelet -->

<node pkg="nodelet" type="nodelet" name="test2" args="load nodelet_
hello_world>Hello standalone_nodelet" output="screen">
</node>

</launch>
```

The preceding launch can be launched using the following commands:

```
$ rosrun nodelet_hello_world hello_world.launch
```

The following message will show up on the terminal if it is launched successfully:

```
setting /run_id to 502aac5e-2f14-11e5-9a2e-0800273c354c
process[rosout-1]: started with pid [5210]
started core service [/rosout]
process[standalone_nodelet-2]: started with pid [5227]
[ INFO] [1437420001.238956883]: Initializing nodelet with 2 worker threads.
process[test1-3]: started with pid [5245]
[ INFO] [1437420001.402022087]: Loading nodelet /test1 of type nodelet_hello_world/Hello to manager standalone_nodelet with the following remappings:
process[test2-4]: started with pid [5284]
[ INFO] [1437420001.704979871]: Loading nodelet /test2 of type nodelet_hello_world/Hello to manager standalone_nodelet with the following remappings:
```

Figure 8: Launching multiple instances of the Hello() class

The list of topics and nodes are shown here. We can see two nodelets instantiated and we can see their topics too.

```
robot@robot-VirtualBox:~$ rostopic list
/rosout
/rosout_agg
/standalone_nodelet/bond
/test1/msg_in
/test1/msg_out
/test2/msg_in
/test2/msg_out
robot@robot-VirtualBox:~$ rosnode list
/rosout
/standalone_nodelet
/test1
/test2
robot@robot-VirtualBox:~$
```

Figure 9: Topics generated by the multiple instances of Hello() class

The following diagram shows how to interconnect these nodelets:

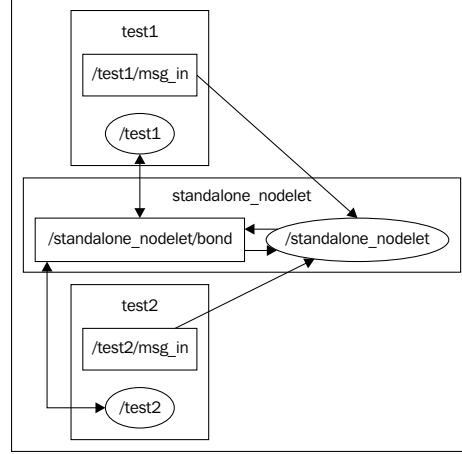


Figure 10: A two-node instance of a nodelet

Run the `rqt_graph` tool to view the preceding node graph view:

```
$rosrun rqt_gui rqt_gui
```

Load the `Node Graph` plugin from the following option **Plugins | Introspection | Node Graph** and you will get a graph as shown in the preceding figure.

Understanding the Gazebo plugins

Gazebo plugins help us to control the robot models, sensors, world properties, and even the way Gazebo runs. Similar to pluginlib and nodelets, Gazebo plugins are a set of C++ code, which can be dynamically loaded/unloaded from the Gazebo simulator.

Using plugins we can access all the components of Gazebo, and also it is independent of ROS, so that it can share with people who are not using ROS too. We can mainly classify the plugins as follows:

- **The world plugin:** Using the world plugin, we can control the properties of a specific world in Gazebo. We can change the physics engine, the lighting, and other world properties using this plugin.
- **The model plugin:** The model plugin is attached to a specific model in Gazebo and controls its properties. The parameters such as joint state of the model, control of the joints, and so, on can be controlled using this plugin.
- **The sensor plugin:** The sensor plugins are for modeling sensors such as camera, IMU, and so on, in Gazebo.
- **The system plugin:** The system plugin is started along with the Gazebo startup. A user can control a system related function in Gazebo using this plugin.
- **The visual plugin:** The visual property of any Gazebo component can be accessed and controlled using the visual plugin.

Before starting development with Gazebo plugins, we might need to install some packages. If you are using ROS Indigo, the package we installed in the previous chapter is sufficient for developing Gazebo plugins. The Gazebo version installed along with ROS Indigo is 2.2.3. But if you are working with ROS Jade, the default Gazebo is Version 5, so you might need to install its development package in Ubuntu using the following command:

```
$ sudo apt-get install libgazebo5-dev
```

The Gazebo plugins are independent of ROS and we don't need ROS libraries to build the plugin.

Creating a basic world plugin

We will look at a basic Gazebo world plugin and try to build and load it in Gazebo.

Create a folder called `gazebo_basic_world_plugin` in the user home folder and create a CPP file called `hello_world.cc`:

```
$ mkdir ~/gazebo_basic_world_plugin
$ cd ~/gazebo_basic_world_plugin
$ nano hello_world.cc
```

The definition of `hello_world.cc` is as follows:

```
//Gazebo header for getting core gazebo functions
#include <gazebo/gazebo.hh>

//All gazebo plugins should have gazebo namespace

namespace gazebo
{

    //The custom WorldpluginTutorials is inheriting from standard
    worldPlugin. Each world plugin has to inheriting from standard plugin
    type.

    class WorldPluginTutorial : public WorldPlugin
    {

        public: WorldPluginTutorial() : WorldPlugin()
        {
            printf("Hello World!\n");
        }

        //The Load function can receive the SDF elements
        public: void Load(physics::WorldPtr _world, sdf::ElementPtr _sdf)
        {
        }
    };

    //Registering World Plugin with Simulator
    GZ_REGISTER_WORLD_PLUGIN(WorldPluginTutorial)
}
```

The header file used in this code is `<gazebo/gazebo.hh>`; the header contains core functionalities of Gazebo. Other headers are as follows:

- `gazebo/physics/physics.hh`: This is the Gazebo header for accessing the physics engine parameters
- `gazebo/rendering/rendering.hh`: This is the Gazebo header for handling rendering parameters
- `gazebo/sensors/sensors.hh`: This is the header for handling sensors

At the end of the code, we have to export the plugin using the statements mentioned below.

The `GZ_REGISTER_WORLD_PLUGIN(WorldPluginTutorial)` macro will register and export the plugin as a world plugin. The following macros are used to register for sensors, models, and so on:

- `GZ_REGISTER_MODEL_PLUGIN`: This is the export macro for Gazebo robot model
- `GZ_REGISTER_SENSOR_PLUGIN`: This is the export macro for Gazebo sensor model
- `GZ_REGISTER_SYSTEM_PLUGIN`: This is the export macro for Gazebo system
- `GZ_REGISTER_VISUAL_PLUGIN`: This is the export macro for Gazebo visuals

After setting the code, we can make the `CMakeLists.txt` for compiling the source. The following is the source of `CMakeLists.txt`:

```
$ nano ~/gazebo_basic_world_plugin/CMakeLists.txt

cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

find_package(Boost REQUIRED COMPONENTS system)
include_directories(${Boost_INCLUDE_DIRS})
link_directories(${Boost_LIBRARY_DIRS})

include (FindPkgConfig)
if (PKG_CONFIG_FOUND)
  pkg_check_modules(GAZEBO gazebo)
endif()
```

```
include_directories(${GAZEBO_INCLUDE_DIRS})
link_directories(${GAZEBO_LIBRARY_DIRS})

add_library(hello_world SHARED hello_world.cc)
target_link_libraries(hello_world ${GAZEBO_LIBRARIES} ${Boost_LIBRARIES})
```

Create a build folder for storing the shared object:

```
$ mkdir ~/gazebo_basic_world_plugin/build
$ cd ~/gazebo_basic_world_plugin/build
```

After switching to the build folder, execute the following command to compile and build the source code:

```
$ cmake ../
$ make
```

After building the code, we will get a shared object called `libhello_world.so` and we have to export the path of this shared object in `GAZEBO_PLUGIN_PATH` and add to the `.bashrc` file:

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/gazebo_basic_world_
plugin/build
```

After setting the Gazebo plugin path, we can use it inside the URDF file or the SDF file. The following is a sample world file called `hello.world`, which includes this plugin:

```
$ nano ~/gazebo_basic_world_plugin/hello.world
```

```
<?xml version="1.0"?>
<sdf version="1.4">
  <world name="default">
    <plugin name="hello_world" filename="libhello_world.so"/>
  </world>
</sdf>
```

Run the Gazebo server and load this world file:

```
$ cd ~/gazebo_basic_world_plugin  
$ gzserver hello.world --verbose
```

```
robot@robot-VirtualBox:~/gazebo_plugin_tutorial$ gzserver hello.world --verbose  
Gazebo multi-robot simulator, version 2.2.3  
Copyright (C) 2012-2014 Open Source Robotics Foundation.  
Released under the Apache 2 License.  
http://gazebosim.org  
  
Msg Waiting for master  
Msg Connected to gazebo master @ http://127.0.0.1:11345  
Msg Publicized address: 10.0.2.15  
Hello World!
```

Figure 11: The Gazebo world plugin printing "Hello World"

We will source the code for various Gazebo plugins from the Gazebo repository.

We can check <https://bitbucket.org/osrf/gazebo>

Browse for the source code. Take the examples folder and then the plugins, as shown in the following figure:



Figure 12: The list of Gazebo plugins in the repository

Questions

1. What is pluginlib and what are its main applications?
2. What is the main application of nodelets?
3. What are the different types of Gazebo plugins?
4. What is the function of the model plugin in Gazebo?

Summary

In this chapter, we covered some advanced concepts such as the pluginlib, nodelets, and Gazebo plugins, which can be used to add more functionalities to a complex ROS application. We discussed the basics of `pluginlib` and saw an example using it. After covering `pluginlib`, we saw the ROS nodelets, which are widely used in high performance applications. Also, we saw an example using the ROS nodelets. Finally, we came to the Gazebo plugins that are used to add functionalities to Gazebo simulators. In the next chapter, we will discuss more on the RViz plugin and the ROS controllers.

6

Writing ROS Controllers and Visualization Plugins

In the last chapter, we have discussed about pluginlib, nodelets, and Gazebo plugins. The base library for making plugins in ROS is pluginlib, and the same library can be used in nodelets and Gazebo plugins. In this chapter, we will continue with pluginlib-based concepts such as ROS controllers and RViz plugins. We have already worked with ROS controllers and have reused some standard controllers such as joint state, position, and trajectory controllers in *Chapter 3, Simulating Robots Using ROS and Gazebo*.

In this chapter, we will see how to write a basic ROS controller for a PR2 robot (<https://www.willowgarage.com/pages/pr2/overview>) and robots similar to PR2. After creating the controller, use the controller in PR2 simulation. The RViz plugins can add more functionality to RViz and in this chapter we can see how to create a basic RViz plugin. The detailed topics that we are going to discuss in this chapter are as follows:

- Understanding packages required for ROS controller development
- Setting the ROS controller development environment
- Understanding `ros_control` packages
- Writing and running a basic ROS controller
- Writing and running a RViz plugin

Let us see how to develop a ROS controller; the first step is to understand the dependency packages required to start building custom controllers for PR2.

The main set of package that helps us to write real-time robot controllers are `pr2_mechanism` stacks. The following is the description of `pr2_mechanism` stacks:

- `pr2_mechanism`: This is a ROS stack consisting of several classes and libraries that can be useful for writing real-time controllers. These packages are for the robot PR2 and we can reuse the packages for other robots. Following are the set of packages inside the `pr2_mechanism` stack.
- `pr2_controller_manager`: The controller manager can load and manage multiple controllers and can work them in a real-time loop.
- `pr2_controller_interface`: This is the controller base class package in which all custom real-time controllers should inherit the controller base class from this package. The controller manager will only load the controller if it inherits from this package.
- `pr2_hardware_interface`: This package consists of PR2 robot hardware interface. There are interfaces for PR2 actuators, sensors, gripper, and so on. Controllers can directly access the hardware components inside a hard real-time loop.
- `pr2_mechanism_model`: This package contains the robot model that can be used inside the controller loaded by the controller manager. The robot model mainly consists of joints, kinematics, and the dynamic model of the robot loaded from the URDF file. The controller mainly handles the main components inside the robot model which need to work in real time.
- `pr2_mechanism_msgs`: This package consists of a message and service definition that is used to communicate with the real-time control loop. The message definition consists of the state of real-time controllers, joints, and actuators.

We should install the above packages for starting with ROS real-time controllers. The following command will install the `pr2_mechanism` stack in Ubuntu 14.04:

- In ROS Indigo:

```
$ sudo apt-get install ros-indigo-pr2-gazebo ros-indigo-pr2-mechanism ros-indigo-pr2-bringup
```

- In ROS Jade, we can install `pr2_mechanism` from the source at
https://github.com/pr2/pr2_mechanism

The description of other ROS packages installing along with the `pr2_mechanism` stack are as follows:

- `pr2-gazebo`: The simulation package of PR2 using Gazebo. It contains the launch file for starting the simulation of the PR2 robot in Gazebo.
- `pr2-bringup`: This has the launch files to start the PR2 hardware and simulation.

Before writing the ROS controller, it will be good if we understand the use of each package of the `pr2_mechanism` stack.

Understanding `pr2_mechanism` packages

The `pr2_mechanism` stack contain packages for writing ROS real-time controllers. The first package that we are going to discuss is the `pr2_controller_interface` package.

`pr2_controller_interface` package

A basic ROS real-time controller must inherit a base class called `pr2_controller_interface::Controller` from this package. This base class contains four important functions: `init()`, `start()`, `update()`, and `stop()`. The basic structure of the `Controller` class is given as follows:

```
namespace pr2_controller_interface
{
    class Controller
    {
        public:
            virtual bool init(pr2_mechanism_model::RobotState *robot,
                              ros::NodeHandle &n);
            virtual void starting();
            virtual void update();
            virtual void stopping();
    };
}
```

The workflow of the controller class is shown as follows.

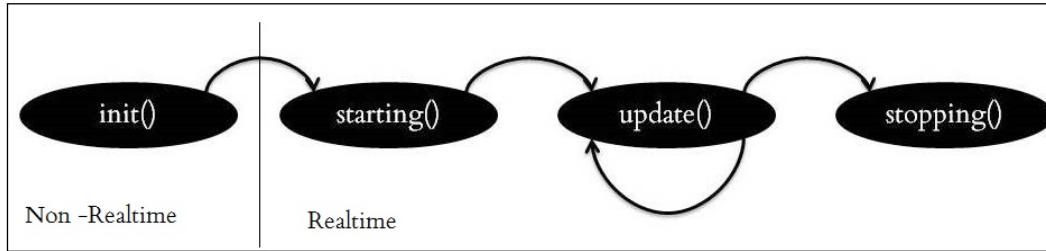


Figure 1: Workflow of the controller

Initialization of the controller

The first function executing when a controller is loaded is `init()`. The `init()` function will not start running the controller. The initialization can take any amount of time before starting the controllers. The declaration of the `init` function is given as follows:

```
virtual bool init(pr2_mechanism_model::RobotState *robot,  
ros::NodeHandle &n);
```

This method will not run as real time.

The function arguments are given as follows:

- `pr2_mechanism_model:: RobotState *robot`: The `pr2_mechanism_model` contains the robot model that can be used by the robot controller. The `pr2_mechanism_model:: RobotState` class helps us to access the joints of the robot model and kinematic/dynamic description of robot.
- `ros::NodeHandle &n`: The controller can read the robot configuration and even advertise topics using this Nodehandle.

The `init()` method only executes once while the controller is loaded by the controller manager. If the `init()` method is not successful, it will unload from the controller manager. We can write a custom message if any error occurs inside the `init()` method.

Starting the ROS controller

The `starting()` method executes once just before running the controller. This method will only execute once before updating/running the controller. This method will work in a hard real-time manner. The `starting()` method declaration is given as follows:

```
virtual void starting();
```

The controller can also call the `starting()` method when it restarts the controller without unloading it.

Updating ROS controller

The `update()` function is the most important method that makes the controller alive. The `update` method executes the code inside it at a rate of 1,000 Hz. It means the controller completes one execution within 1 millisecond.

```
virtual void update();
```

Stopping the controller

This method will call when a controller is stopped. The `stopping()` method will execute as the last `update()` call and only executes once. It is also working in hard real time. The `stopping()` method will not fail and return nothing too. The following is the declaration of the `stopping()` method:

```
virtual void stopping();
```

pr2_controller_manager

The `pr2_controller_manager` package can load/unload the controller in a hard real-time loop. The controller manager also ensures that the controller will not set a goal value that is less than or greater than the safety limits of the joint. The controller manager also publishes the states of the joint in the `/joint_state` (`sensor_msgs/JointState`) topic at a default rate of 100 Hz. The following figure shows the basic workflow of a controller manager:

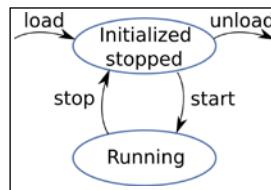


Figure 2: Working of controller manager

The controller manager can load/unload a plugin. When a controller is loaded by the controller manager, it will first initialize it, but will not start running.

After loading the controller, we can start/stop the controller. When we start the controller, it will run the controller, and when we stop it, it will simply stop. Stopping doesn't mean it is unloaded. But if the controller is unloaded from the controller manager, we can't access the controller.

Writing a basic real-time joint controller in ROS

The basic prerequisites for writing a ROS controller are already installed and we have discussed the underlying concepts of controllers. Now we can start creating a package for our own controller.

We are going to develop a controller that can access a joint of the robot and move the robot in a sinusoidal fashion.

The procedure of building a controller is similar to other plugin development that we have seen earlier. The list of procedures to create a ROS controller is given as follows:

- Create a ROS package with necessary dependencies
- Write controller code in C++
- Register or export the C++ class as plugin
- Define the plugin definition in an XML file
- Update the package.xml for exporting the plugin
- Write CMakeLists.txt
- Compile the code
- Writing configuration for our controller
- Start the PR2 simulation in Gazebo
- Load the controller using the controller manager

Step 1 – Creating controller package

The first step is to create the controller package with all its dependencies. The following command can create a package for the controller called my_controller_pkg:

```
$catkin_create_pkg my_controller_pkg roscpp pluginlib  
pr2_controller_interface pr2_mechanism_model
```

We will get the existing package from the chapter_6_codes/my_controller_pkg folder.

Step 2 – Creating controller header file

We will get the header file my_controller_file.h from the chapter_6_codes/my_controller_pkg/include/my_controller_pkg folder.

Given in the following is the header file definition of `my_controller_file.h`. We have discussed each line of this code while discussing `pr2_controller_interface`:

```
#include <pr2_controller_interface/controller.h>
#include <pr2_mechanism_model/joint.h>

namespace my_controller_ns{

    //Inheriting Controller class inside pr2_controller_interface
    class MyControllerClass: public pr2_controller_interface::Controller
    {
    private:
        pr2_mechanism_model::JointState* joint_state_;
        double init_pos_;

    public:

        virtual bool init(pr2_mechanism_model::RobotState *robot,
                          ros::NodeHandle &n);
        virtual void starting();
        virtual void update();
        virtual void stopping();
    };
}
```

In the preceding code, we can see the controller class `MyControllerClass` and we are inheriting the base class `pr2_controller_interface::Controller`. We can see that each function inside the `Controller` class is overriding in our class.

Step 3 – Creating controller source file

Create a folder called `src` inside the package and create a C++ file called `my_controller_file.cpp`, which is the class definition of the above header.

Given in the following is the definition of `my_controller_file.cpp`, which has to be saved inside the `src` folder:

```
#include "my_controller_pkg/my_controller_file.h"
#include <pluginlib/class_list_macros.h>
namespace my_controller_ns {
    /// Controller initialization in non-real-time
    bool MyControllerClass::init(pr2_mechanism_model::RobotState *robot,
                                ros::NodeHandle &n)
    {
        std::string joint_name;
```

```
if (!n.getParam("joint_name", joint_name))
{
    ROS_ERROR("No joint given in namespace: '%s'",
              n.getNamespace().c_str());
    return false;
}
joint_state_ = robot->getJointState(joint_name);
if (!joint_state_)
{
    ROS_ERROR("MyController could not find joint named '%s'",
              joint_name.c_str());
    return false;
}
return true;
}
/// Controller startup in realtime
void MyControllerClass::starting()
{
    init_pos_ = joint_state_->position_;
}
/// Controller update loop in real-time
void MyControllerClass::update()
{
    //Setting a desired position
    double desired_pos = init_pos_ + 15 * sin(ros::Time::now().toSec());
    //Getting current joint position
    double current_pos = joint_state_->position_;
    //Commanding the effort to joint to move into the desired goal
    joint_state_->commanded_effort_ = -10 * (current_pos - desired_pos);
}
/// Controller stopping in realtime
void MyControllerClass::stopping()
{}
} // namespace

// Register controller to pluginlib
PLUGINLIB_EXPORT_CLASS(my_controller_pkg,MyControllerPlugin,
                      my_controller_ns::MyControllerClass,
                      pr2_controller_interface::Controller)
```

Step 4 – Explanation of the controller source file

In this section, we can see the explanation of each section of the code:

```
/// Controller initialization in non-real-time
bool MyControllerClass::init(pr2_mechanism_model::RobotState *robot,
                             ros::NodeHandle &n)
{
    std::string joint_name;
    if (!n.getParam("joint_name", joint_name))
    {
```

The preceding is the `init()` function definition of the controller. This will be called when a controller is loaded by the controller manager. Inside the `init()` function, we are creating an instance of `RobotState` and `NodeHandle`, also retrieving a joint name for attaching our controller. This joint name is defined inside the controller configuration file. We can see the controller configuration file in the next section.

```
    joint_state_ = robot->getJointState(joint_name);
```

This will create a joint state object for a particular joint. Here `robot` is an instance of the `RobotState` class and `joint_name` is the desired joint in which we are attaching the controller:

```
/// Controller startup in realtime
void MyControllerClass::starting()
{
    init_pos_ = joint_state_->position_;
}
```

After loading the controller, the next step is to start the controller. The preceding function will execute when we start a controller. In this function, it will retrieve the current state of the joint into the `init_pos_` variable:

```
/// Controller update loop in real-time
void MyControllerClass::update()
{
    //Setting a desired position
    double desired_pos = init_pos_ + 15 * sin(ros::Time::now().toSec());
    //Getting current joint position
    double current_pos = joint_state_->position_;
    //Commanding the effort to joint to move into the desired goal
    joint_state_->commanded_effort_ = -10 * (current_pos - desired_pos);
}
```

This is the update function of the controller, which will continuously move the joint in a sinusoidal fashion.

Step 5 – Creating plugin description file

We can define the plugin definition file, which is given in the following. The plugin file is being saved inside the package folder with a name of `controller_plugins.xml`:

```
<library path="lib/libmy_controller_lib">
  <class name="my_controller_pkg/MyControllerPlugin"
        type="my_controller_ns::MyControllerClass"
        base_class_type="pr2_controller_interface::Controller" />
</library>
```

Step 6 – Updating package.xml

We need to update the `package.xml` for pointing the `controller_plugins.xml` file:

```
<export>
  <pr2_controller_interface plugin="${prefix}/controller_plugins.xml" />
</export>
```

Step 7 – Updating CMakeLists.txt

After doing all these things, we can compose the `CMakeLists.txt` of the package:

```
## my_controller_file library
add_library(my_controller_lib src/my_controller_file.cpp)
target_link_libraries(my_controller_lib ${catkin_LIBRARIES})
```

You will get the complete `CMakeLists.txt` from `chapter_6_codes/my_controller_pkg`.

Step 8 – Building controller

After completing the `CMakeLists.txt`, we can build our controller using the `catkin_make` command. After building, check that the controller is configured as a plugin using `rospack` command, as given in the following:

```
$ rospack plugins --attrib=plugin pr2_controller_interface
```

If everything has been performed correctly, the output may look like the following:

```
robot@robot-VirtualBox:~/catkin_ws$ rospack plugins --attrib=plugin pr2_controller_interface
pr2_controller_manager /opt/ros/indigo/share/pr2_controller_manager/test/controller_plugin.xml
pr2_calibration_controllers /opt/ros/indigo/share/pr2_calibration_controllers/controller_plugins.xml
pr2_mechanism_controllers /opt/ros/indigo/share/pr2_mechanism_controllers/controller_plugins.xml
ethercat_trigger_controllers /opt/ros/indigo/share/ethercat_trigger_controllers/controller_plugins.xml
robot_mechanism_controllers /opt/ros/indigo/share/robot_mechanism_controllers/controller_plugins.xml
sample_controller /home/robot/catkin_ws/src/sample_controller/controller_plugins.xml
my_controller_pkg /home/robot/catkin_ws/src/my_controller_pkg/controller_plugins.xml
```

Figure 3: List of controllers in the system

Step 9 – Writing controller configuration file

After proper installation of the controller, we can configure and run it. The first procedure is to create the configuration file of the controller that consists of the controller type, joint name, joint limits, and so on. The configuration file is saved as a YAML file that has to be saved inside the package. We are creating a YAML file with a name of `my_controller.yaml` and the definition is given as follows:

```
my_controller_name:
  type: my_controller_pkg/MyControllerPlugin
  joint_name: r_shoulder_pan_joint
```

Step 10 – Writing launch file for the controller

The joint assigned for showing the working of this controller is `r_should_pan_joint` of the robot PR2. After creating the YAML file, we can create a launch file inside the launch folder, which can load the controller configuration file and run the controller. The launch file is called `my_controller.launch`, which is given as follows:

```
<launch>
  <rosparam file="$(find my_controller_pkg)/my_controller.yaml"
  command="load" />

  <!--We can use spawner tool to start running the custom controller -->
  <node pkg="pr2_controller_manager" type="spawner" args="my_
  controller_name" name="my_controller_spawner" />
</launch>
```

Step 11 – Running controller along with PR2 simulation in Gazebo

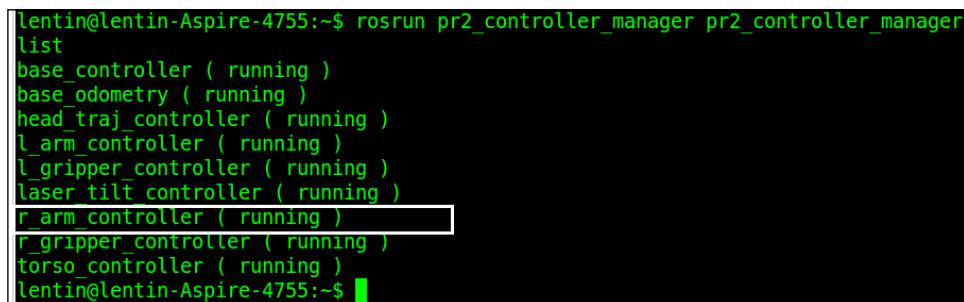
After creating the controller launch files, we have to test it on PR2. We can launch the PR2 robot simulation using following command:

```
$ rosrun pr2_gazebo pr2_empty_world.launch
```

When we launch the PR2 simulation, all controllers associated with PR2 also get started. The purpose of our controller is to move the `r_shoulder_pan_joint` of PR2. If there are existing controllers handling this same joint, our controller can't work properly. To avoid this situation, we need to stop the controller that is handling the right arm of PR2. The following command tells you which are the controllers that are associated with PR2:

```
$ rosrun pr2_controller_manager pr2_controller_manager list
```

The output of this command is given as follows:



```
[lentin@lentin-Aspire-4755:~$ rosrun pr2_controller_manager pr2_controller_manager
list
base_controller ( running )
base_odometry ( running )
head_traj_controller ( running )
l_arm_controller ( running )
l_gripper_controller ( running )
laser_tilt_controller ( running )
r_arm_controller ( running ) [highlight]
r_gripper_controller ( running )
torso_controller ( running )
lentin@lentin-Aspire-4755:~$ ]
```

Figure 4: Running status of PR2 controllers

Stop the `r_arm_controller` using the following command:

```
$ rosrun pr2_controller_manager pr2_controller_manager stop
r_arm_controller
```

After stopping this controller, we can start our own controller using the following command:

```
$ rosrun my_controller_pkg my_controller.launch
```

We can see the right arm of PR2 start moving, and a screenshot of the PR2 pose is given in the following:

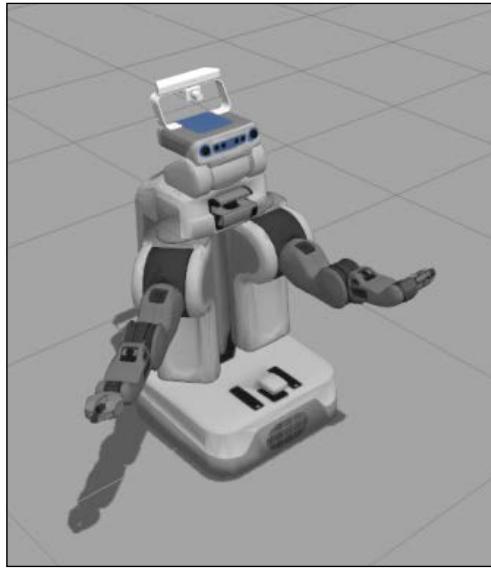


Figure 5: PR2 right hand joint working using our controller

Understanding ros_control packages

In the preceding section, we discussed the `pr2_mechanism` packages which build the controllers for PR2. These packages are exclusively designed for PR2, but they will work in robots that are similar to PR2.

To make these packages more generic to all the robots, the `pr2_mechanism` packages were rewritten and formed a new set of packages called `ros_control` (http://wiki.ros.org/ros_control).

The `ros_control` implements standard set of generic controllers such as `effort_controllers`, `joint_state_controllers`, `position_controllers`, and `velocity_controllers` for any kind of robots.

We have already used these ROS controllers from `ros_control` in *Chapter 3, Simulating Robots Using ROS and Gazebo*. The `ros_control` is still in development. The building procedure of the controllers is almost similar to PR2 controllers.

You can go through the available wiki page of `ros_control` for building new controls at https://github.com/ros-controls/ros_control/wiki.

You will get a sample controller implementation using `ros_control` from the `chapter_6_codes/sample_ros_controller` folder.

Understanding ROS visualization tool (RViz) and its plugins

The RViz tool is an official 3D visualization tool of ROS. Almost all kinds of data from sensors can be viewed through this tool. RViz will be installed along with the ROS desktop full installation. Let's launch RViz and see the basic components present in RViz:

- Start `roscore`
`$ roscore`
- Start RViz
`$ rosrun rviz rviz`

The important sections of the RViz GUI are marked and the uses of each section are given as follows:

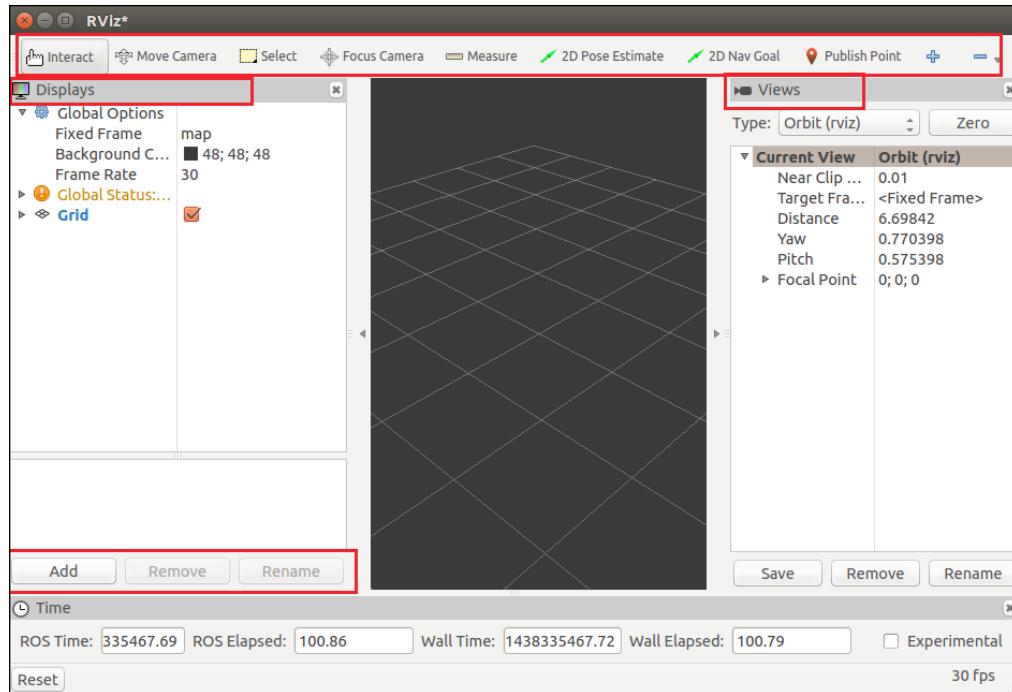


Figure 6: RViz and its toolbars

Displays panel

The panel on the left side of the RViz is called **Displays** panel. The Displays panel contains a list of display plugins of RViz and its properties. The main use of display plugins is to visualize different types of ROS messages, mainly sensor data in the RViz 3D viewport. There are lots of display plugins already present in RViz for viewing images from camera, for viewing 3D point cloud, LaserScan, robot model, Tf, and so on. Plugins can be added by pressing the **Add** button on the left panel. We can also write our own display plugin and add it there. The detail of tutorials for writing a display plugin on RViz is available at http://docs.ros.org/jade/api/rviz_plugin_tutorials/html/display_pluginTutorial.html.

RViz toolbar

There are set of tools present in the RViz toolbar for manipulating the 3D viewport. The toolbar is present on the top of RViz. There are tools present for interacting with the robot model, modifying camera view, giving navigation goals, and giving robot 2D pose estimations. We can add our own custom tools on the toolbar in the form of plugins. One of the official tutorials for building tool plugins is available at http://docs.ros.org/jade/api/rviz_plugin_tutorials/html/tool_pluginTutorial.html.

Views

The **Views** panel is placed on the right side of the RViz. Using Views panel, we can save different views of the 3D viewport and switch to each view by loading the saved configuration.

Time panel

The **Time** panel displays the simulator time elapsed and is mainly useful if there is a simulator running along with RViz. We can also reset to the RViz initial setting using this panel.

Dockable panels

The above toolbar and panels belong to dockable panels. We can create our own dockable panels as a RViz plugin. We are going to create a dockable panel that is having an RViz plugin for robot teleoperation.

Writing a RViz plugin for teleoperation

In this chapter, we design a teleoperation commander in which we can manually enter the teleoperation topic, linear velocity, and angular velocity, as shown in the following:

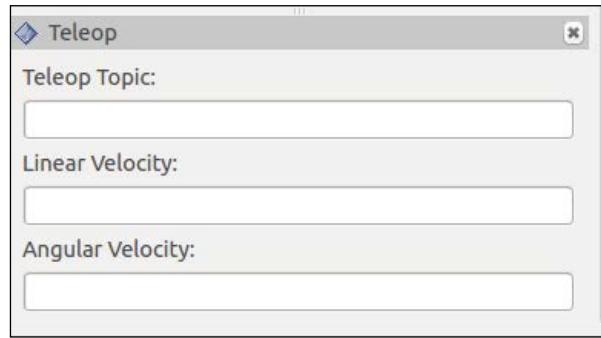


Figure 7: RViz teleop plugin

The following is a detailed procedure to build this plugin.

Methodology of building RViz plugin

Before starting to build this plugin, we should know how to do it. The standard method to build a ROS plugin is applicable for this plugin too. The difference is that the RViz plugin is GUI based. The RViz is written using a GUI framework called **Qt**, so we need to create a GUI in Qt, and using Qt APIs, we have get the GUI values and send them to the ROS system.

The following steps describe how this teleoperation RViz plugin is going to work:

- The dockable panel will have a Qt GUI interface and the user can input the topic, linear velocity, and angular velocity of teleoperation from the GUI.
- Collect the user input from GUI using Qt signals/slots and publish the values using the ROS subscribe and publish method. (The Qt signals and slots are a trigger-invocation technique available in Qt. When a signal/trigger is generated by a GUI field, it can invoke a slot or function like a callback mechanism.)
- Here also, we can use the same procedure to build a plugin like we discussed earlier.

Now we can see the step-by-step procedure to build this plugin as follows:

Step 1 – Creating RViz plugin package

Let's create a new package for creating the teleop plugin:

```
$ catkin_create_pkg rviz_telop_commander roscpp rviz std_msgs
```

Or, you can use the existing package from the following location: chapter_6_codes/rviz_telop_commander.

The package is mainly dependent on the `rviz` package. RViz is built using Qt libraries, so we don't need to include additional Qt libraries in the package.

Step 2 – Creating RViz plugin header file

Let's create a new header inside the `src` folder called `teleop_pad.h`. You will get this source code from the existing package. This header file consists of the class and methods declaration for the plugin.

The following is the explanation of this header file:

```
#include <ros/ros.h>
#include <ros/console.h>
#include <rviz/panel.h>
```

The preceding is the header file required to build this plugin; we need ROS headers for publishing `teleop` topic and `<rviz/panel.h>` for getting the base class of the RViz panel for creating a new panel:

```
class TeleopPanel: public rviz::Panel
{
```

This is a plugin class and is inherited from the `rviz::Panel` base class:

```
Q_OBJECT
public:
```

This class is using Qt signal and slots, and it's also a subclass of `QObject` in Qt. In that case, we should use `Q_OBJECT` macro:

```
TeleopPanel( QWidget* parent = 0 );
```

This is the constructor of the `TeleopPanel()` class and we are initializing a `QWidget` class to 0. We are using the `QWidget` instance inside the `TeleopPanel` class for implementing the GUI of the teleop plugin:

```
virtual void load( const rviz::Config& config );
virtual void save( rviz::Config config ) const;
```

The following is the overriding of `rviz::Panel` functions for saving and loading the RViz config file:

```
public Q_SLOTS:
```

After this line, we can define some public Qt slots:

```
void setTopic( const QString& topic );
```

When we enter the topic name in the GUI and press *Enter*, this slot will be called and will create topic publisher on the given name:

```
protected Q_SLOTS:  
void sendVel();  
void update_Linear_Velocity();  
void update_Angular_Velocity();  
void updateTopic();
```

These are the protected slots for sending velocity, updating linear velocity and angular velocity, and updating the topic name, when we change the name of the existing topic:

```
QLineEdit* output_topic_editor_;  
QLineEdit* output_topic_editor_1;  
QLineEdit* output_topic_editor_2;
```

We are creating Qt `LineEdit` object to create three text fields in the plugin to receive: topic name, linear velocity, and angular velocity.

```
ros::Publisher velocity_publisher_;  
ros::NodeHandle nh_;
```

These are the publisher object and the `Nodehandle` object for publishing topics and handling a ROS node.

Step 3 – Creating RViz plugin definition

In this step, we will create the main C++ file that contains the definition of the plugin. The file is `teleop_pad.cpp`, and you will get it from package `src` folder.

The main responsibilities of this file are as follows:

- It acts as a container for Qt GUI element such as `QLineEdit` to accept text entries
- Publishes the command velocity using ROS publisher
- Saves and restores the RViz config files

The following is the explanation of each section of the code:

```
TeleopPanel::TeleopPanel( QWidget* parent )
    : rviz::Panel( parent )
    , linear_velocity_( 0 )
    , angular_velocity_( 0 )
{
```

This is the constructor and initialize `rviz::Panel` with `QWidget`, setting linear and angular velocity as 0:

```
QVBoxLayout* topic_layout = new QVBoxLayout;
topic_layout->addWidget( new QLabel( "Teleop Topic:" ) );
output_topic_editor_ = new QLineEdit;
topic_layout->addWidget( output_topic_editor_ );
```

This will add a new `QLineEdit` widget on the panel for handling the topic name. Similarly, two other `QLineEdit` widgets handle linear velocity and angular velocity.

```
QTimer* output_timer = new QTimer( this );
```

This will create a `Qt timer` object for updating a function that is publishing the velocity topic:

```
connect( output_topic_editor_, SIGNAL( editingFinished() ), this,
SLOT( updateTopic() ) );
connect( output_topic_editor_, SIGNAL( editingFinished() ), this,
SLOT( updateTopic() ) );

connect( output_topic_editor_1, SIGNAL( editingFinished() ), this,
SLOT( update_Linear_Velocity() ) );

connect( output_topic_editor_2, SIGNAL( editingFinished() ), this,
SLOT( update_Angular_Velocity() ) );
```

This will connect Qt signal to the slots. Here the signal is triggered when `editingFinished()` return true and the slot here is `updateTopic()`. When the editing inside a Qt `LineEdit` is finished by pressing *Enter* key, the signal will trigger and the corresponding slot will execute. Here this slot will set the topic name, angular velocity, and linear velocity value from the text field of the plugin:

```
connect( output_timer, SIGNAL( timeout() ), this, SLOT( sendVel()
) );
output_timer->start( 100 );
```

These lines generate a signal when the Qt timer timesout. The timer will timeout in each 100 ms and execute a slot called `sendVel()`, which will publish the velocity topic.

We can see the definition of each slot after this section. These codes are self-explanatory and finally we can see the following code to export it as a plugin:

```
#include <pluginlib/class_list_macros.h>
PLUGINLIB_EXPORT_CLASS(rviz_telop_commander::TeleopPanel,
rviz::Panel )
```

Step 4 – Creating plugin description file

The definition of `plugin_description.xml` is given as follows:

```
<library path="lib/librviz_telop_commander">
  <class name="rviz_telop_commander/Teleop"
        type="rviz_telop_commander::TeleopPanel"
        base_class_type="rviz::Panel">
    <description>
      A panel widget allowing simple diff-drive style robot base
      control.
    </description>
  </class>
</library>
```

Step 5 – Adding export tags in package.xml

We have to update the `package.xml` file for including the plugin description. The following is the update of `package.xml`:

```
<export>
  <rviz plugin="${prefix}/plugin_description.xml"/>
</export>
```

Step 6 – Editing CMakeLists.txt

We need to add extra lines in the `CMakeLists.txt` definition as given in the following:

```
## This plugin includes Qt widgets, so we must include Qt like so:
find_package(Qt4 COMPONENTS QtCore QtGui REQUIRED)
include(${QT_USE_FILE})

## I prefer the Qt signals and slots to avoid defining "emit",
## "slots",
## etc because they can conflict with boost signals, so define QT_NO_
KEYWORDS here.
```

```

add_definitions(-DQT_NO_KEYWORDS)

## Here we specify which header files need to be run through "moc",
## Qt's meta-object compiler.

qt4_wrap_cpp(MOC_FILES
    src/teleop_pad.h
)

set(SOURCE_FILES
    src/teleop_pad.cpp
    ${MOC_FILES}
)
add_library(${PROJECT_NAME} ${SOURCE_FILES})
target_link_libraries(${PROJECT_NAME} ${QT_LIBRARIES} ${catkin_LIBRARIES})

```

You will get the complete CMakeLists.txt from chapter_6_codes/rviz_telop_commander.

Step 7 – Building and loading plugins

After creating these files, build a package using catkin_make. If the build is successful, we can load the plugin in RViz itself. Take RViz and load the panel by going to **Menu Panel | Add New Panel**; we will get a panel like the following:

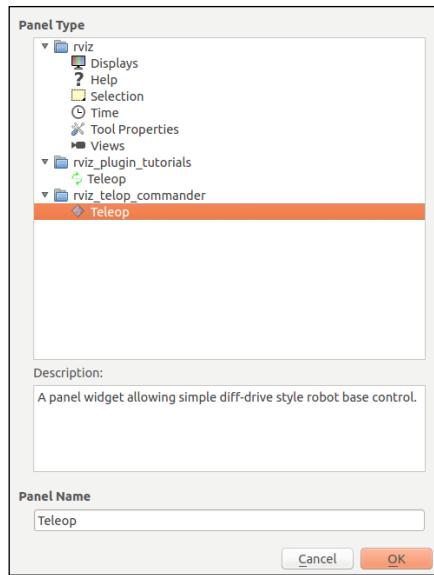


Figure 8: Loading teleop node from RViz

If we load the **Teleop** plugin from the list, we will get a panel like the following:

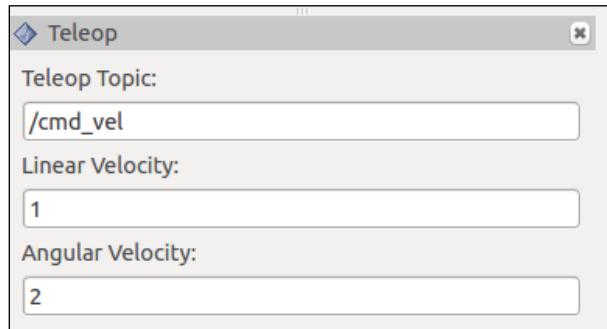


Figure 9: Loading teleop node from RViz

We can put the **Teleop Topic** name and values inside the **Linear Velocity** and **Angular Velocity** and we can echo the **Teleop Topic** and get the topic values like the following:

```
linear:  
  x: 1.0  
  y: 0.0  
  z: 0.0  
angular:  
  x: 0.0  
  y: 0.0  
  z: 2.0  
--  
linear:  
  x: 1.0  
  y: 0.0  
  z: 0.0  
angular:  
  x: 0.0  
  y: 0.0  
  z: 2.0  
--
```

Figure 10: Twist commands from RViz teleop plugin

Questions

1. What are the list of packages needed for writing a real-time controller in ROS?
2. What are the different processes happening inside a ROS controller?
3. What are the main functions of the PR2 mechanism model?
4. What are the different types of RViz plugins?

Summary

In this chapter, we discussed creating plugins for the ROS visualization tool (RViz) and writing basic ROS controllers. We have already worked with default controllers in ROS, and in this chapter, we developed a custom controller for moving joints. After building and testing the controller, we looked at RViz plugins. We created a new RViz panel for teleoperation. We can manually enter the topic name; we need the twist messages and to enter the linear and angular velocity in the panel. This panel is useful for controlling robots without starting another teleoperation node. In the next chapter, we will discuss interfacing of I/O boards and running ROS in embedded systems.

7

Interfacing I/O Boards, Sensors, and Actuators to ROS

In the last two chapters, we discussed different kinds of plugin frameworks that are used in ROS. In this chapter, we are going to discuss interfacing of some hardware components, such as sensors and actuators to ROS. We will see interfacing of sensors using I/O boards such as Arduino, Raspberry Pi, and Odroid-C1 to ROS, and also discuss interfacing of smart actuators such as Dynamixel to ROS. Following is the detailed list of topics we are going to cover in this chapter:

- Understanding the Arduino-ROS interface
- Setting up the Arduino-ROS interface packages
- Arduino-ROS ,example – Chatter and Talker
- Arduino-ROS , example – blink LED and push button
- Arduino-ROS , example – Accelerometer ADXL 335
- Arduino-ROS, example – ultrasonic distance sensor
- Arduino-ROS, example – Odometry Publisher
- Interfacing a non-Arduino board to ROS
- Setting ROS on Odroid-C1 and Raspberry Pi 2
- Working with Raspberry Pi and Odroid GPIOs using ROS
- Interfacing Dynamixel actuators to ROS

Understanding the Arduino–ROS interface

Let's see what an Arduino is first. Arduino is one of the most popular open source I/O boards in the market. The easiness in programmability and the cost effectiveness of the hardware have made Arduino a big success. Most of the Arduino boards are powered by Atmel microcontrollers, which are available from 8-bit to 32-bit and clock speed from 8 MHz to 84 MHz. Arduino can be used for quick prototyping of robots and we can even use it for products as well. The main applications of Arduino in robotics are interfacing sensors and actuators, and communicating with PC for receiving high level commands and sending sensor values to PC using the UART protocol.

There are different varieties of Arduino available in the market. Selecting one board for our purpose will be dependent on the nature of our robotic application. Let's see some boards which we can use for beginners, intermediate, and high end users.

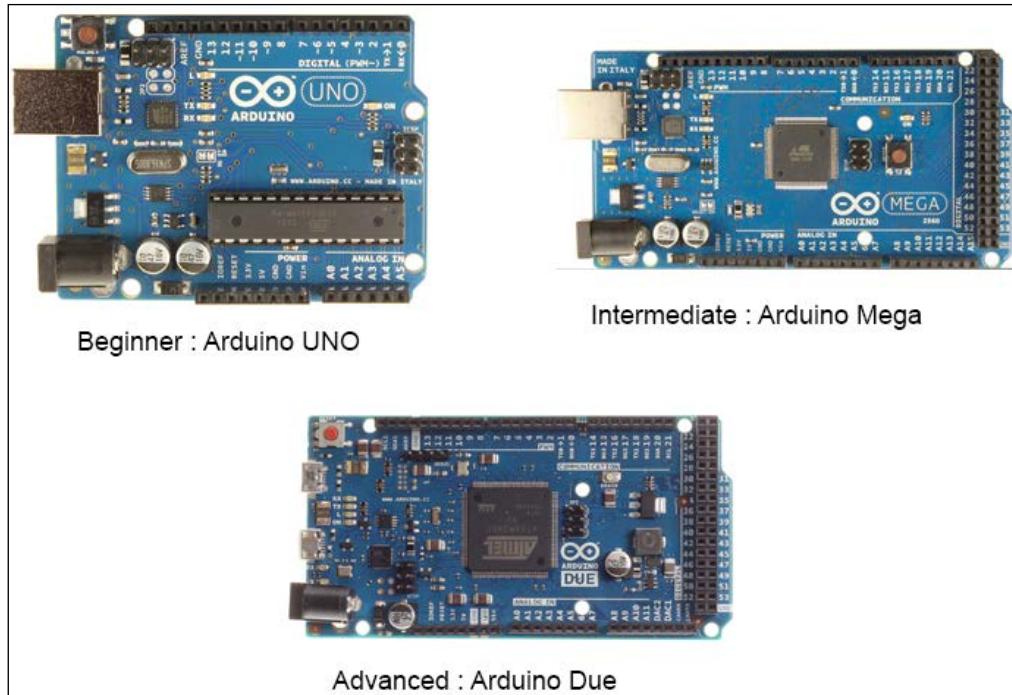


Figure 1 : Different versions of Arduino board

We will look at each Arduino board specification in brief and see where it can be deployed.

Boards	Arduino UNO	Arduino Mega 2560	Arduino Due
Processor	ATmega328P	ATmega2560	ATSAM3X8E
Operating/Input Voltage	5V / 7-12 V	5V/ 7-12V	3.3V / 7 - 12 V
CPU Speed	16 MHz	16 MHz	84 MHz
Analog In/Out	6/0	16/0	12/2
Digital IO/PWM	14/6	54/15	54/12
EEPROM[KB]	1	4	-
SRAM [KB]	2	8	96
Flash [KB]	32	256	512
USB	Regular	Regular	2 Micro
UART	1	4	4
Application	Basic robotics and sensor interfacing	Intermediate robotic application level application	High end robotics application

Let's see how to interface Arduino to ROS.

What is the Arduino–ROS interface?

Most of the communication between PC and I/O boards in robots will be through UART protocol. When both the devices communicate with each other, there should be some program in both the sides which can translate the serial commands from each of these devices. We can implement our own logic to receive and transmit the data from board to PC and vice versa. The interfacing code can be different in each I/O board because there are no standard libraries to do this communication.

The Arduino-ROS interface is a standard way of communication between the Arduino boards and PC. Currently, this interface is exclusive for Arduino. We may need to write custom nodes to interface other I/O boards.

We can use the similar C++ APIs of ROS used in PC in Arduino IDE also, for programming the Arduino board. Detailed information about the interfacing package follows.

Understanding the rosserial package in ROS

The rosserial package is a set of standardized communication protocols implemented for communicating from ROS to character devices such as serial ports, and sockets, and vice versa. The rosserial protocol can convert the standard ROS messages and services data types to embedded device equivalent data types. It also implements multithread support by multiplexing the serial data from a character device. The serial data is sent as data packets by adding header and tail bytes on the packet. The packet representation is shown next:

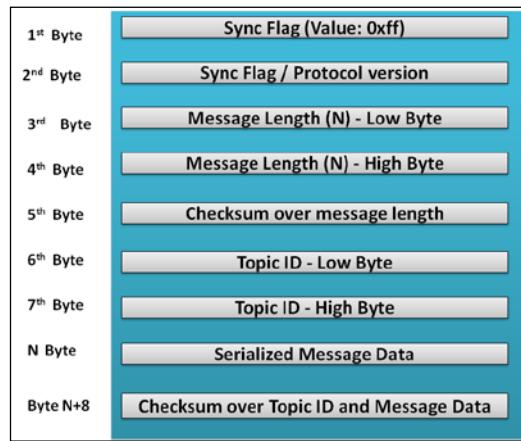


Figure 2 : rosserial packet representation

The function of each byte follows:

- **Sync Flag:** This is the first byte of the packet, which is always 0xff
- **Sync Flag/Protocol version:** This byte was 0xff on ROS Groovy and after that it is set to 0xfe
- **Message Length:** This is the length of the packet
- **Checksum over message length:** This is the checksum of length for finding packet corruption
- **Topic ID:** This is the ID allocated for each topic; the range 0-100 is allocated for the system related functionalities
- **Serialized Message data:** This is the data associated with each topic
- **Checksum of Topic ID and data:** This is the checksum for topic and its serial data for finding packet corruption

The checksum of length is computed using the following equation:

$$\text{Checksum} = 255 - ((\text{Topic ID Low Byte} + \text{Topic ID High Byte} + \dots \text{data byte values}) \% 256)$$

The ROS client libraries, such as `roscpp`, `rospy`, and `roslisp`, enable us to develop ROS nodes which can run from various devices. One of the ports of the ROS clients which enables us to run a ROS node from the embedded devices such as Arduino and embedded Linux based boards, is called the `rosserial_client` library. Using the `rosserial_client` libraries, we can develop the ROS nodes from Arduino, embedded Linux platforms, and windows. Following is the list of `rosserial_client` libraries for each of these platforms:

- `rosserial_arduino`: This `rosserial_client` works on Arduino platforms such as Arduino UNO and Leonardo, and also works in Mega series and Due series for advance robotic projects
- `rosserial_embeddedlinux`: This client supports embedded Linux platforms such as VEXPro, Chumby alarm clock, WRT54GL router, and so on
- `rosserial_windows`: This is a client for Windows platform

In the PC side, we need some other packages to decode the serial message and convert to exact topics from the `rosserial_client` libraries. The following packages help in decoding the serial data:

- `rosserial_python`: This is the recommended PC side node for handling serial data from a device. The receiving node is completely written in Python.
- `rosserial_server`: This is a C++ implementation of `rosserial` in the PC side. The inbuilt functionalities are less compared to `rosserial_python`, but it can be used for high performance applications.
- `rosserial_java`: This is a JAVA based implementation of `rosserial`, but not actively supported now. It is mainly used for communicating with android devices.

We are mainly focusing on running the ROS nodes from Arduino. First we will see how to setup the `rosserial` packages and then discuss how to setup the `rosserial_arduino` client in Arduino IDE.

Installing rosserial packages on Ubuntu 14.04/15.04

We can install the rosserial packages on Ubuntu using the following commands:

1. Installing the rosserial package binaries using apt-get:

- In Indigo:

```
$ sudo apt-get install ros-indigo-rosserial ros-indigo-  
rosserial-arduino ros-indigo-rosserial-server
```

- In Jade:

```
$ sudo apt-get install ros-jade-rosserial ros-jade-  
rosserial-arduino ros-jade-rosserial-server
```

2. For installing the rosserial_client library called ros_lib in Arduino, we have to download the latest Arduino IDE for Linux 32/64 bit. Following is the link for downloading Arduino IDE:

```
https://www.arduino.cc/en/main/software
```

Here we download the Linux 64 bit version and copy the Arduino IDE folder to the Ubuntu desktop.

3. Arduino requires JAVA runtime support to run it. If it is not installed, we can install it using the following command:

```
$ sudo apt-get install java-common
```

4. After installing JAVA runtime, we can switch the arduino folder using the following command:

```
$ cd ~/Desktop/arduino-1.6.5
```

5. Start Arduino using the following command:

```
$ ./arduino
```

Shown next is the Arduino IDE window:

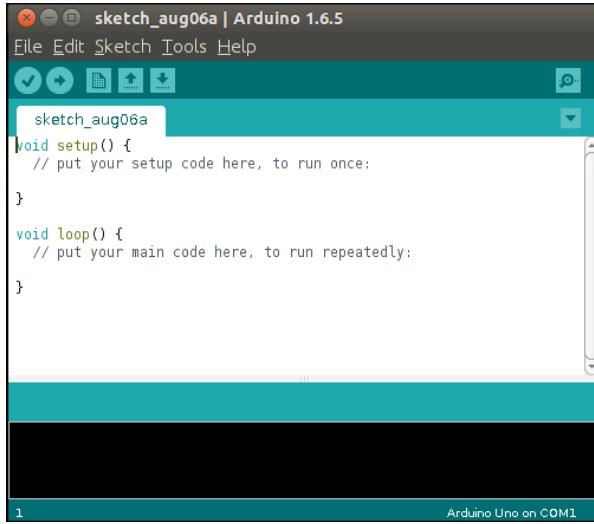


Figure 3 : Arduino IDE

6. Go to **File | Preference** for configuring the sketchbook folder of Arduino. Arduino IDE stores the sketches to this location. We created a folder called **Arduino1** in the user home folder and set this folder as the sketchbook location.

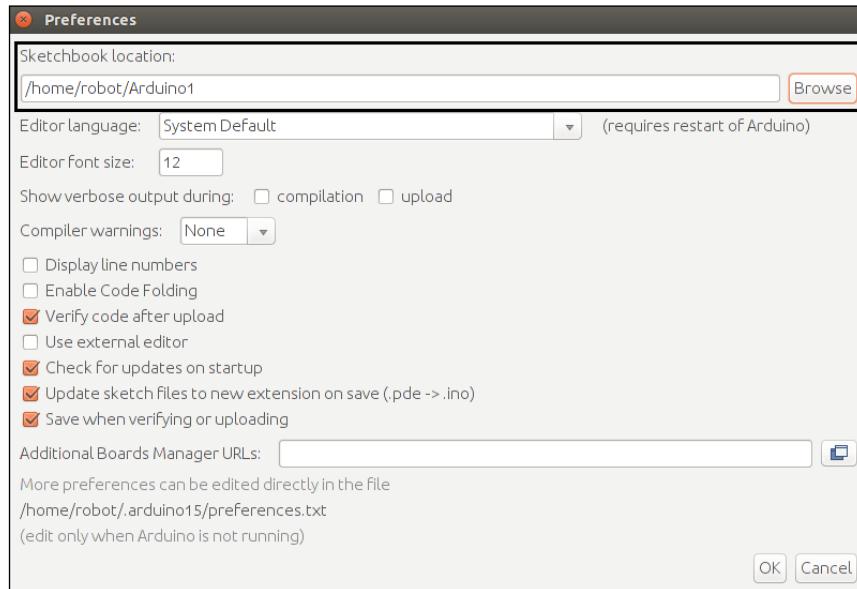


Figure 4 : Preference of Arduino IDE

7. We can see a folder called `libraries` inside the `Arduino1` folder. Switch to this folder using the following command:

```
$ cd ~/Arduino1/libraries/
```

If there is no `libraries` folder, we can create a new one.

8. After switching into this folder, we can generate `ros_lib` using a script called `make_libraries.py`, which is present inside the `rosserial_arduino` package. `ros_lib` is `rosserial_client` for Arduino, which provides the ROS client APIs inside an Arduino IDE environment.

```
$ rosrun rosserial_arduino make_libraries.py .
```

`rosserial_arduino` is ROS client for arduino which can communicate using UART and can publish topics, services, TF, and such others like a ROS node. The `make_libraries.py` script will generate a wrapper of the ROS messages and services which optimized for Arduino data types. These ROS messages and services will convert into Arduino C/C++ code equivalent, as shown next:

- Conversion of ROS messages:

```
ros_package_name/msg/Test.msg  --> ros_package_name::Test
```

- Conversion of ROS services:

```
ros_package_name/srv/Foo.srv  --> ros_package_name::Foo
```

For example, if we include `#include <std_msgs/UInt16.h>`, we can instantiate the `std_msgs::UInt16` number.

If the script `make_libraries.py` works fine, a folder called `ros_lib` will generate inside the `libraries` folder. Restart the Arduino IDE and we will see `ros_lib` examples as follows:

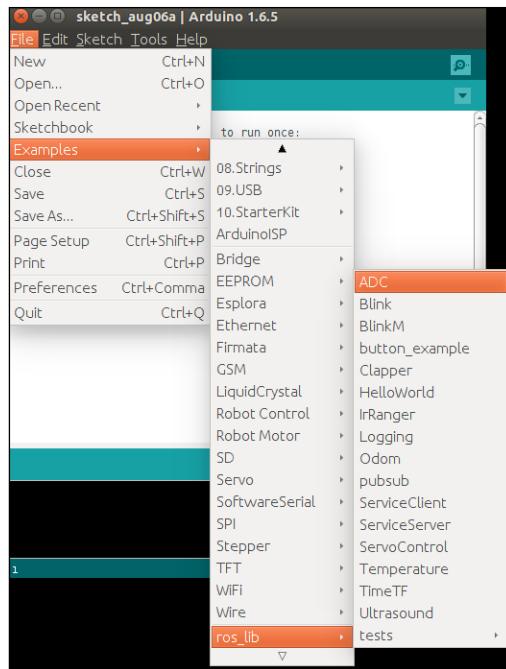


Figure 5 : List of Arduino - ROS examples

We can take any example and make sure that it is building properly to ensure that the `ros_lib` APIs are working fine. The necessary APIs required for building ROS Arduino nodes are discussed next.

Understanding ROS node APIs in Arduino

Following is a basic structure ROS Arduino node. We can see the function of each line of code::

```
#include <ros.h>

ros::NodeHandle nh;

void setup()
{
    nh.initNode();
}

void loop()
{
    nh.spinOnce();
}
```

Creating of `NodeHandle` in Arduino is done using the following line of code:

```
ros::NodeHandle nh;
```

Note that `Nodehandle` should be declared before the `setup()` function, which will give a global scope to the `NodeHandle` instance called `nh`. The initialization of this node is done inside the `setup()` function.

```
nh.initNode();
```

The Arduino `setup()` function will execute only once when the device starts, and note that we can only create one node from a serial device.

Inside the `loop()` function, we have to use the following line of code to execute the ROS callback once:

```
nh.spinOnce();
```

We can create the `Subscriber` and `Publisher` objects in Arduino, similar to the other ROS client libraries. Following are the procedures for defining the subscriber and the publisher.

Here is how we define a subscriber object in Arduino:

```
ros::Subscriber<std_msgs::String> sub("talker", callback);
```

Here we define a subscriber which is subscribing a `String` message, where `callback` is the callback function executing when a `String` message arrives on the `talker` topic. Given next is an example callback for handling the `String` data:

```
std_msgs::String str_msg;

ros::Publisher chatter("chatter", &str_msg);

void callback ( const std_msgs::String& msg) {
    str_msg.data = msg.data;

    chatter.publish( &str_msg );
}
```

Note that the `callback()`, `Subscriber`, and `Publisher` definition will be above the `setup()` function for getting global scope. Here we are receiving `String` data using `const std_msgs::String& msg`.

Following code shows how to define a publisher object in Arduino:

```
ros::Publisher chatter("chatter", &str_msg);
```

This next code shows how we publish the string message:

```
chatter.publish( &str_msg );
```

After defining the publisher and the subscriber, we have to initiate this inside the `setup()` function using the following lines of code:

```
nh.advertise(chatter);  
nh.subscribe(sub);
```

There are ROS APIs for logging from Arduino. Following are the different logging APIs supported:

```
nh.logdebug("Debug Statement");  
nh.loginfo("Program info");  
nh.logwarn("Warnings.");  
nh.logerror("Errors..");  
nh.logfatal("Fatalities!");
```

We can retrieve the current ROS time in Arduino using ROS built-in functions, such as time and duration.

- Current ROS time:

```
ros::Time begin = nh.now();
```

- Convert ROS time in seconds:

```
double secs = nh.now().toSec();
```

- Creating a duration in seconds:

```
ros::Duration ten_seconds(10, 0);
```

ROS – Arduino Publisher and Subscriber example

The first example using Arduino and ROS interface is a chatter and talker interface. Users can send a String message to the talker topic and Arduino will publish the same message in a chatter topic. The following ROS node is implemented for Arduino and we will discuss this example in detail:

```
#include <ros.h>  
#include <std_msgs/String.h>  
  
//Creating Nodehandle  
ros::NodeHandle nh;  
  
//Declaring String variable  
std_msgs::String str_msg;
```

```
//Defining Publisher
ros::Publisher chatter("chatter", &str_msg);
//Defining callback
void callback ( const std_msgs::String& msg) {

    str_msg.data = msg.data;
    chatter.publish( &str_msg );

}

//Defining Subscriber
ros::Subscriber<std_msgs::String> sub("talker", callback);

void setup()
{
    //Initializing node
    nh.initNode();
    //Start advertising and subscribing
    nh.advertise(chatter);
    nh.subscribe(sub);
}

void loop()
{
    nh.spinOnce();
    delay(3);
}
```

We can compile the above code and upload to the Arduino board. After uploading the code, select the desired Arduino board that we are using for this example and the device serial port of the Arduino IDE.

Take **Tools | Boards** to select the board and **Tools | Port** to select the device port name of the board. We are using Arduino Mega for these examples.

After compiling and uploading the code, we can start the ROS bridge nodes in the PC which connects Arduino and the PC using the following command. Ensure that Arduino is already connected to the PC before executing of this command.

```
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

We are using the `rosserial_python` node here as the ROS bridging node. We have to mention the device name and baud-rate as arguments. The default baud-rate of this communication is 57600. We can change the baud-rate according to our application and the usage of `serial_node.py` inside the `rosserial_python` package is given at http://wiki.ros.org/rosserial_python. If the communication between the ROS node and the Arduino node is correct, we will get the following message:

```
lentin@lentin-Aspire-4755:~/Desktop/arduino-1.6.5$ rosrun rosserial_python serial_node.py /dev/ttyACM0
[INFO] [WallTime: 1438880620.972231] ROS Serial Python Node
[INFO] [WallTime: 1438880620.982245] Connecting to /dev/ttyACM0 at 57600 baud
[INFO] [WallTime: 1438880623.117417] Note: publish buffer size is 512 bytes
[INFO] [WallTime: 1438880623.118587] Setup publisher on chatter [std_msgs/String]
[INFO] [WallTime: 1438880623.132048] Note: subscribe buffer size is 512 bytes
[INFO] [WallTime: 1438880623.132745] Setup subscriber on talker [std_msgs/String]
```

Figure 6 : Running the `rosserial_python` node

When `serial_node.py` starts running from the PC, it will send some serial data packets called query packets to get the number of topics, the topic names, and the types of topics which are received from the Arduino node. We have already seen the structure of serial packets which is being used for Arduino ROS communication. Given next is the structure of a query packet which is sent from `serial_node.py` to Arduino:

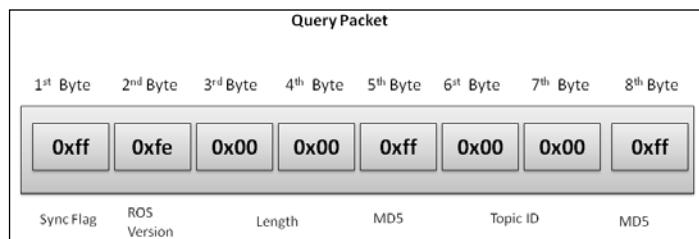


Figure 7 : Structure of Query Packet

The query topic contains fields such as Sync Flag, ROS version, length of the message, MD5 sum, Topic ID, and so on. When the query packet receives on the Arduino, it will reply with a topic info message which contains topic name, type, length, topic data, and so on. Following is a typical response packet from Arduino:

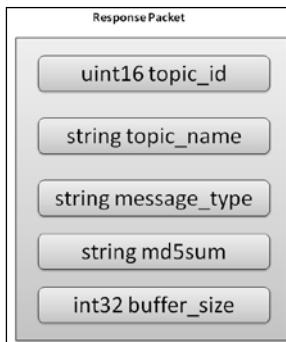


Figure 8 : Structure of Response Packet

If there is no response for the query packet, it will send it again. The synchronization in communication is based on ROS time.

From *Figure 6*, we can see that when we run the `serial_node.py`, the buffer size allocated for publish and subscribe is 512 bytes. The buffer allocation is dependent on the amount of RAM available on each microcontroller that we are working with. Following is a table showing the buffer allocation of each Arduino controller. We can override these settings by changing the `BUFFER_SIZE` macro inside `ros.h`.

AVR Model	Buffer Size	Publishers/Subscribers
ATMEGA 168	150 bytes	6/6
ATMEGA 328P	280 bytes	25/25
All others	512 bytes	25/25

There are also some limitations in the `float64` data type of ROS in Arduino, it will truncate to 32-bit. Also, when we use string data types, use the unsigned char pointer for saving memory.

After running `serial_node.py`, we will get the list of topics using the following command:

```
$ rostopic list
```

We can see that topics such as `chatter` and `talker` are being generated. We can simply publish a message to the `talker` topic using the following command:

```
$ rostopic pub -r 5 talker std_msgs/String "Hello World"
```

It will publish the "Hello World" message with a rate of 5.

We can echo the chatter topic and we will get the same message as we published:

```
$ rostopic echo /chatter
```

The screenshot of this command is shown next:

Figure 9 : Echoing /chatter topic

Arduino-ROS, example – blink LED and push button

In this example, we can interface the LED and push button to Arduino and control using ROS. When the push button is pressed, the Arduino node sends a True value to a topic called pushed, and at the same time, it switches on the LED which is on the Arduino board. The following shows the circuit for doing this example:

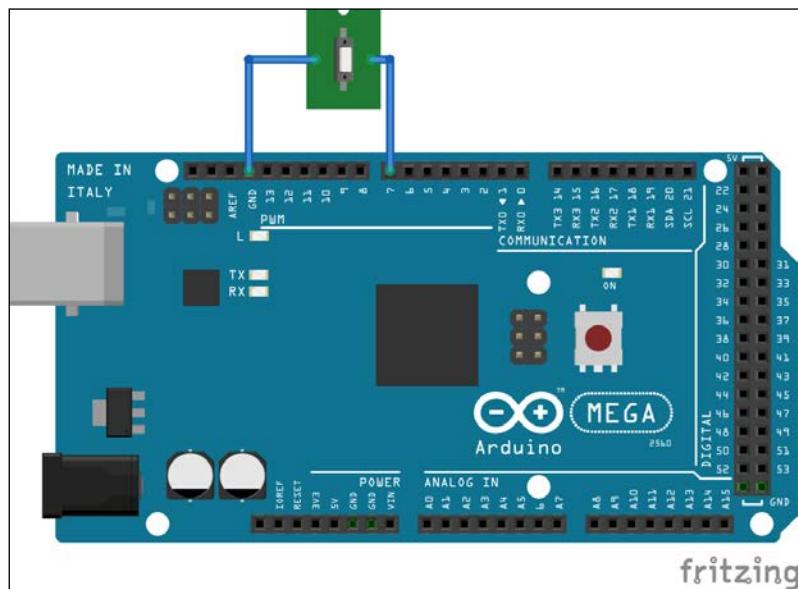


Figure 10 : Interfacing the push button to Arduino

```
/*
 * Button Example for Rosserial
 */

#include <ros.h>
#include <std_msgs/Bool.h>

//Nodehandle
ros::NodeHandle nh;

//Boolean message for Push button
std_msgs::Bool pushed_msg;

//Defining Publisher in a topic called pushed
ros::Publisher pub_button("pushed", &pushed_msg);
```

```
//LED and Push button pin definitions
const int button_pin = 7;
const int led_pin = 13;

//Variables to handle debouncing
//https://www.arduino.cc/en/Tutorial/Debounce

bool last_reading;
long last_debounce_time=0;
long debounce_delay=50;
bool published = true;

void setup()
{
    nh.initNode();
    nh.advertise(pub_button);

    //initialize an LED output pin
    //and a input pin for our push button
    pinMode(led_pin, OUTPUT);
    pinMode(button_pin, INPUT);

    //Enable the pullup resistor on the button
    digitalWrite(button_pin, HIGH);

    //The button is a normally button
    last_reading = ! digitalRead(button_pin);
}

void loop()
{
    bool reading = ! digitalRead(button_pin);

    if (last_reading!= reading){
        last_debounce_time = millis();
        published = false;
    }

    //if the button value has not changed for the debounce delay, we
    know its stable
```

```
    if ( !published && (millis() - last_debounce_time) > debounce_
delay) {
    digitalWrite(led_pin, reading);
    pushed_msg.data = reading;
    pub_button.publish(&pushed_msg);
    published = true;
}

last_reading = reading;

nh.spinOnce();
}
```

The preceding code handles the key debouncing and changes the button state only after the button release. The preceding code can upload to Arduino and can interface to ROS using the following command:

- Start roscore:

```
$ roscore
```

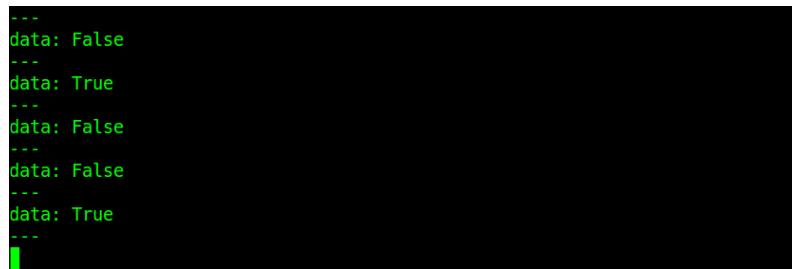
- Start serial_node.py:

```
$ rosrun roserial_python serial_node.py /dev/ttyACM0
```

We can see the button press event by echoing the topic pushed:

```
$ rostopic echo pushed
```

We will get following values when a button is pressed:

A terminal window showing ROS topic output. The text is green on a black background. It shows a sequence of messages on the 'pushed' topic. The messages alternate between 'data: False' and 'data: True', indicating the state of the push button. The output is as follows:

```
...
data: False
...
data: True
...
data: False
...
data: False
...
data: True
...
[REDACTED]
```

Figure 11 : Output of Arduino- Push button

Arduino-ROS, example – Accelerometer ADXL 335

In this example, we are interfacing Accelerometer ADXL 335 to Arduino Mega through ADC pins and plotting the values using the ROS tool called `rqt_plot`.

The following image shows the circuit of the connection between ADXL 335 and Arduino::

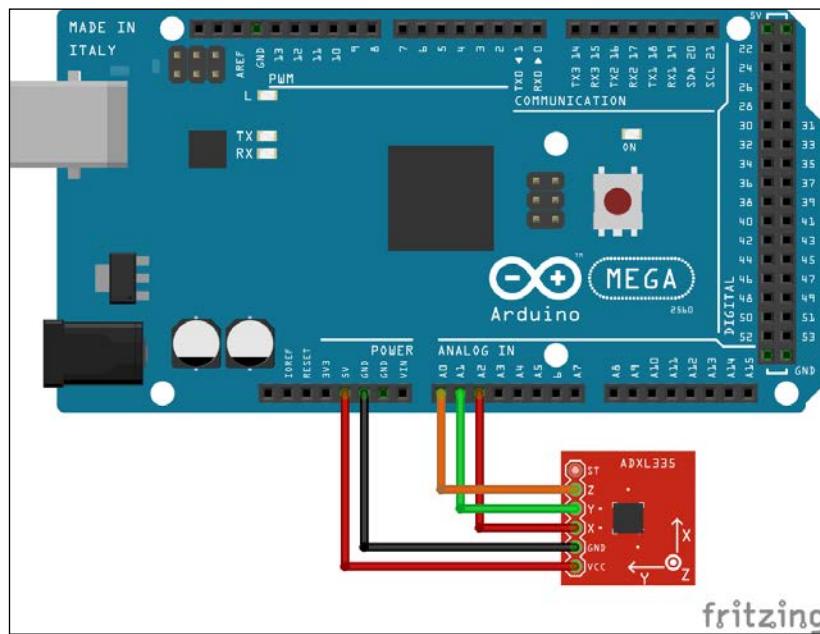


Figure 12 : Interfacing Arduino - ADXL 335

The ADXL 335 is an analog accelerometer. We can simply connect to the ADC port and read the digital value. Following is the embedded code to interface ADXL 335 via Arduino ADC:

```
#if (ARDUINO >= 100)
#include <Arduino.h>
#else
#include <WProgram.h>
#endif
#include <ros.h>
#include <rosserial_arduino/Adc.h>

const int xpin = A2; // x-axis of the accelerometer
const int ypin = A1; // y-axis
```

```
const int zpin = A0;                                // z-axis (only on 3-axis
models)

ros::NodeHandle nh;

//Creating an adc message
rosserial_arduino::Adc adc_msg;

ros::Publisher pub("adc", &adc_msg);

void setup()
{
    nh.initNode();
    nh.advertise(pub);
}

//We average the analog reading to elminate some of the noise
int averageAnalog(int pin){
    int v=0;
    for(int i=0; i<4; i++) v+= analogRead(pin);
    return v/4;
}

void loop()
{
    //Inserting ADC values to ADC message
    adc_msg.adc0 = averageAnalog(xpin);
    adc_msg.adc1 = averageAnalog(ypin);
    adc_msg.adc2 = averageAnalog(zpin);

    pub.publish(&adc_msg);

    nh.spinOnce();

    delay(10);
}
```

The preceding code will publish the ADC values of X, Y, and Z axes in a topic called `/adc`. The code uses the `rosserial_arduino::Adc` message to handle the ADC value. We can plot the values using the `rqt_plot` tool.

Following is the command to plot the three axes values in a single plot:

```
$ rqt_plot adc/adc0 adc/adc1 adc/adc2
```

Next is a screenshot of the plot of the three channels of ADC:

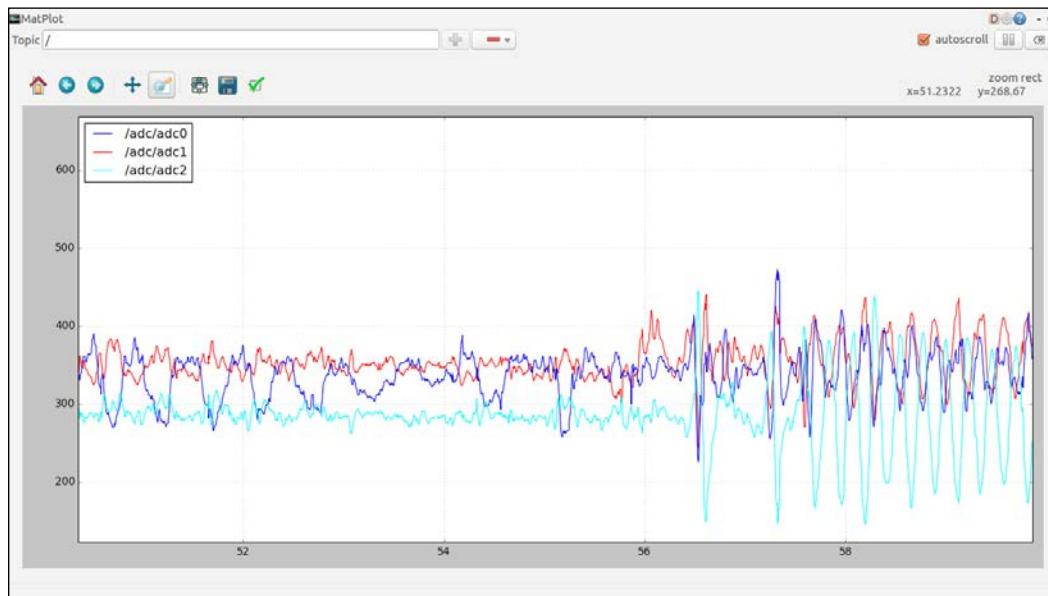


Figure 13 : Plotting ADXL 335 values using `rqt_plot`

Arduino-ROS, example – ultrasonic distance sensor

One of the useful sensors in robots are the range sensors. One of the cheapest range sensor is the ultrasonic distance sensor. The ultrasonic sensor has two pins for handling input and output, called Echo and Trigger. We are using the HC-SR04 ultrasonic distance sensor and the circuit is shown in the following image:

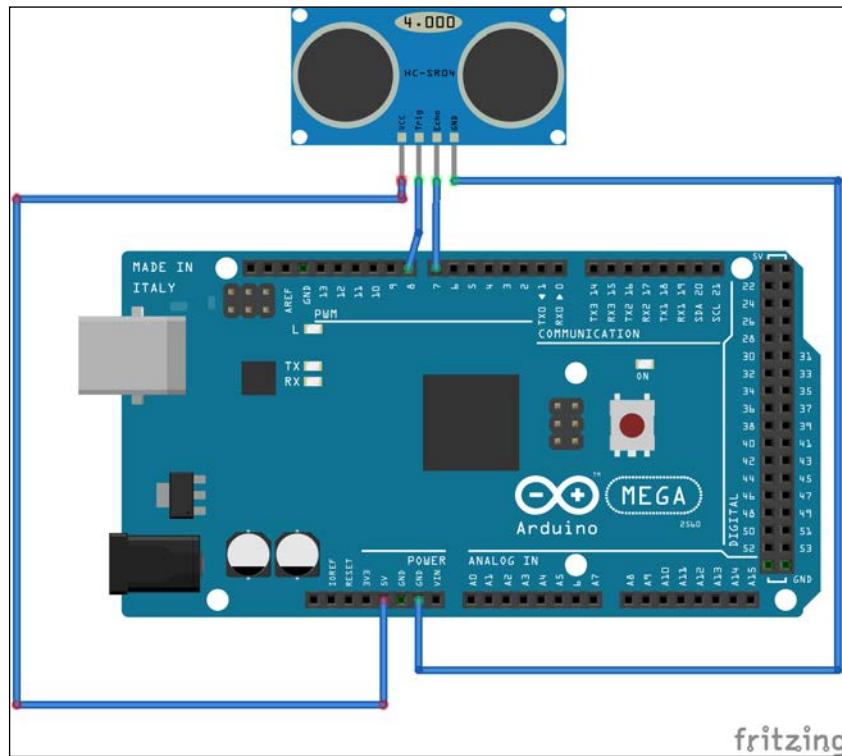


Figure 14 : Plotting ADXL 335 values using rqt_plot

The ultrasonic sound sensor contains two sections: one is the transmitter and the other is the receiver. The working of the ultrasonic distance sensor is, when a trigger pulse of a short duration is applied to the trigger pin of the ultrasonic sensors, the ultrasonic transmitter sends the sound signals to the robot environment. The sound signal sent from the transmitter hits on some obstacles and is reflected back to the sensor. The reflected sound waves are collected by the ultrasonic receiver, generating an output signal which has a relation to the time required to receive the reflected sound signals.

Equations to find distance using the ultrasonic range sensor

Following are the equations used to compute the distance from an ultrasonic range sensor to an obstacle:

$$\text{Distance} = \text{Speed} * \text{Time}/2$$

Speed of sound at sea level = 343 m/s or 34300 cm/s

*Thus, Distance = 17150 * Time (unit cm)*

We can compute the distance to the obstacle using the pulse duration of the output. Following is the code to work with the ultrasonic sound sensor and send value through the ultrasound topic using the range message definition in ROS:

```
#include <ros.h>
#include <ros/time.h>
#include <sensor_msgs/Range.h>

ros::NodeHandle nh;

#define echoPin 7 // Echo Pin
#define trigPin 8 // Trigger Pin

int maximumRange = 200; // Maximum range needed
int minimumRange = 0; // Minimum range needed
long duration, distance; // Duration used to calculate distance

sensor_msgs::Range range_msg;
ros::Publisher pub_range( "/ultrasound", &range_msg );

char frameid[] = "/ultrasound";

void setup() {

    nh.initNode();
    nh.advertise(pub_range);

    range_msg.radiation_type = sensor_msgs::Range::ULTRASOUND;
    range_msg.header.frame_id = frameid;
```

```
range_msg.field_of_view = 0.1; // fake
range_msg.min_range = 0.0;
range_msg.max_range = 60;

pinMode(trigPin, OUTPUT);
pinMode(echoPin, INPUT);

}

float getRange_Ultrasonic() {

    int val = 0;

    for(int i=0; i<4; i++) {

        digitalWrite(trigPin, LOW);
        delayMicroseconds(2);

        digitalWrite(trigPin, HIGH);
        delayMicroseconds(10);

        digitalWrite(trigPin, LOW);
        duration = pulseIn(echoPin, HIGH);

        //Calculate the distance (in cm) based on the speed of sound.
        val += duration;

    }
    return val / 232.8 ;
}

long range_time;

void loop() {
/* The following trigPin/echoPin cycle is used to determine the
   distance of the nearest object by bouncing soundwaves off of it. */

    if ( millis() >= range_time ) {
        int r =0;

        range_msg.range = getRange_Ultrasonic();
        range_msg.header.stamp = nh.now();
        pub_range.publish(&range_msg);
```

```

        range_time = millis() + 50;
    }

nh.spinOnce();

delay(50);
}

```

We can plot the distance value using the following command:

- Start roscore:

```
$ roscore
```

- Start serial_node.py:

```
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

- Plot values using rqt_plot:

```
$ rqt_plot /ultrasound
```

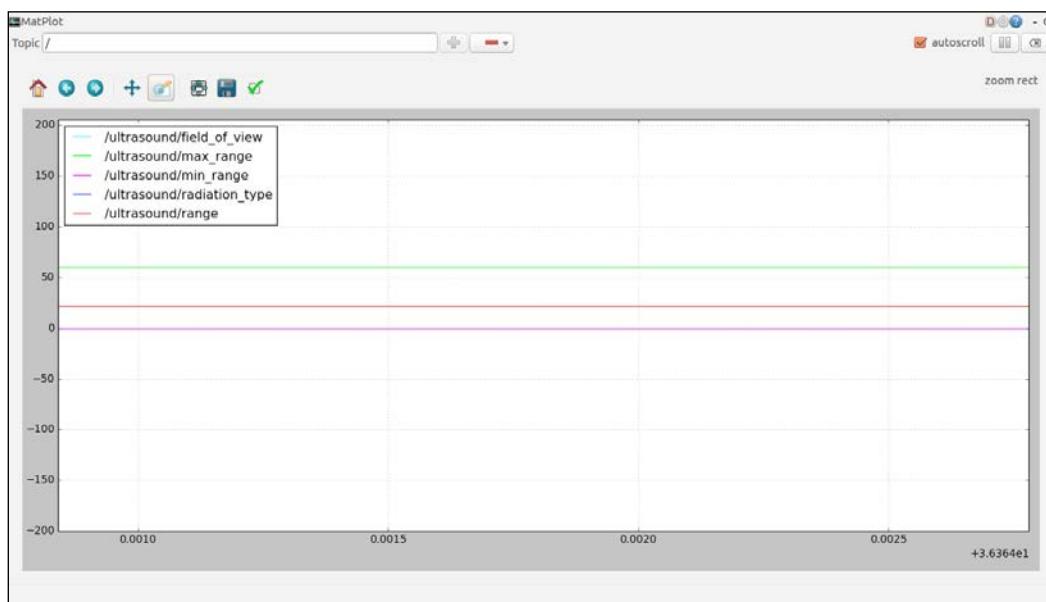


Figure 15 : Plotting ultrasonic sound sensor distance value

The center line indicates the current distance from the sensor. The upper line is the `max_range` and line below is the `minimum range`.

Arduino-ROS, example – Odometry Publisher

In this example, we will see how to send an `odom` message from an Arduino node to a PC. This example can be used in a robot for computing `odom` and send to ROS Navigation stack as the input. The motor encoders can be used for computing `odom` and can send to PC. In this example, we will see how to send `odom` for a robot which is moving in a circle, without taking the motor encoder values.

```
/*
 * rosserial Planar Odometry Example
 */

#include <ros.h>
#include <ros/time.h>
#include <tf/tf.h>
#include <tf/transform_broadcaster.h>

ros::NodeHandle nh;
//Transform broadcaster object
geometry_msgs::TransformStamped t;
tf::TransformBroadcaster broadcaster;

double x = 1.0;
double y = 0.0;
double theta = 1.57;

char base_link[] = "/base_link";
char odom[] = "/odom";

void setup()
{
    nh.initNode();
    broadcaster.init(nh);
}

void loop()
{
    // drive in a circle
    double dx = 0.2;
    double dtheta = 0.18;

    x += cos(theta)*dx*0.1;
    y += sin(theta)*dx*0.1;
    theta += dtheta*0.1;

    if(theta > 3.14)
        theta=-3.14;
```

```

// tf odom->base_link
t.header.frame_id = odom;
t.child_frame_id = base_link;

t.transform.translation.x = x;
t.transform.translation.y = y;

t.transform.rotation = tf::createQuaternionFromYaw(theta);
t.header.stamp = nh.now();

broadcaster.sendTransform(t);
nh.spinOnce();

delay(10);
}

```

After uploading the code, run `roscore` and `rosserial_node.py`. We can view `tf` and `odom` in RViz. Open RViz and view the `tf` as shown next. We will see the `odom` pointer moving in a circle on RViz as follows:

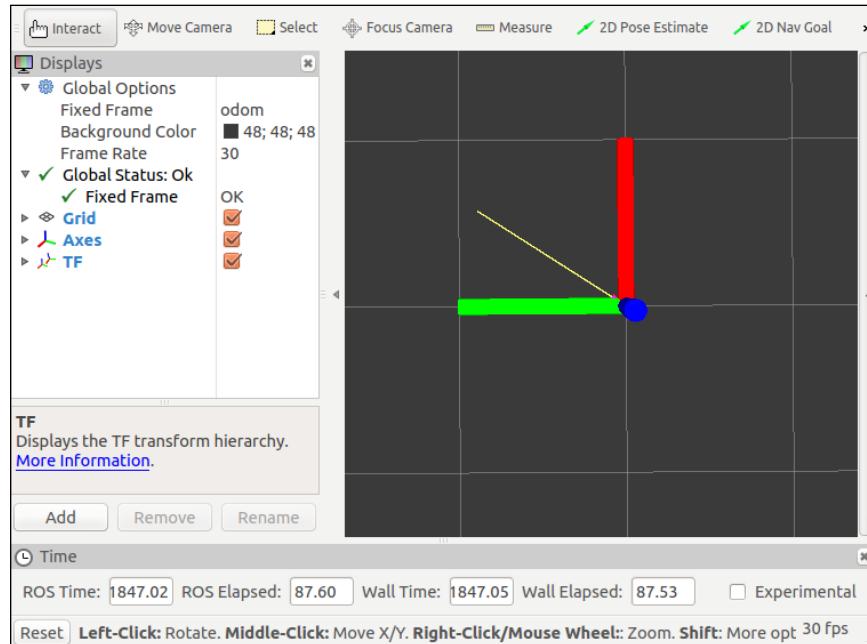


Figure 16 : Visualizing odom data from Arduino

Interfacing Non-Arduino boards to ROS

Arduino boards are commonly used boards in robots but what happens if we want a board which is more powerful than Arduino. In such a case, we may want to write our own driver for the board, which can convert the serial messages into topics.

We will see interfacing of a Non-Arduino board called Tiva C Launchpad to ROS using a Python driver node in *Chapter 9, Building and Interfacing Differential Drive Mobile Robot Hardware in ROS*. This chapter, is about interfacing a real mobile robot to ROS and the robot using Tiva C Launchpad board for its operation.

Setting ROS on Odroid-C1 and Raspberry Pi 2

Odroid-C1 and Raspberry Pi2 are single board computers which have low form factor with a size of a credit card. These single board computers can be installed in robots and we can install ROS on it.

The main specifications comparison of Odroid-C1 and Raspberry Pi2 is shown next:

Device	Odroid-C1	Raspberry Pi 2
CPU	1.5 GHz quad core ARM Cortex-A5 CPU from Amlogic	900 MHz quad core ARM Cortex A7 CPU from Broadcom
GPU	Mali-450 MP2 GPU	VideoCore IV
Memory	1 GB	1 GB
Storage	SD card slot or eMMC module	SD card slot
Connectivity	4 x USB, micro HDMI, Gigabit Ethernet, infra red remote control receiver	4 x USB, HDMI, Ethernet, 3.5mm audio jack
OS	Android, Ubuntu/Linux	Raspbian, Ubuntu/Linux, Windows 10
Connectors	GPIO, SPI, I2C, RTC (Real Time Clock) backup battery connector	Camera interface (CSI), GPIO, SPI, I2C, JTAG
Price	\$35	\$35

Following is an image of the Odroid-C1 board:

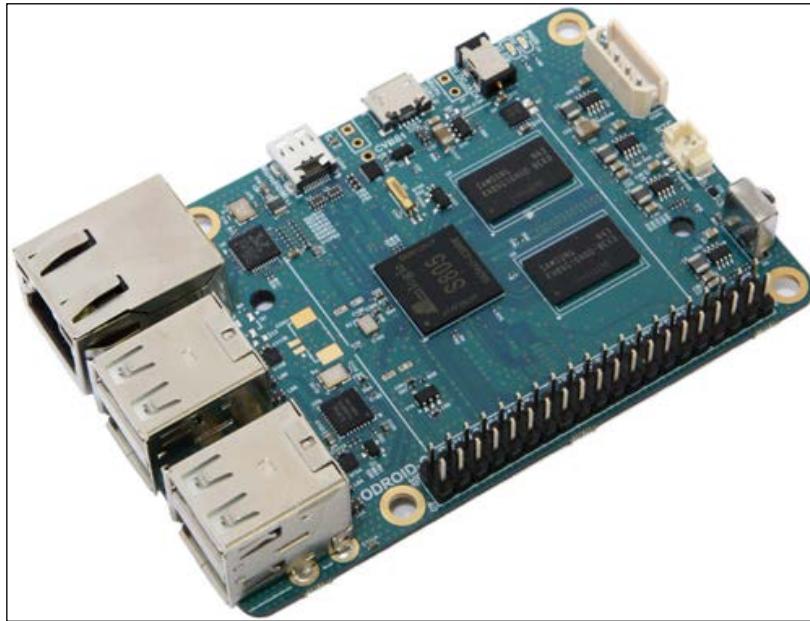


Figure 17 : Odroid-C1 board

The Odroid board is manufactured by a company called **Hard kernel**. The official web page of the Odroid-C1 board is http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141578608433.

The Odroid-C1 is a basic model in the Odroid series. There are more powerful boards as well, such as Odroid-XU4, XU3, and U3. All these boards support ROS.

One of the popular single board computers is Raspberry Pi. The Raspberry Pi boards are manufactured by Raspberry Pi Foundation which is based in the UK. The latest model of Raspberry Pi is Raspberry Pi 2. The official website of Raspberry Pi is <https://www.raspberrypi.org>.

Following is a diagram of Raspberry Pi 2:

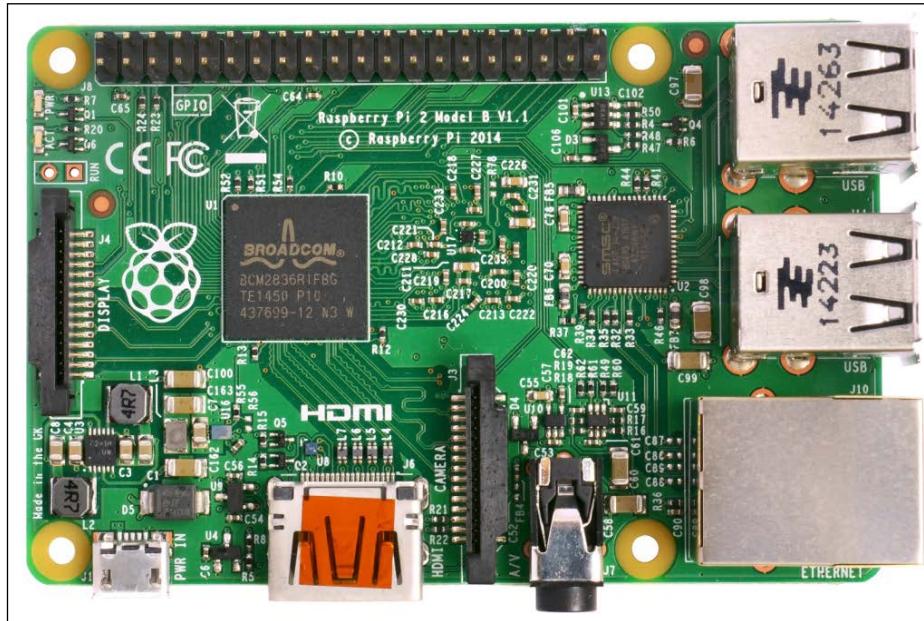


Figure 18 : The Raspberry Pi 2 board

The Odroid GPIO pins and its GPIO handling is much similar to Raspberry Pi 2. We can install Ubuntu and Android on Odroid. There are also unofficial distributions of Linux such as Debian mini, Kali Linux, Arch Linux, and Fedora, and also support libraries such as ROS, OpenCV, PCL, and so on.

For getting ROS on Odroid, we can either install a fresh Ubuntu and install ROS manually or install Ubuntu which is inbuilt with ROS, OpenCV, and PCL.

Installing ROS from the source code and packages will take several hours. For a quick start, we can start with a pre-installed image of Ubuntu with ROS.

The image can be download from <http://forum.odroid.com/viewtopic.php?f=112&t=11994>. This link contains pre-installed images of Ubuntu with ROS, OpenCV, and PCL for Odroid C1.

The list of the other operating systems supported on Odroid-C1 is given on the wiki page of Odroid-C1 at <http://odroid.com/dokuwiki/doku.php?id=en:odroid-c1>.

The official guide of installing ROS on Odroid and Raspberry Pi 2 into their official OS is available at <http://wiki.ros.org/indigo/Installation/UbuntuARM>.

The Raspberry Pi 2 official OS images are given at <https://www.raspberrypi.org/downloads/>.

The official OS supported by Raspberry Pi foundation are Raspbian and Ubuntu. There are unofficial images based on this OS which has ROS pre-installed on them. The following link has some of the Raspberry Pi 2 images which have ROS preinstalled:

<http://www.mauriliodicicco.com/raspberry-pi2-ros-images/>

We can get ROS based images for Raspbian and Ubuntu from the preceding link. In this book, we are using the Raspbian based ROS images for the experiments.

How to install an OS image to Odroid-C1 and Raspberry Pi 2

We can download the Ubuntu image which is prebuilt with ROS, OpenCV, and PCL for Odroid and the ROS built-in Raspbian image for Raspberry Pi 2 and can install to a micro SD card, preferably 16GB. Format the micro SD card in the FAT32 file system and we can either use the SD card adapter or the USB-memory card reader for connecting to a PC.

We can either install OS in Windows or in Linux. The procedure for installing OS on these boards follows.

Installation in Windows

In Windows, there is a tool called `Win32diskimage` which is designed specifically for Odroid. You can download the tool from http://dn.odroid.com/DiskImager_ODROID/Win32DiskImager-odroid-v1.3.zip.

Run **Win32 Disk Imager** with the Administrator privilege. Select the downloaded image, select the memory card drive, and write the image to the drive.

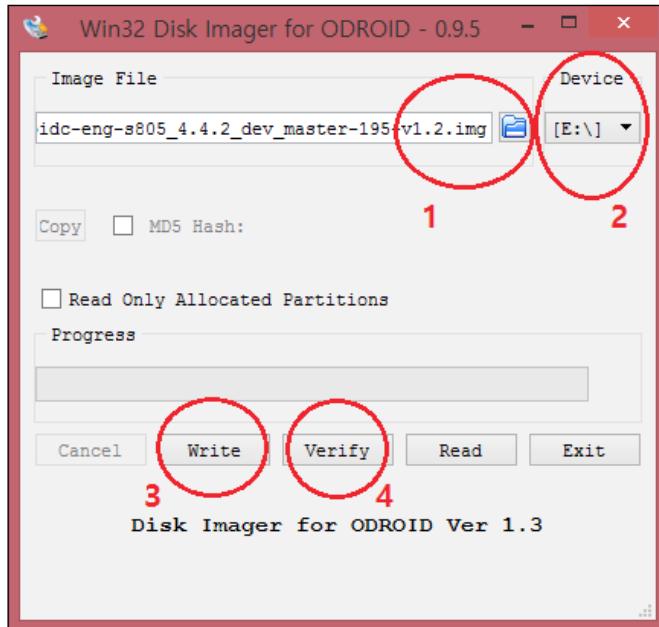


Figure 19 : Win32 Disk Imager for Odroid-C1

After completing this wizard, we can put the micro SD card in Odroid and boot up the OS with ROS support.

The same tool can be used for Raspbian installation in Raspberry Pi 2. We can use the actual version of Win32 Disk Imager for writing Raspbian to a micro SD card from the following link:

<http://sourceforge.net/projects/win32diskimager/>

Installation in Linux

In Linux, there is a tool called **disk dump (dd)**. This tool helps to copy the content of the image to the SD card. dd is a command line tool which is available in all the Ubuntu/Linux based OS. Insert the micro SD card, format to the FAT 32 file system, and use the command mentioned later to write image to the micro SD card.

In the dd tool, there is no progress bar to indicate the copy progress. To get the progress bar, we can install a pipe viewer tool called pv:

```
$ sudo apt-get install pv
```

After installing pv, we can use the following command to install the image file to the micro SD card. Note that you should have the OS image in the same path of the terminal, and also note the micro SD card device name, for example, `mmcblk0`, `sdb`, `sdd`, and so on. You will get the device name using the `dmesg` command.

```
$ dd bs=4M if=image_name.img | pv | sudo dd of=/dev/mmcblk0
```

`image_name.img` is the image name and the device name is `/dev/mmcblk0`. `bs=4M` indicates the block size. If the block size is `4M`, dd will read 4 megabytes from the image and write 4 megabytes to the device. After completing the operation, we can put to Odroid and Raspberry Pi and boot the OS.

Connecting to Odroid-C1 and Raspberry Pi 2 from a PC

We can work with Odroid-C1 and Raspberry Pi 2 by connecting to the HDMI display port and connect the keyboard and mouse to the USB like a normal PC. This is the simplest way of working with Odroid and Raspberry Pi.

In most of the projects, the boards will be placed on the robot, so we can't connect the display and the keyboards to it. There are several methods for connecting these boards to the PC. It will be good if we can share the Internet to these boards too. The following methods can share the Internet to these boards, and at the same time, we can remotely connect via SSH protocol:

- **Remote connection using Wi-Fi router and Wi-Fi dongle through SSH:** In this method, we need a Wi-Fi router with Internet connectivity and Wi-Fi dongle in the board for getting the Wi-Fi support. Both the PC and board will connect to the same network, so each will have an IP address and can communicate using that address.
- **Direct connection using an Ethernet hotspot:** We can share the Internet connection and communicate using SSH via Dnsmasq, a free software DNS forwarder and DHCP server using low system resources. Using this tool, we can tether the Wi-Fi Internet connection of the laptop to the Ethernet and we can connect the board to the Ethernet port of the PC. This kind of communication can be used for robots which are static in operation.

The first method is very easy to configure; it's like connecting two PCs on the same network. The second method is a direct connection of board to laptop through the Ethernet. This method can be used when the robot is not moving. In this method, the board and the laptop can communicate via SSH at the same time and it can share Internet access too. We are using this method in this chapter for working with ROS.

Configuring an Ethernet hotspot for Odroid-C1 and Raspberry Pi 2

The procedure for creating an Ethernet hotspot in Ubuntu and sharing Wi-Fi Internet through this connection follows:

- Take **Edit Connection...** from the network settings and **Add** a new connection as shown next:

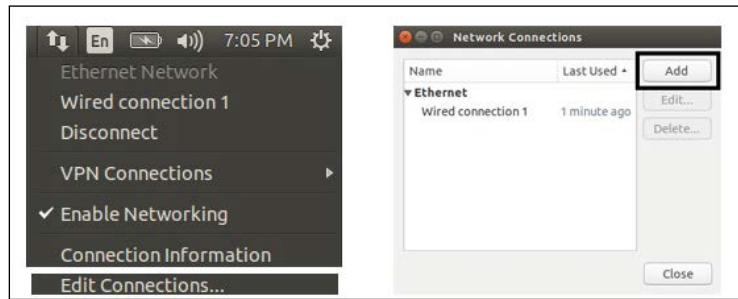


Figure 20 : Configuring a network connection in Ubuntu

- Create an **Ethernet** connection and in **IPv4** setting, change the method to **Shared to Other Computers** and give the connection name as **Share**, as shown next:

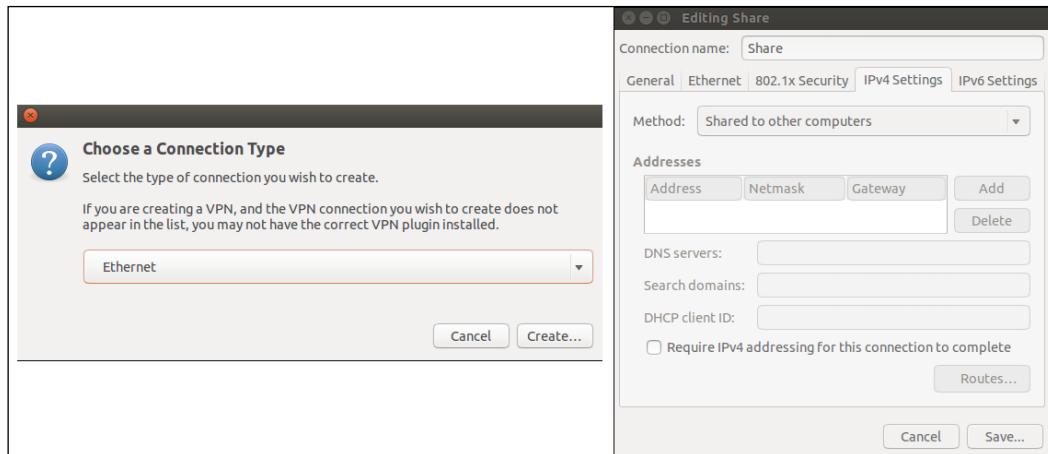


Figure 21 : Creating a new connection for sharing through the Ethernet

- Plugin the micro SD card, power up the Odroid or Raspberry Pi, and connect the Ethernet port from the board to the PC. When the board boots up, we will see that the shared network is automatically connected to the board.
- The following command helps to get the board IP for communicating using SSH:

```
$ cat /var/lib/misc/dnsmasq.leases
```

```
lentin@lentin-Aspire-4755:~$ cat /var/lib/misc/dnsmasq.leases
1420622928 b8:27:eb:a3:dd:cd 10.42.0.65 raspberrypi *
lentin@lentin-Aspire-4755:~$
```

Figure 22 : Listing IP of Raspberry connected via Dnsmasq

- We can communicate with the board using the following commands:

- In Odroid:

```
$ ssh odroid@ip_address
password is odroid
```

- In Raspberry Pi 2:

```
$ ssh pi@ip_adress
password is raspberry
```

After doing SSH into the board, we can launch `roscore` and most of the ROS commands on the board similar to our PC. We will do two examples using these boards. One is for blinking and LED, and the other is for handling a push button. The library we are using for handling GPIO pins of Odroid and Raspberry is called **Wiring Pi**.

The odroid and Raspberry pi have the same pin layout and most of the Raspberry pi GPIO libraries are ported to Odroid, which will make the programming easier. One of the libraries we are using in this chapter for GPIO programming is wiring Pi. Wiring Pi is based on C++ APIs which can access the board GPIO using C++ APIs.

Following are the instructions for installing Wiring Pi on Odroid and Raspberry 2:

Installing Wiring Pi on Odroid-C1

The following procedure can be used to install Wiring Pi on Odroid-C1. This is customized version of Wiring Pi which is used in Raspberry Pi 2.

```
$ git clone https://github.com/hardkernel/wiringPi.git
$ cd wiringPi
$ sudo ./build
```

The Wiring Pi pin out of Odroid-C1 is given next:

ODROID-C1 40pin Layout											
WiringPi GPIO#	Export GPIO#	ODROID-C PIN	Label	HEADER	Label	ODROID-C PIN	Export GPIO#	WiringPi GPIO#	Power Pin		
			3V3	1	2	5V0			Special Function		
		I2CA_SDA	SDA1	3	4	5V0					
		I2CA_SCL	SCL1	5	6	GND					
7	83	GPIOY.BIT3	#83	7	8	TXD1	TXD_B		113		
			GND	9	10	RXD1	RXD_B		114		
0	88	GPIOY.BIT8	#88	11	12	#87	GPIOY.BIT7		87		
2	116	GPIOX.BIT19	#116	13	14	GND					
3	115	GPIOX.BIT18	#115	15	16	#104	GPIOX.BIT7		104		
			3V3	17	18	#102	GPIOX.BIT5		102		
12	107	MOSI	GPIOX.BIT10	MOSI	19	20	GND				
13	106	MISO	GPIOX.BIT9	MISO	21	22	#103	GPIOX.BIT6		103	
14	105	SCLK	GPIOX.BIT8	SCLK	23	24	CE0	GPIOX.BIT20	CE0	117	
			GND	25	26	#118	GPIOX.BIT21		118		
		I2CB_SDA	SDA2	27	28	SCL2	I2CB_SCL				
21	101		GPIOX.BIT4	#101	29	30	GND				
22	100		GPIOX.BIT3	#100	31	32	#99	GPIOX.BIT2		99	
23	108		GPIOX.BIT11	#108	33	34	GND				
24	97		GPIOX.BIT0	#97	35	36	#98	GPIOX.BIT1		98	
			ADC.AIN1	AIN1	37	38	1V8	1V8			
			GND	39	40	A1N0	ADC.AIN0				

Figure 23 : Pin out of Odroid - C1

Installing Wiring Pi on Raspberry Pi 2

The following procedure can be used to install Wiring Pi on Raspberry Pi 2.

```
$ git clone git clone git://git.drogon.net/wiringPi
$ cd wiringPi
$ sudo ./build
```

The pin out of Raspberry Pi 2 and Wiring Pi is shown next:

P1: The Main GPIO connector							
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin
		3.3v	1	2	5v		
8	Rv1:0 - Rv2:2	SDA	3	4	5v		
9	Rv1:1 - Rv2:3	SCL	5	6	0v		
7	4	GPIO7	7	8	TxD	14	15
		0v	9	10	RxD	15	16
0	17	GPIO0	11	12	GPIO1	18	1
2	Rv1:21 - Rv2:27	GPIO2	13	14	0v		
3	22	GPIO3	15	16	GPIO4	23	4
		3.3v	17	18	GPIO5	24	5
12	10	MOSI	19	20	0v		
13	9	MISO	21	22	GPIO6	25	6
14	11	SCLK	23	24	CE0	8	10
		0v	25	26	CE1	7	11
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin

P5: Secondary GPIO connector (Rev. 2 Pi only)							
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin
		5v	1	2	3.3v		
17	28	GPIO8	3	4	GPIO9	29	18
19	30	GPIO10	5	6	GPIO11	31	20
		0v	7	8	0v		
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin

Figure 24 : Pin out of Raspberry Pi 2

The following are the ROS examples for Odroid-C1 and Raspberry Pi 2.

Blinking LED using ROS on Odroid-C1 and Raspberry Pi 2

This is a basic LED example which can blink the LED connected to the first pin of Wiring Pi, that is the 12th pin on the board. The LED cathode is connected to the GND pin and 12th pin as an anode. The following image shows the circuit of Raspberry Pi with an LED. The same pin out can be used in Odroid too.

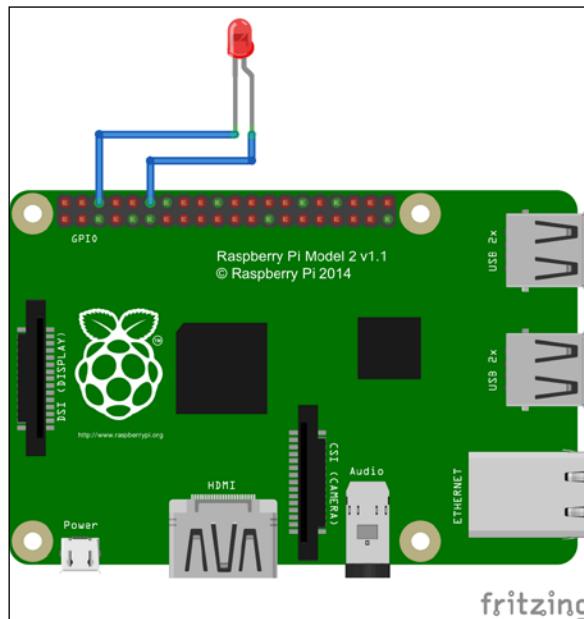


Figure 25 : Blinking an LED using Raspberry Pi 2

We can create the example ROS package using the following command:

```
$ catkin_create_pkg ros_wiring_example roscpp std_msgs
```

You will get the existing package from the chapter_7_codes/ROS_Odroid_Examples/ ros_wiring_examples folder.

Create a `src` folder and create the following code called `blink.cpp` inside the `src` folder:

```
#include "ros/ros.h"
#include "std_msgs/Bool.h"
#include <iostream>
```

```
//Wiring Pi header
#include "wiringPi.h"

//Wiring PI first pin

#define LED 1

//Callback to blink the LED according to the topic value
void blink_callback(const std_msgs::Bool::ConstPtr& msg)
{

    if(msg->data == 1){
        digitalWrite (LED, HIGH) ;
        ROS_INFO("LED ON");
    }
    if(msg->data == 0){
        digitalWrite (LED, LOW) ;
        ROS_INFO("LED OFF");
    }
}
int main(int argc, char** argv)
{
    ros::init(argc, argv,"blink_led");
    ROS_INFO("Started Odroid-C1 Blink Node");
    //Setting WiringPi
    wiringPiSetup () ;
    //Setting LED pin as output
    pinMode(LED, OUTPUT);
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("led_blink",10,blink_callback);
    ros::spin();
}
```

This code will subscribe a topic called `led_blink`, which is a Boolean type. If we publish 1 to this topic, it will switch on the LED. If we publish 0, the LED will turn off.

Push button + blink LED using ROS on Odroid-C1 and Raspberry Pi 2

The next example is handling input from a button. When we press the button, the code will publish to the `led_blink` topic and blink the LED. When the switch is off, LED will also be OFF. The LED is connected to the 12th pin and GND, and the button is connected to the 11th pin and GND. The following image shows the circuit of this example. The circuit is the same for Odroid also.

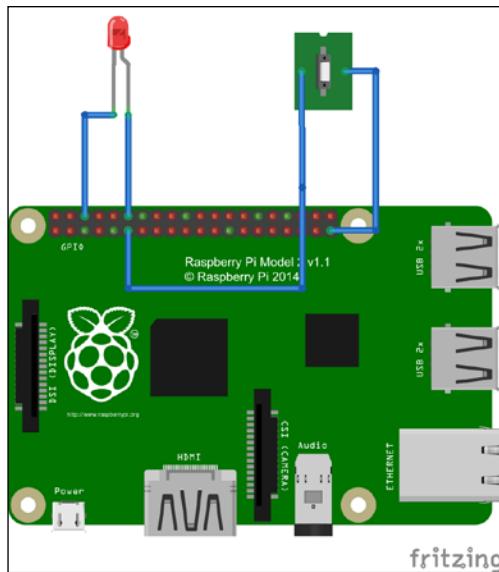


Figure 26 : LED + button in Raspberry Pi 2

The code for interfacing LED and button is given next. The code can be saved with the name `button.cpp` inside the `src` folder.

```
#include "ros/ros.h"
#include "std_msgs/Bool.h"

#include <iostream>
#include "wiringPi.h"

//Wiring PI 1
#define BUTTON 0
#define LED 1

void blink_callback(const std_msgs::Bool::ConstPtr& msg)
```

```
{  
  
    if(msg->data == 1){  
  
        digitalWrite (LED, HIGH) ;  
        ROS_INFO("LED ON");  
    }  
  
    if(msg->data == 0){  
        digitalWrite (LED, LOW) ;  
        ROS_INFO("LED OFF");  
    }  
  
}  
  
int main(int argc, char** argv)  
{  
  
    ros::init(argc, argv,"button_led");  
    ROS_INFO("Started Odroid-C1 Button Blink Node");  
  
    wiringPiSetup ();  
  
    pinMode(LED, OUTPUT);  
    pinMode(BUTTON, INPUT);  
    pullUpDnControl(BUTTON, PUD_UP); // Enable pull-up resistor on  
button  
  
    ros::NodeHandle n;  
    ros::Rate loop_rate(10);  
  
    ros::Subscriber sub = n.subscribe("led_blink",10,blink_callback);  
    ros::Publisher chatter_pub = n.advertise<std_msgs::Bool>("led_  
blink", 10);  
  
    std_msgs::Bool button_press;  
    button_press.data = 1;  
  
    std_msgs::Bool button_release;  
    button_release.data = 0;
```

```
while (ros::ok())
{
    if (!digitalRead(BUTTON)) // Return True if button pressed
    {

        ROS_INFO("Button Pressed");
        chatter_pub.publish(button_press);

    }
    else
    {

        ROS_INFO("Button Released");
        chatter_pub.publish(button_release);

    }

    ros::spinOnce();
    loop_rate.sleep();
}

}
```

CMakeLists.txt for building these two examples is given next. The Wiring Pi code needs to link with the Wiring Pi library. We have added this in the CMakeLists.txt file.

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_wiring_examples)

find_package(catkin REQUIRED COMPONENTS
    roscpp
    std_msgs
)

find_package(Boost REQUIRED COMPONENTS system)

//Include directory of wiring Pi
set(wiringPi_include "/usr/local/include")

include_directories(
    ${catkin_INCLUDE_DIRS}
    ${wiringPi_include}
```

```
)  
  
//Link directory of wiring Pi  
LINK_DIRECTORIES("/usr/local/lib")  
  
add_executable(blink_led src/blink.cpp)  
  
add_executable(button_led src/button.cpp)  
  
target_link_libraries(blink_led  
    ${catkin_LIBRARIES} wiringPi  
)  
  
target_link_libraries(button_led  
    ${catkin_LIBRARIES} wiringPi  
)
```

Build the project using `catkin_make` and we can run each example. For executing the Wiring Pi based code, we need root permission.

Running LED blink in Odroid-C1

After building the project, first we can run the LED blink example. We have to login to Odroid using SSH from PC in multiple terminals for running this example.

- Start roscore in one terminal:
`$ roscore`
- Run the executable as root in the another terminal:
`$ sudo -s
cd /home/odroid/catkin_ws/build/ros_wiring_examples
#./blink_led`

After starting the `blink_led` node, publish 1 to the `led_blink` topic in another terminal.

- For LED to ON state:
`$ rostopic pub /led_blink std_msgs/Bool 1`
- For LED to OFF state:
`$ rostopic pub /led_blink std_msgs/Bool 0`

Running button handling and LED blink in Odroid-C1

The button handling + LED Blink should have same setup in the above example. We should login to Odroid via SSH in multiple terminal and execute each command on each terminals.

Start roscore in one terminal:

```
$ roscore
```

Run the button LED node in another terminal:

```
$ sudo -s  
# cd /home/odroid/catkin_ws/build/ros_wiring_examples  
#./button_led
```

Press the button and we can see the LED blinking. We can also check the button state by echoing the topic led_blink:

```
$ rostopic echo /led_blink
```

Running LED blink in Raspberry Pi 2

The examples which work on Odroid-C1 will work on Raspberry Pi-2 too. Before running the examples, first we should do the following setup in Raspberry Pi. You can do this setup by login to Raspberry Pi through SSH.

We need to add the following lines to the .bashrc file of the root user. Take the .bashrc file of the root user:

```
$ sudo -i  
$ nano .bashrc
```

Add the following lines to the end of this file:

```
source /opt/ros/indigo/setup.sh  
source /home/pi/catkin_ws/devel/setup.bash  
export ROS_MASTER_URI=http://localhost:11311
```

After adding these lines, we can follow the same command we did in Odroid. Note that the user name is pi, not odroid.

Interfacing Dynamixel actuators to ROS

One of the latest smart actuators available on the market is **Dynamixel**, which is manufactured by a company called Robotis. The Dynamixel servos are available in various versions and shown in the following image are some of the different versions of Dynamixel servos:



Figure 27 : Different types of Dynamixel servos

These smart actuators have complete support in ROS and clear documentation is also available for them.

The official ROS wiki page of Dynamixel is http://wiki.ros.org/dynamixel_controllers/Tutorials.

Questions

1. What are the different `rosserial` packages?
2. What is the main function of `rosserial_arduino`?
3. How does `rosserial` protocol work?
4. What are the main differences between Odroid-C1 and Raspberry Pi?

Summary

This chapter was about interfacing I/O boards to ROS and adding sensors on it. We have discussed interfacing of the popular I/O board called Arduino to ROS, and interface basic components such as LEDs, buttons, accelerometers, ultrasonic sound sensors, and so on. After seeing the interfacing of Arduino, we discussed how to setup ROS on Raspberry Pi 2 and Odroid-C1. We also did few basic examples in Odroid and Raspberry Pi based on ROS and Wiring Pi. In the end, we saw the interfacing of smart actuators called Dynamixel in ROS.

8

Programming Vision Sensors using ROS, Open-CV, and PCL

In the last chapter, we discussed interfacing of sensors and actuators using I/O board in ROS. In this chapter, we are going to discuss how to interface various vision sensors in ROS and program it using libraries such as **OpenCV** (**Open Source Computer Vision**) and **PCL** (**Point Cloud Library**). The vision in a robot is an important aspect of the robot for manipulating object and navigation. There are lots of 2D/3D vision sensors available in the market and most of the sensors have an interface driver package in ROS. We will discuss interfacing of new vision sensors to ROS and programming it using OpenCV and PCL.

We will cover the following topics in this chapter:

- Understanding ROS—OpenCV interfacing packages
- Understanding ROS—PCL interfacing packages
- Installing OpenCV and PCL interfaces in ROS
- Interfacing USB webcams in ROS
- Working with ROS camera calibration
- Converting images between ROS and OpenCV using `cv_bridge`
- Displaying images from webcam using OpenCV and `cv_bridge`
- Interfacing Kinect and Asus Xtion Pro in ROS
- Interfacing Intel Real Sense Camera in ROS
- Working with the ROS `depthimage_to_laserscan` package

- Interfacing Hokuyo Laser in ROS
- Interfacing Velodyne in ROS
- Programming using PCL-ROS interface
- Streaming webcam from Odroid using ROS

Understanding ROS – OpenCV interfacing packages

OpenCV is one of the popular open source real time computer vision libraries, which is mainly written in C/C++. OpenCV comes with a BSD license and is free for academic and commercial application. OpenCV can be programmed using C/C++, Python, and Java, and it has multi-platform support such as Windows, Linux, OSX, Android, and iOS. OpenCV has tons of computer vision APIs, which can be used for implementing computer vision applications. The web page of OpenCV library is <http://opencv.org/>.

The OpenCV library is interfaced to ROS via ROS stack called `vision_opencv`. `vision_opencv` consists of two important packages for interfacing OpenCV to ROS. They are:

- `cv_bridge`: The `cv_bridge` package contains a library that provides APIs for converting the OpenCV image data type `cv::Mat` to the ROS image message called `sensor_msgs/Image` and vice versa. In short, it can act as a bridge between OpenCV and ROS. We can use OpenCV APIs to process the image and convert to ROS image messages whenever we want to send to another node. We will discuss how to do this conversion in the upcoming sections.
- `image_geometry`: One of the first processes that we should do before working with cameras is its calibration. The `image_geometry` package contains libraries written in C++ and Python, which helps to correct the geometry of the image using calibration parameters. The package uses a message types called `sensor_msgs/CameraInfo` for handling the calibration parameters and feed to the OpenCV image rectification function.

Understanding ROS – PCL interfacing packages

The point cloud data can be defined as a group of data points in some coordinate system. In 3D, it has X, Y, and Z coordinates. PCL library is an open source project for handling 2D/3D image and point clouds processing.

Like OpenCV, it is under BSD license and free for academic and commercial purposes. It is also a cross platform, which has support in Linux, Windows, Mac OS, and Android/iOS.

The library consists of standard algorithms for filtering, segmentation, feature estimation, and so on, which is required to implement different point cloud applications. The main web page of point cloud library is <http://pointclouds.org/>.

The point cloud data can be acquired by sensors such as Kinect, Asus Xtion Pro, Intel Real Sense, and such others. We can use this data for robotic applications such as robot object manipulation and grasping. PCL is tightly integrated into ROS for handling point cloud data from various sensors. The `perception_pcl` stack is the ROS interface for PCL library. It consists of packages for pumping the point cloud data from ROS to PCL data type and vice versa. `perception_pcl` consists of the following packages:

- `pcl_conversions`: This package provides APIs to convert PCL data types to ROS messages and vice versa.
- `pcl_msgs`: This package contains definition of PCL related messages in ROS. The PCL messages are:
 - `ModelCoefficients`
 - `PointIndices`
 - `PolygonMesh`
 - `Vertices`
- `pcl_ros`: This is the PCL bridge of ROS. This package contains tools and nodes to bridge ROS messages to PCL data types and vice versa.
- `pointcloud_to_laserscan`: The main function of this package is to convert 3D point cloud into 2D laser Scan. This package is useful for converting an inexpensive 3D vision sensor such as Kinect and Asus Xtion Pro to a laser scanner. The laser scanner data is mainly used for 2D-SLAM for the purpose of robot navigation.

Installing ROS perception

We are going to install a single package called **perception**, which is a meta package of ROS containing all the perception related packages such as OpenCV, PCL, and so on.

- In ROS Jade
`$ sudo apt-get install ros-jade-perception`
- In ROS Indigo
`$ sudo apt-get install ros-indigo-perception`

The ROS perception stack contains the following ROS packages:

- **image-common**: This meta package contains common functionalities to handle an image in ROS. The meta package consists of the following list of packages (http://wiki.ros.org/image_common):
 - **image_transport**: This package helps to compress the image during publishing and subscribes the images to save the band width (http://wiki.ros.org/image_transport). The various compression methods are JPEG/PNG compression and Theora for streaming videos. We can also add custom compression methods to `image_transport`.
 - **camera_calibration_parsers**: This package contains routine to read/write camera calibration parameters from an XML file. This package is mainly used by camera drivers for accessing calibration parameters.
 - **camera_info_manager**: This package consists of routine to save, restore, and load the calibration information. This is mainly used by camera drivers.
 - **polled_camera**: This package contains interface for requesting images from a polling camera driver (for example, `prosilica_camera`).
- **image-pipeline**: This meta package contains packages to process the raw image from the camera driver. The various processing done by this meta package are calibration, distortion removal, stereo vision processing, depth image processing, and so on. The following packages are present in this meta package for this processing (http://wiki.ros.org/image_pipeline):
 - **camera_calibration**: One of the important tools for relating the 3D world to the 2D camera image is calibration. This package provides tools for doing monocular and stereo image calibration in ROS.

- `image_proc`: The nodes in this package act between the camera driver and the vision processing nodes. It can handle the calibration parameters, correct image distortion from the raw image, and convert to color image.
- `depth_image_proc`: This package contains nodes and nodelets for handling depth image from Kinect and 3D vision sensors. The depth image can be processed by these nodelets to produce point cloud data.
- `stereo_image_proc`: This package has nodes to perform distortion removal for a pair of cameras. It is same as the `image_proc` package, except that it handles two cameras for stereo vision and for developing point cloud and disparity images.
- `image_rotate`: This package contains nodes to rotate the input image.
- `image_view`: This is a simple ROS tool for viewing ROS message topic. It can also view stereo and disparity images.
- `image-transport-plugins`: These are the plugins of ROS image transport for publishing and subscribing the ROS images in different compression levels or different video codec to reduce the bandwidth and latency.
- `laser-pipeline`: This is a set of packages that can process laser data such as filtering and converting into 3D Cartesian points and assembling points to form a cloud. The `laser-pipeline` stack contains the following packages:
 - `laser_filters`: This package contains nodes to filter the noise in the raw laser data, remove the laser points inside the robot footprint, and remove spurious values inside the laser data.
 - `laser_geometry`: After filtering the laser data, we have to transform the laser ranges and angles into 3D Cartesian coordinates efficiently by taking into account the tilt and skew angle of laser scanner.
 - `laser_assembler`: This package can assemble the laser scan into a 3D point cloud or 2.5 D scan.
- `perception-pcl`: This is the stack of PCL-ROS interface.
- `vision-opencv`: This is the stack of OpenCV-ROS interface.

Interfacing USB webcams in ROS

We can start interfacing with an ordinary webcam or a laptop cam in ROS. There are no exact specific packages for webcam - ROS interfaces. If the camera is working in Ubuntu/Linux, it may be supported by the ROS driver too. After plugging the camera, check whether a /dev/videoX device file has been created, or check with some application such as Cheese, VLC, and such others. The guide to check whether the web cam is supported on Ubuntu is available at <https://help.ubuntu.com/community/Webcam>.

We can find the video devices present on the system using the following command:

```
$ ls /dev/ | grep video
```

If you get an output of video0, you can confirm a USB cam is available for use.

After ensuring the webcam support in Ubuntu, we can install a ROS webcam driver called `usb_cam` using the following command:

- In ROS Jade

```
$ sudo apt-get install ros-jade-usb-cam
```

- In ROS Indigo

```
$ sudo apt-get install ros-indigo-usb-cam
```

We can install the latest package of `usb_cam` from the source code. The driver is available on GitHub at https://github.com/bosch-ros-pkg/usb_cam

The `usb_cam` package contains a node called `usb_cam_node`, which is the driver of USB cams. There are some parameters that need to be set before running this node. We can run the ROS node along with its parameters. The `usb_cam-test.launch` launch file can launch the USB cam driver with the necessary parameters:

```
<launch>
  <node name="usb_cam" pkg="usb_cam" type="usb_cam_node"
output="screen" >
    <param name="video_device" value="/dev/video0" />
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap" />
  </node>
```

```
<!-- Launching image_view node -->

<node name="image_view" pkg="image_view" type="image_view"
respawn="false" output="screen">
    <remap from="image" to="/usb_cam/image_raw"/>
    <param name="autosize" value="true" />
</node>
</launch>
```

This launch file will start `usb_cam_node` with the video device `/dev/video0`, with a resolution of 640x480. The pixel format here is YUV (<https://en.wikipedia.org/wiki/YUV>). After initiating `usb_cam_node`, it will start an `image_view` node for displaying the raw image from the driver. We can launch the previous file using the following command:

```
$ roslaunch usb_cam usb_cam-test.launch
```

We will get the following message with an image view as shown next:

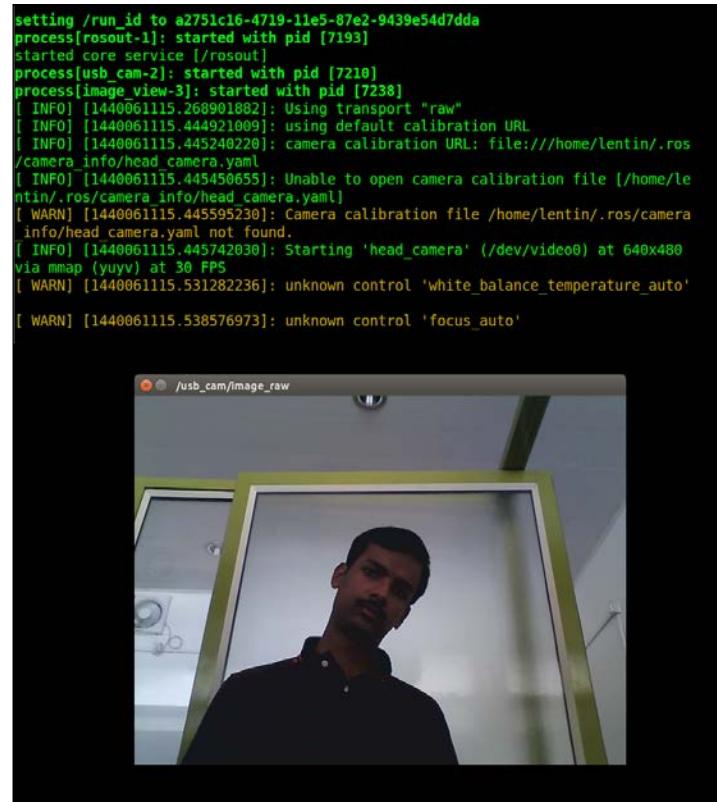
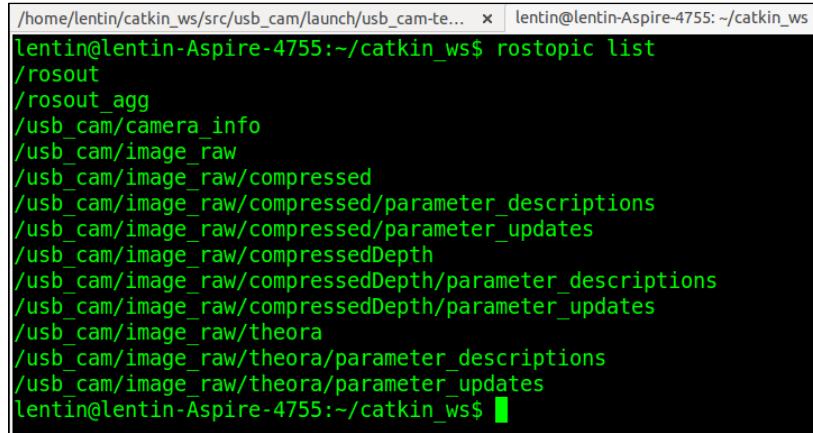


Figure 1 : USB camera view using image view tool

The topics generated by the driver are shown next. There are raw, compressed, and Theora codec topics generated by the driver.



```
/home/lentin/catkin_ws/src/usb_cam/launch/usb_cam-te... x lentin@lentin-Aspire-4755: ~/catkin_ws
lentin@lentin-Aspire-4755:~/catkin_ws$ rostopic list
/rosout
/rosout_agg
/usb_cam/camera_info
/usb_cam/image_raw
/usb_cam/image_raw/compressed
/usb_cam/image_raw/compressed/parameter_descriptions
/usb_cam/image_raw/compressed/parameter_updates
/usb_cam/image_raw/compressedDepth
/usb_cam/image_raw/compressedDepth/parameter_descriptions
/usb_cam/image_raw/compressedDepth/parameter_updates
/usb_cam/image_raw/theora
/usb_cam/image_raw/theora/parameter_descriptions
/usb_cam/image_raw/theora/parameter_updates
lentin@lentin-Aspire-4755:~/catkin_ws$
```

Figure 2 : List of topics generated by the USB camera driver

We can visualize the image in another window using the following command:

```
$ rosrun image_view image_view image:=/usb_cam/image_raw
```

After getting the camera image message, the first thing we have to do is camera calibration.

Working with ROS camera calibration

Like all sensors, cameras also need calibration for correcting the distortions in the camera images due to the camera's internal parameters and for finding the world coordinates from the camera coordinates.

The primary parameters that cause image distortions are radial distortions and tangential distortions. Using camera calibration algorithm, we can model these parameters and also calculate the real world coordinates from the camera coordinates by computing the camera calibration matrix, which contains the focal distance and the principle points.

Camera calibration can be done using a classic black-white chessboard, symmetrical circle pattern, or asymmetrical circle pattern. According to each different pattern, we use different equations to get the calibration parameters. Using the calibration tools, we detect the patterns and each detected pattern is taken as a new equation. When the calibration tool gets enough detected patterns, it can compute the final parameters for the camera.

ROS provides a package named `camera_calibration` (http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration) to do camera calibration, which is a part of the image pipeline stack. We can calibrate monocular, stereo, and even 3D sensors such as Kinect and Asus Xtion pro.

The first thing we have to do before calibration is download the check board pattern mentioned in the ROS Wiki page, and print it and paste it onto a card board. This is the pattern we are going to use for calibration. This check board has 8x6 with 108mm squares.

Run the `usb_cam` launch file to start the camera driver. We are going to run the camera calibration node of ROS using the raw image from the `/usb_cam/image_raw` topic. Following command will run the calibration node with the necessary parameters:

```
$ rosrn camera_calibration cameracalibrator.py --size 8x6 --square 0.108  
image:=/usb_cam/image_raw camera:=/usb_cam
```

A calibration window will pop up, and when we show the calibration pattern to the camera, and the detection made, is seen in the following screenshot:

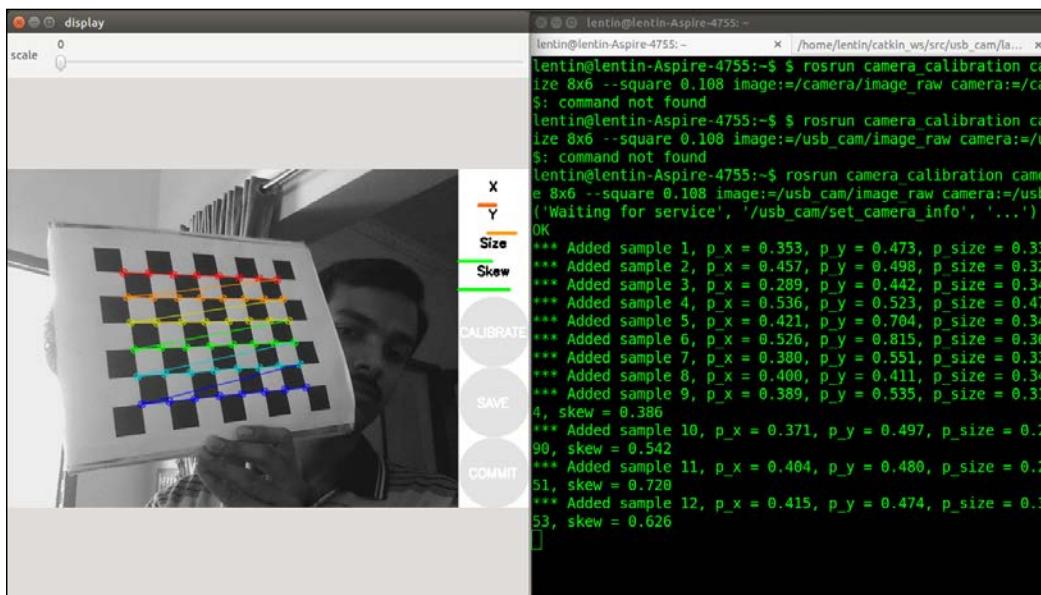


Figure 3: ROS camera calibration

Move the calibration pattern in X direction and Y direction. If the calibrator node gets a sufficient amount of samples, a calibration button will get active on the window. When we press the **CALIBRATE** button, it will compute the camera parameters using these samples. It will take some time for calculation. After computation, two buttons, **SAVE** and **COMMIT**, will become active inside the window, which is shown in the following image. If we press the **SAVE** button, it will save the calibration parameters to a file in the `/tmp` folder. If we press the **COMMIT** button, it will save them to `./ros/camera_info/head_camera.yaml`.

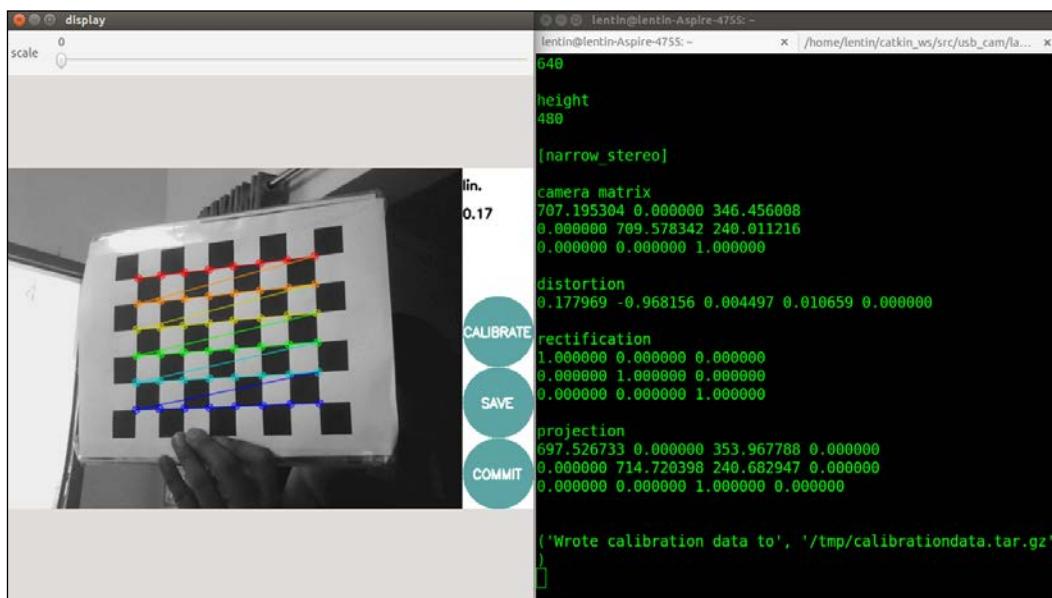


Figure 4 : Generating camera calibration file

Restart the camera driver and we will see the YAML calibration file loaded along with the driver. The calibration file that we generated will look as follows:

```

image_width: 640
image_height: 480
camera_name: head_camera
camera_matrix:
  rows: 3
  cols: 3
  data: [707.1953043273086, 0, 346.4560078627374, 0,
    709.5783421541863, 240.0112155124814, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5

```

```

data: [0.1779688561999974, -0.9681558538432319,
0.004497434720139909, 0.0106588921249554, 0]
rectification_matrix:
rows: 3
cols: 3
data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:
rows: 3
cols: 4
data: [697.5267333984375, 0, 353.9677879190494, 0, 0,
714.7203979492188, 240.6829465337159, 0, 0, 0, 1, 0]

```

Converting images between ROS and OpenCV using cv_bridge

In this section, we will see how to convert between ROS image message (`sensor_msgs/Image`) to OpenCV image data type(`cv::Mat`). The main ROS package used for this conversion is `cv_bridge`, which is part of the `vision_opencv` stack. The ROS library inside `cv_bridge` called `CvBridge` helps to perform this conversion. We can use the `CvBridge` library inside our code and perform the conversion. The following figure shows how the conversion is performed between ROS and OpenCV:

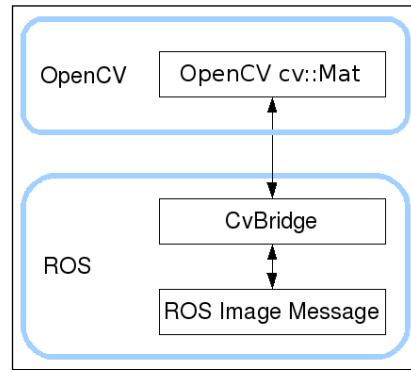


Figure 5 : Converting images using CvBridge

Here, the `CvBridge` library acts as a bridge for converting the ROS messages to OpenCV image and vice versa.

We will see how the conversion between ROS and OpenCV is performed using the following example.

Image processing using ROS and OpenCV

In this section, we will see an example of using `cv_bridge` for acquiring images from a camera driver, and converting and processing the images using OpenCV APIs. Following is how the example works:

- Subscribe the images from the camera driver from the topic `/usb_cam/image_raw` (`sensor_msgs/Image`)
- Convert the ROS images to OpenCV image type using `CvBridge`
- Process the OpenCV image using its APIs and find the edges on the image
- Convert the OpenCV image type of edge detection to ROS image messages and publish into the topic `/edge_detector/processed_image`

The step by step procedure to build this example follows:

Step 1: Creating ROS package for the experiment

You can get the existing package `cv_bridge_tutorial_pkg` from the `chapter_8_codes` folder, or you can create a new package using the following command:

```
$ catkin_create_pkg cv_bridge_tutorial_pkg cv_bridge image_transport  
roscpp sensor_msgs std_msgs
```

This package is mainly dependent on `cv_bridge`, `image_transport`, and `sensor_msgs`.

Step 2: Creating source files

You can get the source code of the example `sample_cv_bridge_node.cpp` from the `chapter_8_codes/cv_bridge_tutorial_pkg/src` folder.

Step 3: Explanation of the code

Following is the explanation of the complete code:

```
#include <image_transport/image_transport.h>
```

We are using the `image_transport` package in this code for publishing and subscribing to image in ROS.

```
#include <cv_bridge/cv_bridge.h>  
#include <sensor_msgs/image_encodings.h>
```

This header includes the CvBridge class and image encoding related functions in the code.

```
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
```

These are main OpenCV image processing module and GUI modules which provide image processing and GUI APIs in our code.

```
image_transport::ImageTransport it_;
public:
    Edge_Detector()
        : it_(nh_)
    {
        // Subscribe to input video feed and publish output video feed
        image_sub_ = it_.subscribe("/usb_cam/image_raw", 1,
            &ImageConverter::imageCb, this);

        image_pub_ = it_.advertise("/edge_detector/processed_image", 1);
```

We will look in more detail at the line `image_transport::ImageTransport it_`. This line creates an instance of `ImageTransport` which is used to publish and subscribe the ROS image messages. More information about the `ImageTransport` API is given next.

Publishing and subscribing images using `image_transport`

ROS image transport is very similar to ROS Publishers and Subscribers and it is used to publish/subscribe the images along with the camera information. We can publish the image data using `ros::Publishers`, but image transport is a more efficient way of sending the image data.

The image transport APIs are provided by the `image_transport` package. Using these APIs, we can transport an image in different compression formats; for example, we can transport it as an uncompressed image, JPEG/PNG compression, or Theora compression in separate Topics. We can also add different transport formats by adding plugins. By default, we can see the compressed and Theora transports.

```
image_transport::ImageTransport it_;
```

In the following line, we are creating an instance of the `ImageTransport` class:

```
image_transport::Subscriber image_sub_;
image_transport::Publisher image_pub_;
```

After that, we declare the `Subscriber` and `Publisher` objects for subscribing and publishing the images using the `image_transport` object:

```
image_sub_ = it_.subscribe("/usb_cam/image_raw", 1,
                           &ImageConverter::imageCb, this);
image_pub_ = it_.advertise("/edge_detector/processed_image", 1);
```

The following is how we subscribe and publish an image:

```
cv::namedWindow(OPENCV_WINDOW);
}
~Edge_Detector()
{
    cv::destroyWindow(OPENCV_WINDOW);
}
```

This is how we subscribe and publish an `image`. `cv::namedWindow()` is an OpenCV function to create a GUI for displaying an image. The argument inside this function is the window name. Inside the class destructor, we are destroying the named window.

Converting OpenCV-ROS images using `cv_bridge`

This is an image callback function and it basically converts the ROS image messages into OpenCV `cv::Mat` type using the `CvBridge` APIs. Following is how we can convert ROS to OpenCV, and vice versa:

```
void imageCb(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;
    namespace enc = sensor_msgs::image_encodings;

    try
    {
        cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_
encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }
```

To start with CvBridge, we should start with creating an instance of a CvImage. Given next is the creation of the CvImage pointer:

```
cv_bridge::CvImagePtr cv_ptr;
```

The CvImage type is a class provided by cv_bridge, which consists of information such as an OpenCV image, its encoding, ROS header, and so on. Using this type, we can easily convert an ROS image to OpenCV, and vice versa.

```
cv_ptr = cv_bridge::toCvCopy(msg,
sensor_msgs::image_encodings::BGR8);
```

We can handle the ROS image message in two ways: either we can make a copy of the image or we can share the image data. When we copy the image, we can process the image, but if we use shared pointer, we can't modify the data. We use `toCvCopy()` for creating a copy of the ROS image, and the `toCvShare()` function is used to get the pointer of the image. Inside these functions, we should mention the ROS message and the type of encoding.

```
if (cv_ptr->image.rows > 400 && cv_ptr->image.cols > 600) {
    detect_edges(cv_ptr->image);
    image_pub_.publish(cv_ptr->toImageMsg());
}
```

In this section, we are extracting the image and its properties from the CvImage instance, and accessing the `cv::Mat` object from this instance. This code simply checks whether the rows and columns of the image are in a particular range, and if it is true, it will call another method called `detect_edges(cv::Mat)`, which will process the image given as argument and display the edge detected image.

```
image_pub_.publish(cv_ptr->toImageMsg());
```

The preceding line will publish the edge detected image after converting to ROS image message. Here we are using the `toImageMsg()` function for converting the CvImage instance to a ROS image message.

Finding edges on the image

After converting the ROS images to OpenCV type, the function `detect_edges(cv::Mat)` will be called for finding the edges on the image using the following inbuilt OpenCV functions:

```
cv::cvtColor( img, src_gray, CV_BGR2GRAY );
cv::blur( src_gray, detected_edges, cv::Size(3,3) );
cv::Canny( detected_edges, detected_edges, lowThreshold,
lowThreshold*ratio, kernel_size );
```

Here, the `cvtColor()` function will convert an RGB image to a GRAY color space and `cv::blur()` will add blurring to the image. After that, using Canny edge detector, we extract the edges of the image.

Visualizing raw and edge detected image

```
cv::imshow(OPENCV_WINDOW, img);
cv::imshow(OPENCV_WINDOW_1, dst);
cv::waitKey(3);
```

Here we are displaying the image data using the OpenCV function called `imshow()`, which consists of the window name and the image name.

Step 4: Editing the CMakeLists.txt file

The definition of the `CMakeLists.txt` file is given next. In this example, we need OpenCV support, so we should include the OpenCV header path and also link the source code against the OpenCV library path.

```
include_directories(
    ${catkin_INCLUDE_DIRS}
    ${OpenCV_INCLUDE_DIRS}
)

add_executable(sample_cv_bridge_node src/sample_cv_bridge_node.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(sample_cv_bridge_node
    ${catkin_LIBRARIES}
    ${OpenCV_LIBRARIES}
)
```

Step 5: Building and running example

After building the package using `catkin_make`, we can run the node using the following command:

- Launch webcam driver:

```
$ roslaunch usb_cam usb_cam-test.launch
```

- Run the `cv_bridge` sample node:

```
$ rosrun cv_bridge_tutorial_pkgs sample_cv_bridge_node
```

If everything works fine, we will get two windows, as shown in the following image. The first window shows the raw image and the second is the processed edge detected image.

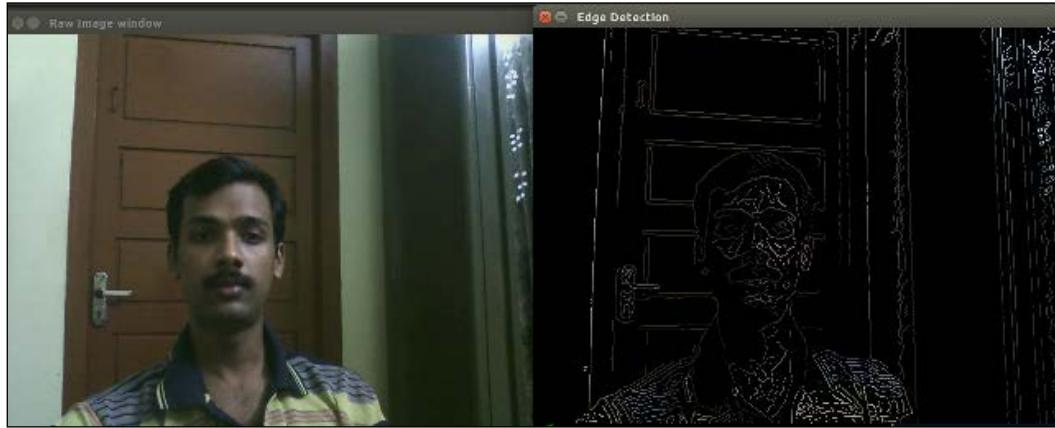


Figure 6 : Raw image and edge detected image

Interfacing Kinect and Asus Xtion Pro in ROS

The web cams that we have worked with till now can only provide 2D visual information of the surroundings. For getting 3D information about the surroundings, we have to use 3D vision sensors or range finders such as laser finders. Some of the 3D vision sensors that we are discussing in this chapter are Kinect, Asus Xtion Pro, Intel Real sense, Velodyne, and Hokuyo laser scanner.



Figure 7 : Top: Kinect , Bottom: Asus Xtion Pro

The first two sensors we are going to discuss are Kinect and Asus Xtion Pro. Both of these devices need OpenNI (**Open source Natural Interaction**) driver library for operating in Linux system. OpenNI acts as a middleware between the 3D vision devices and the application software. The OpenNI driver is integrated to ROS and we can install these drivers using the following commands. These packages help to interface the OpenNI complaint device, such as Kinect and Asus Xtion Pro.

- In Jade:

```
$ sudo apt-get install ros-jade-openni-launch
```

- In Indigo:

```
$ sudo apt-get install ros-indigo-openni-launch
```

The preceding command will install OpenNI drivers and launch files for starting the RGB/Depth streams. After successful installation of these packages, we can launch the driver using the following command:

```
$ rosrun openni_launch openni.launch
```

This launch file will convert the raw data from the devices into useful data, such as 3D point cloud, disparity images, and depth, and the RGB images using ROS nodelets.

Other than the OpenNI drivers, there is another driver available called `libfreenect`. The common launch files of the drivers are organized into a package called `rgbd_launch`. This package consists of common launch files that are used for the freenect and openni drivers.

We can visualize the point cloud generated by the OpenNI ROS driver using RViz.

Run RViz using the following command:

```
$ rosrun rviz rviz
```

Set Fixed frame to `/camera_depth_optical_frame`, add a `PointCloud2` display and set topic as `/camera/depth/points`. This is the unregistered point cloud from IR camera, that is, it may have complete match with the RGB camera and it only uses depth camera for generating point cloud.

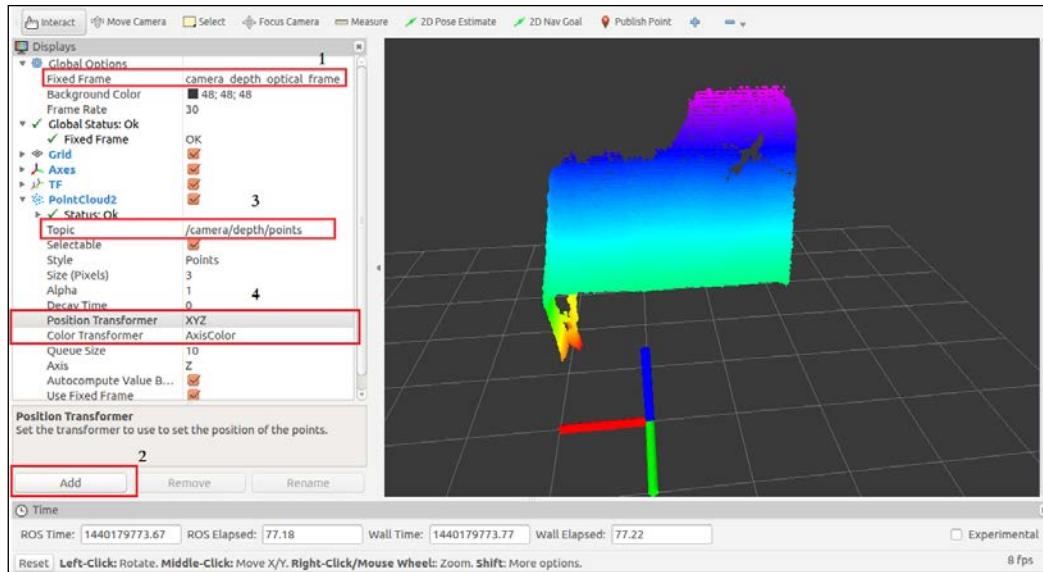


Figure 8: Unregistered point cloud view in RViz

We can enable the registered point cloud by using **Dynamic Reconfigure GUI**, by using the following command:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

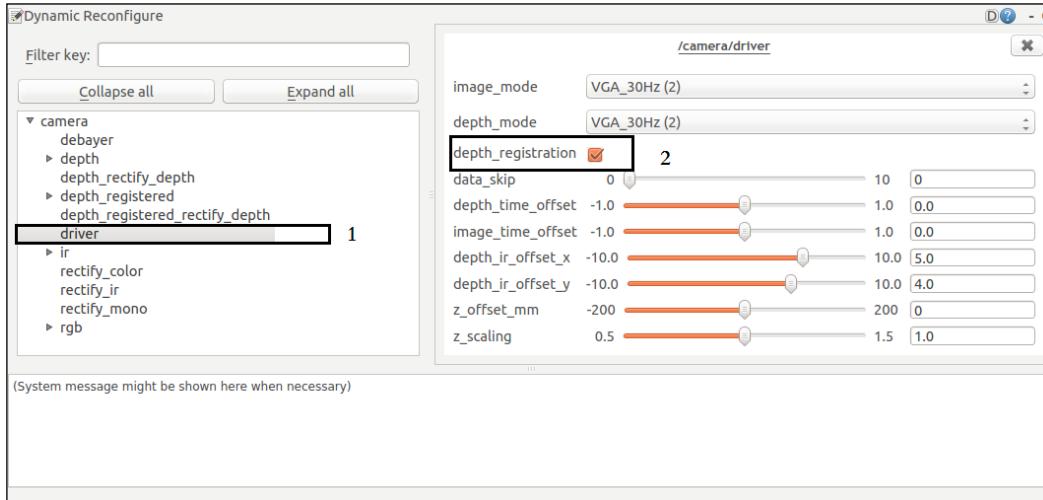


Figure 9: Dynamic Reconfigure GUI

Click on **camera | driver** and tick **depth_registration**. Change the point cloud to **/camera/depth_registered/points** and **Color Transformer** to **RGB8** in RViz. We will see the registered point cloud in RViz as it appears in the following image. Registered point cloud takes information from the depth and the RGB camera to generate the point cloud.

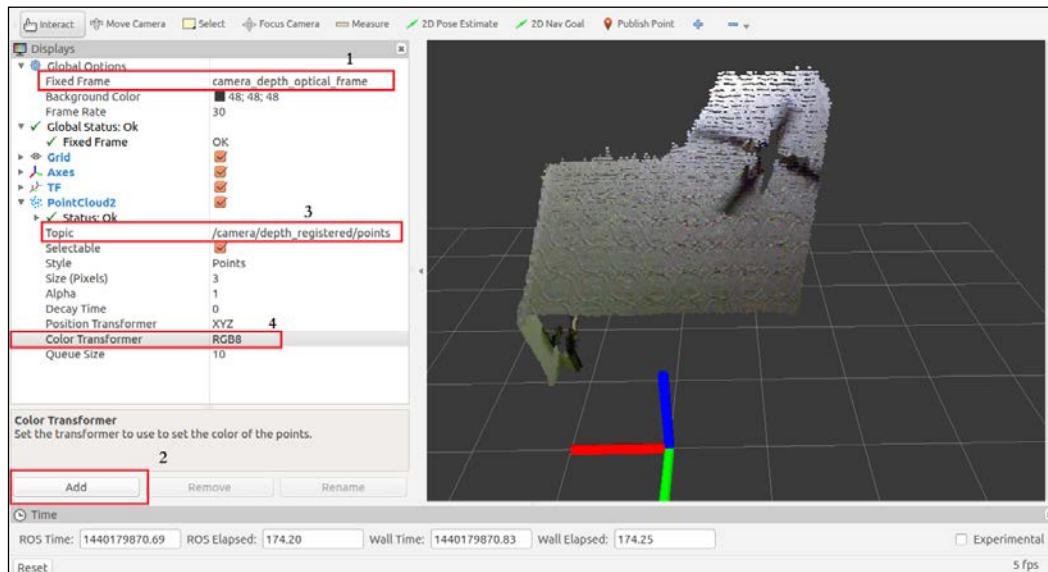


Figure 10: The registered point cloud

Interfacing Intel Real Sense camera with ROS

One of the new 3D depth sensors from Intel is Real Sense. The following link is the ROS interface of Intel Real Sense: https://github.com/BlazingForests/realsense_camera



Figure 11: Intel RealSense

Before installing the ROS driver, we have to install the following packages for building the source code:

```
$ sudo apt-get install libudev-dev libv4l-dev
```

After installing, clone the ROS package to the `src` folder of catkin workspace:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/BlazingForests/realsense_camera.git
$ catkin_make
```

Launch the Real Sense camera driver and RViz using the following command:

```
$ roslaunch realsense_camera realsense_rviz.launch
```

Launch Real Sense camera driver only:

```
$ roslaunch realsense_camera realsense_camera.launch
```

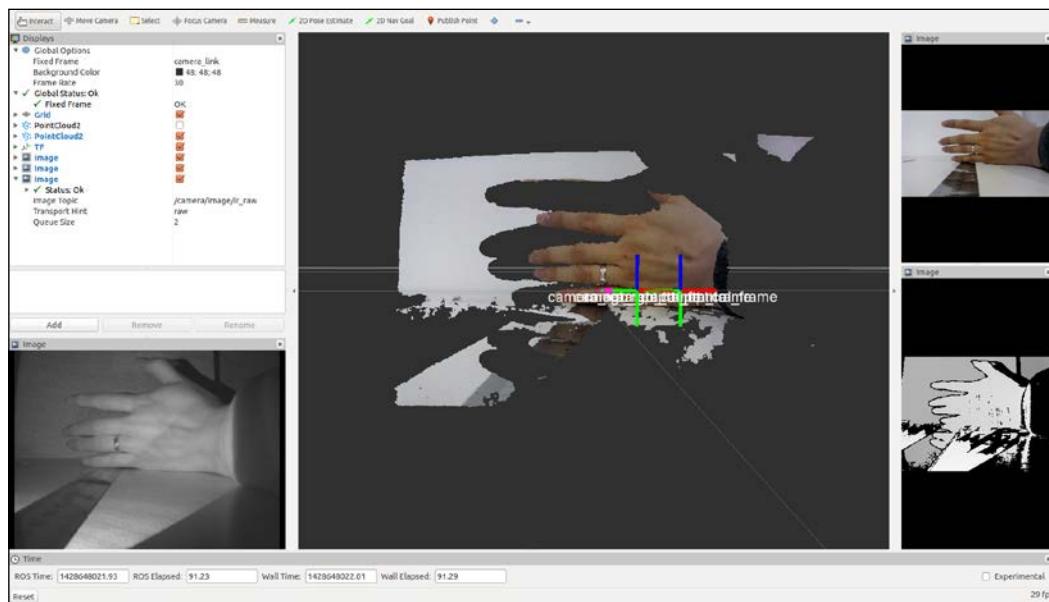


Figure 12: Intel Real Sense view in RViz

Following are the topics generated by the Real Sense driver:

sensor_msgs::PointCloud2	
/camera/depth/points	point cloud without RGB
/camera/depth_registered/points	point cloud with RGB
sensor_msgs::Image	
/camera/image/rgb_raw	raw image for RGB sensor
/camera/image/depth_raw	raw image for depth sensor
/camera/image/ir_raw	raw image for infrared sensor

Working with point cloud to laser scan package

One of the important applications of 3D vision sensors is mimicking the functionalities of a laser scanner. We need the laser scanner data for working with autonomous navigation algorithms such as SLAM. We can make a fake laser scanner using a 3D vision sensor. We can take a slice of point cloud data/depth image and convert it to laser range data. In ROS, we have a set of packages to convert the point cloud to laser scans:

- `depthimage_to_laserscan`: This package contains nodes that take the depth image from the vision sensor and generate 2D laser scan based on the provided parameters. The input of the node are depth image and camera info parameters, which include calibration parameters. After conversion to laser scan data, it will publish laser scanner data in the `/scan` topic. The node parameters are `scan_height`, `scan_time`, `range_min`, `range_max`, and output frame ID. The official ROS wiki page of this package is http://wiki.ros.org/depthimage_to_laserscan.
- `pointcloud_to_laserscan`: This package converts the real point cloud data into 2D laser scan, instead of taking depth image as the previous package. The official wiki page of this package is http://wiki.ros.org/pointcloud_to_laserscan.

The first package is suitable for normal applications; however, if the sensor is placed in an angle, it is better to use the second package. Also, the first package takes less processing than the second one. Here we are using the `depthimage_to_laserscan` package to convert laser scan. We can install `depthimage_to_laserscan` using the following commands:

- In Jade:

```
$ sudo apt-get install ros-jade-depthimage-to-laserscan
```

- In Indigo:

```
$ sudo apt-get install ros-indigo-depthimage-to-laserscan
```

We can install the pointcloud_to_laser scanner package using the following commands:

- In Jade:

```
$ sudo apt-get install ros-jade-pointcloud-to-laserscan
```

- In Indigo:

```
$ sudo apt-get install ros-indigo-pointcloud-to-laserscan
```

We can start converting from the depth image of OpenNI device to 2D laser scanner using the following package.

Creating a package for performing the conversion:

```
$ catkin_create_pkg fake_laser_pkg depthimage_to_laserscan nodelet  
roscpp
```

Create a folder called launch and inside this folder create the following launch file called start_laser.launch. You will get this package and file from the chapter_8_codes/fake_laser_pkg/launch folder.

```
<launch>  
    <!-- "camera" should uniquely identify the device. All topics  
        are pushed down  
        into the "camera" namespace, and it is prepended to tf  
        frame ids. -->  
    <arg name="camera"      default="camera"/>  
    <arg name="publish_tf" default="true"/>  
    .....  
    .....  
  
    <group if="$(arg scan_processing)">  
        <node pkg="nodelet" type="nodelet"  
            name="depthimage_to_laserscan" args="load  
            depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg  
            camera)/$(arg camera)_nodelet_manager">  
            <!-- Pixel rows to use to generate the laserscan. For each  
                column, the scan will return the minimum value for those  
                pixels centered vertically in the image. -->  
            <param name="scan_height" value="10"/>  
            <param name="output_frame_id" value="/$(arg  
            camera)_depth_frame"/>  
            <param name="range_min" value="0.45"/>
```

```

<remap from="image" to="$(arg camera) /$(arg
depth)/image_raw"/>
<remap from="scan" to="$(arg scan_topic)"/>

. . .
. . .

</launch>
```

The following code snippet will launch the nodelet for converting the depth image to laser scanner:

```

<node pkg="nodelet" type="nodelet"
name="depthimage_to_laserscan" args="load
depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg
camera) /$(arg camera)_nodelet_manager">
```

Launch this file and we can view the laser scanner in RViz.

Launch this file using the following command:

```
$ rosrun fake_laser_pkg start_laser.launch
```

We will see the data in RViz, as shown in the following image:

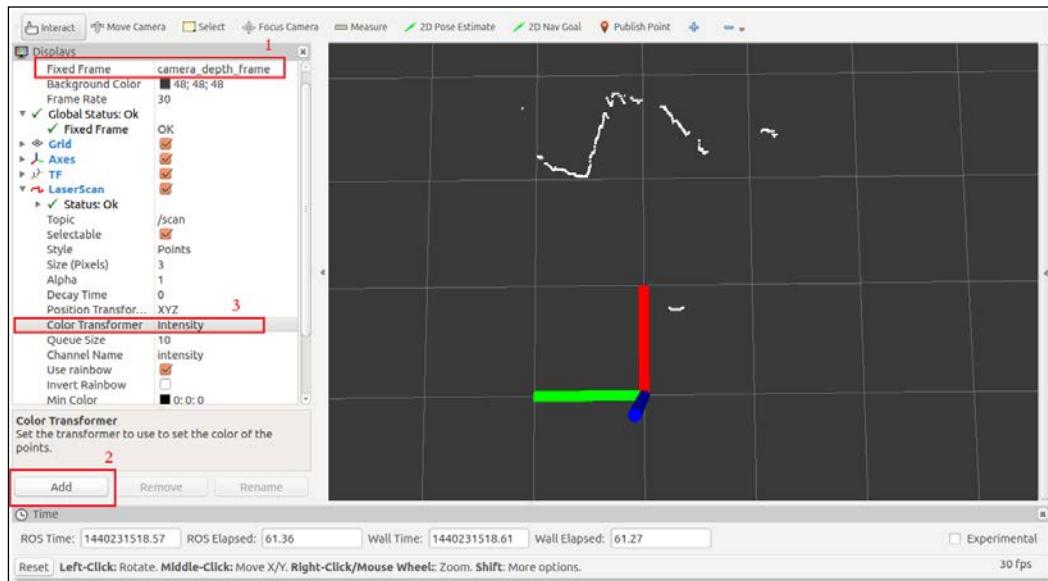


Figure 13: Laser scan in RViz

Set **Fixed Frame** as `camera_depth_frame` and **Add the LaserScan** in topic /`scan`. We can see the laser data in the view port.

Interfacing Hokuyo Laser in ROS

We can interface different ranges of laser scanners in ROS. One of the popular laser scanner available in the market is Hokuyo Laser scanner (<http://www.robotshop.com/en/hokuyo-utm-03lx-laser-scanning-rangefinder.html>).



Figure 14: Different series of Hokuyo laser scanner

One of the commonly used Hokuyo laser scanner models is UTM-30LX. This sensor is fast and accurate, suitable for robotic applications. The device has USB 2.0 interface for communication, and has up to 30 meter range with millimeter resolution. The arc range of the scan is about 270 degrees.



Figure 15 : Hokuyo UTM-30LX

There is already a driver available in ROS for interfacing these scanners. One of the interfaces is called `hokuyo_node` (http://wiki.ros.org/hokuyo_node).

We can install this package using the following command:

- In Jade:

```
$ sudo apt-get install ros-jade-hokuyo-node
```

- In Indigo:

```
$ sudo apt-get install ros-indigo-hokuyo-node
```

When the device connects to the Ubuntu system, it will create a device called `ttyACMx`. Check the device name by entering the `dmesg` command in the terminal. Change the USB device permission by using the following command:

```
$ sudo chmod a+rwx /dev/ttyACMx
```

Start the laser scan device using the following launch file called `hokuyo_start.launch`:

```
<launch>
  <node name="hokuyo" pkg="hokuyo_node" type="hokuyo_node"
    respawn="false" output="screen">

    <!-- Starts up faster, but timestamps will be inaccurate. -->
    <param name="calibrate_time" type="bool" value="false"/>

    <param name="min_ang" type="double" value="-0.7854"/>
    <param name="max_ang" type="double" value="0.7854"/>

    <!-- Set the port to connect to here -->
    <param name="port" type="string" value="/dev/ttyACM0"/>

    <param name="intensity" type="bool" value="false"/>
  </node>

  <node name="rviz" pkg="rviz" type="rviz" respawn="false"
    output="screen" args="-d $(find hokuyo_node)/hokuyo_test.vcg"/>

</launch>
```

This launch file starts a hokuyo node for getting the laser data from the device /dev/ttyACM0. The laser data can be viewed inside the RViz window, as shown in the following image:

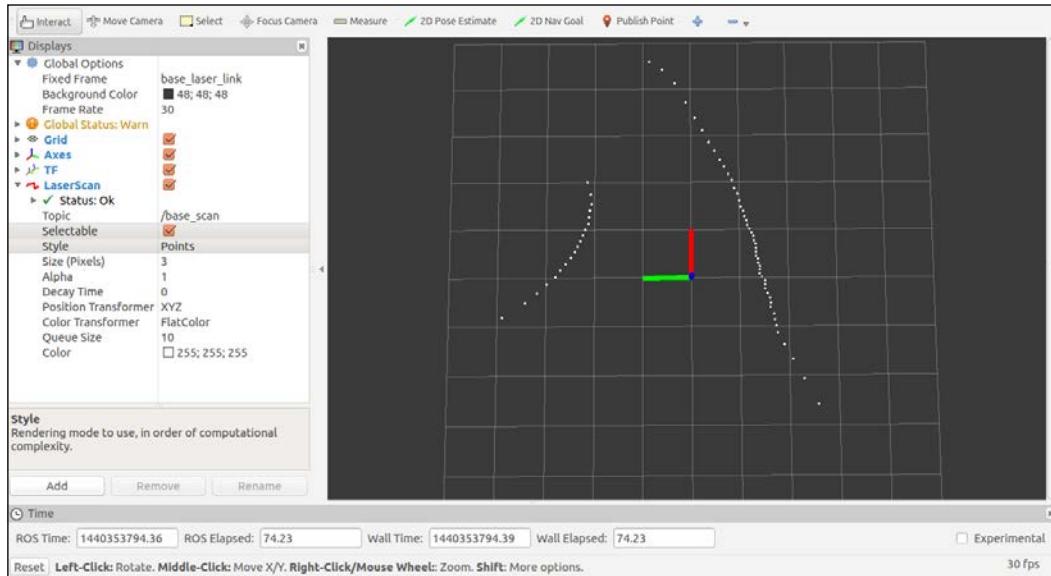


Figure 16: Hokuyo Laser scan data in RViz

Interfacing Velodyne LIDAR in ROS

One of the trending areas in robotics is autonomous cars or driverless cars. One of the essential ingredients in this robot is a **Light Detection and Ranging (LIDAR)**. One of the commonly used LIDARs is Velodyne LIDAR. Velodyne LIDARs are used in Google driverless cars and also in most of the research in driver less cars. There are three models of Velodyne LIDAR available in the market. Following are the three models and their diagrams:

Velodyne HDL-64E, Velodyne HDL-32E, and Velodyne VLP-16/Puck.



Figure 17: Different series of Velodyne

Velodyne can interface to ROS and can generate point cloud data from its raw data. The link for the velodyne ROS package for model HDL-32E is <http://wiki.ros.org/velodyne>.

We can install the velodyne driver in Ubuntu using the following command:

- In Jade:
`$ sudo apt-get install ros-jade-velodyne`
- In Indigo:
`$ sudo apt-get install ros-indigo-velodyne`

After installing these packages, connect the LIDAR power supply and connect Ethernet cable from the PC to Velodyne.

Assign a static IP of the PC in the range 192.168.3.x using the following command:

```
$ sudo ifconfig eth0 192.168.3.100
```

After setting the static IP of the PC, assign a route to Velodyne. The IP of LIDAR will be present on the CD gotten along with the Velodyne.

```
$ sudo route add 192.168.XX.YY eth0
```

After setting the network, we need to generate calibration data in YAML file. The following command will generate the calibration data in a YAML file from the standard Velodyne XML file:

```
$ rosrun velodyne_pointcloud gen_calibration.py 32db.xml
```

Launch the point cloud generation nodes from the raw data of LIDAR. We have to mention the generated calibration YAML file along with the launch file:

```
$ roslaunch velodyne_pointcloud 32e_points.launch
calibration:=~/home/robot/32db.yaml
```

After launching the converter nodes, we can start RViz to view the point cloud data generated from LIDAR using the following command. Set the **Fixed Frame** as Velodyne and **Add** display **Point Cloud 2** and set **Topic** as /velodyne_points:

```
$ rosrun rviz rviz -f velodyne
```

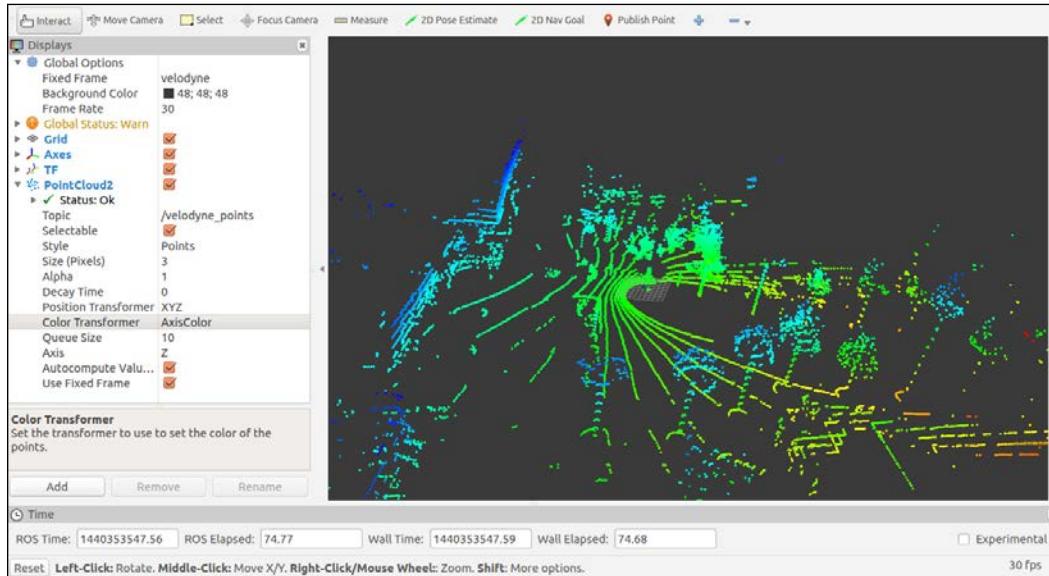


Figure 18: Velodyne point cloud view in RViz

Working with point cloud data

We can handle the point cloud data from Kinect or the other 3D sensors for performing wide variety of tasks such as 3D object detection and recognition, obstacle avoidance, 3D modeling, and so on. In this section, we will see some basic functionalities using the PCL library and its ROS interface. We will discuss the following examples:

- How to publish a point cloud in ROS
- How to subscribe and process point cloud
- How to write point cloud data to a PCD file
- How to read and publish point cloud from a PCD file

How to publish a point cloud

In this example, we will see how to publish a point cloud data using the `sensor_msgs/PointCloud2` message. The code will use PCL APIs for handling and creating the point cloud, and converting the PCL cloud data to `PointCloud2` message type. You will get the example code `pcl_publisher.cpp` from the `chapter_8_codes/pcl_ros_tutorial/src` folder.

```
#include <ros/ros.h>

// point cloud headers
#include <pcl/point_cloud.h>
//Header which contain PCL to ROS and ROS to PCL conversion functions
#include <pcl_conversions/pcl_conversions.h>

//sensor_msgs header for point cloud2
#include <sensor_msgs/PointCloud2.h>

main (int argc, char **argv)
{
    ros::init (argc, argv, "pcl_create");

    ROS_INFO("Started PCL publishing node");

    ros::NodeHandle nh;

    //Creating publisher object for point cloud
```

```
ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
("pcl_output", 1);

//Creating a cloud object
pcl::PointCloud cloud;

//Creating a sensor_msg of point cloud

sensor_msgs::PointCloud2 output;

//Insert cloud data
cloud.width = 50000;
cloud.height = 2;
cloud.points.resize(cloud.width * cloud.height);

//Insert random points on the clouds

for (size_t i = 0; i < cloud.points.size (); ++i)
{
    cloud.points[i].x = 512 * rand () / (RAND_MAX + 1.0f);
    cloud.points[i].y = 512 * rand () / (RAND_MAX + 1.0f);
    cloud.points[i].z = 512 * rand () / (RAND_MAX + 1.0f);
}

//Convert the cloud to ROS message
pcl::toROSMsg(cloud, output);
output.header.frame_id = "point_cloud";

ros::Rate loop_rate(1);
while (ros::ok())
{
    //publishing point cloud data
    pcl_pub.publish(output);
    ros::spinOnce();
    loop_rate.sleep();
}

return 0;
}
```

The creation of PCL cloud is done as follows:

```
//Creating a cloud object  
pcl::PointCloud<pcl::PointXYZ> cloud;
```

After creating this cloud, we insert random points to the clouds. We convert the PCL cloud to a ROS message using the following function:

```
//Convert the cloud to ROS message  
pcl::toROSMsg(cloud, output);
```

After converting to ROS messages, we can simply publish the data on the topic /pcl_output.

How to subscribe and process the point cloud

In this example, we will see how to subscribe the generated point cloud on the topic pcl_output. After subscribing the point cloud, we apply a filter called the VoxelGrid class in PCL to down sample the input cloud by keeping the same centroid of the input cloud. You will get the example code pcl_filter.cpp from the src folder of the package.

```
#include <ros/ros.h>  
#include <pcl/point_cloud.h>  
#include <pcl_conversions/pcl_conversions.h>  
#include <sensor_msgs/PointCloud2.h>  
//Vortex filter header  
#include <pcl/filters/voxel_grid.h>  
  
//Creating a class for handling cloud data  
class cloudHandler  
{  
public:  
    cloudHandler()  
    {  
  
        //Subscribing pcl_output topics from the publisher  
        //This topic can change according to the source of point cloud  
  
        pcl_sub = nh.subscribe("pcl_output", 10, &cloudHandler::cloudCB,  
        this);  
        //Creating publisher for filtered cloud data  
        pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_  
        filtered", 1);  
    }  
};
```

```
//Creating cloud callback
void cloudCB(const sensor_msgs::PointCloud2& input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::PointCloud<pcl::PointXYZ> cloud_filtered;

    sensor_msgs::PointCloud2 output;
    pcl::fromROSMsg(input, cloud);

    //Creating VoxelGrid object
    pcl::VoxelGrid<pcl::PointXYZ> vox_obj;
    //Set input to voxel object
    vox_obj.setInputCloud (cloud.makeShared());

    //Setting parameters of filter such as leaf size
    vox_obj.setLeafSize (0.1f, 0.1f, 0.1f);

    //Performing filtering and copy to cloud_filtered variable
    vox_obj.filter(cloud_filtered);
    pcl::toROSMsg(cloud_filtered, output);
    output.header.frame_id = "point_cloud";
    pcl_pub.publish(output);
}

protected:
ros::NodeHandle nh;
ros::Subscriber pcl_sub;
ros::Publisher pcl_pub;
};

main(int argc, char** argv)
{
    ros::init(argc, argv, "pcl_filter");
    ROS_INFO("Started Filter Node");
    cloudHandler handler;
    ros::spin();
    return 0;
}
```

This code subscribes the point cloud topic called `/pcl_output`, filters using `VoxelGrid`, and publishes the filtered cloud through the `/cloud_filtered` topic.

Writing a point cloud data to a PCD file

We can save the point cloud to a **PCD (Point Cloud Data)** file by using the following code. The file name is `pcl_write.cpp` inside the `src` folder.

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
//Header file for writing PCD file
#include <pcl/io/pcd_io.h>

void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::fromROSMsg(input, cloud);

    //Save data as test.pcd file
    pcl::io::savePCDFileASCII ("test.pcd", cloud);
}

main (int argc, char **argv)
{
    ros::init (argc, argv, "pcl_write");

    ROS_INFO("Started PCL write node");

    ros::NodeHandle nh;
    ros::Subscriber bat_sub = nh.subscribe("pcl_output", 10, cloudCB);

    ros::spin();

    return 0;
}
```

Read and publish point cloud from a PCD file

This code can read a PCD file and publish the point cloud in the `/pcl_output` topic. The code `pcl_read.cpp` is available in the `src` folder.

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
```

```
#include <pcl/io/pcd_io.h>

main(int argc, char **argv)
{
    ros::init (argc, argv, "pcl_read");

    ROS_INFO("Started PCL read node");

    ros::NodeHandle nh;
    ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
    ("pcl_output", 1);

    sensor_msgs::PointCloud2 output;
    pcl::PointCloud cloud;

    //Load test.pcd file
    pcl::io::loadPCDFile ("test.pcd", cloud);

    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "point_cloud";

    ros::Rate loop_rate(1);
    while (ros::ok())
    {
        //Publishing the cloud inside pcd file
        pcl_pub.publish(output);
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

We can create a ROS package called `pcl_ros_tutorial` for compiling these examples:

```
$ catkin_create_pkg pcl_ros_tutorial pcl pcl_ros roscpp sensor_msgs
```

Otherwise, we can use the existing package.

Create the preceding examples inside `src` as `pcl_publisher.cpp`, `pcl_filter.cpp`, `pcl_write.cpp`, and `pcl_read.cpp`.

Create CMakeLists.txt for compiling all the sources:

```
## Declare a cpp executable
add_executable(pcl_publisher_node src/pcl_publisher.cpp)
add_executable(pcl_filter src/pcl_filter.cpp)
add_executable(pcl_write src/pcl_write.cpp)
add_executable(pcl_read src/pcl_read.cpp)

target_link_libraries(pcl_publisher_node
    ${catkin_LIBRARIES}
)
target_link_libraries(pcl_filter
    ${catkin_LIBRARIES}
)
target_link_libraries(pcl_write
    ${catkin_LIBRARIES}
)
target_link_libraries(pcl_read
    ${catkin_LIBRARIES}
)
```

Build this package using catkin_make, and we can run pcl_publisher_node and view point cloud inside RViz using the following command:

```
$ rosrun rviz rviz -f point_cloud
```

A screenshot of the point cloud from pcl_output is shown in the following image:

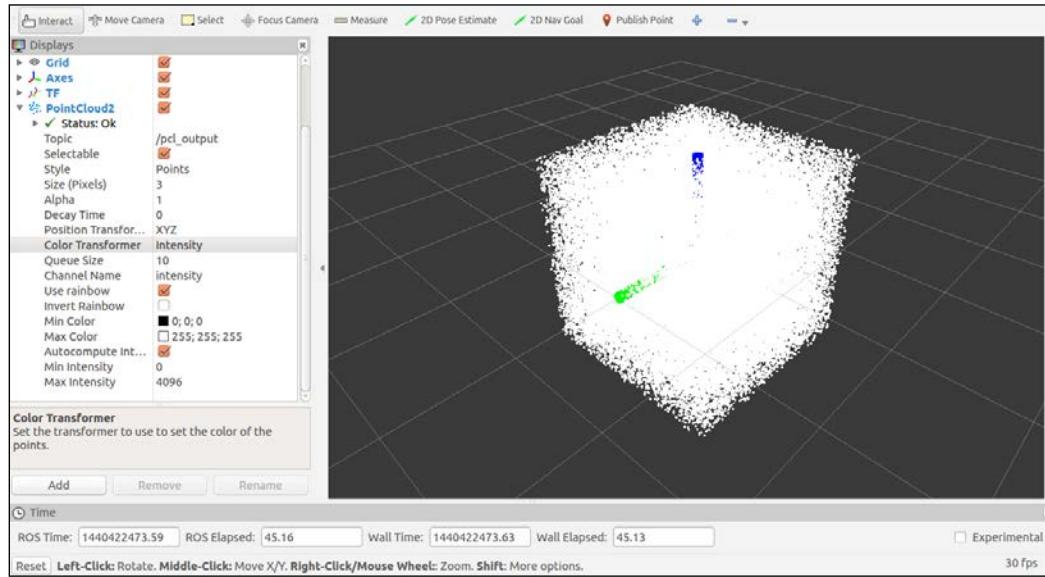


Figure 19: PCL cloud in RViz

We can run the `pcl_filter` node to subscribe this same cloud and do voxel grid filtering. The following screenshot shows the output from `/pcl_filtered` topic, which is the resultant down sampled cloud:

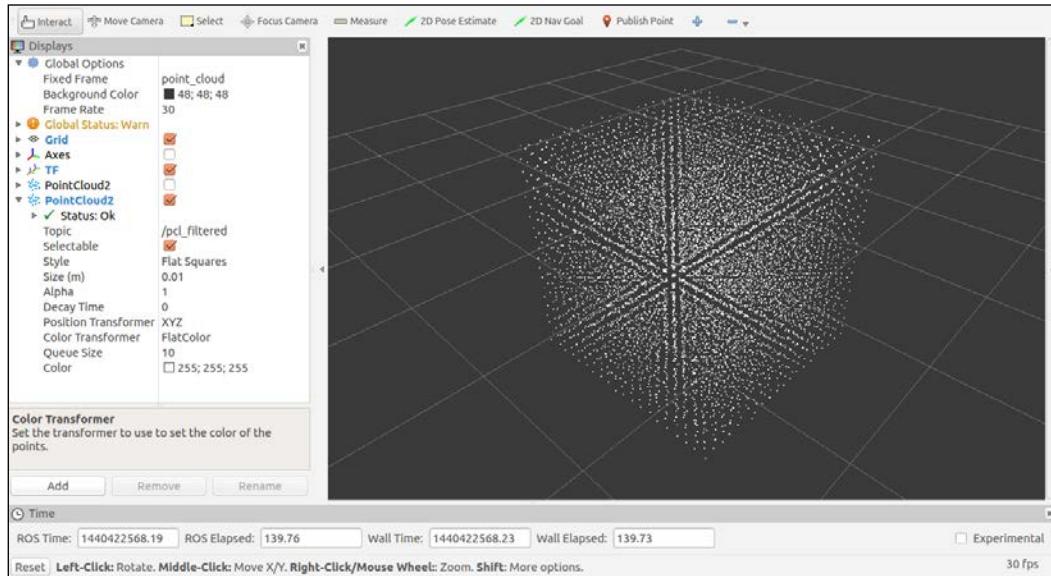


Figure 20 : Filtered PCL cloud in RViz

We can write the `pcl_output` cloud using the `pcl_write` node and read/publish using the `pcl_read` nodes.

Streaming webcam from Odroid using ROS

ROS system is designed mainly for distributive computing. We can write and run the ROS nodes on multiple machines and communicate each node to a single master. For communicating between two devices using ROS, we should follow the following rules:

- Only single ROS master should run; we can decide which machine should run the master
- All machines should be configured to use the same master URI through `ROS_MASTER_URI`
- Bi-directional connectivity should be ensured between all the pairs of machines
- Each machine should have a name that can be identified by the other machines

In this section, we will see how to run the ROS master in Odroid and stream the camera images to a PC. First, we will look at the setup required for the distributing computing between Odroid and PC.

Connect Odroid to the PC directly using the LAN cable and create a Ethernet hotspot, as we mentioned in the previous chapter. Find the IPs of Odroid and the PC and set the following lines of command in their .bashrc files. We are going to run the Odroid board as the ROS master and the PC as a computing node. Following is a sample configuration of Odroid and PC:

- Configuring Odroid as ROS master:

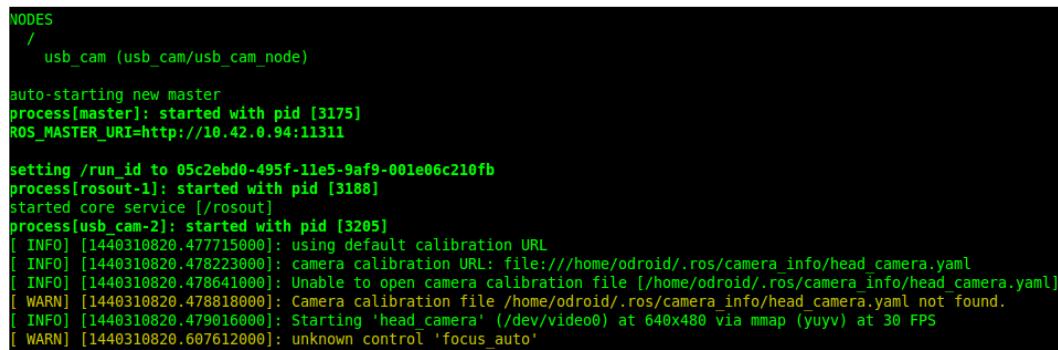
```
#Setting MY_IP as Odroid IP
export MY_IP=10.42.0.94
#Setting ROS_IP variable as MY_IP
export ROS_IP=$MY_IP
#Setting ROS_MASTER_URI as Odroid IP
export ROS_MASTER_URI="http://10.42.0.94:11311"
```

- Configuring PC as ROS computing node:

```
#Setting MY_IP as P.C IP
export MY_IP=10.42.0.1
#Setting ROS_IP variable as MY_IP
export ROS_IP=$MY_IP
#Setting ROS_MASTER_URI as Odroid IP
export ROS_MASTER_URI="http://10.42.0.94:11311"
```

Install a `usb_cam` ROS package in Odroid, connect a USB web cam to it, and start running `usb_cam` on it using the following command:

```
$ roslaunch usb_cam usb_cam-test.launch
```



The terminal window displays the following ROS log output:

```
NODES
/
  usb_cam (usb_cam/usb_cam_node)

auto-starting new master
process[master]: started with pid [3175]
ROS_MASTER_URI=http://10.42.0.94:11311

setting /run_id to 05c2ebd0-495f-11e5-9af9-001e06c210fb
process[rosout-1]: started with pid [3188]
started core service [/rosout]
process[usb_cam-2]: started with pid [3205]
[ INFO] [1440310820.477715000]: using default calibration URL
[ INFO] [1440310820.478223000]: camera calibration URL: file:///home/odroid/.ros/camera_info/head_camera.yaml
[ INFO] [1440310820.478641000]: Unable to open camera calibration file [/home/odroid/.ros/camera_info/head_camera.yaml]
[ WARN] [1440310820.478818000]: Camera calibration file /home/odroid/.ros/camera_info/head_camera.yaml not found.
[ INFO] [1440310820.479016000]: Starting 'head_camera' (/dev/video0) at 640x480 via mmap (yuyv) at 30 FPS
[ WARN] [1440310820.607612000]: unknown control 'focus_auto'
```

Figure 21: Terminal message generated by `usb_cam` node

In the PC terminal, we can access the camera topics and display the image data in RViz.

Following are the camera topics in the PC that are running on Odroid:

```
$ rostopic list
```

```
lentin@lentin-Aspire-4755:~$ rostopic list
/clicked_point
/initialpose
/move_base_simple/goal
/rosout
/rosout_agg
/tf
/tf_static
/usb_cam/camera_info
/usb_cam/image_raw
/usb_cam/image_raw/compressed
/usb_cam/image_raw/compressed/parameter_descriptions
/usb_cam/image_raw/compressed/parameter_updates
/usb_cam/image_raw/compressedDepth
/usb_cam/image_raw/compressedDepth/parameter_descriptions
/usb_cam/image_raw/compressedDepth/parameter_updates
/usb_cam/image_raw/theora
/usb_cam/image_raw/theora/parameter_descriptions
/usb_cam/image_raw/theora/parameter_updates
lentin@lentin-Aspire-4755:~$
```

Figure 22: Topics generated by Odroid, which is viewed on PC terminal

We can view the image from the Odroid cam on the PC using RViz or the image_view tool. Following is an image of Odroid camera stream in RViz:

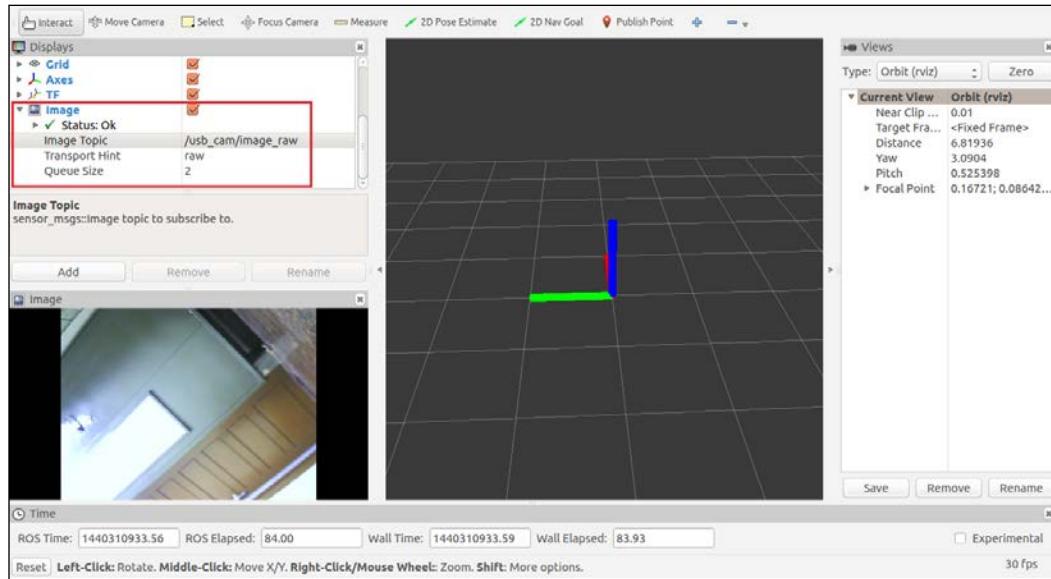


Figure 21 : Odroid camera view in RViz running on PC

Questions

1. What are the packages in the `vision_opencv` stack?
2. What are the packages in the `perception_pcl` stack?
3. What are the functions of `cv_bridge`?
4. How do we convert PCL cloud to ROS message?
5. How do we do distributive computing using ROS?

Summary

This chapter was about vision sensors and its programming in ROS. We saw the interfacing packages to interface the cameras and 3D vision sensors such as `vision_opencv` and `perception_pcl`. We looked at each package and its functions on these stacks. We saw interfacing of basic webcam and processing image using ROS `cv_bridge`. After discussing `cv_bridge`, we looked at the interfacing of various 3D vision sensors and laser scanners with ROS. After interfacing, we learned how to process the data from these sensors using PCL library and ROS. At the end of the chapter, we understood how to stream a camera from an embedded device called Odroid to the PC. In the next chapter, we will see the interfacing of robotic hardware in ROS.

9

Building and Interfacing Differential Drive Mobile Robot Hardware in ROS

In the previous chapter, we have discussed about robotic vision using ROS. In this chapter, we can see discuss how to build an autonomous mobile robot hardware with differential drive configuration and how to interface it into ROS. We will see how to configure ROS Navigation stack for this robot and perform SLAM and AMCL to move the robot autonomously. This chapter aims to give you an idea about building a custom mobile robot and interfacing it on ROS.

You will see the following topics in this chapter:

- Introduction to Chefbot: a DIY autonomous mobile robot
- Flashing Chefbot firmware using Energia IDE
- Discussing Chefbot interface package in ROS
- Developing base controller and odometry node for Chefbot in ROS
- Configuring Navigation stack for Chefbot
- Understanding AMCL
- Understanding RViz for working with Navigation stack
- Obstacle avoidance using Navigation stack
- Working with Chefbot simulation
- Sending a goal to the Navigation stack from a ROS node

The first topic we are going to discuss in this chapter is how to build a **DIY (Do It Yourself)** autonomous mobile robot, developing its firmware, and interface it to ROS Navigation stack. The robot called Chefbot was built as a part of my first book called *Learning Robotics using Python* for PACKT (<http://learn-robotics.com>). This book discusses step by step procedure to build this robot and its interfacing to ROS.

In this chapter, we will cover abstract information about this robot hardware and we will learn more about configuring ROS Navigation stack and its fine tuning for performing autonomous navigation using SLAM and AMCL. We have already discussed about ROS Navigation stack in *Chapter 4, Using the ROS MoveIt! and Navigation Stack* and we have simulated a differential robot using Gazebo and performed SLAM and AMCL. In this chapter, we will see how to interface a real differential drive robot hardware to navigation package.

Introduction to Chefbot- a DIY mobile robot and its hardware configuration

In *Chapter 4, Using the ROS MoveIt! and Navigation Stack* we have discussed some mandatory requirements for interfacing a mobile robot with ROS navigation package. The following are the mandatory requirements:

- **Odometry source:** Robot should publish its odometry/position data with respect to the starting position. The necessary hardware components that provide odometry information are wheel encoders, IMU, and 2D/3D cameras (visual odometry).
- **Sensor source:** There should be a laser scanner or a 3D vision sensor sensor, which can act as a laser scanner. The laser scanner data is essential for map building process using SLAM.
- **Sensor transform using tf:** The robot should publish the transform of the sensors and other robot components using ROS transform.
- **Base controller:** The base controller is a ROS node, which can convert a twist message from Navigation stack to corresponding motor velocities.



Figure 1: Chefbot prototype

We can check the components present in the robot and determine whether they satisfy the Navigation stack requirements. The following components are present in the robot:

- **Pololu DC Gear motor with Quadrature encoder** (<https://www.pololu.com/product/1447>): The motor is operated in 12 V, 80 RPM, and 18 kg-cm torque. It takes current of 300 mA in free run and 5 A in stall condition. The motor shaft is attached to a quadrature encoder, which can deliver a maximum count of 8400 counts per revolution of the gearbox's output shaft. Motor encoders are one source of odometry of robot.
- **Pololu motor drivers** (<https://www.pololu.com/product/708>): These are dual motor controllers for Pololu motors that can support up to 30 A and motor voltage from 5.5 V to 16 V.
- **Tiva C Launchpad Controller** (<http://www.ti.com/tool/ek-tm4c123gx1>): This robot has a Tiva C LaunchPad controller for interfacing motors, encoders, sensors, and so on. Also, it can receive control commands from the PC and can send appropriate signals to the motors according to the command. This board can act as a embedded controller board of the robot. Tiva C LaunchPad board runs on 80 MHz.

- **MPU 6050 IMU:** The IMU used in this robot is **MPU 6050**, which is a combination of accelerometer, gyroscope, and **Digital Motion Processor (DMP)**. This motion processor can run sensor fusion algorithm onboard and can provide accurate results of roll, pitch, and yaw. The IMU values can be taken to calculate the odometry along with the wheel encoders.
- **Xbox Kinect/Asus Xtion Pro:** These are 3D vision sensors and we can use these sensors to mock a laser scanner. The point cloud generated from these sensors can be converted into laser scan data and used in the Navigation stack.
- **Intel NUC PC:** This is a mini PC from Intel, and we have to load this with Ubuntu and ROS. The PC is connected to Kinect and LaunchPad to retrieve the sensor values and the odometry details. The program running on the PC can compute TF of the robot and can run the Navigation stack and associated packages such as SLAM and AMCL. This PC is placed in the robot itself.

From the robot components lists, it is clear that it satisfies the requirements of the ROS navigation packages. The following figure shows the block diagram of this robot:

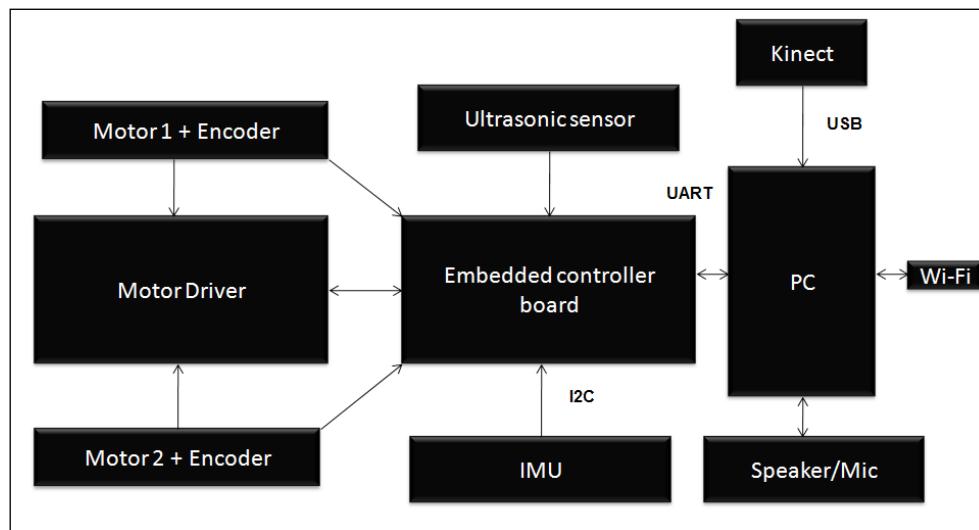


Figure 2: Block diagram of Chefbot

In this robot, the embedded controller board is the Tiva C LaunchPad. All the sensors and actuators are connected to the controller board and it is connected to Intel NUC PC for receiving higher level commands. The board and the PC communicate in UART protocol, IMU and the board communicate using I2C, Kinect is interfaced to PC via USB, and all the other sensors are interfaced through GPIO pins. A detailed connection diagram of the robot components follows:

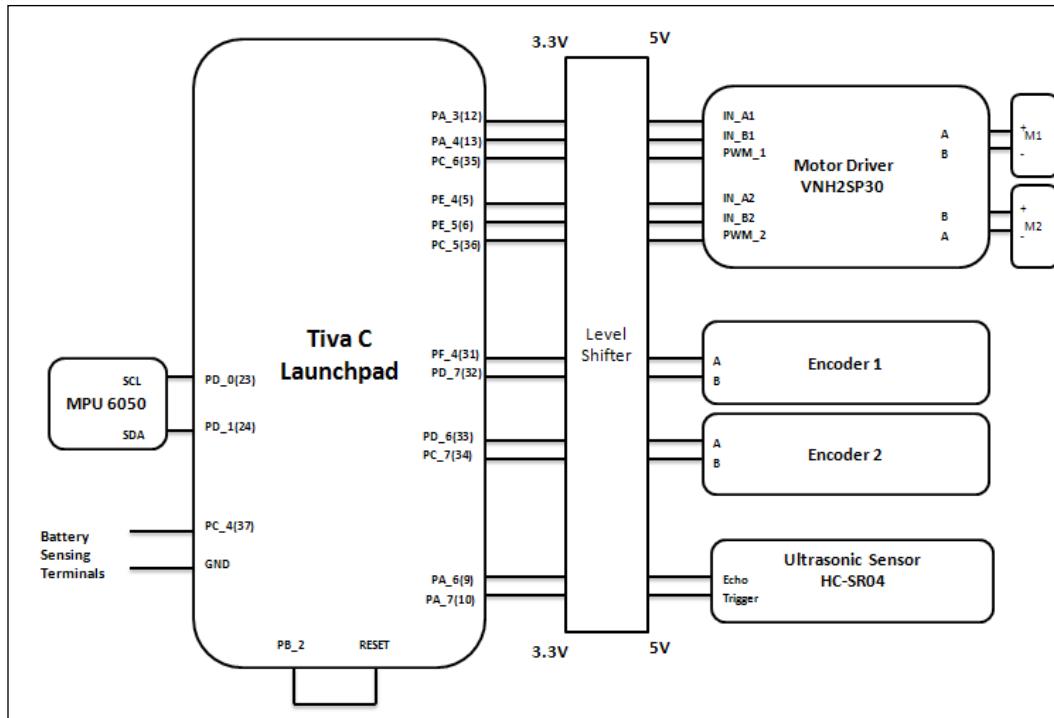


Figure 3: Connection diagram of Chefbot

Flashing Chefbot firmware using Energia IDE

After developing the preceding connections, we can program the Launchpad using Energia IDE (<http://energia.nu/>). After setting Energia IDE on the PC (Ubuntu is preferred), we can flash the robot firmware to the board. We will get the firmware code and the ROS interface package by using the following command:

```
$ git clone https://github.com/qboticslabs/Chefbot_ROS_pkg
```

The folder contains a folder called `tiva_c_energia_code`, which has the firmware code that flashes to the board after compilation in Energia IDE.

The firmware can read the encoder, ultrasonic sensor, and IMU values, and can receive values of the motor velocity command.

The important section of the firmware is discussed here. The programming language in the LaunchPad is the same as Arduino. Here we are using Energia IDE to program the controller, which is built from Arduino IDE.

The following code snippet is the `setup()` function definition of the code. This function starts serial communication with a baud rate of 115200. It also configures pins of motor encoder, motor driver pins, ultrasonic distance sensor, and IMU. Also, through this code, we are configuring a pin to reset the LaunchPad.

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

    //Setup Encoders
    SetupEncoders();
    //Setup Motors
    SetupMotors();
    //Setup Ultrasonic
    SetupUltrasonic();
    //Setup MPU 6050
    Setup_MP6050();
    //Setup Reset pins
    SetupReset();
    //Set up Messenger
    Messenger_Handler.attach(OnMessageCompleted);
}
```

In the `loop()` function, the sensor values are continuously polled and the data is sent through serial port and incoming serial data are continuously polled for getting the robot commands. The following convention protocols are used to send each sensor value from the LaunchPad to the PC using serial communication (UART).

Serial data sending protocol from LaunchPad to PC

For the encoder, the protocol will be as follows:

```
e<space><left_encoder_ticks><space><right_encoder_ticks>
```

For the ultrasonic sensor, the protocol will be as follows:

```
u<space><distance_in_centimeter>
```

For IMU, the protocol will be as follows:

```
i<space><value_of_x_quaternion><space><value_of_y_quaternion>
<space><value_of_z_quaternion><space><value_of_w_quaternion>
```

Serial data sending protocol from PC to Launchpad

For the motor, the protocol will be as follows:

```
s<space><pwm_value_of_motor_1><space><pwm_value_of_motor_2>
```

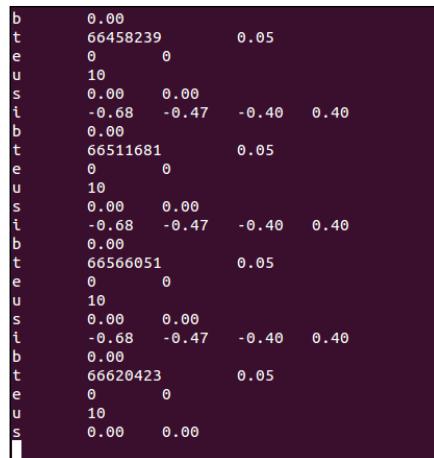
For resetting the device, the protocol will be as follows:

```
r<space>
```

We can check the serial values from the LaunchPad using a command line tool called `miniterm.py`. This tool can view the serial data coming from a device. This script is already installed with the `python-serial` package, which is installed along with the `rosserial-python` Debian package. The following command will display the serial values from the robot controller:

```
$ miniterm.py /dev/ttyACM0 115200
```

We will get values like the following screenshot:



```
b      0.00
t    66458239      0.05
e      0      0
u      10
s      0.00      0.00
i     -0.68     -0.47     -0.40     0.40
b      0.00
t    66511681      0.05
e      0      0
u      10
s      0.00      0.00
i     -0.68     -0.47     -0.40     0.40
b      0.00
t    66566051      0.05
e      0      0
u      10
s      0.00      0.00
i     -0.68     -0.47     -0.40     0.40
b      0.00
t    66620423      0.05
e      0      0
u      10
s      0.00      0.00
```

Figure 4: Checking serial data using `miniterm.py`

Discussing Chefbot interface packages on ROS

After confirming the serial values from the board, we can install the Chefbot ROS package. The Chefbot package contains the following files and folders:

- `chefbot_bringup`: This package contains python scripts, C++ nodes, and launch files to start publishing robot odometry and `tf`, and performing gmapping and AMCL. It contains the python/C++ nodes to read/write values from the LaunchPad, convert the encoder ticks to `tf`, and `twist` message to motor commands. It also has the PID node for handling velocity commands from the motor commands.
- `chefbot_description`: This package contains the Chefbot URDF model.
- `chefbot_simulator`: This package contains launch files to simulate the robot in Gazebo.
- `chefbot_navig_cpp`: This package contains C++ implementation of few nodes which are already implemented in `chefbot_bringup` as the python node.

The following launch file will start the robot odometry and `tf` publishing nodes:

```
$ roslaunch chefbot_bringup robot_standalone.launch
```

The following figure shows the nodes started by this launch file and how they are interconnected:

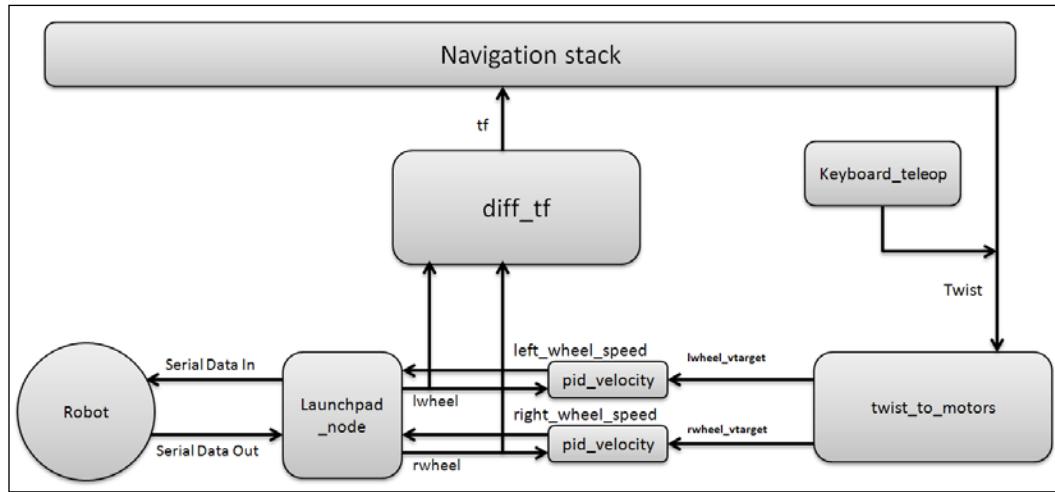


Figure 5: Interconnection of each nodes in Chefbot

The nodes run by this launch file and their working are described next:

- `launchpad_node.py`: We know that this robot uses Tiva C LaunchPad board as its controller. This node acts as a bridge between the robot controller and ROS. The basic functionality of this node is to receive serial values from the LaunchPad and convert each sensor data into ROS topics. This acts as the ROS driver for the LaunchPad board.
- `twist_to_motors.py`: This node converts the `geometry_msgs/Twist` message to motor velocity targets. It subscribes the command velocity, which is either from a teleop node or from a ROS Navigation stack, and publishes `lwheel_vtarget` and `rwheel_vtarget`.
- `pid_velocity.py`: This node subscribes `wheel_vtarget` from the `twist_to_motors` node and the `wheel` topic, which is the encoder ticks from `launchpad_node`. We have to start two PID nodes for each wheel of the robot, as shown in the previous figure. This node finally generates the motor speed commands for each motor.
- `diff_tf.py`: This node subscribes the encoder ticks from the two motors and computes odometry, and publishes `tf` for the Navigation stack.

The list of topics generated after running `robot_standalone.launch` are shown in the following image:

```

lentin@lentin-Aspire-4755:~$ rostopic list
/battery_level
/cmd_vel_mux/input/teleop
 imu/data
/joint_states
/left_wheel_speed
/lwheel
/lwheel_vel
/lwheel_vtarget
/odom
/qw
/qx
/qy
/qz
/right_wheel_speed
/rosout
/rosout_agg
/rwheel
/rwheel_vel
/rwheel_vtarget
/serial
/tf
/ultrasonic_distance

```

Figure 6: List of topic generated when executing `robot_standalone.launch`

The following is the content of the `robot_standalone.launch` file:

```
<launch>
  <arg name="simulation" default="$(optenv TURTLEBOT_SIMULATION
false)"/>
  <param name="/use_sim_time" value="$(arg simulation)"/>

  <!-- URDF robot model -->
  <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find
chefbot_description)/urdf/chefbot_base.xacro'" />
  <param name="robot_description" command="$(arg urdf_file)" />

  <!-- important generally, but specifically utilised by the current
app manager -->
  <param name="robot/name" value="$(optenv ROBOT turtlebot)"/>
  <param name="robot/type" value="turtlebot"/>

  <!-- Starting robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher">
    <param name="publish_frequency" type="double" value="5.0" />
  </node>

  <!-- Robot parameters -->
  <rosparam param="base_width">0.3</rosparam>
  <rosparam param="ticks_meter">14865</rosparam>

  <!-- Starting launchpad_node -->
  <node name="launchpad_node" pkg="chefbot_bringup" type="launchpad_
node.py">
    <rosparam file="$(find chefbot_bringup)/param/serial.yaml"
command="load" />
  </node>

  <!-- PID node for left motor , setting PID parameters -->
  <node name="lpid_velocity" pkg="chefbot_bringup" type="pid_velocity.
py" output="screen">
    <remap from="wheel" to="lwheel"/>
    <remap from="motor_cmd" to="left_wheel_speed"/>
    <remap from="wheel_vtarget" to="lwheel_vtarget"/>
    <remap from="wheel_vel" to="lwheel_vel"/>

    <rosparam param="Kp">400</rosparam>
```

```
<rosparam param="Ki">100</rosparam>
<rosparam param="Kd">0</rosparam>
<rosparam param="out_min">-1023</rosparam>
<rosparam param="out_max">1023</rosparam>
<rosparam param="rate">30</rosparam>
<rosparam param="timeout_ticks">4</rosparam>
<rosparam param="rolling_pts">5</rosparam>
</node>

<!-- PID node for right motor, setting PID parameters -->
<node name="rpid_velocity" pkg="chefbot_bringup" type="pid_velocity.py" output="screen">
    <remap from="wheel" to="rwheel"/>
    <remap from="motor_cmd" to="right_wheel_speed"/>
    <remap from="wheel_vtarget" to="rwheel_vtarget"/>
    <remap from="wheel_vel" to="rwheel_vel"/>
    <rosparam param="Kp">400</rosparam>
    <rosparam param="Ki">100</rosparam>
    <rosparam param="Kd">0</rosparam>
    <rosparam param="out_min">-1023</rosparam>
    <rosparam param="out_max">1023</rosparam>
    <rosparam param="rate">30</rosparam>
    <rosparam param="timeout_ticks">4</rosparam>
    <rosparam param="rolling_pts">5</rosparam>
</node>

<!-- Starting twist to motor and diff_tf nodes -->

<node pkg="chefbot_bringup" type="twist_to_motors.py" name="twist_to_motors" output="screen"/>
<node pkg="chefbot_bringup" type="diff_tf.py" name="diff_tf" output="screen"/>

</launch>
```

After running `robot_standalone.launch`, we can visualize the robot in RViz using the following command:

```
$ rosrun chefbot_bringup view_robot.launch
```

We will see the robot model as shown in this next screenshot:

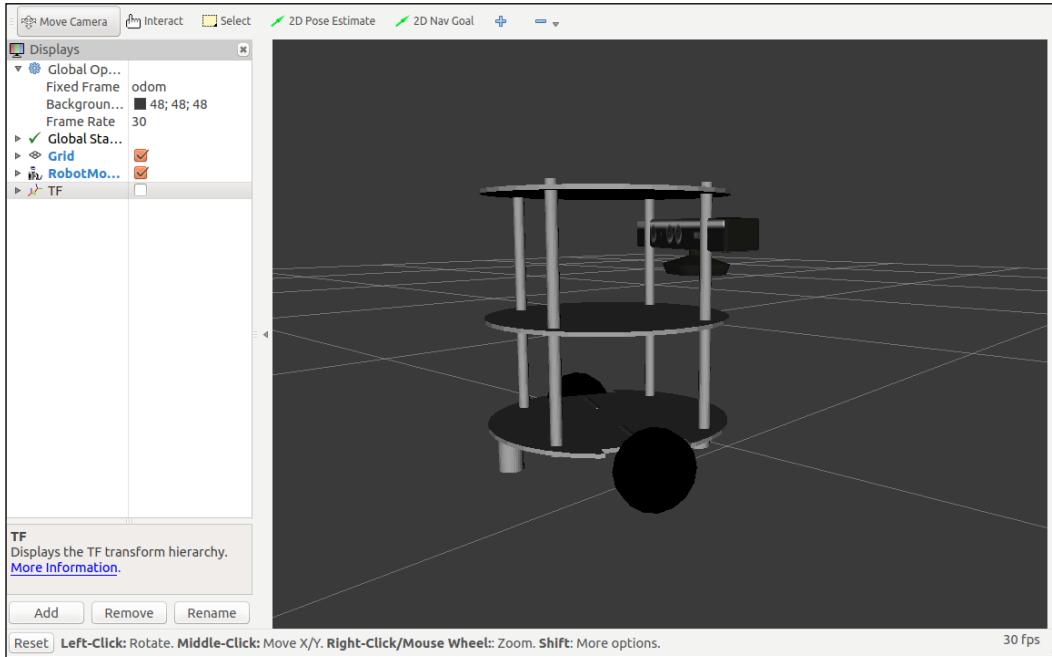


Figure 7: Visualization of robot model using real robot values.

Launch the keyboard teleop node and we can start moving the robot:

```
$ roslaunch chefbot_bringup keyboard_teleop.launch
```

Move the robot using the keys and we will see that the robot is moving around. If we enable TF of the robot in RViz, we can view the odometry as shown in the following screenshot:

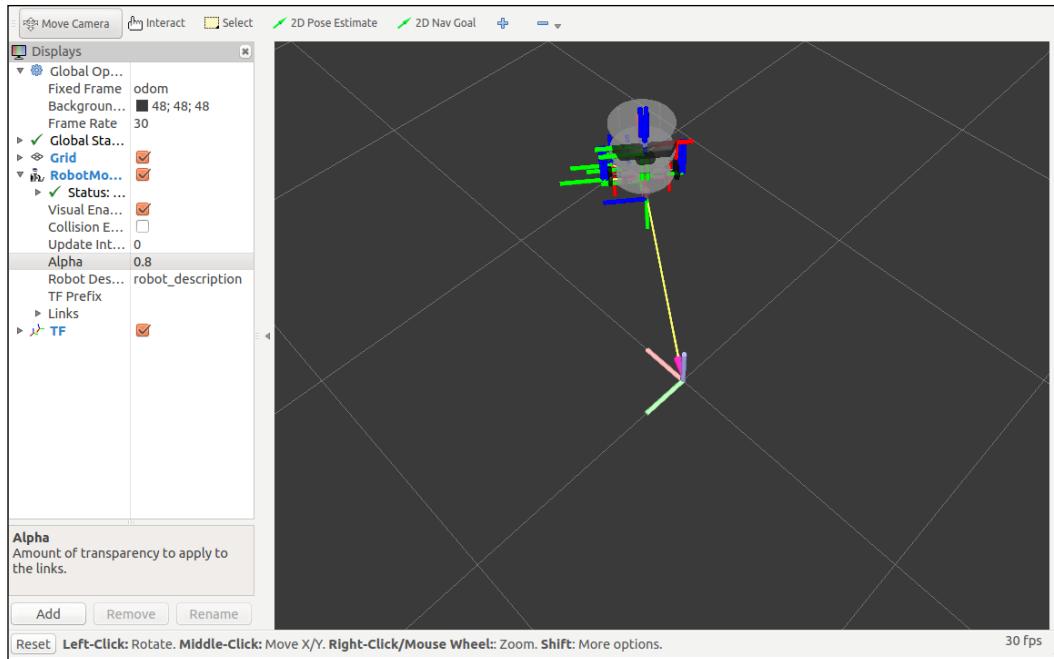


Figure 8: Visualizing robot odometry

The graph of the connection between each node is given next. We can view it using the `rqt_graph` tool.

```
$ rqt_graph
```

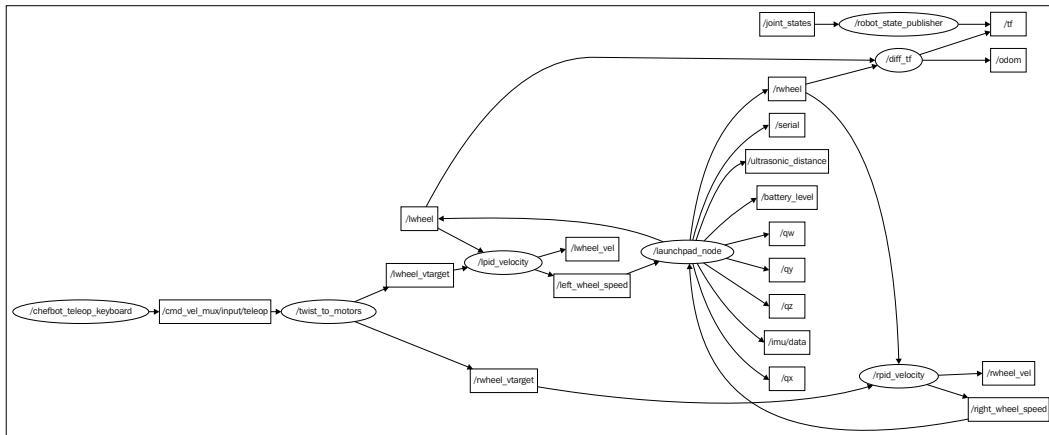


Figure 9: Interconnection of nodes in Chefbot

Till now we have discussed the Chefbot interfacing on ROS. The coding of Chefbot is completely done in Python. There are some nodes implemented in C++ for computing odometry from the encoder ticks and generating motor speed commands from the `twist` messages.

Computing odometry from encoder ticks

In this section, we will see the C++ interpretation of the `diff_tf.py` node, which subscribes the encoder data and computes the odometry, and publishes the odometry and `tf` of the robot. We can see the C++ interpretation of this node, called `diff_tf.cpp`, which can be found in the `src` folder of a package named `chefbot_navig_cpp`.

Discussed next are the important code snippets of this code and their explanations. The following code snippet is the constructor of the class `Odometry_calc`. This class contains the definition of computing odometry. The following code declares the subscriber for the left and right wheel encoders along with the publisher for `odom` value:

```
Odometry_calc::Odometry_calc() {  
  
    //Initialize variables used in the node  
    init_variables();  
  
    ROS_INFO("Started odometry computing node");  
  
    //Subscribing left and right wheel encoder values  
    l_wheel_sub = n.subscribe("/lwheel",10, &Odometry_  
    calc::leftencoderCb, this);  
  
    r_wheel_sub = n.subscribe("/rwheel",10, &Odometry_  
    calc::rightencoderCb, this);  
  
    //Creating a publisher for odom  
    odom_pub = n.advertise<nav_msgs::Odometry>("odom", 50);  
  
    //Retrieving parameters of this node  
    get_node_params();  
}
```

The following code is the update loop of computing odometry. It computes the delta distance moved and the angle rotated by the robot using the encoder values, base width of the robot, and ticks per meter of the encoder. After calculating the delta distance and the delta theta, we can compute the final x, y, and theta using the standard differential drive robot equations.

```
if ( now > t_next) {  
  
    elapsed = now.toSec() - then.toSec();  
  
    if(enc_left == 0){  
        d_left = 0;  
        d_right = 0;  
    }  
    else{  
        d_left = (left - enc_left) / ( ticks_meter);  
        d_right = (right - enc_right) / ( ticks_meter);  
    }  
  
    enc_left = left;  
    enc_right = right;  
  
    d = (d_left + d_right) / 2.0;  
  
    th = ( d_right - d_left ) / base_width;  
  
    dx = d /elapsed;  
  
    dr = th / elapsed;  
  
    if ( d != 0){  
  
        x = cos( th ) * d;  
        y = -sin( th ) * d;  
  
        // calculate the final position of the robot  
        x_final = x_final + ( cos( theta_final ) * x - sin(  
theta_final ) * y );  
        y_final = y_final + ( sin( theta_final ) * x + cos(  
theta_final ) * y );  
    }  
    if( th != 0)  
        theta_final = theta_final + th;
```

After computing the robot position and the orientation from the preceding code snippet, we can feed the odom values to the odom message header and in the tf header, which will publish the topics in /odom and /tf.

```
geometry_msgs::Quaternion odom_quat ;  
  
odom_quat.x = 0.0;  
odom_quat.y = 0.0;  
odom_quat.z = 0.0;  
  
odom_quat.z = sin( theta_final / 2 );  
odom_quat.w = cos( theta_final / 2 );  
  
//first, we'll publish the transform over tf  
geometry_msgs::TransformStamped odom_trans;  
odom_trans.header.stamp = now;  
odom_trans.header.frame_id = "odom";  
odom_trans.child_frame_id = "base_footprint";  
  
odom_trans.transform.translation.x = x_final;  
odom_trans.transform.translation.y = y_final;  
odom_trans.transform.translation.z = 0.0;  
odom_trans.transform.rotation = odom_quat;  
  
//send the transform  
odom_broadcaster.sendTransform(odom_trans);  
  
//next, we'll publish the odometry message over ROS  
nav_msgs::Odometry odom;  
odom.header.stamp = now;  
odom.header.frame_id = "odom";  
  
//set the position  
odom.pose.pose.position.x = x_final;  
odom.pose.pose.position.y = y_final;  
odom.pose.pose.position.z = 0.0;  
odom.pose.pose.orientation = odom_quat;  
  
//set the velocity  
odom.child_frame_id = "base_footprint";  
odom.twist.twist.linear.x = dx;  
odom.twist.twist.linear.y = 0;  
odom.twist.twist.angular.z = dr;
```

```
//publish the message  
odom_pub.publish(odom);
```

Computing motor velocities from ROS twist message

The C++ implementation of `twist_to_motor.py` is discussed in this section. This node will convert the `twist` message (`geometry_msgs/Twist`) to motor target velocities. The topics subscribing by this node is the `twist` message from teleop node or Navigation stack and it publishes the target velocities for the two motors. The target velocities are fed into the PID nodes, which will send appropriate commands to each motor. The CPP file name is `twist_to_motor.cpp` and you can get it from the `chapter_9_codes/chefbot_navig_cpp/src` folder.

```
TwistToMotors::TwistToMotors()  
{  
    init_variables();  
    get_parameters();  
  
    ROS_INFO("Started Twist to Motor node");  
  
    cmd_vel_sub = n.subscribe("cmd_vel_mux/input/teleop",10,  
    &TwistToMotors::twistCallback, this);  
  
    pub_lmotor = n.advertise<std_msgs::Float32>("lwheel_vtarget", 50);  
  
    pub_rmotor = n.advertise<std_msgs::Float32>("rwheel_vtarget", 50);  
}
```

The following code snippet is the callback function of the `twist` message. The linear velocity X is assigned as `dx`, Y as `dy`, and angular velocity Z as `dr`.

```
void TwistToMotors::twistCallback(const geometry_msgs::Twist &msg)  
{  
    ticks_since_target = 0;  
  
    dx = msg.linear.x;  
    dy = msg.linear.y;  
    dr = msg.angular.z;  
}
```

After getting dx , dy , and dr , we can compute the motor velocities using the following equations:

$$dx = (l + r) / 2$$

$$dr = (r - l) / w$$

Here r and l are the right and left wheel velocities, and w is the base width. The preceding equations are implemented in the following code snippet. After computing the wheel velocities, it is published to the `lwheel_vtarget` and `rwheel_vtarget` topics.

```
right = ( 1.0 * dx ) + (dr * w /2);
left = ( 1.0 * dx ) - (dr * w /2);

std_msgs::Float32 left_;
std_msgs::Float32 right_;

left_.data = left;
right_.data = right;

pub_lmotor.publish(left_);
pub_rmotor.publish(right_);

ticks_since_target += 1;

ros::spinOnce();
```

Running robot stand alone launch file using C++ nodes

The following command can launch `robot_stand_alone.launch`, which uses the C++ nodes:

```
$ roslaunch chefbot_navig_cpp robot_standalone.launch
```

Configuring the Navigation stack for Chefbot

After setting the odometry nodes, the base controller node, and the PID nodes, we need to configure the Navigation stack to perform SLAM and **Adaptive Monte Carlo Localization (AMCL)** for building the map, localizing the robot, and performing autonomous navigation.

In *Chapter 4, Using the ROS MoveIt! and Navigation Stack*, we saw the basic packages in the Navigation stack. To build the map of the environment, we need to configure mainly two nodes: the `gmapping` node for performing SLAM and the `move_base` node. We also need to configure the global planner, the local planner, the global cost map, and the local cost map inside the Navigation stack. Let's see the configuration of the `gmapping` node first.

Configuring the gmapping node

The `gmapping` node is the package to perform SLAM (<http://wiki.ros.org/gmapping>).

The `gmapping` node inside this package mainly subscribes and publishes the following topics:

The following are the subscribed topics:

- `tf` (`tf/tfMessage`): Robot transform that relates to Kinect, robot base and odometry
- `scan` (`sensor_msgs/LaserScan`): Laser scan data that is required to create the map

The following are the published topics:

- `map` (`nav_msgs/OccupancyGrid`): Publishes the occupancy grid map data
- `map_metadata` (`nav_msgs/MapMetaData`): Basic information about the occupancy grid

The `gmapping` node is highly configurable using various parameters. The `gmapping` node parameters are defined inside the `chapter_9_codes/chefbot/chefbot Bringup/launch/include/gmapping.launch.xml` file. Following is a code snippet of this file and its uses:

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

  <!-- Starting gmapping node -->
  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
        output="screen">

    <!-- Frame of mobile base -->
    <param name="base_frame" value="base_footprint"/>
    <param name="odom_frame" value="odom"/>
    <!-- The interval of map updation, reducing this value will speed of
        map generation but increase computation load -->
```

```
<param name="map_update_interval" value="5.0"/>
<!-- Maximum usable range of laser/kinect -->
<param name="maxUrange" value="6.0"/>
<!-- Maximum range of sensor, max range should be > maxUrange -->
<param name="maxRange" value="8.0"/>
<param name="sigma" value="0.05"/>
<param name="kernelSize" value="1"/>
</node>
</launch>
```

By fine tuning these parameters, we improve the accuracy of the gmapping node.

The main gmapping launch file is given next. It is placed in chefbot_bringup/launch/includes/gmapping_demo.launch. This launch file launches the openni_launch file and the depth_to_laserscan node to convert the depth image to laser scan. After launching the Kinect nodes, it launches the gmapping node and the move_base configurations.

```
<launch>
<!-- Launches 3D sensor nodes -->
<include file="$(find chefbot_bringup)/launch/3dsensor.launch">
  <arg name="rgb_processing" value="false" />
  <arg name="depth_registration" value="false" />
  <arg name="depth_processing" value="false" />
  <arg name="scan_topic" value="/scan" />
</include>

<!-- Start gmapping nodes and its configurations -->
<include file="$(find chefbot_bringup)/launch/includes/gmapping.launch.xml"/>

<!-- Start move_base node and its configuration -->
<include file="$(find chefbot_bringup)/launch/includes/move_base.launch.xml"/>
</launch>
```

Configuring the Navigation stack packages

The next node we need to configure is move_base. Along with the move_base node, we need to configure the global and the local planners, and also the global and the local cost maps. We will first look at the launch file to load all these configuration files. The following launch file chefbot_bringup/launch/includes/move_base.launch.xml will load all the parameters of move_base, planners, and costmaps:

```
<launch>
    <arg name="odom_topic" default="odom" />
    <!-- Starting move_base node -->
    <node pkg="move_base" type="move_base" respawn="false" name="move_
base" output="screen">

    <!-- common parameters of global costmap -->
        <rosparam file="$(find chefbot_bringup)/param/costmap_common_
params.yaml" command="load" ns="global_costmap" />

    <!-- common parameters of local costmap -->
        <rosparam file="$(find chefbot_bringup)/param/costmap_common_
params.yaml" command="load" ns="local_costmap" />

    <!-- local cost map parameters -->
        <rosparam file="$(find chefbot_bringup)/param/local_costmap_
params.yaml" command="load" />

    <!-- global cost map parameters -->
        <rosparam file="$(find chefbot_bringup)/param/global_costmap_
params.yaml" command="load" />

    <!-- base local planner parameters -->
        <rosparam file="$(find chefbot_bringup)/param/base_local_planner_
params.yaml" command="load" />

    <!-- dwa local planner parameters -->
        <rosparam file="$(find chefbot_bringup)/param/dwa_local_planner_
params.yaml" command="load" />

    <!-- move_base node parameters -->
        <rosparam file="$(find chefbot_bringup)/param/move_base_params.
yaml" command="load" />

        <remap from="cmd_vel" to="/cmd_vel_mux/input/navi"/>
        <remap from="odom" to="$(arg odom_topic)"/>
    </node>
</launch>
```

We will now take a look at each configuration file and its parameters.

Common configuration (`local_costmap`) and (`global_costmap`)

The common parameters of the local and global costmaps are discussed in this section. The costmap is created using the obstacles present around the robot. Fine tuning the parameters of the costmap can increase the accuracy of map generation. The customized file `costmap_common_params.yaml` of Chefbot follows. This configuration file contains the common parameters of both the global and the local cost maps. It is present in the `chefbot_bringup/param` folder. For more about costmap common parameters, check http://wiki.ros.org/costmap_2d/flat.

```
#The maximum value of height which has to be taken as an obstacle
max_obstacle_height: 0.60

#This parameters set the maximum obstacle range. In this case, the
robot will only look at obstacles within 2.5 meters in front of robot
obstacle_range: 2.5

#This parameter helps robot to clear out space in front of it upto 3.0
meters away given a sensor reading
raytrace_range: 3.0

#If the robot is circular, we can define the robot radius, otherwise
we need to mention the robot footprint

robot_radius: 0.45
#footprint: [[-0.1,-0.1],[0.1,-0.1],[0.1,0.1],[-0.1,0.1]]

#This parameter will actually inflate the obstacle up to this distance
from the actual obstacle. This can be taken as a tolerance value of
obstacle. The cost of map will be same as the actual obstacle up to
the inflated value.

inflation_radius: 0.50

#This factor is used for computing cost during inflation
cost_scaling_factor: 5

#We can either choose map type as voxel which will give a 3D view of
the world, or the other type, costmap which is a 2D view of the map.
Here we are opting voxel.
map_type: voxel

#This is the z_origin of the map if it voxel
origin_z: 0.0
```

```
#z resolution of map in meters
z_resolution: 0.2

#No of voxel in a vertical column
z_voxels: 2

#This flag set whether we need map for visualization purpose
publish voxel map: false

#A list of observation source in which we get scan data and its
parameters
observation_sources: scan

#The list of scan, which mention, data type of scan as LaserScan,
marking and clearing indicate whether the laser data is used for
marking and clearing costmap.

scan: {data_type: LaserScan, topic: scan, marking: true, clearing:
true, min_obstacle_height: 0.0, max_obstacle_height: 3}
```

After discussing the common parameters, we will now look at the global costmap configuration.

Configuring global costmap parameters

The following are the main configurations required for building a global costmap. The definition of the costmap parameters are dumped in `chefbot_bringup/param/global_costmap_params.yaml`. The following is the definition of this file and its uses:

```
global_costmap:
  global_frame: /map
  robot_base_frame: /base_footprint
  update_frequency: 1.0
  publish_frequency: 0.5
  static_map: true
  transform_tolerance: 0.5
```

The `global_frame` here is `/map`, which is the coordinate frame of the costmap. The `robot_base_frame` parameter is `/base_footprint`; it is the coordinate frame in which the costmap should reference as the robot base. The `update_frequency` is frequency at which the cost map runs its main update loop. The `publishing_frequency` of the cost map is given as `publish_frequency`, which is 0.5. If we are using an existing map, we have to set `static_map` as `true`, otherwise as `false`. The `transform_tolerance` is the rate at which the transform has to perform. The robot would stop if the transforms are not updated at this rate.

Configuring local costmap parameters

Following is the local costmap configuration of this robot. The configuration of this file is located in `chefbot_bringup/param/local_costmap_params.yaml`.

```
local_costmap:  
  global_frame: odom  
  robot_base_frame: /base_footprint  
  update_frequency: 5.0  
  publish_frequency: 2.0  
  static_map: false  
  rolling_window: true  
  width: 4.0  
  height: 4.0  
  resolution: 0.05  
  transform_tolerance: 0.5
```

The `global_frame`, `robot_base_frame`, `publish_frequency`, and `static_map` are the same as the global costmap. The `rolling_window` parameter makes the costmap centered around the robot. If we set this parameter to `true`, we will get a costmap that is built centered around the robot. The `width`, `height`, and `resolution` parameters are the width, height, and resolution of the costmap.

The next step is to configure the base local planner.

Configuring base local planner parameters

The main function of the base local planner is to compute the velocity commands from the goal sent from the ROS nodes. This file mainly contains the configurations related to velocity, acceleration, and so on. The base local planner configuration file of this robot is in `chefbot_bringup/param/base_local_planner_params.yaml`. The definition of this file is as follows:

```
TrajectoryPlannerROS:  
  
  # Robot Configuration Parameters, these are the velocity limit of the  
  # robot  
  max_vel_x: 0.3  
  min_vel_x: 0.1  
  
  #Angular velocity limit  
  max_vel_theta: 1.0  
  min_vel_theta: -1.0  
  min_in_place_vel_theta: 0.6
```

```
#These are the acceleration limits of the robot
acc_lim_x: 0.5
acc_lim_theta: 1.0

# Goal Tolerance Parameters: The tolerance of robot when it reach the
goal position

yaw_goal_tolerance: 0.3
xy_goal_tolerance: 0.15

# Forward Simulation Parameters
sim_time: 3.0
vx_samples: 6
vtheta_samples: 20

# Trajectory Scoring Parameters
meter_scoring: true
pdist_scale: 0.6
gdist_scale: 0.8
occdist_scale: 0.01
heading_lookahead: 0.325
dwa: true

# Oscillation Prevention Parameters
oscillation_reset_dist: 0.05

# Differential-drive robot configuration : If the robot is holonomic
# configuration, set to true other vice set to false. Chefbot is a non
holonomic robot.

holonomic_robot: false
max_vel_y: 0.0
min_vel_y: 0.0
acc_lim_y: 0.0
vy_samples: 1
```

Configuring DWA local planner parameters

The DWA planner is another local planner in ROS. Its configuration is almost the same as the base local planner. It is located in `chefbot_bringup/param/ dwa_local_planner_params.yaml`. We can either use the base local planner or the DWA local planner for our robot.

Configuring move_base node parameters

There are some configurations to the `move_base` node too. The `move_base` node configuration is placed in the `param` folder. Following is the definition of `move_base_params.yaml`:

```
#This parameter determine whether the cost map need to shutdown when
move_base in inactive state
shutdown_costmaps: false

#The rate at which move base run the update loop and send the velocity
commands
controller_frequency: 5.0

#Controller wait time for a valid command before a space-clearing
operations
controller_patience: 3.0

#The rate at which the global planning loop is running, if it is 0,
planner only plan when a new goal is received
planner_frequency: 1.0

#Planner wait time for finding a valid path before the space-clearing
operations

planner_patience: 5.0

#Time allowed for oscillation before starting robot recovery
operations
oscillation_timeout: 10.0

#Distance that robot should move to be considered which not be
oscillating. Moving above this distance will reset the oscillation_
timeout

oscillation_distance: 0.2

# local planner - default is trajectory rollout
base_local_planner: "dwa_local_planner/DWAPlannerROS"
```

We have discussed most of the parameters used in the Navigation stack, the `gmapping` node, and the `move_base` node. Now we can start running a `gmapping` demo for building the map.

Start the robot's `tf` nodes and base controller nodes:

```
$ roslaunch chefbot Bringup robot_standalone.launch
```

Start the gmapping nodes using the following command:

```
$ rosrun chefbot Bringup gmapping_demo.launch
```

This gmapping_demo.launch will launch the 3Dsensor, which launches the OpenNI drivers and depth to the laser scan node, and launches gmapping node and movebase node with necessary parameters.

We can launch a teleop node for moving the robot to build the map of environment. The following command will launch the teleop node for moving the robot:

```
$ rosrun chefbot Bringup keyboard_teleop.launch
```

We can see the map building in RViz, which can be invoked using the following command:

```
$ rosrun chefbot Bringup view_navigation.launch
```

We are testing this robot in a plane room; we can move robot in all areas inside the room. If we move the robot in all the areas, we will get a map as shown in the following screenshot:

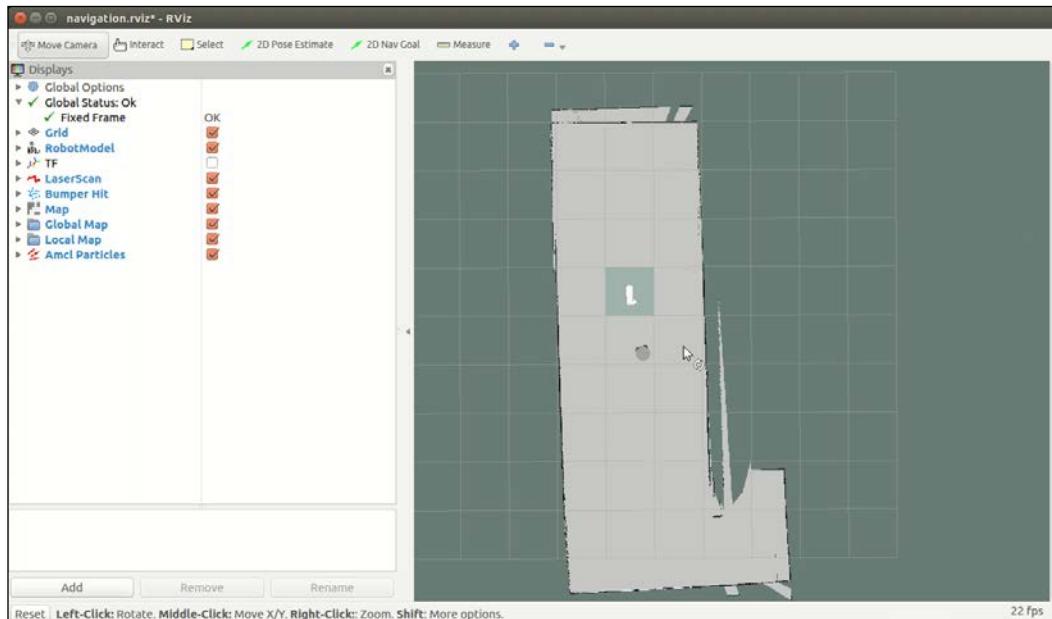


Figure 10: Creating a map using gmapping is shown in RViz

After completing the mapping process, we can save the map using the following command:

```
$ rosrun map_server map_saver -f /home/lentin/room
```

The `map_server` package in ROS contains the `map_server` node, which provides the current map data as an ROS service. It provides a command utility called `map_saver`, which helps to save the map.

It will save the current map as two files: `room.pgm` and `room.yaml`. The first one is the map data and the next is its meta data which contains the map file's name and its parameters. The following screenshot shows map generation using the `map_server` tool, which is saved in the `home` folder:

```
lentin@lentin-Aspire-4755:~$ rosrn map_server map_saver -f room
[ INFO] [1441544530.992319268]: Waiting for the map
[ INFO] [1441544531.226293214]: Received a 2560 X 2336 map @ 0.010 m/pix
[ INFO] [1441544531.226483203]: Writing map occupancy data to room.pgm
[ INFO] [1441544531.497796388, 101.846000000]: Writing map occupancy data to room.yaml
[ INFO] [1441544531.498148723, 101.846000000]: Done
```

Figure 11: Terminal messages while saving a map

The following is the `room.yaml`:

```
image: room.pgm
resolution: 0.010000
origin: [-11.560000, -11.240000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

The definition of each parameter follows:

- `image`: The image contains the occupancy data. The data can be absolute or relative to the origin mentioned in the YAML file.
- `resolution`: This parameter is the resolution of the map, which is meters/pixels.
- `origin`: This is the 2D pose of the lower left pixel in the map as (x, y, yaw) in which yaw as counter clockwise(yaw = 0 means no rotation).
- `negate`: This parameter can reverse the semantics of white/black in the map and the free space/occupied space representation.
- `occupied_thresh`: This is the threshold deciding whether the pixel is occupied or not. If the occupancy probability is greater than this threshold, it is considered as free space.
- `free_thresh`: The map pixel with occupancy probability less than this threshold is considered completely occupied.

After mapping the environment, we can quit all the terminals and rerun the following commands to start AMCL. Before starting the `amcl` nodes, we will look at the configuration and main application of AMCL.

Understanding AMCL

After building a map of the environment, the next thing we need to implement is localization. The robot should localize itself on the generated map. We have worked with AMCL in *Chapter 4, Using the ROS MoveIt! and Navigation Stack*. In this section, we will see a detailed study of the `amcl` package and the `amcl` launch files used in Chefbot.

AMCL is probabilistic localization technique for robot working in 2D. This algorithm uses particle filter for tracking the pose of the robot with respect to the known map. To know more about this localization technique, you can refer to a book called *Probabilistic Robotics* by Thrun (<http://www.probabilistic-robotics.org/>).

The AMCL algorithm is implemented in the AMCL ROS package (<http://wiki.ros.org/amcl>), which has an `amcl` node that subscribes the `scan` (`sensor_msgs/LaserScan`), the `tf` (`tf/tfMessage`), the initial pose (`geometry_msgs/PoseWithCovarianceStamped`), and the map (`nav_msgs/OccupancyGrid`).

After processing the sensor data, it publishes `amcl_pose` (`geometry_msgs/PoseWithCovarianceStamped`), `particlecloud` (`geometry_msgs/PoseArray`) and `tf` (`tf/Message`).

The `amcl_pose` is the estimated pose of the robot after processing, where the particle cloud is the set of pose estimates maintained by the filter.

If the initial pose of the robot is not mentioned, the particle will be around the origin. We can set the initial pose of the robot in RViz using the **2D Pose estimate** button. We can see the `amcl` launch file used in this robot. Following is the main launch file for starting `amcl`, called `amcl_demo.launch`:

```
<launch>
  <rosparam command="delete" ns="move_base" />
  <include file="$(find chefbot Bringup)/Launch/3dsensor.launch">
    <arg name="rgb_processing" value="false" />
    <arg name="depth_registration" value="false" />
    <arg name="depth_processing" value="false" />

    <!-- We must specify an absolute topic name because if not it will
    be prefixed by "$(arg camera)". -->
    <arg name="scan_topic" value="/scan" />
  </include>

  <!-- Map server -->
```

```
<arg name="map_file" default="$(find turtlebot_navigation)/maps/
willow-2010-02-18-0.10.yaml"/>
<node name="map_server" pkg="map_server" type="map_server"
args="$(arg map_file) " />

<arg name="initial_pose_x" default="0.0"/> <!-- Use 17.0 for
willow's map in simulation -->
<arg name="initial_pose_y" default="0.0"/> <!-- Use 17.0 for
willow's map in simulation -->
<arg name="initial_pose_a" default="0.0"/>

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

<include file="$(find chefbot_bringup)/launch/includes/move_base.
launch.xml"/>

</launch>
```

The preceding launch file starts the 3D sensors related nodes, the map server for providing the map data, the amcl node for performing localization, and the move_base node to move the robot from the commands getting from higher level.

The complete amcl launch parameters are mentioned inside another sub file called amcl.launch.xml. It is placed in chefbot_bringup/launch/include. Following is the definition of this file:

```
<launch>
  <arg name="use_map_topic"  default="false"/>
  <arg name="scan_topic"      default="scan"/>
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>

  <node pkg="amcl" type="amcl" name="amcl">
    <param name="use_map_topic"           value="$(arg use_map_
topic) "/>
    .....
    .....
```

```
<!-- Increase tolerance because the computer can get quite busy
-->
<param name="transform_tolerance"           value="1.0"/>
<param name="recovery_alpha_slow"          value="0.0"/>
<param name="recovery_alpha_fast"          value="0.0"/>
<param name="initial_pose_x"              value="$(arg initial_
pose_x)"/>
<param name="initial_pose_y"              value="$(arg initial_
pose_y)"/>
<param name="initial_pose_a"              value="$(arg initial_
pose_a)"/>
<remap from="scan"                      to="$(arg scan_topic)"/>
</node>
</launch>
```

We can refer the ROS `amcl` package wiki for getting more details about each parameter.

We will see how to localize and path plan the robot using the existing map.

Rerun the robot hardware nodes using the following command:

```
$ rosrun chefbot Bringup robot_standalone.launch
```

Run the `amcl` launch file using the following command:

```
$ rosrun chefbot Bringup amcl_demo.launch map_file:=~/home/lentin/room.
yaml
```

We can launch RViz for commanding the robot to move to a particular pose on the map.

We can launch RViz for navigation using the following command:

```
$ rosrun chefbot Bringup view_navigation.launch
```

The following is the screenshot of RViz:

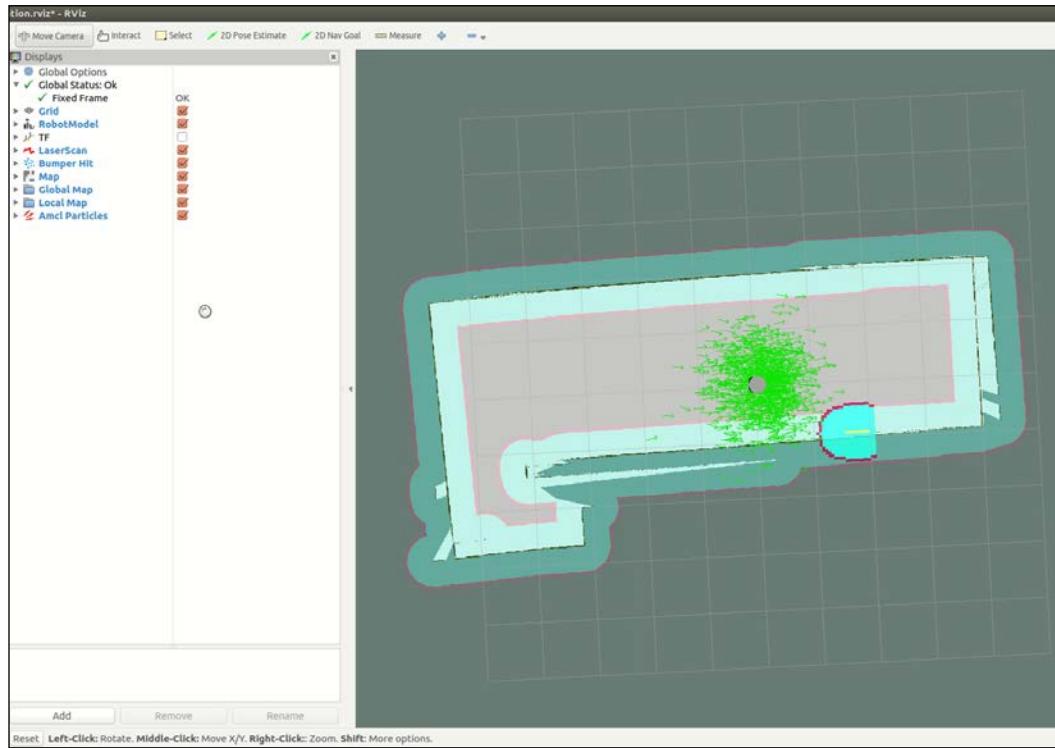


Figure 12: Robot autonomous navigation using AMCL

We will see more about each option in RViz and how to command the robot in the map in the following section.

Understanding RViz for working with the Navigation stack

We will explore various GUI options inside RViz to visualize each parameter in the Navigation stack.

2D Pose Estimate button

The first step in RViz is to set the initial position of the robot on the map. If the robot is able to localize on the map by itself, there is no need to set the initial position. Otherwise, we have to set the initial position using the **2D Pose Estimate** button in RViz, as shown in the following screenshot:

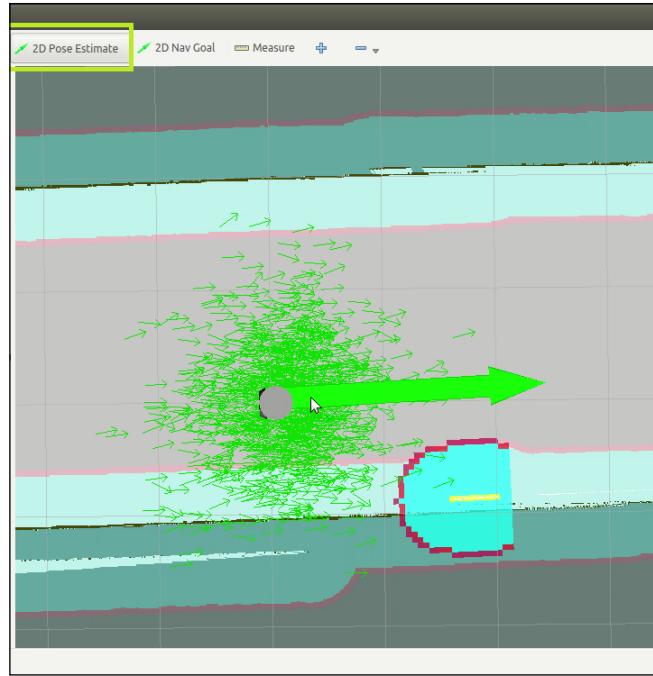


Figure 13: RViz 2D Pose Estimate button

Press the **2D Pose Estimate** button and select a pose of the robot using the left mouse button, as shown in the preceding figure. Check if the actual pose of the robot and the robot model in RViz are the same. After setting the pose, we can start path plan the robot.

The green color cloud around the robot is the particle cloud of `amcl`. If the particle amount is high, it means the uncertainty in the robot position is high, and if the cloud is less, it means that uncertainty is low and the robot is almost sure about its position. The topic handling the robot's initial pose is:

- **Topic Name:** `initialpose`
- **Topic Type:** `geometry_msgs/PoseWithCovarianceStamped`

Visualizing the particle cloud

The particle cloud around the robot can be enabled using the `PoseArray` display topic. Here the `PoseArray` topic is `/particlecloud` displayed in RViz. The `PoseArray` type is renamed as `Amcl Particles`.

- **Topic:** `/particlecloud`

- **Type:** geometry_msgs/PoseArray

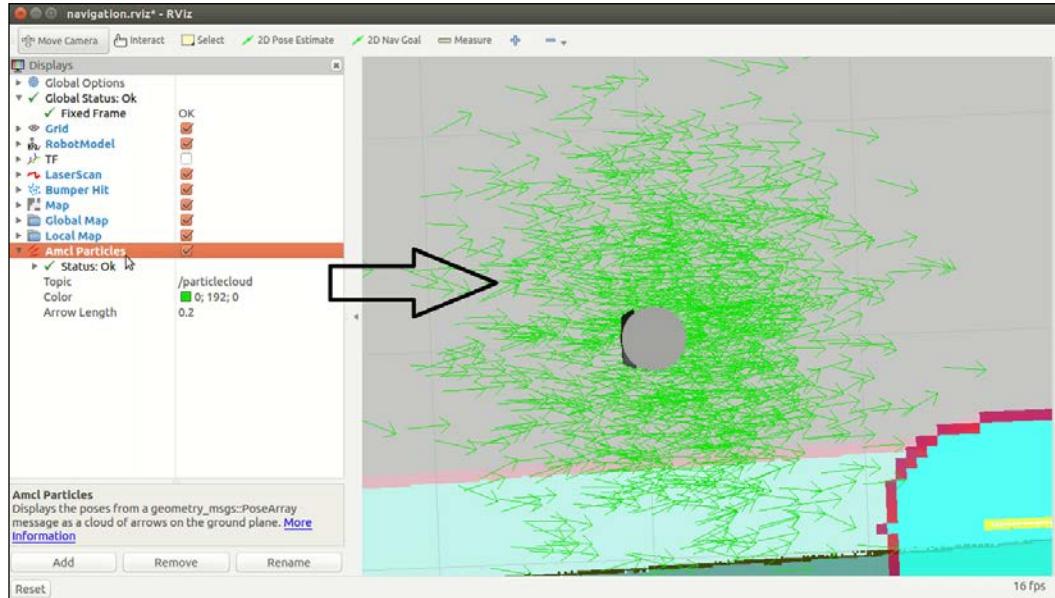


Figure 14: Visualizing AMCL particles

The 2D Nav Goal button

The **2D Nav Goal** button is used to give a goal position to the `move_base` node in the ROS Navigation stack through RViz. We can select this button from the top panel of RViz and can give the goal position inside the map by left clicking the map using the mouse. The goal position will send to the `move_base` node for moving the robot to that location.

- **Topic:** `move_base_simple/goal`
- **Topic Type:** `geometry_msgs/PoseStamped`

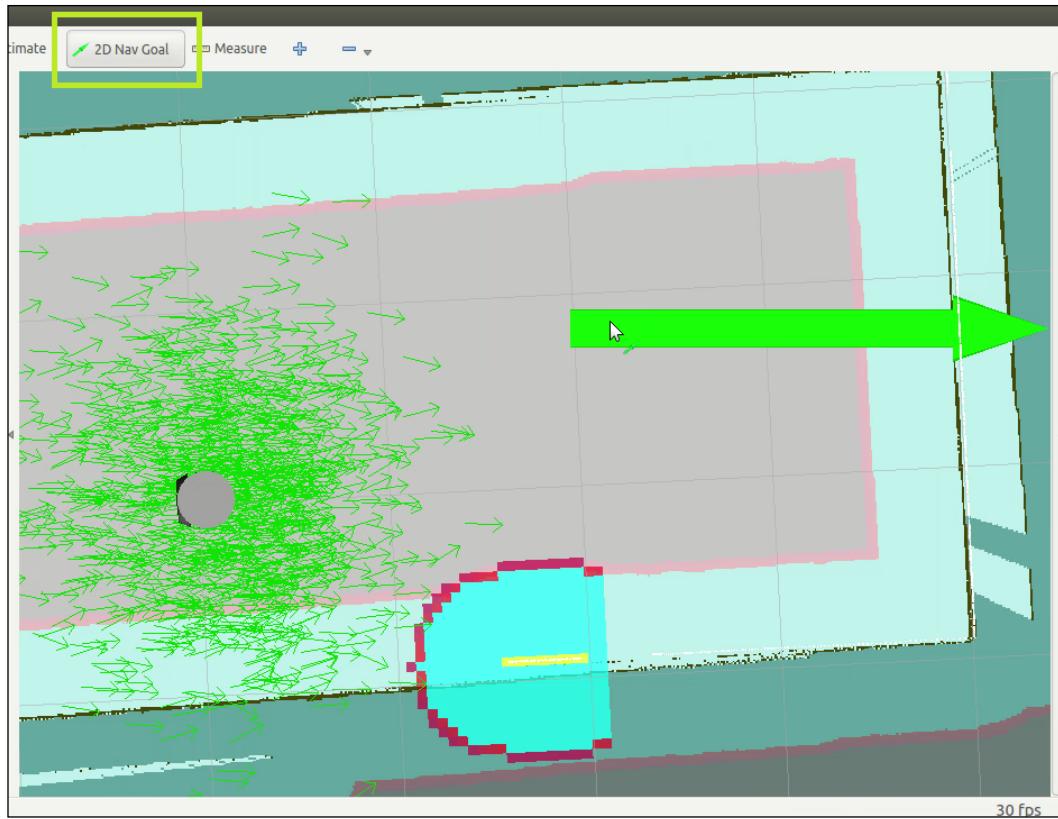


Figure 15 []:Setting robot goal position in RViz using 2D Nav Goal

Displaying the static map

The static map is the map that we feed into the `map_server` node. The `map_server` node serves the static map in the `/map` topic.

- **Topic:** `/map`
- **Type:** `nav_msgs/GetMap`

The following is the static map in RViz:

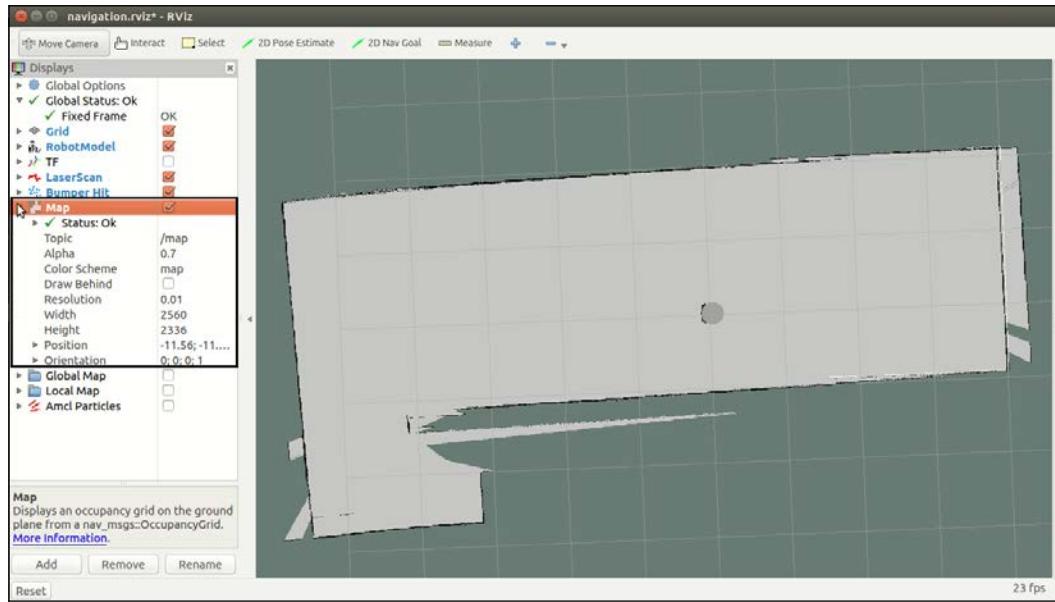


Figure 16: Visualizing static map in RViz

Displaying the robot footprint

We have defined the robot footprint in the configuration file called `costmap_common_params.yaml`. This robot has a circular shape, and we have given the radius as 0.45 meters. It can visualize using the **Polygon** display type in RViz. The following is the circular footprint of the robot around the robot model and its topics:

- **Topic:** `/move_base/global_costmap/obstacle_layer_footprint/footprint_stamped`
- **Topic:** `/move_base/local_costmap/obstacle_layer_footprint/footprint_stamped`
- **Type:** `geometry_msgs/Polygon`

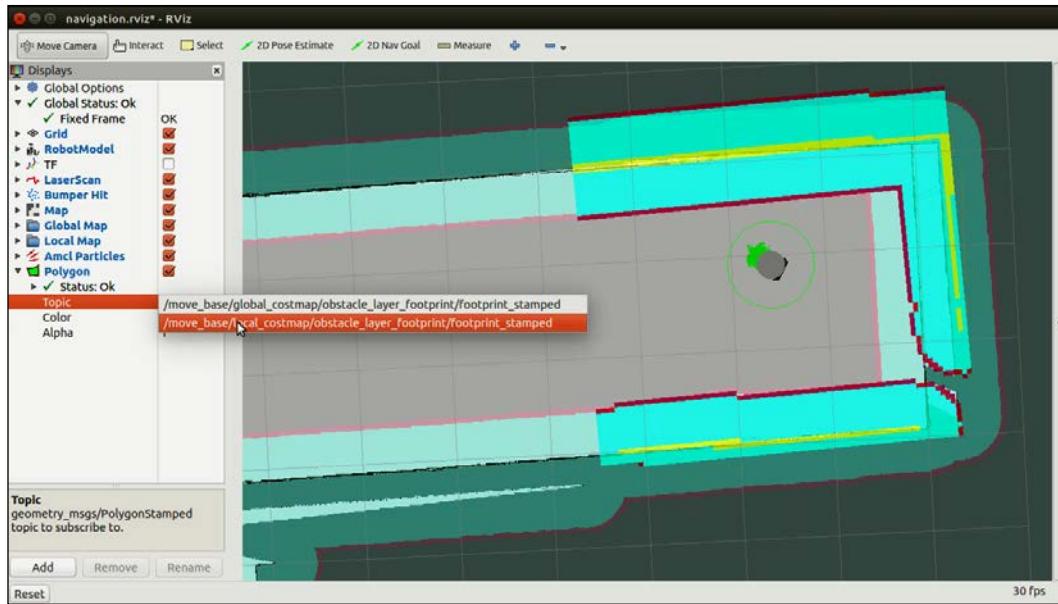


Figure 17: global and local robot footprint in RViz

Displaying the global and local cost map

The following RViz screenshot shows the local cost map, the global cost map, the real obstacles, and the inflated obstacles. The display type of each of these maps is `map` itself.

- **Local cost map topic:** `/move_base/local_costmap/costmap`
- **Local cost map topic type:** `nav_msgs/OccupancyGrid`
- **Global cost map topic:** `/move_base/global_costmap/costmap`

- **Global cost map topic type:** nav_msgs/OccupancyGrid

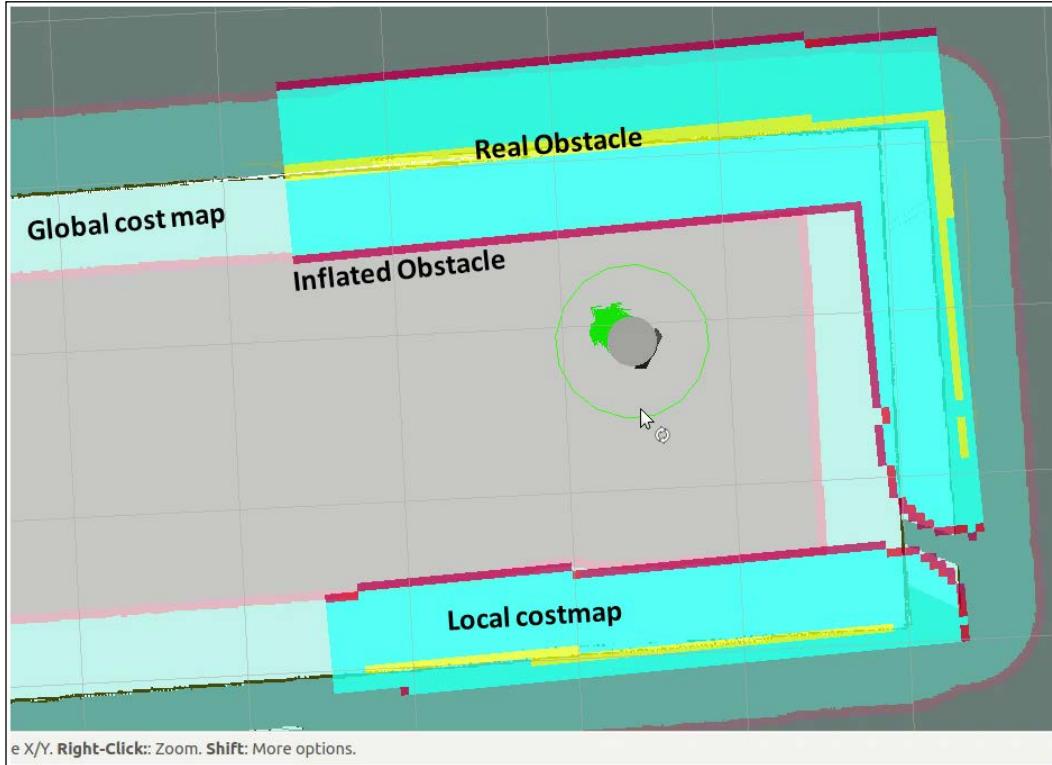


Figure 18 : Visualizing global and local map, and real and inflated obstacle in RViz

To avoid collision with the real obstacles, it is inflated to some distance from real obstacles called inflated obstacle as per the values in the configuration files. The robot only plans a path beyond the inflated obstacle; inflation is a technique to avoid collision with the real obstacles.

Displaying the global plan, local plan, and planner plan

The global plan from the global planner is shown as green in the next screenshot. The local plan is shown as red and the planner plan as black. The local plan is each section of the global plan and the planner plan is the complete plan to the goal. The global plan and the planner plan can be changed if there are any obstacles. The plans can be displayed using the Path display type in RViz.

- **Global plan topic:** /move_base/DWAPlannerROS/global_plan
- **Global plan topic type:** nav_msgs/Path
- **Local plan topic:** /move_base/DWAPlannerROS/local_plan
- **Local plan topic type:** nav_msgs/Path
- **Planner plan topic:** /move_base/NavfnROS/plan
- **Planner plan topic type:** nav_msgs/Path

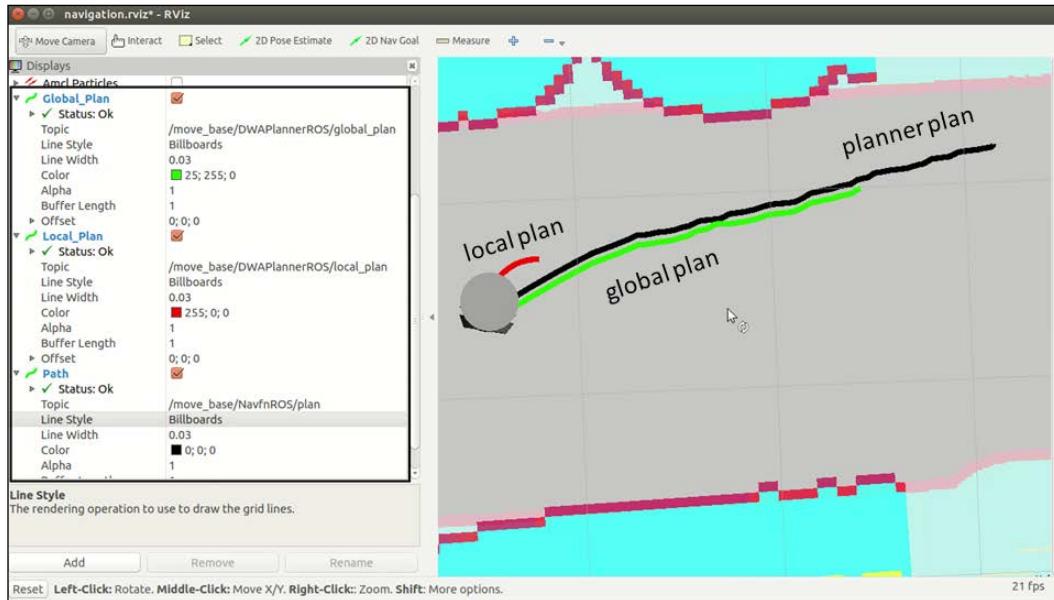


Figure 19: Visualizing global, local, and planner plan in RViz

The current goal

The current goal is the commanded position of the robot using the **2D Nav Goal** button or using the ROS client nodes. The red arrow indicates the current goal of the robot.

- **Topic:** /move_base/current_goal

- **Topic type:** geometry_msgs/PoseStamped

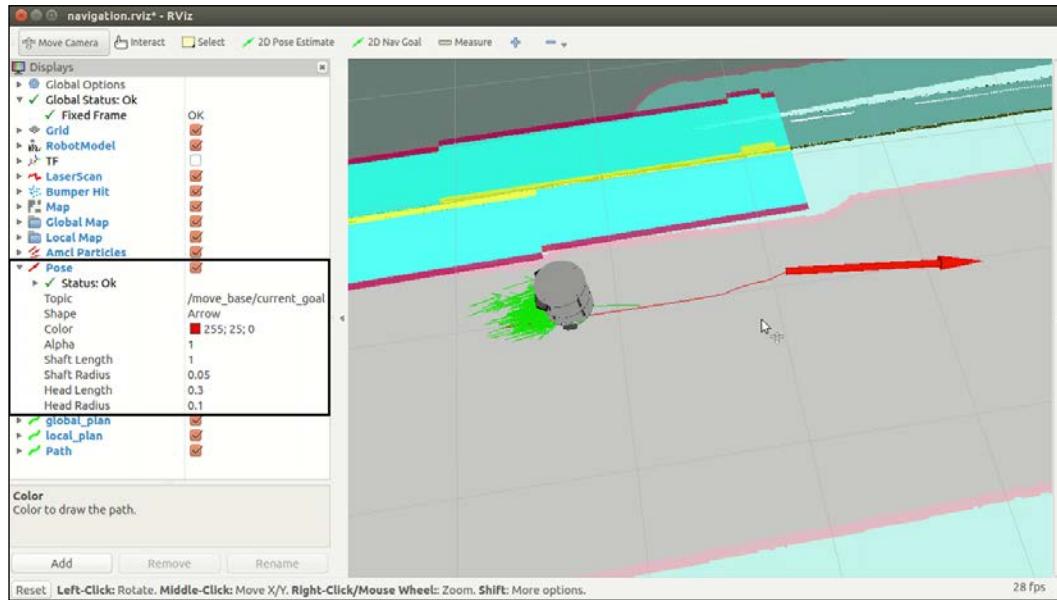


Figure 20: Visualizing robot goal position

Obstacle avoidance using the Navigation stack

The Navigation stack can avoid a random obstacle in the path. The following is a scenario where we have placed a dynamic obstacle in the planned path of the robot.

The first figure shows a path planning without any obstacle on the path. When we place a dynamic obstacle on the robot path, we can see it planning a path by avoiding the obstacle.

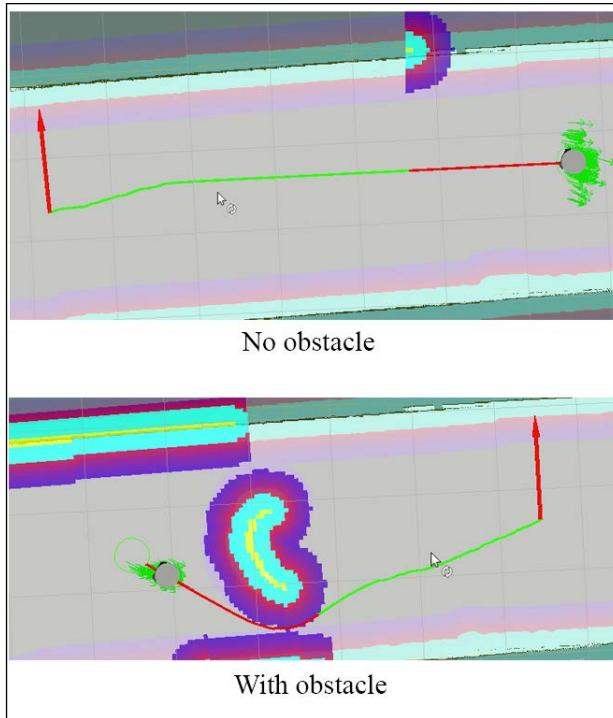


Figure 21: Visualizing obstacle avoidance capabilities in RViz

Working with Chefbot simulation

The `chefbot_gazebo` simulator package is available along with the `chefbot_bringup` package, and we can simulate the robot in Gazebo. We will see how to build a room similar to the room we tested with hardware. First we will check how to build a virtual room in Gazebo.

Building a room in Gazebo

We will start building the room in Gazebo, save into **Semantic Description Format (SDF)**, and insert in the Gazebo environment.

Launch Gazebo with Chefbot robot in an empty world:

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

It will open the Chefbot model in an empty world on Gazebo. We can build the room using walls, windows, doors, and stairs.

There is a Building Editor in Gazebo. We can take this editor from the menu **Edit | Building Editor**. We will get an editor in Gazebo viewport.

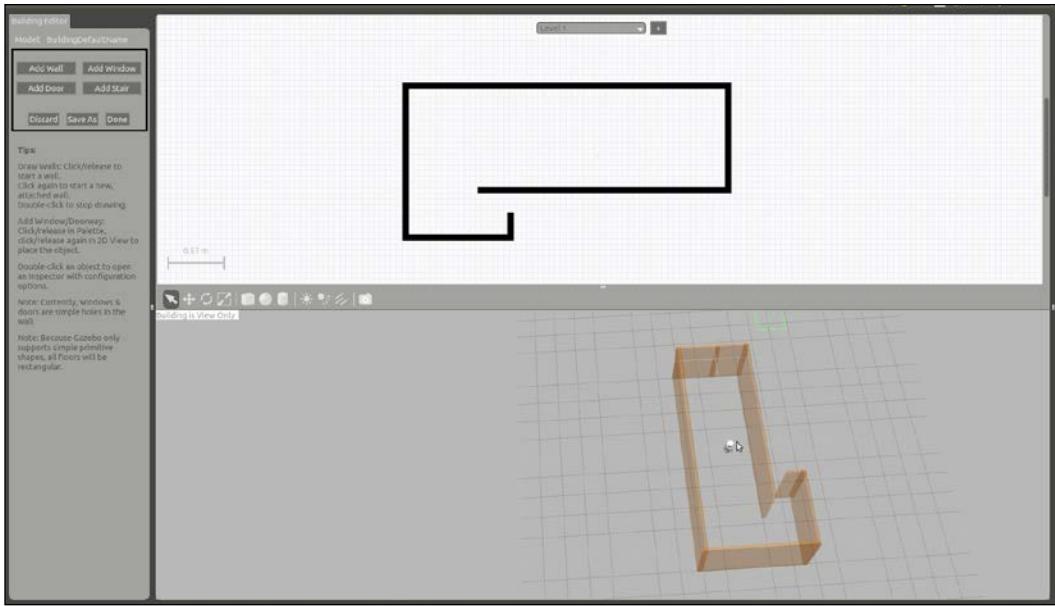


Figure 22: Building walls in Gazebo

We can add walls by clicking the **Add Wall** option on the left side pane of Gazebo. In the **Building Editor**, we can draw the walls by clicking the left mouse button. We can see adding walls in editor will build real 3D walls in Gazebo. We are building a similar layout of the room that we tested for the real robot.

Save the room through the **Save As** option, or press the **Done** button; a box will pop up to save the file. The file will get saved in the .sdf format. We can save this example as `final_room`.

After saving the room file, we can add the model of this room in the `gazebo model` folder, so that we can access the model in any simulation.

Adding model files to the Gazebo model folder

The following procedure is to add a model to the gazebo folder:

1. Locate the default `model` folder of Gazebo, which is located in the folder `~/ .gazebo/models`.
2. Create a folder called `final_room` and copy `final_room.sdf` inside this folder. Also, create a file called `model.config`, which contains the details of the `model` file. The definition of this file follows:

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

<model>
  <!-- Name of model which is displaying in Gazebo -->
  <name>Test Room</name>
  <version>1.0</version>
  <!-- Model file name -->
  <sdf version="1.2">final_room.sdf</sdf>

  <author>
    <name>Lentin Joseph</name>
    <email>qboticslabs@gmail.com</email>
  </author>

  <description>
    A test room for performing SLAM
  </description>
</model>
```

After adding this model in the `model` folder, restart the Gazebo and we can see the model named `Test Room` in the entry in the **Insert** tab, as shown in the next screenshot. We have named this model as **Test Room** in the `model.config` file; that name will show on this list. We can select this file and add to the viewport, as shown next:

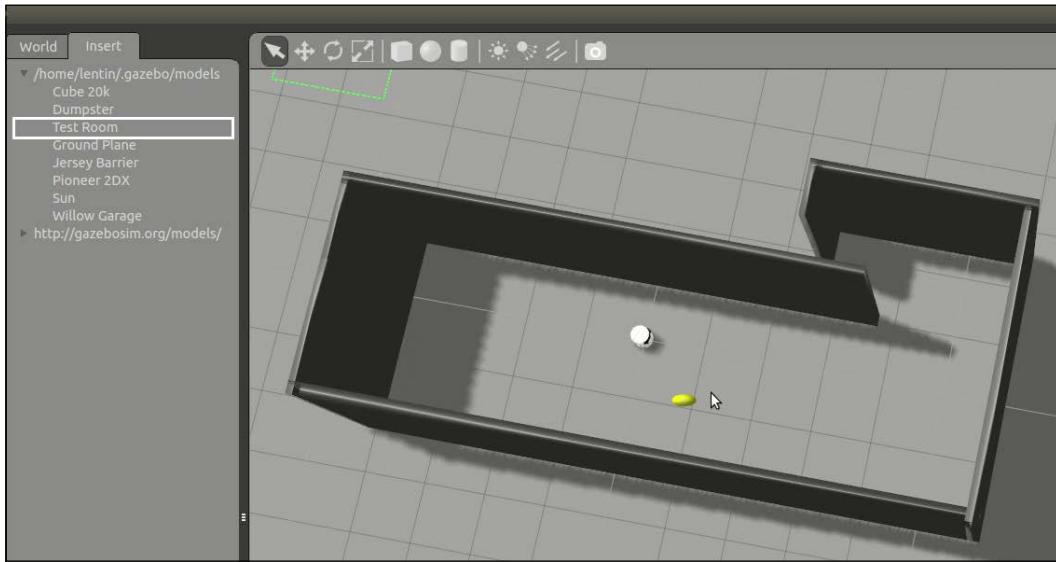


Figure 23: Inserting the walls in Chefbot simulation

After adding to the viewport, we can save the current world configuration. Take **File** from the **Gazebo** menu and **Save World As** option. Save the file as `test_room.sdf` in the `worlds` folder of the `chefbot_gazebo` ROS package.

After saving the world file, we can add it into the `chefbot_empty_world.launch` file and save this launch file as the `chefbot_room_world.launch` file, which is shown next:

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

  <!-- Adding world test_room.sdf as argument -->
  <arg name="world_name" value="$(find chefbot_gazebo)/worlds/test_
room.sdf"/>
</include>
```

After saving this launch file, we can start the launch file `chefbot_room_world.launch` for simulating the same environment as the hardware robot. We can add obstacles in Gazebo using the primitive shapes available in it.

Instead of launching the `robot_standalone.launch` file from `chefbot_bringup` for hardware, we can start `chefbot_room_world.launch` for getting the same environment of the robot, and the `odom` and `tf` data in simulation.

```
$ roslaunch chefbot_gazebo chefbot_room_world.launch
```

Other operations, such as SLAM and AMCL, have the same procedure as we followed for the hardware. The following launch files are used to perform SLAM and AMCL in simulation:

Running SLAM in simulation:

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

Running the Teleop node:

```
$ roslaunch chefbot_bringup keyboard_keyboard_teleop.launch
```

Running AMCL in simulation:

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

Sending a goal to the Navigation stack from a ROS node

We have seen how to send a goal position to a robot for moving it from point A to B, using the RViz 2D Nav Goal button. Now we will see how to command the robot using actionlib client and ROS C++ APIs. Following is a sample package and node for communicating with Navigation stack `move_base` node.

The `move_base` node is `SimpleActionServer`. We can send and cancel the goals to the robot if the task takes a lot of time to complete.

The following code is `SimpleActionClient` for the `move_base` node, which can send the `x`, `y`, and `theta` from the command line arguments. The following code is in the `chefbot_bringup/src` folder with the name of `send_robot_goal.cpp`:

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_broadcaster.h>
#include <sstream>
#include <iostream>
//Declaring a new SimpleActionClient with action of move_base_
msgs::MoveBaseAction
typedef
```

```
actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;

int main(int argc, char** argv){
    ros::init(argc, argv, "navigation_goals");
    //Initiating move_base client
    MoveBaseClient ac("move_base", true);
    //Waiting for server to start
    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting for the move_base action server");
    }
    //Declaring move base goal
    move_base_msgs::MoveBaseGoal goal;

    //Setting target frame id and time in the goal action
    goal.target_pose.header.frame_id = "map";
    goal.target_pose.header.stamp = ros::Time::now();

    //Retrieving pose from command line other vice execute a default value
    try{
        goal.target_pose.pose.position.x = atof(argv[1]);
        goal.target_pose.pose.position.y = atof(argv[2]);
        goal.target_pose.pose.orientation.w = atof(argv[3]);
    }
    catch(int e){
        goal.target_pose.pose.position.x = 1.0;
        goal.target_pose.pose.position.y = 1.0;
        goal.target_pose.pose.orientation.w = 1.0;
    }
    ROS_INFO("Sending move base goal");

    //Sending goal
    ac.sendGoal(goal);

    ac.waitForResult();

    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
        ROS_INFO("Robot has arrived to the goal position");
    else{
        ROS_INFO("The base failed for some reason");
    }
    return 0;
}
```

The following lines are added to `CMakeLists.txt` for building this node:

```
add_executable(send_goal src/send_robot_goal.cpp)
target_link_libraries(send_goal ${catkin_LIBRARIES})
```

Build the package using `catkin_make` and test the working of the client using the following set of commands using Gazebo.

Start Gazebo simulation in a room:

```
$ roslaunch chefbot_gazebo chefbot_room_world.launch
```

Start the `amcl` node with the generated map:

```
$ roslaunch chefbot_gazebo amcl_demo.launch map_file:=final_room.yaml
```

Start RViz for navigation:

```
$ roslaunch chefbot_bringup view_navigation.launch
```

Run the `send_goal` node for sending the move base goal:

```
$ rosrun chefbot_bringup send_goal 1 0 1
```

We will see the red arrow appear when this node runs, which shows that the pose is set on RViz.

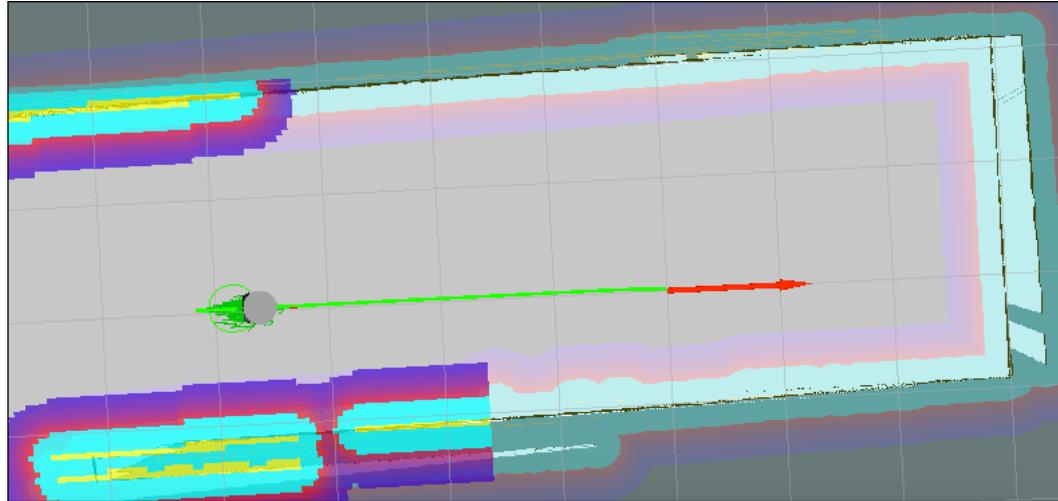


Figure 24: Sending a goal to move_base node from C++ APIs

After completing the operation, we will see the following messages in the send goal terminal:

```
Clentin@lentin-Aspire-4755:~$ rosrun chefbot Bringup send_goal 1 0 1
[INFO] [1441650740.309473041, 210.432000000]: Waiting for the move_base
action server
[INFO] [1441650740.472899452, 210.471000000]: Sending move base goal
[INFO] [1441650767.998997498, 223.440000000]: Robot has arrived to the
goal position
```

Figure 25: Terminal messages printing when a goal is send from action client

We will get the desired pose of the robot in the map by using the RViz **2D Nav goal** button. Simply echoing the topic `/move_base/goal` will print the pose that we commanded through RViz. We can use these values as command line arguments in the `send_goal` node.

Questions

1. What are the basic requirements for working with ROS Navigation stack?
2. What are the main configuration files for working with ROS Navigation stack?
3. How does AMCL package in ROS work?
4. What are the methods to send a goal pose to Navigation stack?

Summary

In this chapter, we mainly covered interfacing a DIY autonomous mobile robot to ROS and navigation package. We saw an introduction of this robot and the necessary components and connection diagrams of the same. We saw the robot firmware and how to flash it into the real robot. After flashing the firmware, we learned how to interface it to ROS and saw the Python nodes for interfacing the LaunchPad controller in the robot and the nodes for converting `twist` message to motor velocities and encoder ticks to odom and `tf`.

After discussing the interconnection of the Chefbot nodes, we covered the C++ port of some important nodes for odometry calculation and the base controller node. After discussing these nodes, we saw detailed configurations of the ROS Navigation stack. We also did gmapping, AMCL and came into detail description of each options in RViz for working with Navigation stack. We also covered the obstacle avoidance using the Navigation stack and worked with Chefbot simulation. We set up a similar environment in Gazebo like the environment of the real robot and went through the steps to perform SLAM and AMCL. At the end of this chapter, we saw how we can send a goal pose to the Navigation stack using `actionlib`.

10

Exploring the Advanced Capabilities of ROS-MoveIt!

In the previous chapter, we covered ROS navigation stack and interfacing a mobile robotic hardware to the navigation stack. Similarly, in this chapter, we are going to cover the capabilities of MoveIt!, such as collision avoidance, perception using 3D sensors, grasping, picking, and placing. After this, we will see the interfacing of a robotic manipulator hardware to MoveIt!.

The following are the main topics discussed in this chapter:

- Motion planning of arm using MoveIt! C++ APIs
- Working with collision checking in robot arm using MoveIt!
- Working with perception in MoveIt! and Gazebo
- Understanding grasping using the `moveit_simple_grasps` ROS package
- Simple robot pick and place using MoveIt!
- Understanding Dynamixel ROS servo controllers for robot hardware interfacing
- Interfacing 7-DOF Dynamixel based robotic arm to ROS MoveIt!

In *Chapter 3, Simulating Robots Using ROS and Gazebo* and *Chapter 4, Using the ROS MoveIt! and Navigation Stack*, we discussed MoveIt! and how to simulate an arm in Gazebo and motion plan using MoveIt!. In this chapter, we can see some of the advanced capabilities of MoveIt! and how to interface a real robotic manipulator to ROS MoveIt!.

The first topic that we are going to discuss is how to motion plan our robot using MoveIt! C++ APIs.

Motion planning using the move_group C++ interface

In *Chapter 4, Using the ROS MoveIt! and Navigation Stack*, we discussed about how to interact with a robot arm and how to plan its path using **MoveIt! RViz motion planning** plugin. In this section, we will see how to program the robot motion using the `move_group` C++ APIs. Motion planning using RViz can also be done programmatically through the `move_group` C++ APIs.

The first step to start working with C++ APIs is to create another ROS package that has the MoveIt! packages as dependencies. You can get an existing package `seven_dof_arm_test` from `chapter_10_codes/`. We can create this same package using the following command:

```
$ catkin_create_pkg seven_dof_arm_test catkin cmake_modules
interactive_markers moveit_core moveit_ros_perception
moveit_ros_planning_interface pluginlib roscpp std_msgs
```

Motion planning a random path using MoveIt! C++ APIs

The first example that we are going to see is random motion planning using MoveIt! C++ APIs. You will get the code named `test_random.cpp` from the `src` folder. The code and the description of each line follows. When we execute this node, it will plan a random path and execute it:

```
//MoveIt! header file
#include <moveit/move_group_interface/move_group.h>
int main(int argc, char **argv)
{
    ros::init(argc, argv, "test_random_node",
    ros::init_options::AnonymousName);
    // start a ROS spinning thread
    ros::AsyncSpinner spinner(1);
    spinner.start();
    // this connects to a running instance of the move_group node
    // Here the Planning group is "arm"
    move_group_interface::MoveGroup group("arm");
    // specify that our target will be a random one
    group.setRandomTarget();
    // plan the motion and then move the group to the sampled target
    group.move();
    ros::waitForShutdown();
}
```

To build the source code, we should add the following lines of code to CMakeLists.txt. You will get the complete CMakeLists.txt file from the existing package itself:

```
add_executable(test_random_node src/test_random.cpp)
add_dependencies(test_random_node
seven_dof_arm_test_generate_messages_cpp)
target_link_libraries(test_random_node
${catkin_LIBRARIES} )
```

We can build the package using the `catkin_make` command. Check whether `test_random.cpp` is built properly or not. If the code is built properly, we can start testing the code.

The following command will start the RViz with 7-DOF arm with motion planning plugin:

```
$ rosrun seven_dof_arm_config demo.launch
```

Move the end-effector to check whether everything is working properly in RViz.

Run the C++ node for planning to a random position using the following command:

```
$ rosrun seven_dof_arm_test test_random_node
```

The output of RViz is shown next. The arm will select a random position that has a valid IK and motion plan from the current position:

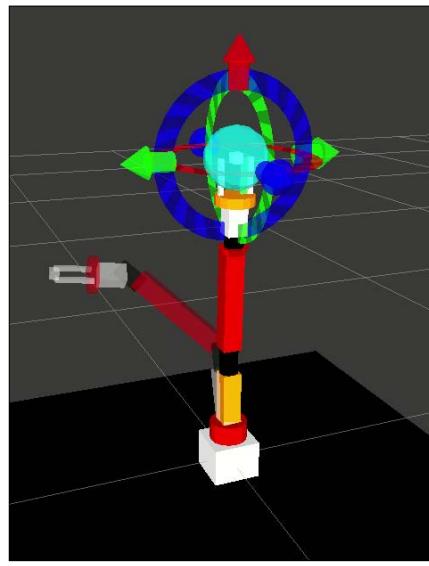


Figure 1: Random motion planning using move_group APIs

Motion planning a custom path using MoveIt! C++ APIs

We saw random motion planning in the preceding example. In this section, we will check how to command the robot end-effector to move to a custom goal position. The following example `test_custom.cpp` will do that job:

```
//Move It header files
#include <moveit/move_group_interface/move_group.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>
int main(int argc, char **argv)
{
    ros::init(argc, argv, "test_custom_node");
    ros::NodeHandle node_handle;
    ros::AsyncSpinner spinner(1);
    spinner.start();
    moveit::planning_interface::MoveGroup group("arm");
    moveit::planning_interface::PlanningSceneInterface
planning_scene_interface;
    ros::Publisher display_publisher =
node_handle.advertise<moveit_msgs::DisplayTrajectory>("/move_group/
display_planned_path", 1, true);
    moveit_msgs::DisplayTrajectory display_trajectory;

    ///Setting custom goal position
geometry_msgs::Pose target_pose1;
target_pose1.orientation.w = 0.726282;
target_pose1.orientation.x= 4.04423e-07;
target_pose1.orientation.y = -0.687396;
target_pose1.orientation.z = 4.81813e-07;
target_pose1.position.x = 0.0261186;
target_pose1.position.y = 4.50972e-07;
target_pose1.position.z = 0.573659;
group.setPoseTarget(target_pose1);

    ///Motion plan from current location to custom position
moveit::planning_interface::MoveGroup::Plan my_plan;
bool success = group.plan(my_plan);
ROS_INFO("Visualizing plan 1 (pose goal")
%s", success?"":"FAILED");
```

```
/* Sleep to give RViz time to visualize the plan. */
sleep(5.0);
ros::shutdown();
return 0;
}
```

The following are the extra lines of code added on `CMakeLists.txt` for building the source code:

```
add_executable(test_custom_node src/test_custom.cpp)
add_dependencies(test_custom_node
seven_dof_arm_test_generate_messages_cpp)
target_link_libraries(test_custom_node
${catkin_LIBRARIES} )
```

Following is the command to execute the custom node:

```
$ rosrun seven_dof_arm_test test_custom_node
```

The following screenshot shows the result of `test_custom_node`:

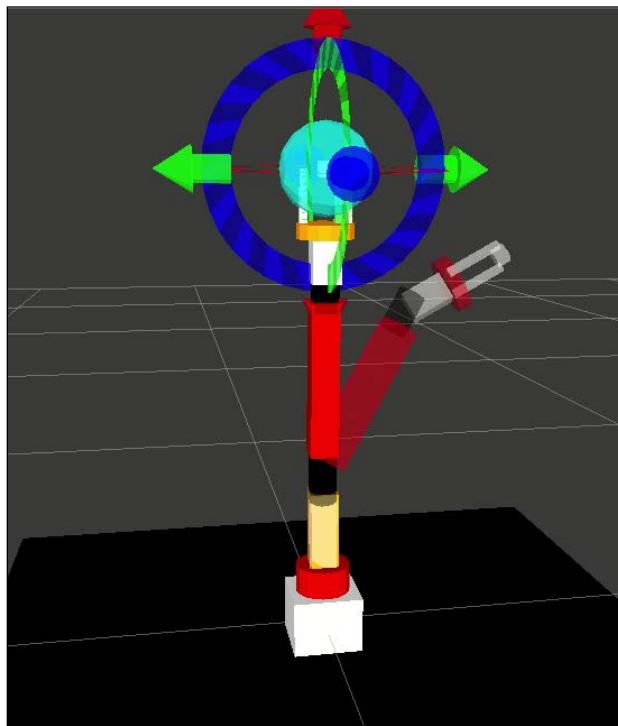


Figure 2: Custom motion planning using MoveIt! C++ APIs

Collision checking in robot arm using MoveIt!

Along with motion planning and IK solving algorithm, one of the important tasks that is done in parallel in MoveIt! is collision checking and its avoidance. The collision can be self collision or environmental collision. MoveIt! can handle both the environment collision and the self collision. The MoveIt! package is inbuilt with **FCL (Flexible Collision Library)** (http://gamma.cs.unc.edu/FCL/fcl_docs/webpage/generated/index.html), which is an open source project that implements various collision detection and avoidance algorithms. MoveIt! takes the power of FCL and handles collision inside planning scene using a `collision_detection::CollisionWorld` class. The MoveIt! collision checking includes objects such as meshes, primitives shapes such as boxes and cylinders, and OctoMap. The **OctoMap** (<http://octomap.github.io/>) library implements a 3D occupancy grid called octree that consists of probabilistic information of obstacles in the environment. The MoveIt! package can build an OctoMap using 3D point cloud information and can directly feed the OctoMap to FCL for collision checking.

Similar to motion planning, collision checking is also very computationally intensive. We can fine tune the collision checking between two bodies, say a robot link or with the environment, using a parameter called **ACM (Allowed Collision Matrix)**. If the value of a collision between two links is set to 1 in ACM, there will not be any collision checks. We may set this for links that are far from each other. We can optimize the collision checking process by optimizing this matrix.

Adding a collision object in MoveIt!

We can add a collision object to the MoveIt! planning scene and can see how the motion planning works. For adding a collision object, we can use mesh files, which can directly be imported from the MoveIt! interface, and also can be added by writing a ROS node using MoveIt! APIs.

We will first discuss how to add a collision object using the ROS node:

1. In the node `add_collision_objct.cpp` which is inside the `seven_dof_arm_test/src` folder, we are starting an ROS node and creating an object of `moveit::planning_interface::PlanningSceneInterface`, which can access the planning scene of MoveIt! and can perform any action on the current scene. We are adding a sleep of 5 seconds to wait for the `PlanningSceneInterface` object instantiation:

```
moveit::planning_interface::PlanningSceneInterface  
current_scene;  
sleep(5.0);
```

2. In the next step, we need to create an instance of the collision object message `moveit_msgs::CollisionObject`. This message is going to be sent to the current planning scene. Here we are making a collision object message for a cylinder shape and the message is given as `seven_dof_arm_cylinder`. When we add this object to the planning scene, the name of the object is its ID:

```
moveit_msgs::CollisionObject cylinder;
cylinder.id = "seven_dof_arm_cylinder";
```

3. After making the collision object message, we have to define another message of type `shape_msgs::SolidPrimitive`, which is used to define what kind of primitive shape we are using and its properties. In this example, we are creating a cylinder object as shown next. We have to define the type of shape, the resizing factor, the width, and the height of the cylinder:

```
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.CYLINDER;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.6;
primitive.dimensions[1] = 0.2;
```

4. After creating the shape message, we have to create a `geometry_msgs::Pose` message to define the pose of this object. We define a pose which may be closer to robot. We can change the pose after the creation of the object in the planning scene:

```
geometry_msgs::Pose pose;
pose.orientation.w = 1.0;
pose.position.x = 0.0;
pose.position.y = -0.4;
pose.position.z = -0.4;
```

5. After defining the pose of the collision object, we need to add the defined primitive object and the pose to the `cylinder` collision object. The operation we need to perform is adding the planning scene:

```
cylinder.primitives.push_back(primitive);
cylinder.primitive_poses.push_back(pose);
cylinder.operation = cylinder.ADD;
```

6. In the next step, we create a vector called `collision_objects` of type `moveit_msgs::CollisionObject`. After creating the vector, we push the collision object to this vector:

```
std::vector<moveit_msgs::CollisionObject>
collision_objects;
collision_objects.push_back(cylinder);
```

7. After pushing the collision object, we will add this vector to the current planning scene using the following line of code. `addCollisionObjects()` inside the `PlanningSceneInterface` class is used to add the object to the planning scene:

```
current_scene.addCollisionObjects(collision_objects);
```

Following are the compile and build lines of the code in `CMakeLists.txt`:

```
add_executable(add_collision_objc src/add_collision_objc.cpp)
add_dependencies(add_collision_objc
seven_dof_arm_test_generate_messages_cpp)
target_link_libraries(add_collision_objc
${catkin_LIBRARIES} )
```

Let's see how this node works in RViz with MoveIt! motion planning Plugin:

1. We will start `demo.launch` inside the `seven_dof_arm_config` package for testing this node:

```
$ roslaunch seven_dof_arm_config demo.launch
```

2. Next, add the following collision object:

```
$ rosrun seven_dof_arm_test add_collision_objc
```

When we run the `add_collision_objc` node, a green cylinder will pop up and we can move the collision object as shown in the following screenshot. When the collision object is successfully added to the planning scene, it will list out in the **Scene Objects** tab. We can click on the object and modify its pose. We can also attach the new model in any links of robots too. There is a **Scale** option to scale down the collision model:

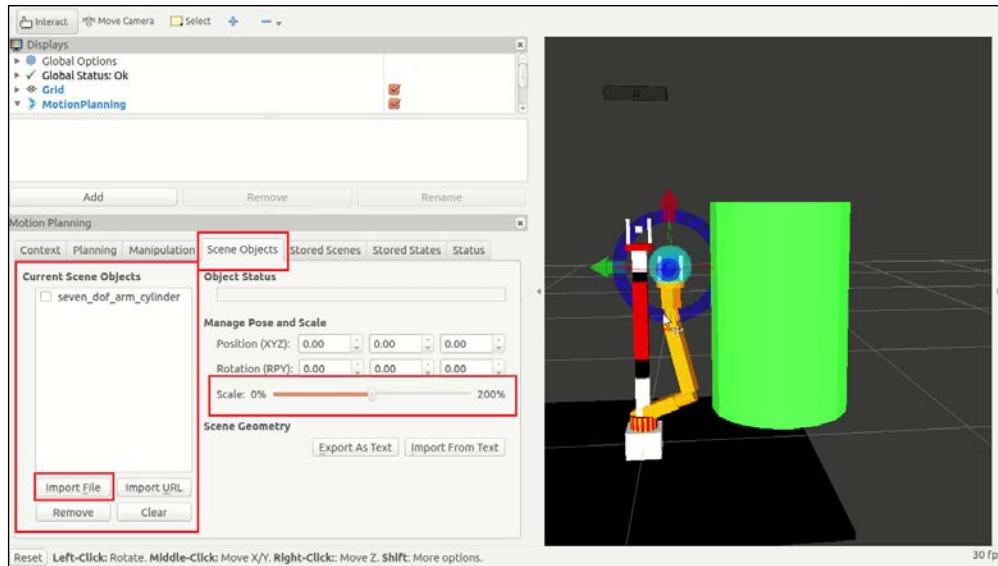


Figure 3 : Adding collision objects to RViz using MoveIt!! C++ APIs

The RViz Motion Planning plugin also gives an option to import a 3D mesh to the planning scene. Click the **Import File** button for importing the meshes. The following image shows our importing a cube mesh DAE file, which is imported along with the cylinder in the planning scene:

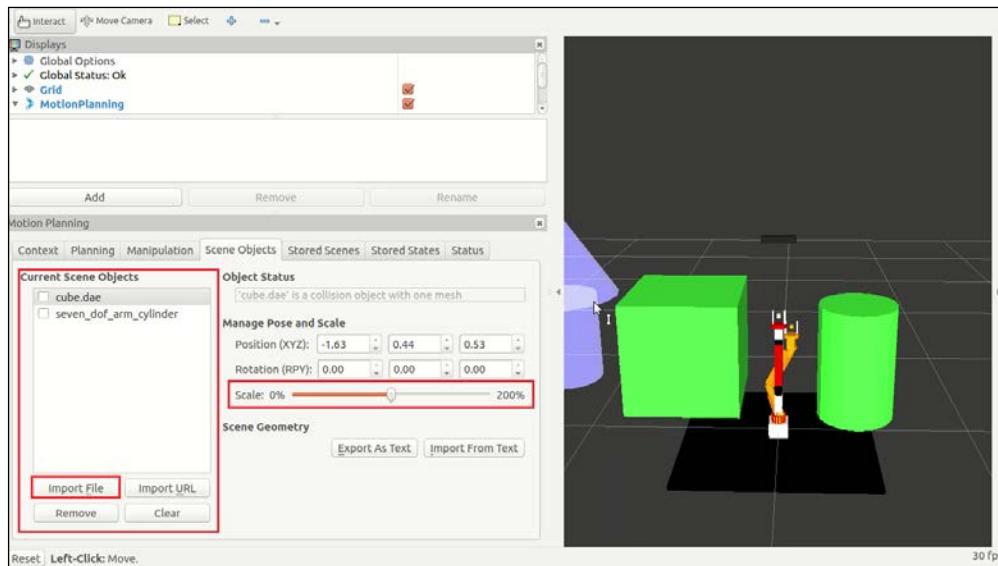


Figure 4: Adding collision objects by importing meshes

We can scale up the collision object using the **Scale** slider and set the desired pose using the **Manage Pose** option. When we move the arm end effector to any of these collision objects, MoveIt! detects it as collision. The MoveIt! collision detection can detect environment collision as well as self collision. Following is a snapshot of a collision with the environment:

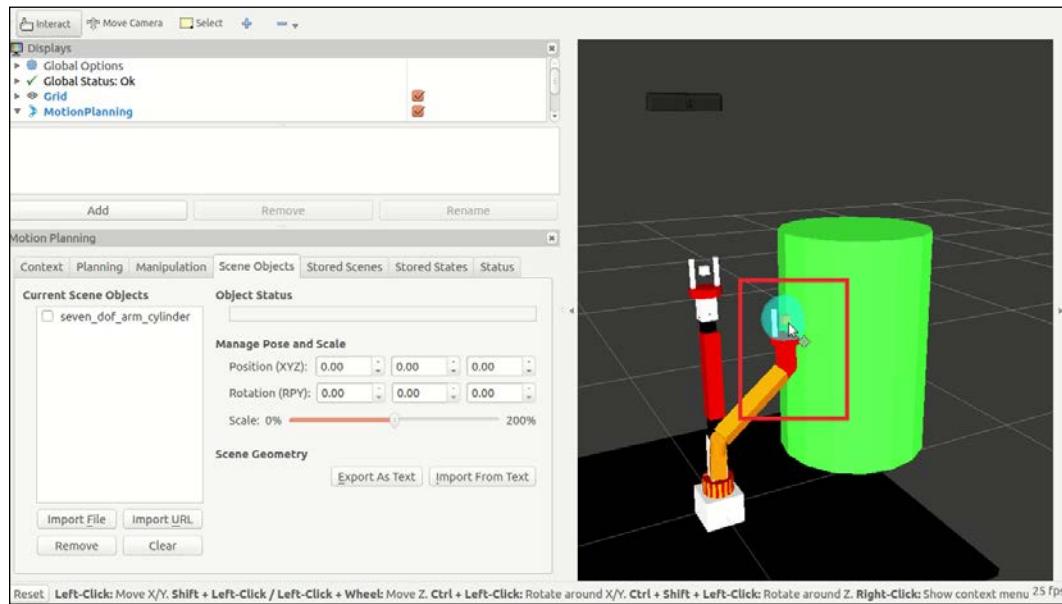


Figure 5: Visualizing collided link.

The collided link will turn red when the arm touches the object. In self collision also, the collided link will turn red. We can change the color setting of the collision in the **Motion Planning** plugin settings.

Removing a collision object from the planning scene

Removing the collision object from the planning scene is pretty easy. We have to create an object of `moveit::planning_interface::PlanningSceneInterface`, like we did in the previous example, along with some delay:

```
moveit::planning_interface::PlanningSceneInterface current_scene;  
sleep(5.0);
```

Next, create a vector of the string that contains the collision object IDs. Here our collision object ID is `seven_dof_arm_cylinder`. After pushing the string to this vector, we will call `removeCollisionObjects(object_ids)`, which will remove the collision objects from the planning scene:

```
std::vector<std::string> object_ids;
object_ids.push_back("seven_dof_arm_cylinder");
current_scene.removeCollisionObjects(object_ids);
```

This code is placed in `seven_dof_arm_test/src/remove_collision_objct.cpp`.

Checking self collision using MoveIt! APIs

We have seen how to detect collision in RViz, but what do we have to do if we want to get collision information in our ROS node. In this section, we will discuss how to get the collision information of our robot in an ROS code. This example can check self collision and environment collision, and also tell which links were collided.

The example called `check_collision.cpp` is placed in the `seven_dof_arm_test/src` folder. This code is a modified version of the collision checking example of PR2 MoveIt! robot tutorials (https://github.com/ros-planning/moveit_pr2/tree/indigo-devel/pr2_moveit_tutorials).

In this code, the following snippet loads the kinematic model of the robot to the planning scene:

```
robot_model_loader::RobotModelLoader
robot_model_loader("robot_description");
robot_model::RobotModelPtr kinematic_model =
robot_model_loader.getModel();
planning_scene::PlanningScene planning_scene(kinematic_model);
```

To test self collision in the robot's current state, we can create two instances of class `collision_detection::CollisionRequest` and `collision_detection::CollisionResult`, which have the name of `collision_request` and `collision_result`. After creating these objects, pass it MoveIt! collision checking function, `planning_scene.checkSelfCollision()`, which can give the collision result in `collision_result` object and we can print the details, which are shown next:

```
planning_scene.checkSelfCollision(collision_request,
collision_result);
ROS_INFO_STREAM("1. Self collision Test: "<<
(collision_result.collision ? "in" : "not in")
<< " self collision");
```

If we want to test collision in a particular group, we can do that by mentioning `group_name` as shown next. Here `group_name` is `arm`:

```
collision_request.group_name = "arm";
current_state.setToRandomPositions();
//Previous results should be cleared
collision_result.clear();
planning_scene.checkSelfCollision(collision_request,
collision_result);
ROS_INFO_STREAM("3. Self collision Test(In a group): <<
(collision_result.collision ? "in" : "not in"));

```

For performing a full collision check, we have to use the following function called `planning_scene.checkCollision()`. We need to mention the current robot state and the ACM matrix in this function.

The following is the code snippet to perform full collision checking using this function:

```
collision_detection::AllowedCollisionMatrix acm =
planning_scene.getAllowedCollisionMatrix();
robot_state::RobotState copied_state =
planning_scene.getCurrentState();
planning_scene.checkCollision(collision_request, collision_result,
copied_state, acm);
ROS_INFO_STREAM("6. Full collision Test: <<
(collision_result.collision ? "in" : "not in")
<< " collision");

```

We can launch the demo of motion planning and run this node using the following command:

```
$ rosrun seven_dof_arm_config demo.launch
```

Run the collision checking node:

```
$ rosrun seven_dof_arm_test check_collision
```

You will get a report such as the one shown in the following image. The robot is now not in collision; if it is in collision, it will send a report of it:

```
[ INFO] [1442483406.124802329]: 1. Self collision Test: not in self collision
[ INFO] [1442483406.125105624]: 2. Self collision Test(Change the state): in
[ INFO] [1442483406.125339959]: 3. Self collision Test(In a group): not in
[ INFO] [1442483406.125421092]: 4. Collision points valid
[ INFO] [1442483406.125672041]: 5. Self collision Test: not in self collision
[ INFO] [1442483406.125928019]: 6. Self collision Test after modified ACM: not in self collision
[ INFO] [1442483406.126233457]: 6. Full collision Test: not in collision
```

Figure 6: Collision information messages.

Working with perception using MoveIt! and Gazebo

Till now, in MoveIt!, we have worked with arm only. In this section, we will see how to interface a 3D vision sensor data to MoveIt!. The sensor can be either simulated using Gazebo or you can directly interface an RGB-D sensor such as Kinect or Xtion Pro using the `openni.launch` package. Here we will work using Gazebo simulation.

We will add sensors to MoveIt! for vision assisted pick and place. We will create a grasp table and a grasp object in gazebo for the pick and place operation. We will add two custom models called `grasp_table` and `grasp_object`. The sample models are located along with the chapter codes and it should copy to the `~/gazebo/models` folder for accessing the models from gazebo.

The following command will launch the robot arm and the Asus Xtion pro simulation in gazebo:

```
$ roslaunch seven_dof_arm_gazebo seven_dof_arm_bringup_grasping
```

This command will open up gazebo with arm joint controllers and gazebo plugin for 3D vision sensor. We can add a grasp table and grasp objects to the simulation, as shown in the following image, by simply clicking and dragging them to the workspace. We can create any kind of table or object. The objects shown in the following image are only for demonstration purposes. We can edit the model SDF file for changing the size and shape of the model:

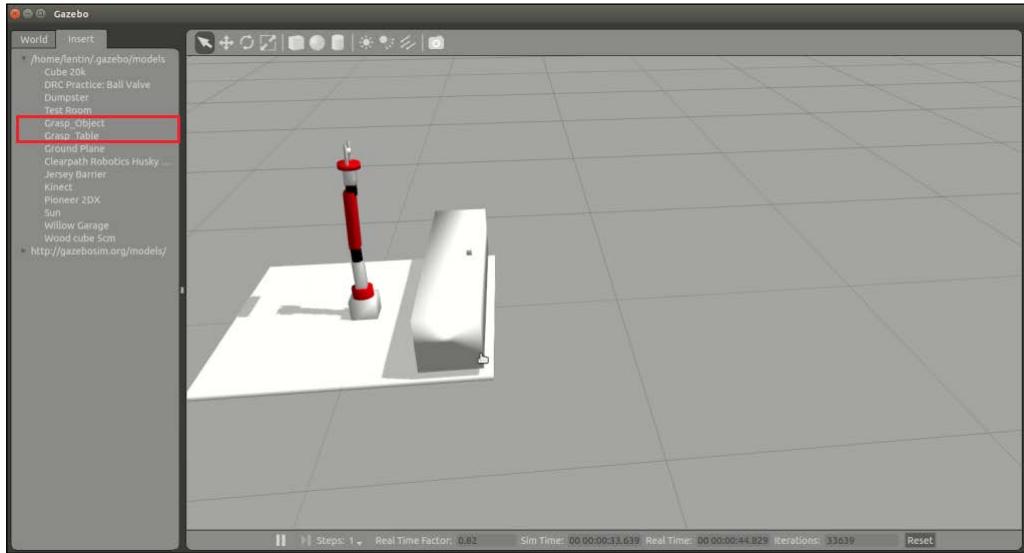


Figure 7: Robot arm with grasp table and object in Gazebo

Check the topics generated after starting the simulation:

```
$ rostopic list
```

Make sure that we are getting the RGB-D camera topics, as shown next:

```
/rgbd_camera/depth/camera_info
/rgbd_camera/depth/image_raw
/rgbd_camera/depth/points
/rgbd_camera/ir/camera_info
/rgbd_camera/ir/image_raw
/rgbd_camera/ir/image_raw/compressed
/rgbd_camera/ir/image_raw/compressed/parameter_descriptions
/rgbd_camera/ir/image_raw/compressed/parameter_updates
/rgbd_camera/ir/image_raw/theora
/rgbd_camera/ir/image_raw/theora/parameter_descriptions
/rgbd_camera/ir/image_raw/theora/parameter_updates
/rgbd_camera/parameter_descriptions
/rgbd_camera/parameter_updates
/rgbd_camera/rgb/camera_info
/rgbd_camera/rgb/image_raw
/rgbd_camera/rgb/image_raw/compressed
/rgbd_camera/rgb/image_raw/compressed/parameter_descriptions
/rgbd_camera/rgb/image_raw/compressed/parameter_updates
/rgbd_camera/rgb/image_raw/compressedDepth
/rgbd_camera/rgb/image_raw/compressedDepth/parameter_descriptions
/rgbd_camera/rgb/image_raw/compressedDepth/parameter_updates
/rgbd_camera/rgb/image_raw/theora
/rgbd_camera/rgb/image_raw/theora/parameter_descriptions
/rgbd_camera/rgb/image_raw/theora/parameter_updates
/rgbd_camera/rgb/points
```

Figure 8: Listing RGB-D sensor topics

We can view the point cloud in RViz using the following command:

```
$ rosrun rviz rviz -f base_link
```

The following is the output generated:

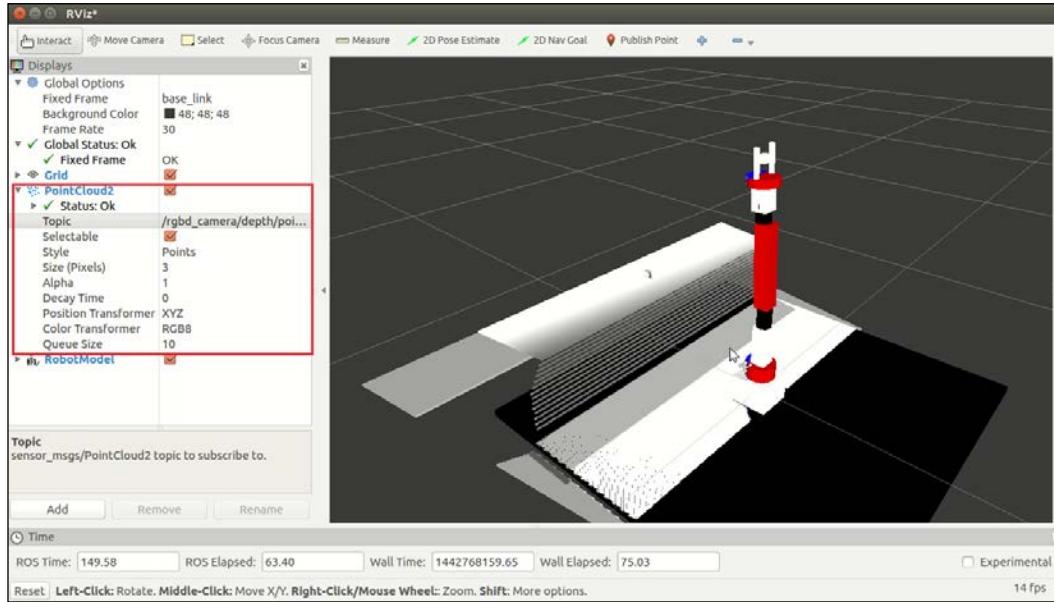


Figure 9: Visualizing point cloud data in RViz

After confirming the point cloud data from the Gazebo plugins, we have to add some files to the MoveIt! configuration package of this arm, that is, `seven_dof_arm_config`, for bringing the point cloud data from Gazebo into the MoveIt! planning scene.

The robot environment is mapped as octree representation (<https://en.wikipedia.org/wiki/Octree>), which can be built using a library called OctoMap, which we have already seen in the previous section. The OctoMap is incorporated as a plugin in MoveIt!, called the **Occupany Map Updater** plugin, which can update octree from different kinds of sensor inputs such as point cloud and depth images from 3D vision sensors. Currently, there are following plugins for handling 3D data:

- **PointCloud Occupancy Map Updater:** This plugin can take input in the form of point clouds (`sensor_msgs/PointCloud2`)
- **Depth Image Occupancy Map Updater:** This plugin can take input in the form of input depth images (`sensor_msgs/Image`)

The first step is to write a configuration file for these plugins. This file contains information about which plugin are we using in this robot and what are its properties. We are using point cloud data and the configuration is saved in `sensors_rgbd.yaml`, which is included in the `seven_dof_arm_config/config` folder. The definition of this file follows:

```
sensors:  
  - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater  
    point_cloud_topic: /rgbd_camera/depth/points  
    max_range: 10  
    padding_offset: 0.01  
    padding_scale: 1.0  
    point_subsample: 1  
    filtered_cloud_topic: output_cloud
```

The explanation of a general parameter is:

- `sensor_plugin`: This parameter specifies the name of the plugin we are using in the robot

Following are the parameters of the given `sensor_plugin`:

- `point_cloud_topic`: The plugin will listen to this topic for point cloud data.
- `max_range`: This is the distance limit in meters in which points above the range will not be used for processing.
- `padding_offset`: This value will be taken into account for robot links and attached objects when filtering clouds containing the robot links (self-filtering).
- `padding_scale`: This value will also be taken into account while self-filtering.
- `point_subsample`: If the update process is slow, points can be subsampled. If we make this value greater than 1, the points will be skipped instead of processed.
- `filtered_cloud_topic`: This is the final filtered cloud topic. We will get the processed point cloud through this topic. It can be used mainly for debugging.

If we are using the `DepthImageUpdater` plugin, we can have a different configuration file. We are not using this plugin in this robot, but we can see its usage and properties.

```
sensors:  
  - sensor_plugin: occupancy_map_monitor/DepthImageOctomapUpdater  
    image_topic: /head_mount_kinect/depth_registered/image_raw  
    queue_size: 5
```

```
near_clipping_plane_distance: 0.3
far_clipping_plane_distance: 5.0
skip_vertical_pixels: 1
skip_horizontal_pixels: 1
shadow_threshold: 0.2
padding_scale: 4.0
padding_offset: 0.03
filtered_cloud_topic: output_cloud
```

- `queue_size`: This is the queue size for the depth image transport subscriber.
- `near_clipping_plane_distance`: This is the minimum valid distance from the sensor.
- `far_clipping_plane_distance`: This is the maximum valid distance from the sensor.
- `skip_vertical_pixels`: This is the number of pixels have to skip from top and bottom of the image. If we give a value of 5, it will skip five columns from first and last of the image.
- `skip_horizontal_pixels`: Skipping pixels in horizontal direction.
- `shadow_threshold`: In some situations, points can appear below the robot links. This happens because of padding. `shadow_threshold` removes those points whose distance is greater than `shadow_threshold`.

After discussing about the OctoMap update plugin and its properties, we can switch to the launch files, necessary to initiate this plugin and parameters.

The first file we need to create is inside the `seven_dof_arm_config/launch` folder with the name `seven_dof_arm_moveit_sensor_manager.launch`.

Following is the definition of this file. This launch file basically loads the plugin parameters:

```
<launch>
  <rosparam command="load" file="$(find seven_dof_arm_config)/config/
sensors_rgbd.yaml" />
</launch>
```

The next file that we need an editing is `sensor_manager.launch`, which is located inside the `launch` folder. The definition of this file follows:

```
<launch>
  <!-- This file makes it easy to include the settings for sensor
managers -->

  <!-- Params for the octomap monitor -->
```

```
<!-- <param name="octomap_frame" type="string" value="some frame in  
which the robot moves" /> -->  
<param name="octomap_resolution" type="double" value="0.015" />  
<param name="max_range" type="double" value="5.0" />  
  
<!-- Load the robot specific sensor manager; this sets the moveit_  
sensor_manager ROS parameter -->  
  
<arg name="moveit_sensor_manager" default="seven_dof_arm" />  
<include file="$(find seven_dof_arm_config)/launch/$(arg moveit_  
sensor_manager)_moveit_sensor_manager.launch.xml" />  
  
</launch>
```

The following line is commented because it can be used if the robot is mobile. In our case, our robot is static. If it is a fixed on a mobile robot, we can give the frame value as `odom` or `odom_combined` of the robot:

```
<param name="octomap_frame" type="string" value="some frame in which  
the robot moves" />
```

The following parameter is the resolution of OctoMap, which is visualizing in RViz measured in meters. The rays beyond the `max_range` value will be truncated.

```
<param name="octomap_resolution" type="double" value="0.015" />  
<param name="max_range" type="double" value="5.0" />
```

The interfacing is now complete. We can test the MoveIt! interface using the following command.

Launch Gazebo for perception using the following command, and add the desired grasp table and grasp object model:

```
$ rosrun seven_dof_arm_gazebo seven_dof_arm_bringup_grasping.launch
```

Start the MoveIt! planner with sensor support:

```
$ rosrun seven_dof_arm_config moveit_planning_execution.launch
```

Now RViz has sensor support. We can see the OctoMap in front of the robot in the following screenshot:

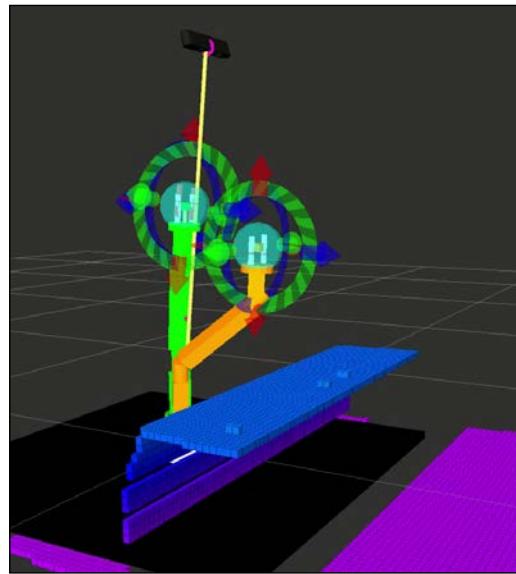


Figure 10 : Visualizing octomap in RViz

Grasping using MoveIt!

One of the main applications of robot manipulators is picking an object and placing it. Grasping is the process of picking the object by the robot end-effector. It is actually a complex process because lot of constraints are required in picking an object.

We humans handle our grasping using our intelligence, but in the robot we have to create rules for it. One of the constraints in grasping is force; the gripper/end-effector should adjust the grasping force for picking the object but not make any deformation on the object while grasping.

One of the ROS packages for generating the grasp poses for simple objects such as blocks and cylinders, which can work along with the MoveIt! pick and place pipeline, is `moveit_simple_grasps`. It's a simple grasp generator. It takes the pose of grasping object as input and generates the grasping sequences for picking the object. It filters and removes kinematically infeasible grasps via threaded IK solvers. The package provides grasp generators, grasp filters, and visualization tools.

This package already supports robots such as Baxter, REEM, and Clam arm. We can interface a custom arm to this package with a minor tweak of the package code. The package used for this experiment is inside the chapter codes. The main package code is on GitHub, and we can find it at:

https://github.com/davetcoleman/moveit_simple_grasps

It is also available as a Debian package as `moveit-simple-grasps`, but it will be better to use our own customized package to work with this experiment. Copy the `moveit_simple_grasps` package from `chapter_10_codes/` to your catkin workspace and build it using the `catkin_make` command

After building the package, we have to check whether the following launch file is working:

```
$ roslaunch seven_dof_arm_gazebo grasp_generator_server.launch
```

If it is working well, we will get log messages as in the following screenshot:



```
0.0]
  * /moveit_simple_grasps_server/gripper/pregrasp_time_from_start
  : 4.0
  * /moveit_simple_grasps_server/group: arm
  * /rosdistro: indigo
  * /rosversion: 1.11.13

NODES
/
  moveit_simple_grasps_server (moveit_simple_grasps/moveit_simple_grasps_server)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[moveit_simple_grasps_server-1]: started with pid [30638]
```

Figure 11 : Launching grasp generator server.

The definition of this launch file follows. Basically, it starts a grasp server that provides grasp combination to a grasp client node. We have to provide the planning group and the end-effector group inside this launch file, which is needed by the grasp server. The grasp server will execute feasible motion plan on the arm. It needs detailed configuration of the gripper:

```
<launch>
  <arg name="robot" default="seven_dof_arm"/>
  <arg name="group"        default="arm"/>
  <arg name="end_effector" default="gripper"/>
```

```
<node pkg="moveit_simple_grasps" type="moveit_simple_grasps_server"
name="moveit_simple_grasps_server">
  <param name="group" value="$(arg group)"/>
  <param name="end_effector" value="$(arg end_effector)"/>

  <rosparam command="load" file="$(find seven_dof_arm_gazebo)/
config/$(arg robot)_grasp_data.yaml"/>

</node>

</launch>
```

Next is the definition of `seven_dof_arm_grasp_data.yaml` and its explanation:

```
base_link: 'base_link'

gripper:

#The end effector name for grasping
end_effector_name: 'gripper'

# Gripper joints
joints: ['finger_joint1', 'finger_joint2']

#Posture of grippers before grasping
pregrasp_posture: [0.0, 0.0]

pregrasp_time_from_start: 4.0

grasp_posture: [1.0, 1.0]

grasp_time_from_start: 4.0

postplace_time_from_start: 4.0

# Desired pose from end effector to grasp [x, y, z] + [R, P, Y]
grasp_pose_to_eef: [0.0, 0.0, 0.0]
grasp_pose_to_eef_rotation: [0.0, 0.0, 0.0]

end_effector_parent_link: 'gripper_roll_link'
```

These parameters are all related to the grasping task. We can fine tune these parameters for better grasping.

Next, we will see how to do a pick and place task using this grasp server and a custom client.

Working with robot pick and place task using MoveIt!

We can do pick and place in various ways. One is by using pre defined sequences of joint values; in this case, we put the object in a predefined position and move the robot into that position by providing direct joint values or forward kinematics. Another method of pick and place is by using inverse kinematics without any visual feedback; in this case, we command the robot to go to an X,Y, and Z position with respect to the robot, and by solving IK, the robot can reach that position and pickup that object. One more method is vision assisted pick and place; in this case, a vision sensor is used to identify the object's position and the arm goes to that location by solving IK and picks the object.

In this section, we will demonstrate a pick and place in which we will give the grasping object position and the robot will move to that coordinate and pick the object. It can be tied up with vision in such a way that we need to tell the object position, which is seen by the sensor in robot coordinate system. Here we are not performing object recognition and finding position of the object. Instead of that, we are directly giving the object position.

We can work with this example along with Gazebo or simply use the MoveIt! demo interface. First, we will look at a direct pick and place mechanism by giving the grasp object position in MoveIt! using the python grasp client.

Launch MoveIt! demo:

```
$ roslaunch seven_dof_config demo.launch
```

Launch MoveIt! Grasp server:

```
$ roslaunch seven_dof_arm_gazebo grasp_generator_server
```

Run the Grasp client:

```
$ rosrun seven_dof_arm_gazebo pick_and_place.py
```

This will do a basic pick and place routine with a grasp object inserted in the planning scene.

Following is the screenshot of the grasping process:

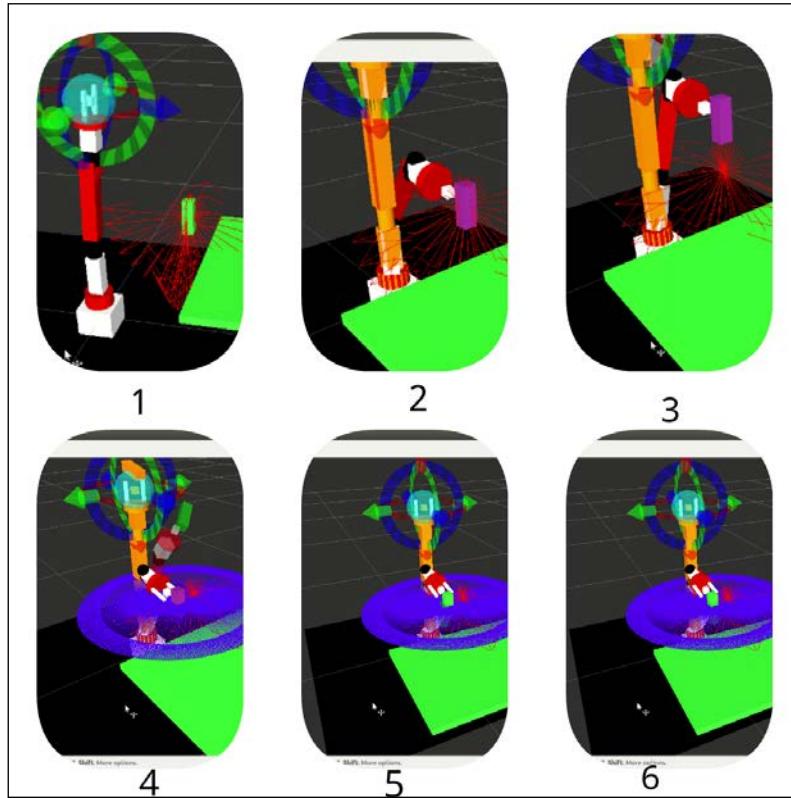


Figure 12 : Pick and place sequences using MoveIt!.

The various steps in the grasping process are explained next:

- **Step 1 - Grasp Pose:** In the first step, we can see a green block, which is the object that is going to be grasped by the robot gripper. We have created this object inside the planning scene using the `pick_and_place.py` node and it gives the block position to the grasp server. When the pick and place starts, we can see a `Pose Array` of values from the `/grasp` topic, indicating that this is the grasp object position.
- **Step 2 - Pick Action:** After getting the grasp object position, this grasp client sends this position of pick and place to the grasp server to generate IK and check whether any feasible IK for this object position. If it is a valid IK, the arm gripper will come to pick the object.

- Step 3,4,5,6 Place Action - After picking the block, the grasp server checks for the valid IK pose in the place pose. If there is a valid IK in the place pose, the gripper holds the object in a trajectory and places it in the appropriate position. The place **Pose Array** is shown as blue color from the topic /place.

We can have a look on to the `pick_and_place.py` code, this is a modified version of sample code mentioned in the following Git repository. (https://github.com/AaronMR/Learning_ROS_for_Robotics_Programming_2nd_edition.git)

Creating Grasp Table and Grasp Object in MoveIt!

We have to create a table and a grasp object similar to the robot environment. Here we are creating a table almost the same as in the gazebo simulation. If the table size and the pose of gazebo and MoveIt! are same, we can set the position of the grasping object:

```
def _add_table(self, name):  
    p = PoseStamped()  
    p.header.frame_id = self._robot.get_planning_frame()  
    p.header.stamp = rospy.Time.now()  
  
    #Table position  
    p.pose.position.x = 0.45  
    p.pose.position.y = 0.0  
    p.pose.position.z = 0.22  
  
    q = quaternion_from_euler(0.0, 0.0, numpy.deg2rad(90.0))  
    p.pose.orientation = Quaternion(*q)  
  
    # Table size from ~/.gazebo/models/grasp_table/model.sdf,  
    using the values  
    # for the surface link.  
    self._scene.add_box(name, p, (0.5, 0.4, 0.02))  
  
    return p.pose
```

Following is the creation of the grasp object. We are creating a random grasp object here. You can change the pose and size of it according to your environment:

```
def _add_grasp_block_(self, name):  
    p = PoseStamped()  
    p.header.frame_id = self._robot.get_planning_frame()  
    p.header.stamp = rospy.Time.now()
```

```
p.pose.position.x = 0.25
p.pose.position.y = 0.05
p.pose.position.z = 0.32

q = quaternion_from_euler(0.0, 0.0, 0.0)
p.pose.orientation = Quaternion(*q)

# Grasp Object can size from ~/.gazebo/models/grasp_object/
model.sdf,
    self._scene.add_box(name, p, (0.03, 0.03, 0.09))

return p.pose
```

After creating the grasp object and the grasp table, we will see how to set the pick position and the place position from the following code snippet. Here the pose of the grasp object created in the planning scene is retrieved and fed into the place pose in which the Y axis of the place pose is subtracted by 0.06. So when the pick and place happens, the object will place into 0.06 away from the object in the Y direction.

```
# Add table and grasp object to the planning scene:

self._pose_table      = self._add_table(self._table_object_name)
self._pose_grasp_obj = self._add_grasp_block_(self._grasp_
object_name)

rospy.sleep(1.0)

# Define target place pose:
self._pose_place = Pose()

self._pose_place.position.x = self._pose_grasp_obj.position.x
self._pose_place.position.y = self._pose_grasp_obj.position.y
- 0.06
self._pose_place.position.z = self._pose_grasp_obj.position.z

self._pose_place.orientation = Quaternion(*quaternion_from_
euler(0.0, 0.0, 0.0))
```

The next step is to generate the grasp **Pose Array** data for visualization and then send the grasp goal to the grasp server. If there is a grasp sequence, it will be published, else it will show as an error.

```
def _generate_grasps(self, pose, width):

    # Create goal:
    goal = GenerateGraspsGoal()
```

```
goal.pose  = pose
goal.width = width

.....
.....



state = self._grasps_ac.send_goal_and_wait(goal)
if state != GoalStatus.SUCCEEDED:
    rospy.logerr('Grasp goal failed!: %s' % self._grasps_
ac.get_goal_status_text())
    return None

grasps = self._grasps_ac.get_result().grasps

# Publish grasps (for debugging/visualization purposes):
self._publish_grasps(grasps)

return grasps
```

This function will create a **Pose Array** data for the pose of the place.

```
def _generate_places(self, target):

    # Generate places:
    places = []
    now = rospy.Time.now()
    for angle in numpy.arange(0.0, numpy.deg2rad(360.0), numpy.
deg2rad(1.0)):
        # Create place location:
        place = PlaceLocation()

        .....
        .....

    # Add place:
    places.append(place)

    # Publish places (for debugging/visualization purposes):
    self._publish_places(places)
```

The following function will create a goal object for picking the object, which has to send into MoveIt!.

```
def _create_pickup_goal(self, group, target, grasps):
    """
    Create a MoveIt!! PickupGoal
```

```
"""
# Create goal:
goal = PickupGoal()

goal.group_name = group
goal.target_name = target

.....
return goal
```

Also, there is the `_create_place_goal(self, group, target, places)` function to create place goal for MoveIt!.

The important functions which are performing picking and placing are given below.

These functions will generate a pick and place sequence, which will be sent to MoveIt! and print the result of the motion planning, whether it is succeeded or not:

```
def _pickup(self, group, target, width)
def _place(self, group, target, place)
```

Pick and place action in Gazebo and real Robot

The grasping sequence executed in the MoveIt! demo uses fake controllers. We can send the trajectory to the actual robot or Gazebo. In Gazebo, we can launch the grasping world to perform this action. The following commands will perform pick and place in Gazebo.

Launch Gazebo for grasping:

```
$ roslaunch seven_dof_arm_gazebo seven_dof_arm_bringup_grasping.launch
```

Start MoveIt! motion planning:

```
$ roslaunch seven_dof_arm_config moveit_planning_execution.launch
```

Launch MoveIt! Grasp server:

```
$ roslaunch seven_dof_arm_gazebo grasp_generator_server
```

Run the Grasp client:

```
$ rosrun seven_dof_arm_gazebo pick_and_place.py
```

In the real hardware, the only difference is that we need to create joint Trajectory controllers for the arm. One of the commonly used hardware controllers is Dynamixel controller. We will learn more about the Dynamixel controllers in the next section.

Understanding Dynamixel ROS Servo controllers for robot hardware interfacing

Till now, we have learned about MoveIt! interfacing using Gazebo simulation. In this section, we will see how to replace Gazebo and put a real robot interface to MoveIt!. Let's discuss the Dynamixel Servos and the ROS controllers.

The Dynamixel Servos

The Dynamixel Servos are smart, high performance networked actuators for high end robotics applications. These servos are manufactured by a Korean company called ROBOTIS (<http://en.robotis.com/>). These servos are very popular among robotics enthusiasts because they can provide excellent position and torque control, and also provide variety of feedback, such as position, speed, temperature, voltage, and so on.

One of their useful features is that they can be networked as a daisy chain manner. This feature is very useful in multijoint systems such as a robotic arm, humanoid robots, robotic snakes, and such others.

The servos can be directly connected to PCs using a USB to Dynamixel controller, which is provided from ROBOTIS. This controller has a USB interface and when it is plugged into the PC, it acts as a virtual COM port. We can send data to this port and internally it will convert the RS 232 protocol to TTL (Transistor-Transistor Logic) and in RS 485 standards. The Dynamixel can be powered and connect the USB to dynamixel controller to start working with it. Dynamixel servos support both TTL and RS 485 level standards. The following figure shows the Dynamixel servos called MX-106 and **USB To Dynamixel** controller.



Figure 13 : Dynamixel Servo and USB to Dynamixel controller.

There are different series of Dynamixel available in the market. Some of the series are MX - 28, 64 and 106, RX - 28,64, 106, and so on. The following is the connection diagram of Dynamixel, USB to Dynamixel to PC:

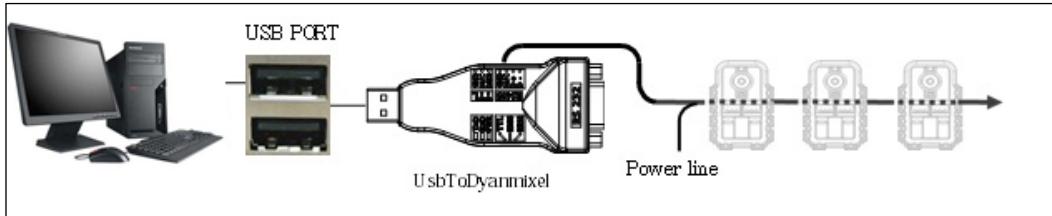


Figure 14 : Dynamixel Servos connected to PC using USB To Dynamixel controller.

The Dynamixel can be connected as daisy chain, as shown in the preceding figure. Each Dynamixel has a firmware setting inside its controller. We can assign the ID of servo, the joint limits, the position limits, the position commands, the PID values, the voltage limits, and so on, inside the controller. There are ROS drivers and controllers for Dynamixel which are available at:

http://wiki.ros.org/dynamixel_motor.

Dynamixel-ROS interface

The ROS stack for interfacing the Dynamixel motor is `dynamixel_motor`. This stack contains interface for Dynamixel motors such as MX-28, MX64, MX-106, RX-28, RX64, EX106, AX-12, and AX-18. The stack consists of the following packages:

- `dynamixel_driver`: This package is the driver package of Dynamixel, which can do low level IO communication with Dynamixel from PC. This driver has hardware interface for the previously mentioned series of servos and can do the read / write operation to Dynamixel through this package. This package is used by high level packages such as `dynamixel_controllers`. There are only few cases when the user directly interacts with this package.
- `dynamixel_controllers`: This is a higher level package that works using the `dynamixel_motor` package. Using this package, we can create a ROS controller for each Dynamixel joint of the robot. The package contains a configurable node, services, and spawner script to start, stop, and restart one or more controller plugins. In each controller, we can set the speed and the torque. Each Dynamixel controller can be configured using the ROS parameters or can be loaded by a YAML file. The `dynamixel_controllers` packages support the following kinds of controllers:
 - Joint Position controllers
 - Joint Torque controllers
 - Joint Trajectory Action controller
- `dynamixel_msgs`: These are the message definitions which are used inside the `dynamixel_motor` stack.

Interfacing seven DOF Dynamixel based robotic arm to ROS MoveIt!

In this section, we will discuss a 7 DOF robot manipulator called **COOL arm-5000**, which is manufactured by a company called ASIMOV Robotics (<http://asimovrobotics.com/>). The robot is built using Dynamixel servos (http://www.robotis.com/xe/dynamixel_en). We will see how to interface a Dynamixel-based robotic arm to ROS using `dynamixel_controllers`.

The following is a diagram of a COOL arm-5000:

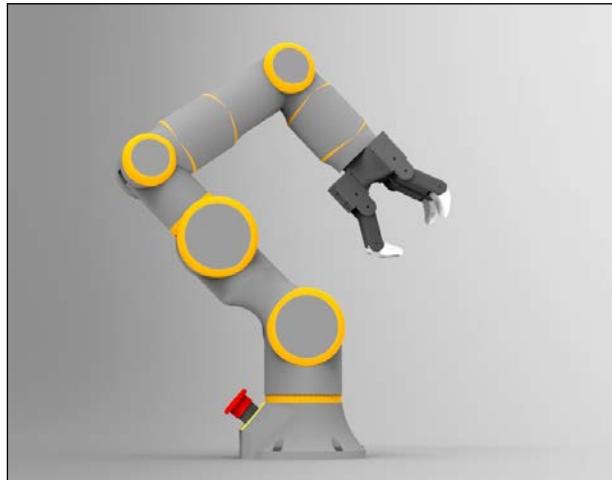


Figure 15 : COOL Arm illustration.

COOL arm robots are fully compatible with ROS and MoveIt! and are mainly used in education and research. The price range is between 5K - 10K USD.

Following are the details of the arms:

- Degree of Freedom: 7 DOF
- Types of Actuators: Dynamixel MX-64 and MX-28
- List of Joints: Shoulder Roll, Shoulder Pitch, Elbow Roll, Elbow Pitch, Wrist Yaw, Wrist Pitch, and Wrist Roll
- Payload: 5K
- Reach: 1 meter
- Work Volume: 2.09 m³
- Repeatability: +/- 0.05 mm
- Gripper with 3 fingers
- ROS support

Creating a controller package for COOL arm robot

The first step is to create a controller package for COOL arm for interfacing to ROS. The cool arm controller package is available for download along with the book codes.

The following command will create the controller package with necessary dependencies. The important dependency of this package is the dynamixel_controller package.

```
$ catkin_create_pkg cool5000_controller roscpp rospy dynamixel_controller  
std_msgs sensor_msgs
```

The next step is to create configuration file for each joint. The configuration file is called cool5000.yaml, which contains definition of each controller name, its type, and its parameters. We can see this file in the cool5000_controller/config folder. We have to create parameters for the seven joints in this arm. Following is a snippet of this config file:

```
joint1_controller:  
    controller:  
        package: dynamixel_controllers  
        module: joint_position_controller  
        type: JointPositionController  
    joint_name: joint1  
    joint_speed: 0.1  
    motor:  
        id: 0  
        init: 2048  
        min: 320  
        max: 3823  
  
joint2_controller:  
    controller:  
        package: dynamixel_controllers  
        module: joint_position_controller  
        type: JointPositionController  
    joint_name: joint2  
    joint_speed: 0.1  
    motor:  
        id: 1  
        init: 2048  
        min: 957  
        max: 3106
```

The controller configuration file mentions the joint name, package of the controller, controller type, joint speed, motor ID, initial position, and minimum and maximum limits of the joint. We can connect as many motors as we want and can create a controller parameters by including in this configuration file.

Next configuration file is to create a Joint Trajectory controller configuration. MoveIt! can only interface if the robot has the `FollowJointTrajectory` action server. The file called `cool5000_trajectory_controller.yaml` is put in the `cool5000_controller/config` folder and its definition is given next:

```
cool5000_trajectory_controller:  
  controller:  
    package: dynamixel_controllers  
    module: joint_trajectory_action_controller  
    type: JointTrajectoryActionController  
  joint_trajectory_action_node:  
    min_velocity: 0.0  
    constraints:  
      goal_time: 0.01
```

After creating the `JointTrajectory` controller, we need to create a `joint_state_aggregator` node for combining and publishing the joint states of the robotic arm. You can find this node from the `cool5000_controller/src` folder named `joint_state_aggregator.cpp`. The function of this node is to subscribe controller states of each controller having message type of `dynamixel::JointState` and combine each message of the controller into the `sensor_msgs::JointState` messages and publish in the `/joint_states` topic. This message will be the aggregate of the joint states of all the dynamixel controllers.

The definition of `joint_state_aggregator.launch`, which runs the `joint_state_aggregator` node with its parameters, follows. It is placed in the `cool5000_controller/launch` folder:

```
<launch>  
  <node name="joint_state_aggregator" pkg="cool5000_controller"  
        type="joint_state_aggregator" output="screen">  
    <rosparam>  
      rate: 50  
      controllers:  
        - joint1_controller  
        - joint2_controller  
        - joint3_controller  
        - joint4_controller  
        - joint5_controller
```

```
- joint6_controller  
- joint7_controller  
- gripper_controller  
</rosparam>  
</node>  
</launch>
```

We can launch the entire controller using the following launch file called `cool5000_controller.launch`, which is inside the launch folder.

The code inside this launch file will start communication between the PC and the Dynamixel servos and start the controller manager. The controller manager parameters are serial port, baud rate, servo ID range, and update rate.

```
<launch>  
  
    <!-- Start the Dynamixel motor manager to control all cool5000  
    servos -->  
  
        <node name="dynamixel_manager" pkg="dynamixel_controllers"  
        type="controller_manager.py" required="true" output="screen">  
            <rosparam>  
                namespace: dxl_manager  
                serial_ports:  
                    dynamixel_port:  
                        port_name: "/dev/ttyUSB0"  
                        baud_rate: 1000000  
                        min_motor_id: 0  
                        max_motor_id: 6  
                        update_rate: 20  
            </rosparam>  
        </node>  
</launch>
```

In the next step, it should launch the controller spawner by reading the controller config file:

```
<!-- Load joint controller configuration from YAML file to  
parameter server -->  
    <rosparam file="$(find cool5000_controller)/config/cool5000.yaml"  
    command="load"/>  
  
    <!-- Start all Cool Arm joint controllers -->  
        <node name="controller_spawner" pkg="dynamixel_controllers"  
        type="controller_spawner.py"  
        args="--manager=dxl_manager"
```

```
--port dynamixel_port
joint1_controller
joint2_controller
    joint3_controller
    joint4_controller
    joint5_controller
    joint6_controller
joint7_controller
    gripper_controller"
output="screen"/>
```

In the next section of the code, it will launch the `JointTrajectory` controller from the controller configuration file:

```
<!-- Start the cool5000 arm trajectory controller -->
<rosparam file="$(find cool5000_controller)/config/cool5000_
trajectory_controller.yaml" command="load"/>
<node name="controller_spawner_meta" pkg="dynamixel_controllers"
type="controller_spawner.py"
args="--manager=dxl_manager
--type=meta
cool5000_trajectory_controller
joint1_controller
joint2_controller
joint3_controller
joint4_controller
joint5_controller
joint6_controller"
output="screen"/>
```

The following section will launch the joint state aggregator node and the robot description from the `cool5000_description` package:

```
<!-- Publish combined joint info -->
<include file="$(find cool5000_controller)/launch/joint_state_
aggregator.launch" />

<param name="robot_description" command="$(find xacro)/xacro.py
'$(find cool5000_description)/robots/cool5000.xacro'" />
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" output="screen">
    <rosparam param="source_list">[joint_states]</rosparam>
    <rosparam param="use_gui">FALSE</rosparam>
</node>
```

This is all about the cool arm controller package. Next, we need to setup the controllers configuration inside the MoveIt! configuration package of cool arm called `cool5000_moveit_config`.

MoveIt! configuration of the COOL Arm

The first step is to configure `controllers.yaml`, which is inside the `cool5000_moveit_config/config` folder. The definition of this file follows. We are only focusing on moving the arm and not on handling the gripper control for now. So the configuration only contains the arm group joints:

```
controller_list:  
  - name: cool5000_trajectory_controller  
    action_ns: follow_joint_trajectory  
    type: FollowJointTrajectory  
    default: true  
    joints:  
      - joint1  
      - joint2  
      - joint3  
      - joint4  
      - joint5  
      - joint6  
      - joint7
```

The following is the definition of `cool5000_description_moveit_controller_manager.launch.xml` inside `cool5000_moveit_config/launch`:

```
<launch>  
<!--  
 Set the param that trajectory_execution_manager needs to find the  
 controller plugin  
-->  
<arg name="moveit_controller_manager" default="MoveIt_simple_  
 controller_manager/MoveItSimpleControllerManager"/>  
  
<param name="MoveIt_controller_manager" value="$(arg MoveIt_  
 controller_manager)"/>  
  
<!-- load controller_list -->  
  
<rosparam file="$(find cool5000_moveit_config)/config/controllers.  
 yaml"/>  
</launch>
```

After configuring MoveIt!, we can start working on the arm. Apply proper power supply on the arm and connect it to USB To Dynamixel. Plug the USB TO Dynamixel to a PC. We will see a serial device generate; it may be either /dev/ttyUSB0 or /dev/ttyACM0. According to the device, change the port name inside the controller launch file.

Start the cool5000 arm controller using the following command:

```
$ rosrun cool5000_controller cool5000_controller.launch
```

Start the RViz demo and start path planning. If we press the **Execute** button, the trajectory will execute on the hardware arm:

```
$ rosrun cool5000_moveit_config demo.launch
```

A random pose, which is shown in RViz, and the cool arm is shown in the following image:



Figure 16 : COOL-Arm-5000 prototype with MoveIt! visualization

Questions

1. What is the role of the FCL library in MoveIt!?
2. How does MoveIt! build OctoMap of the environment?
3. What is the main function of grasp server?
4. What are the main features of dynamixel servos?

Summary

In this chapter, we explored some advanced features of MoveIt! and how we can interface it into a real hardware. The chapter started with a discussion on collision checking using MoveIt!. We saw how to add a collision object using MoveIt! APIs and also saw the direct importing of mesh to the planning scene. We discussed a ROS node to check collision using MoveIt! APIs. After learning about collisions, we moved to perception using MoveIt!. We connected the simulated point cloud data to MoveIt! and created an OctoMap in MoveIt!. The next topic we discussed was grasping, using the `moveit_simple_grasp` package. We saw the grasp generator using this package and we made a simple pick and place task using the grasp server and the pick and place node. After discussing these things, we switched to hardware interfacing of MoveIt! using dynamixel servos and its ROS controllers. In the end, we saw a real robotic arm called COOL arm and its interfacing to MoveIt!, which was completely built using dynamixel controllers.

11

ROS for Industrial Robots

In the previous chapter, we have seen some advanced concepts in **ROS-MoveIt!** Until now, we have been discussing mainly about interfacing personal and research robots with ROS, but one of the main areas where robots are extensively used are industries. Does ROS support industrial robots? Do any of the companies use ROS for the manufacturing process? The ROS-Industrial packages comes with a solution to interface industrial robot manipulators to ROS and controlling it using the power of ROS, such as MoveIt!, Gazebo, RViz, and so on.

In this chapter, we will discuss the following topics:

- Understanding ROS-Industrial packages
- Installing ROS-Industrial packages in ROS
- Block diagram of ROS-Industrial packages
- Creating URDF for an industrial robot
- Creating the MoveIt! interface for an industrial robot
- Installing ROS-Industrial packages of Universal robotic arms
- Understanding MoveIt! configuration of a universal robotic arm
- Working with MoveIt! configuration of ABB robots
- Understanding ROS-Industrial robot support packages
- ROS-Industrial robot client package
- ROS-Industrial robot driver package
- Understanding MoveIt! IKFast plugin
- Creating the MoveIt! IKFast plugin for an ABB-IRB6640 robot

Let's start with a brief overview of ROS-Industrial.

Understanding ROS-Industrial packages

ROS-Industrial basically extends the advanced capabilities of ROS software to industrial robots working in the production process. ROS-Industrial consists of many software packages, which can be used for interfacing industrial robots. These packages are **BSD** (legacy) / **Apache 2.0** (preferred) licensed program, which contain libraries, drivers, and tools for industrial hardware. The ROS-Industrial is now guided by the ROS-Industrial Consortium. The official website of ROS-I is <http://rosindustrial.org/>. The following diagram is the logo of ROS-I:



Figure 1: Logo of ROS-Industrial

Goals of ROS-Industrial

The main goals behind developing ROS-Industrial are given as follows:

- Combine strengths of ROS to the existing industrial technologies for exploring advanced capabilities of ROS in the manufacturing process
- Developing a reliable and robust software for industrial robots application.
- Provide an easy way for doing research and development in industrial robotics
- Create a wide community supported by researchers and professionals for industrial robotics
- Provide industrial grade ROS application and become a one-stop location of industry-related applications

ROS-Industrial – a brief history

In 2012, the ROS-Industrial open source project started as the collaboration of Yaskawa Motoman Robotics (<http://www.motoman.com/>), Willow Garage (<https://www.willowgarage.com/>) and **Southwest Research Institute (SwRI)** at <http://www.swri.org/> for using ROS research and development in Industrial manufacturing. The ROS-I was founded by Shaun Edwards in January 2012.

In 2013, the ROS-I Consortium Americas launched in March 2013 led by SwRI and ROS-I Consortium Europe led by Fraunhofer IPA in Germany.

Benefits of ROS-Industrial

Let's see the benefits ROS-I provides to the community:

- **Explore the features in ROS:** The ROS-Industrial packages are tied to the ROS framework so that we can use all ROS features in industrial robots too. Using ROS, we can create custom IK solvers for each robot, object manipulation using 2D/3D perception. ROS also provides a rich toolset, such as RViz, Gazebo, and rqt_gui for visualization, simulation, and debugging
- **Out-of-the-box applications:** The ROS interface enables advanced perception in robots for working with picking and placing complex objects.
- **Simplifies robotic programming:** ROS-I eliminates teaching and planning paths of robots and instead of it, automatically calculates a collision-free optimal path for the given points.
- **Low Cost:** Instead of costly proprietary robotic simulators, ROS-I is an open source software that allows commercial use without any restrictions.

Installing ROS-Industrial packages

Installing ROS-I packages can be done using package managers or building from the source code. If we have installed the ros-desktop-full installation, we can use the following command to install ROS-Industrial packages on Ubuntu 14.04.3. The following command will install ROS-Industrial packages on ROS Indigo:

```
$ sudo apt-get install ros-indigo-industrial-core ros-indigo-open-industrial-ros-controllers
```

The preceding command will install the core packages of ROS-Industrial packages. The `industrial-core` stack includes the following set of ROS packages:

- `industrial-core`: This stack contains packages and libraries for supporting industrial robotic systems. The stack consists of nodes for communicating with industrial robot controllers, industrial robot simulators, and also provides ROS controllers for industrial robots.
- `industrial_DEPRECATED`: This package contains nodes, launch files, and so on that are going to be deprecated. The files inside this package will delete soon from the repository, so we should look for the replacement of these files before the content is going to be deleted.
- `industrial_msgs`: This package contains message definitions, which are specific to the ROS-Industrial packages.
- `simple_message`: This is a part of ROS-Industrial stacks, which is a standard message protocol containing a simple messaging framework for communicating with industrial robot controllers.
- `industrial_robot_client`: This package contains a generic robot client for connecting to industrial robot controllers, which is running an industrial robot server and can communicate using a simple message protocol.
- `industrial_robot_simulator`: This package simulates the industrial robot controller, which follows the ROS-Industrial driver standard. Using this simulator, we can simulate and visualize the industrial robot.
- `industrial_trajectory_filters`: This package contains libraries and plugins for filtering the trajectories, which is sent to the robot controller.

Block diagram of ROS-Industrial packages

The following diagram a simple block diagram representation of ROS-I packages, which are organized on top of ROS. We can see the ROS-I layer on top of the ROS layers. We can see a brief description of each of the layers for better understanding. The following diagram is taken from ROS-I wiki page (<http://wiki.ros.org/Industrial>).

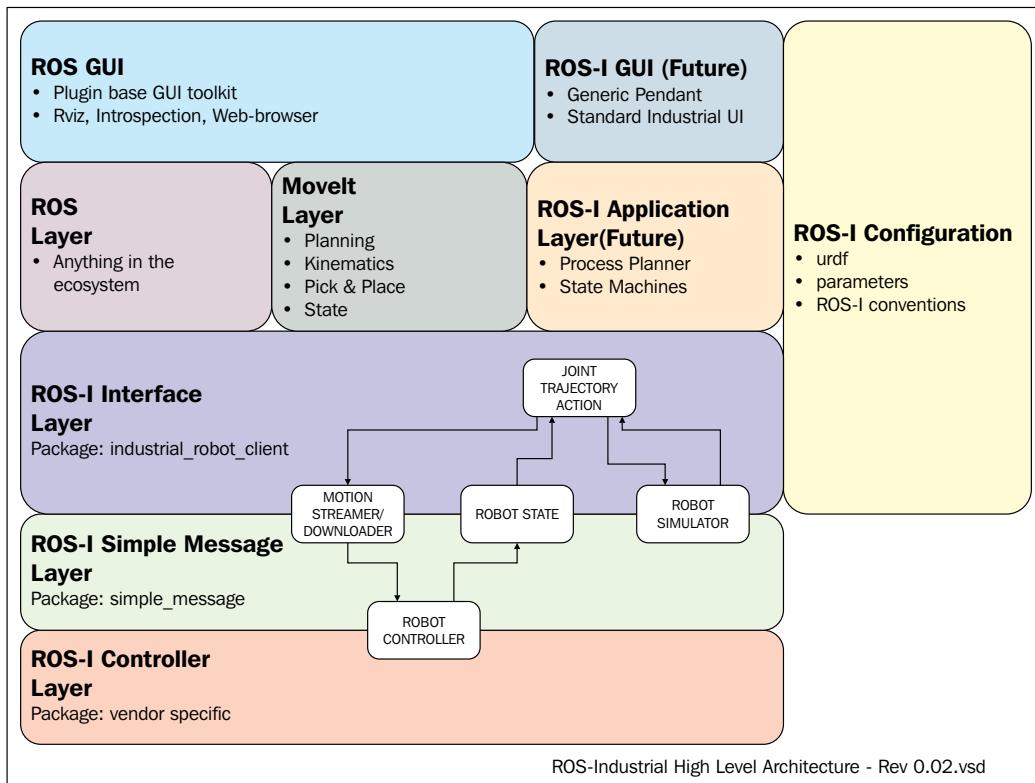


Figure 2: The block diagram of ROS-Industrial

- **The ROS GUI:** This layer includes the ROS plugin-based GUI tools layer, which consists of tools such as RViz, rqt_gui, and so on
- **The ROS-I GUI:** These GUIs are standard industrial UI for working with industrial robots which may be implemented in the future
- **The ROS Layer:** This is the base layer in which all communications are taking place
- **The MoveIt! Layer:** The MoveIt! layer provides a direct solution to industrial manipulators in planning, kinematics, and pick and place
- **The ROS-I Application Layer:** This layer consists of an industrial process planner, which is used to plan what is to be manufactured, how it will be manufactured, and what resources are needed for the manufacturing process

- **The ROS-I Interface Layer:** This layer consists of the industrial robot client, which can connect to the industrial robot controller using the simple message protocol
- **The ROS-I- Simple Message Layer:** This is the communication layer of the industrial robot, which is a standard set of protocol that will send data from the robot client to the controller and vice versa.
- **The ROS-I Controller Layer:** This layer has vendor-specific industrial robot controllers.

After discussing the basic concepts, we can start working on interfacing an industrial robot to ROS using ROS-I. The following are the major issues we need to address:

- How to create a URDF model for an industrial manipulator
- How to create MoveIt! interface for an industrial manipulator
- What are industrial robot driver packages?
- What are support packages in ROS-I and how are created?
- How to create IKFast MoveIt! plugins for Industrial robots

We can see each issue and its solutions with an example

Creating URDF for an industrial robot

Creating the URDF file for an ordinary robot and industrial robot are the same, but in industrial robots, there are some standards that should be strictly followed during its URDF modeling, which are as follows:

- **Simplify the URDF design:** The URDF file should be simple and readable and only need the important tags
- **Common design:** Developing a common design formula for all industrial robots by various vendors
- **Modularizing URDF:** The URDF needs to modularize using XACRO macros and it can be included in a large URDF file without much hassle.

The following points are the main difference in the URDF design followed by ROS-I.

- **Collision-Aware:** The industrial robot IK planners are collision aware so the URDF should contain accurate collision 3D mesh for each link. Every link in the robot should export to STL or DAE with a proper coordinate system. The coordinate system which ROS-I is following are X-axis pointing forward and Z-axis pointing up when each joint is in zero position. It is also to be noted that if the joint's origin coincides with the base of the robot, the transformation will be simpler. It will be good if we are putting robot-based joints in zero position (origin), which can simplify the robot design.
- In ROS-I, the mesh file used for visual purpose is highly detailed, but the mesh file used for collision will not be detailed, because it takes more time to perform collision checking. In order to remove the mesh details, we can use tools such as MeshLab (<http://meshlab.sourceforge.net/>) using its option (Filters -> Remeshing, Simplification and Reconstruction -> Convex Hull).
- **URDF Joint conventions:** The orientation value of each robot joint is limited to single rotation, that is, out of the two orientation (roll, pitch, and yaw) values, only one value will be there.
- **Xacro Macros:** In ROS-I, the entire manipulator section is written as a macro using xacro. We can add an instance of this macro in another macro file, which can be used for generating a URDF file. We can also include additional end effector definitions on this same file.
- **Standards Frames:** In ROS-I, the `base_link` frame should be the first link and `tool0` (tool-zero) should be the end effector link. Also, the `base` frame should match with the base of the robot controller. In most cases, transform from `base` to `base_link` is treated as fixed.

After building the xacro file for the industrial robot, we can convert to URDF and verify it using the following command:

```
$ rosrun xacro xacro.py -o <urdf_file> <xacro_file>
$ check_urdf <urdf_file>
```

Next, we can discuss the differences in creating the MoveIt! configuration for an industrial robot.

Creating MoveIt! configuration for an industrial robot

The procedure for creating the MoveIt! interface for industrial robots are same as the other ordinary robot manipulators except in some standard conventions. The following procedures give a clear idea about these standard conventions:

- Launch the MoveIt! setup assistant using the following command:
`$ rosrun moveit_setup_assistant setup_assistant.launch`
- Load the URDF from the robot description folder or convert xacro to URDF and load to the setup assistant
- Create a **Self-Collision** matrix with **Sampling Density** about ~ 80,000. This value can increase the collision checking in the arm
- Add a **Virtual Joint** matrix as shown in the following screenshot. Here the virtual and parent frame names are arbitrary.

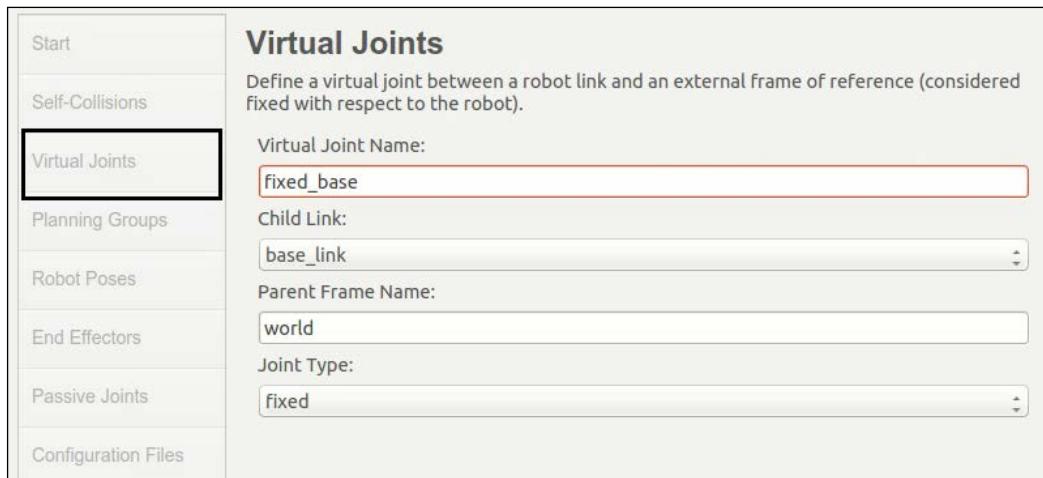


Figure 3: Adding MoveIt! - Virtual joints

- In the next step, we are adding **Planning Groups for manipulator and End Effector**, here also the group names are arbitrary. The default plugin is KDL, we can change it even after creating the MoveIt! configuration.

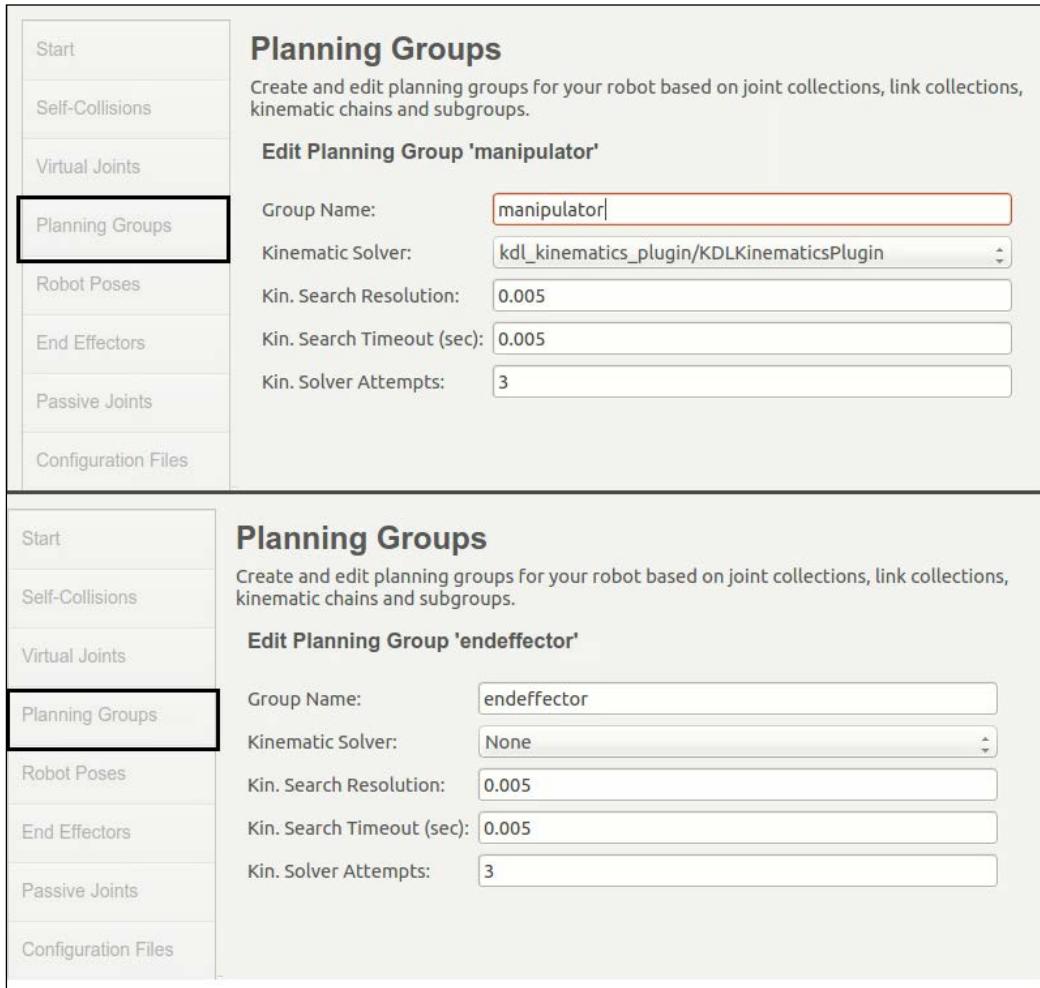


Figure 4: Creating Planning Groups in MoveIt!

The planning groups, that is, the manipulator plus the end effector configuration, will be shown like this:

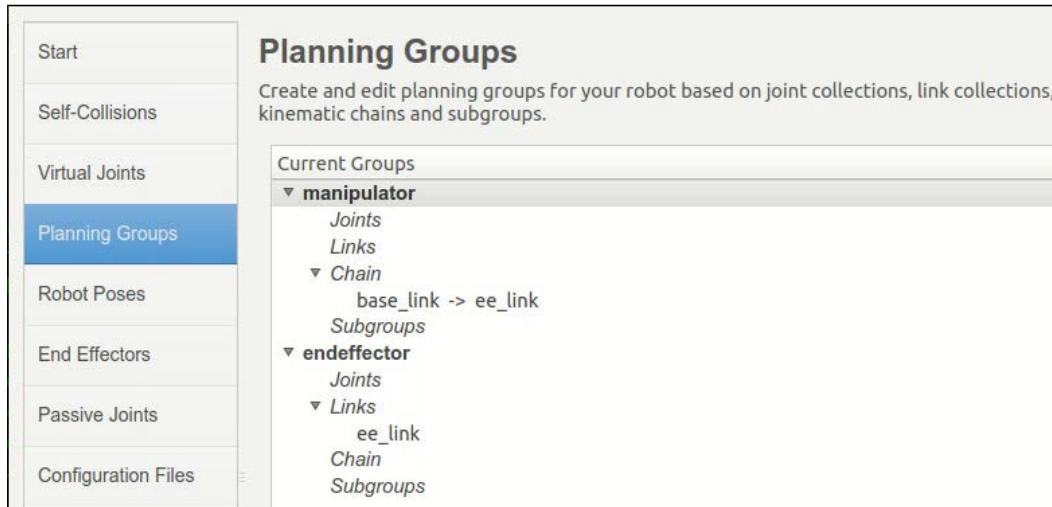


Figure 5: Planning groups of manipulator + end effector in MoveIt!

- We can assign **Robot Poses**, such as home position, up position, and so on. This setting is an optional one.
- We can assign **End Effectors** as shown in the following screenshot; this is also an optional setting:

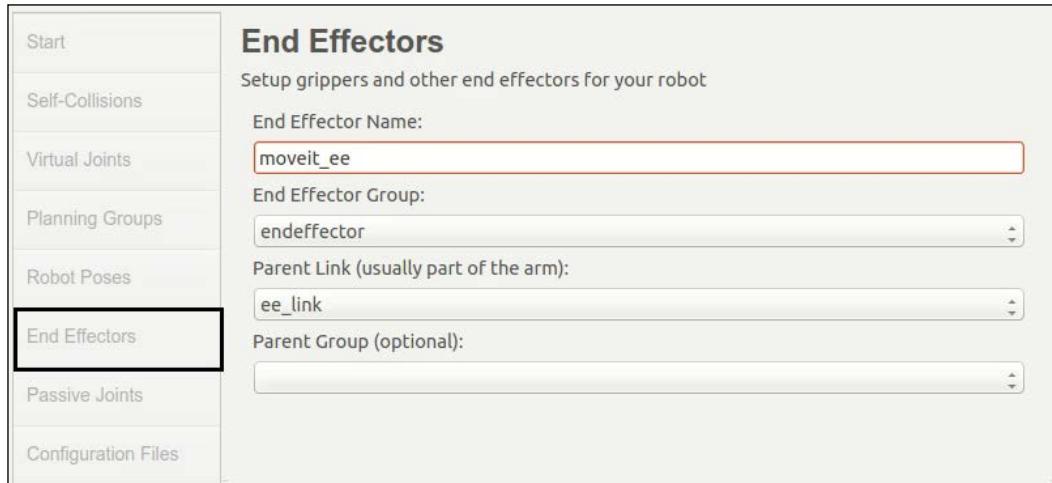


Figure 6: Setting End Effectors in MoveIt! Setup Assistant

- After setting the end effector, we can directly generate the configuration files. It should be noted that the `moveit-config` package should be named as `<robot_name>_moveit_config`, where `robot_name` is the name of the URDF file. Also, if we want to move this generated config package to another PC, we need to edit the `.setup_assistant` file which is inside the `moveit` package. We should change the absolute path to the relative path. Here is an example of the `abb_irb2400` robot. We should mention the relative path of URDF and SRDF in this file, as follows:

```
moveit_setup_assistant_config:  
    URDF:  
        package: abb_irb2400_support  
        relative_path: urdf/irb2400.urdf  
    SRDF:  
        relative_path: config/abb_irb2400.srdf  
    CONFIG:  
        generated_timestamp: 1402076252
```

Updating the MoveIt! configuration files

After creating the MoveIt! configuration, we should update the `controllers.yaml` file inside the `config` folder of the MoveIt! package. Here is an example of `controllers.yaml`:

```
controller_list:  
    - name: ""  
        action_ns: follow_joint_trajectory  
        type: FollowJointTrajectory  
        joints:  
            - shoulder_pan_joint  
            - shoulder_lift_joint  
            - elbow_joint  
            - wrist_1_joint  
            - wrist_2_joint  
            - wrist_3_joint
```

We should also update `joint_limits.yaml` about the joint information. Here is a code snippet of `joint_limits.yaml`:

```
joint_limits:  
    shoulder_pan_joint:  
        has_velocity_limits: true  
        max_velocity: 2.16  
        has_acceleration_limits: true  
        max_acceleration: 2.16
```

We can also change the Kinematic solver plugin by editing the `kinematics.yaml` file. After editing all the configuration files, we need to edit the controller manager launch file (`<robot>_moveit_config/launch/<robot>_moveit_controller_manager.launch`).

Here is an example of the `controller_manager.launch` file:

```
<launch>

<rosparam file="$(find ur10_moveit_config)/config/
    controllers.yaml"/>

<param name="use_controller_manager" value="false"/>

<param name="trajectory_execution/execution_duration_monitoring"
    value="false"/>

<param name="moveit_controller_manager" value=
    "moveit_simple_controller_manager/
        MoveItSimpleControllerManager"/>
</launch>
```

After creating the controller manager, we need to create the `<robot>_moveit_planning_execution.launch` file. Here is an example of this file:

```
<launch>
    <arg name="sim" default="false" />
    <arg name="limited" default="false"/>
    <arg name="debug" default="false" />

    <!-- Remap follow_joint_trajectory -->
    <remap if="$(arg sim)" from="/follow_joint_trajectory" to="/arm_
        controller/follow_joint_trajectory"/>

    <!-- Launch moveit -->
    <include file="$(find ur10_moveit_config)/launch/move_group.launch">
        <arg name="limited" default="$(arg limited)"/>
        <arg name="debug" default="$(arg debug) " />
    </include>
</launch>
```

Testing the MoveIt! configuration

After editing the configuration and launch files in the MoveIt! configuration, we can start running the robot simulation and can check whether the MoveIt! configuration is working well or not. Ensure the `ros-industrial-simulator` package is installed properly. Here are the steps to test an industrial robot.

- Start the robot simulator
- Start the MoveIt! planning execution launch file using the following command line:
`$ rosrun <robot>_moveit_config moveit_planning_execution.launch`
- Open RViz and load RViz **Motion planning plugin** using the **Plan and Execute** button. We can plan and execute the trajectory on the simulated robot.

Installing ROS-Industrial packages of universal robotic arm

The Universal Robots (<http://www.universal-robots.com/>) is an industrial robot manufacturer based in Denmark. The company mainly manufactures three arms **UR3**, **UR5**, and **UR10**. The robots are shown in the following screenshot:

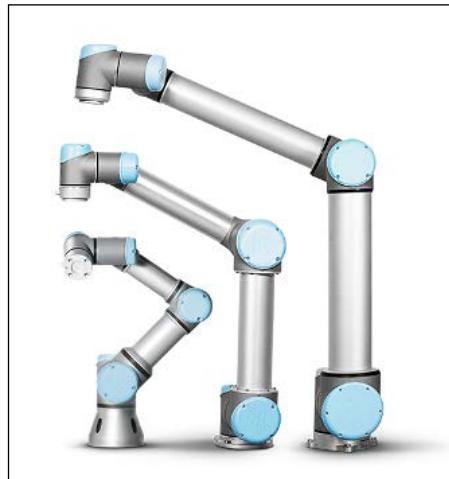


Figure 7: UR-3, UR-5, and UR-10 robots

The smaller one is **UR-3**, **UR-5** is the one in the center, and the big one is **UR-10**. The specifications of these robots are given in the following table:

Robot	UR-3	UR-5	UR-10
Working radius	500 mm	850 mm	1300 mm
Payload	3 kg	5 kg	10 kg
Weight	11 kg	18.4 kg	28.9 kg
Footprint	118 mm	149 mm	190 mm

We are mainly discussing ROS interfacing of UR-5 and UR-10 using ROS-I packages.

We can install the packages of these robots and can work with the MoveIt! interface and simulation interface of these robots in Gazebo.

Installing the ROS interface of universal robots

We can install the latest packages of the universal robot using the source installation.

Create a workspace for the industrial robot packages called `ros_industrial_ws` and clone the universal robot code to the `src` folder as follows:

```
ros_industrial_ws/src$ git clone https://github.com/ros-industrial/universal_robot.git
```

We can also install its Ubuntu binary packages using the following command:

```
$ sudo apt-get install ros-indigo-universal-robot
```

The universal robot stack consists of the following packages:

- `ur_description`: This package consists of the robot description and gazebo description of UR-5 and UR-10.
- `ur_driver`: This package contains client nodes, which can communicate to the UR-5 and UR-10 robot hardware controllers.
- `ur_bringup`: This package consists of launch files to start communication with the robot hardware controllers to start working with the real robot.

- `ur_gazebo`: This package consists of gazebo simulations of both UR-5 and UR-10.
- `ur_msgs`: This package contains ROS messages used for communication between various UR nodes.
- `ur10_moveit_config/ur5_moveit_config`: These are the moveit config files of UR-5 and UR-10 robots.
- `ur_kinematics`: This package contains kinematic solver plugins for UR-5 and UR-10. We can use this solver plugin in MoveIt!.

Build the packages using the `catkin_make` command and add the following line to the `.bashrc` file for accessing the preceding packages:

```
source ~/ros_industrial_ws/devel/setup.bash
```

We can launch the simulation in Gazebo of UR-10 robot using the following command:

```
$ roslaunch ur_gazebo ur10.launch
```

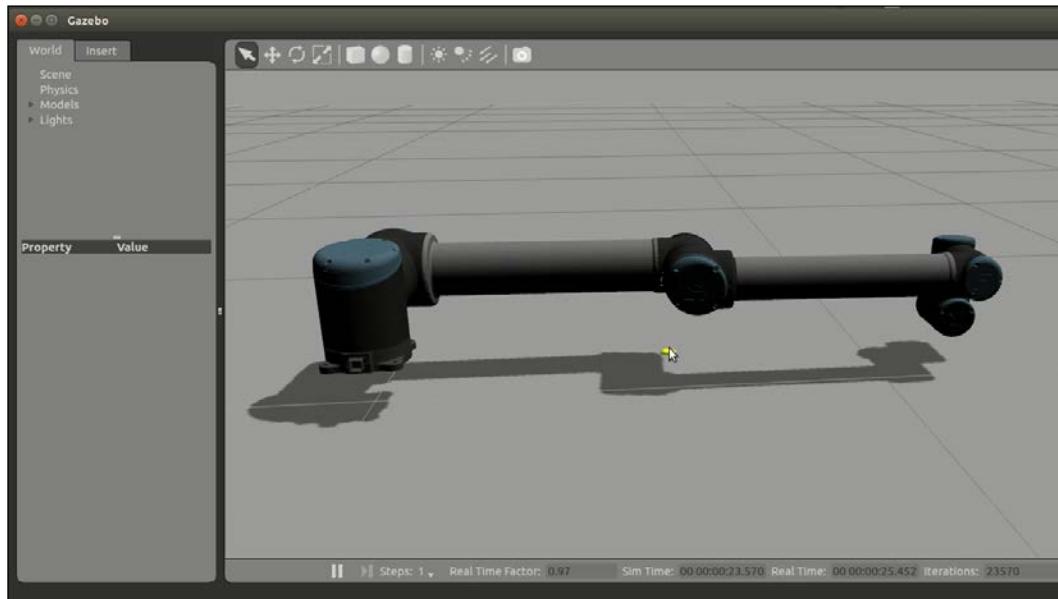


Figure 8: Universal robot, UR-10 model simulation in Gazebo

We can see the robot controller configuration file for interfacing into the MoveIt! package. The following YAML file defines the JointTrajectory controller. It is placed in the `ur_gazebo/controller` folder with a name `arm_controller_ur10.yaml`:

```
arm_controller:  
    type: position_controllers/JointTrajectoryController  
    joints:  
        - shoulder_pan_joint  
        - shoulder_lift_joint  
        - elbow_joint  
        - wrist_1_joint  
        - wrist_2_joint  
        - wrist_3_joint  
    constraints:  
        goal_time: 0.6  
        stopped_velocity_tolerance: 0.05  
        shoulder_pan_joint: {trajectory: 0.1, goal: 0.1}  
        shoulder_lift_joint: {trajectory: 0.1, goal: 0.1}  
        elbow_joint: {trajectory: 0.1, goal: 0.1}  
        wrist_1_joint: {trajectory: 0.1, goal: 0.1}  
        wrist_2_joint: {trajectory: 0.1, goal: 0.1}  
        wrist_3_joint: {trajectory: 0.1, goal: 0.1}  
    stop_trajectory_duration: 0.5  
    state_publish_rate: 25  
    action_monitor_rate: 10
```

We can see the necessary settings which have to be done in the robot Moveit! config package for interfacing the Gazebo controller.

Understanding the Moveit! configuration of a universal robotic arm

The changes that we need to make in the industrial MoveIt! configuration are almost the same as the arm we already worked with.

First, we have to define the `controller.yaml` file, which has to create inside `ur10_moveit_config/config`. Here is the definition of the `controller.yaml` of UR-10:

```
controller_list:  
    - name: ""  
        action_ns: follow_joint_trajectory
```

```
type: FollowJointTrajectory
joints:
  - shoulder_pan_joint
  - shoulder_lift_joint
  - elbow_joint
  - wrist_1_joint
  - wrist_2_joint
  - wrist_3_joint
```

The `kinematics.yaml` file inside the `config` folder contains the IK solvers used for this arm; we can use the following IK solvers. The contents of this file are given as follows:

```
#manipulator:
#  kinematics_solver: ur_kinematics/UR10KinematicsPlugin
#  kinematics_solver_search_resolution: 0.005
#  kinematics_solver_timeout: 0.005
#  kinematics_solver_attempts: 3
manipulator:
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
  kinematics_solver_search_resolution: 0.005
  kinematics_solver_timeout: 0.005
  kinematics_solver_attempts: 3
```

The UR-10 and UR-5 have their custom IK solver plugins and we can switch from the default KDL kinematics plugins to the robot specific solver.

The definition of `ur10_moveit_controller_manager.launch` inside the `launch` folder is given as follows. This launch file loads the trajectory controller configuration and starts the trajectory controller manager:

```
<launch>
  <rosparam file="$(find ur10_moveit_config)/config/controllers.yaml"/>
  <param name="use_controller_manager" value="false"/>
  <param name="trajectory_execution/execution_duration_monitoring" value="false"/>
  <param name="moveit_controller_manager" value="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
</launch>
```

After discussing these files, let's see how to execute motion planning in MoveIt! and executing in Gazebo:

1. Start the simulation of UR-10 with joint trajectory controllers:
`$ roslaunch ur_gazebo ur10.launch`
2. Start the MoveIt! nodes for motion planning. We need to use `sim:=true`, if we are trying MoveIt! along with the simulation:
`$ roslaunch ur10_moveit_config ur10_moveit_planning_execution.launch sim:=true`
3. Launch RViz with the MoveIt! visualization plugin:
`$ roslaunch ur10_moveit_config moveit_rviz.launch config:=true`

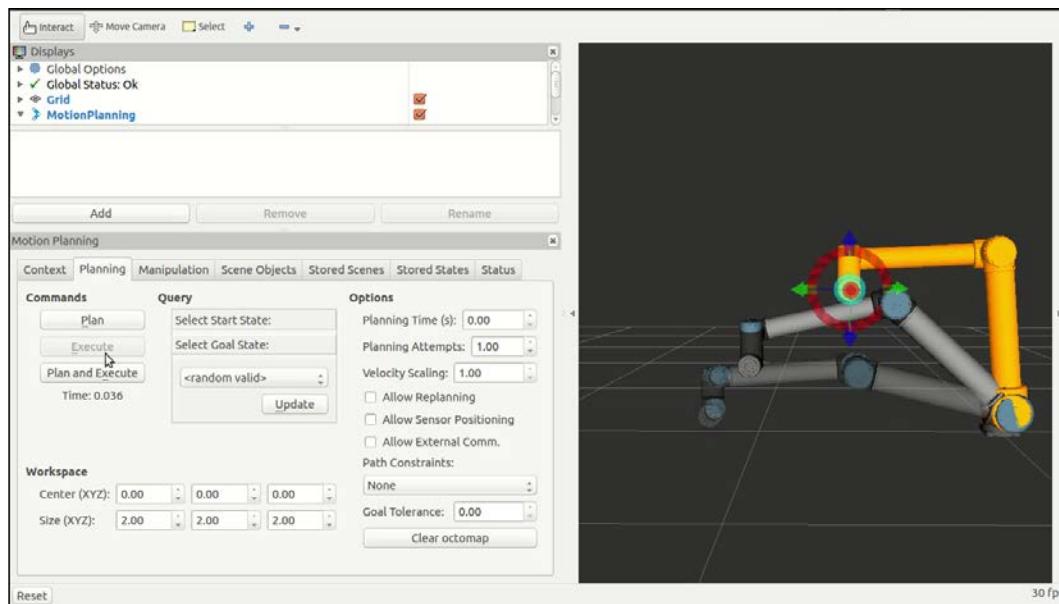


Figure 9: Motion planning in UR-10 model in RViz

We can move the end effector position of the robot and plan the path using the **Plan** button. When we press the **Execute** button or the **Plan and Execute** button, the trajectory should send to the simulated robot, which is shown as follows.

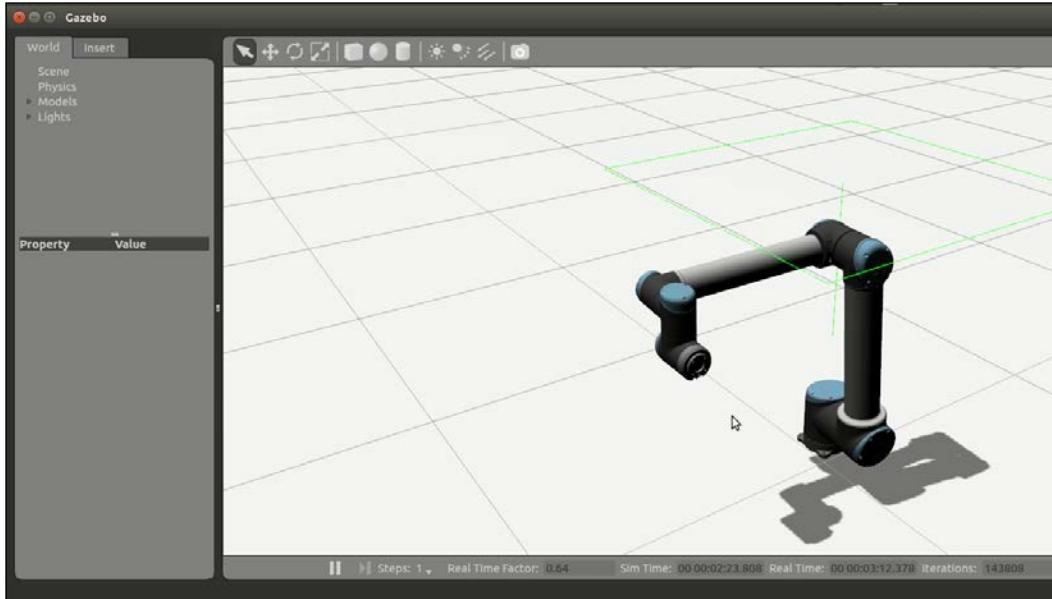


Figure 10 : Motion planned trajectory from MoveIt! executing in Gazebo

We have seen a universal robot and its simulation in Gazebo. Next, we can work with ABB robots.

Working with MoveIt! configuration of ABB robots

We will work with the motion planning of the popular ABB industrial robot models such as IRB 2400 and IRB 6640. The following are the images of these two robots and their specifications.



Figure 11: ABB IRB 2400 and IRB 6640

The arm specification of the IRB 2400-10 and 6640-130 models are given in the following table:

Robot	IRB 2400-10	IRB 6640-130
Working radius	1.55 m	3.2 m
Payload	12 kg	130 kg
Weight	380 kg	1310-1405 kg
Footprint	723x600 mm	1107 x 720 mm

To work with ABB packages, clone the ROS packages of the robot into the `catkin` workspace. We can use the following command to do this task:

```
$ git clone https://github.com/ros-industrial/abb
```

We can also install packages using the Ubuntu binary packages. The following package will install a complete set of ABB robot packages:

```
$ sudo apt-get install ros-<distro>-abb
```

Build the source packages using `catkin_make` and the following command will launch ABB IRB 6640 in RViz for motion planning:

```
$ roslaunch abb_irb6640_moveit_config demo.launch
```

The following RViz window will appear and we can start motion planning the robot in RViz:

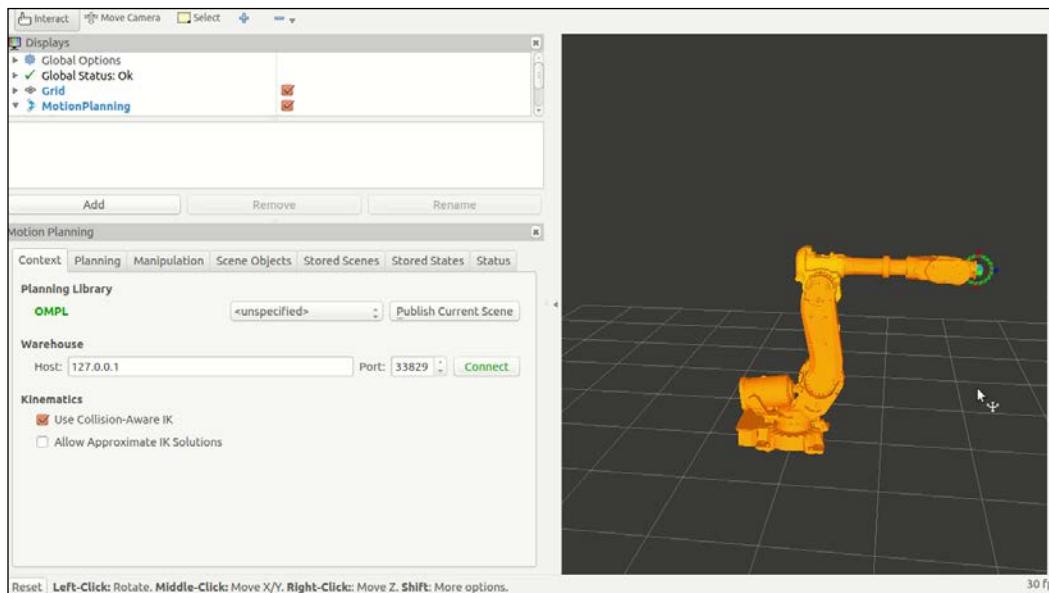


Figure 12: Motion planning of ABB IRB 6640

One of the other ABB robot model is IRB 2400. We can launch the robot in RViz using the following command:

```
$ roslaunch abb_irb2400_moveit_config demo.launch
```

The following is a screenshot of motion planning this robot:

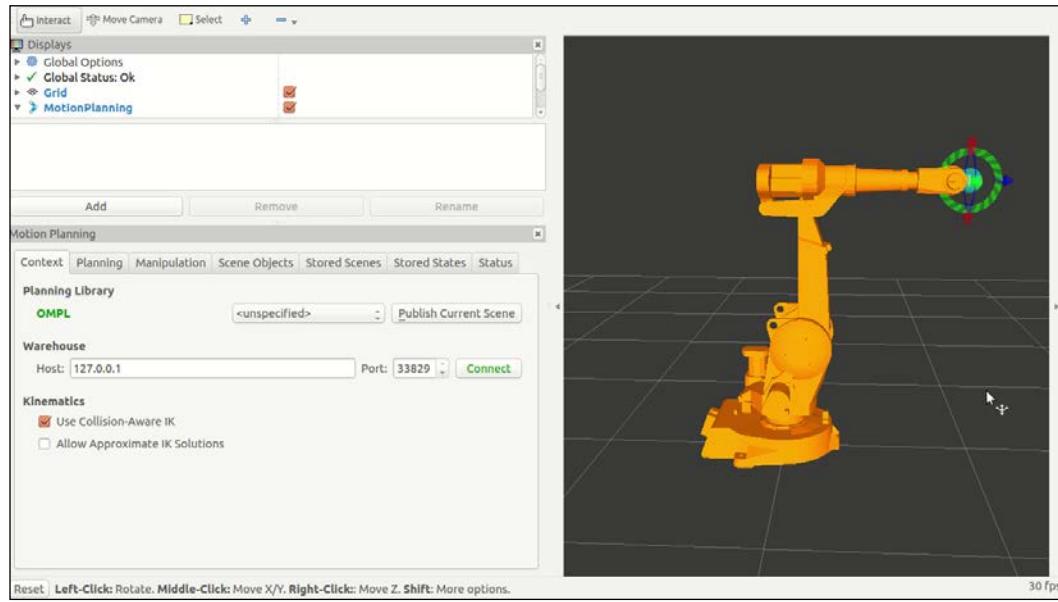


Figure 13: Motion planning of ABB IRB 2400

Understanding the ROS-Industrial robot support packages

The ROS-I robot support packages are a new convention followed for industrial robots. The aim of these support packages are to standardize the ways of maintaining ROS packages for a wide variety of industrial robot types of different vendors. Because of a standardized way of keeping files inside support packages, we don't have any confusion in accessing the files inside it. We can demonstrate a support package of an ABB robot and can see the folders and files and its uses.

We have already cloned the ABB robot packages and inside this folder we can see three support packages that support three variety of ABB robots. Here we are taking the ABB IRB 2,400 model support package: `abb_irb2400_support`. This is the support package of the ABB industrial robot model called IRB 2400. The following list shows the folders and files inside this package:

- config: As the name of the folder, this contains the configuration files of joint names, RViz configuration, and robot model specific configuration
 - joint_names_irb2400: Inside the config folder, there is a configuration file, which contains the joint names of the robot which is used by the ROS controller.
- launch: This folder contains the launch file definitions of this robot. These files are following a common convention in all industrial robots.
 - load_irb2400.launch: This file simply loads `robot_description` on the parameter server. According to the complexity of the robot the number of xacro files can be increased. This file loads all xacro in a single launch file. Instead of writing separate code for adding `robot_description` in other launch files, we can simply include this launch file.
 - test_irb2400.launch: This launch file can visualize the loaded URDF. We can inspect and verify the URDF in RViz. This launch file includes the preceding launch files and starts `joint_state_publisher` and `robot_state_publisher` nodes, which helps to interact with the user on RViz. This will work without the need for real hardware.
 - robot_state_visualize_irb2400.launch: This launch file visualizes the current state of the real robot by running nodes from the ROS-Industrial driver package with appropriate parameters. The current state of the robot is visualized by running RViz and the `robot_state_publisher` node. This launch file needs a real robot or simulation interface. One of the main arguments provided along with this launch file is the IP address of the industrial controller. Also note that the controller should run a ROS-Industrial server node.
 - robot_interface_download_irb2400.launch: This launch file starts bi-directional communication with the industrial robot controller to ROS and vice versa. There are industrial robot client nodes for reporting the state of robot (`robot_state` node) and subscribing the joint command topic and issuing the joint position to the controller (`joint_trajectory` node). This launch file also requires access to the simulation or real robot controller and needs to mention the IP address of the industrial controllers. The controller should run the ROS-Industrial server programs too.

- **urdf:** This folder contains the set of standardized xacro files of the robot model:
 - `irb2400_macro.xacro`: This is the xacro definition of a specific robot. It is not a complete URDF, but it's a macro definition of the manipulator section. We can include this file inside another file and create an instance of this macro.
 - `irb2400.xacro`: This is the top level xacro file, which creates an instance of the macro, which is discussed in the preceding section. This file doesn't include any other files other than the macro of the robot. This xacro file will be loading inside the `load_irb2400.launch` file that we have already discussed.
 - `irb2400.urdf`: This is the URDF generated from the preceding xacro file using the `xacro` tool. This file is used when the tools or packages can't load xacro directly. This is the top-level URDF for this robot
- `meshes`: This contains meshes for visualization and collision checking
- `irb2400`: This folder contains mesh files for a specific robot
- `visual`: This folder contains STL files used for visualization
- `collision`: This folder contains STL files used for collision checking
- `tests`: The folder contains the test launch file to test all the preceding launch files
- `roslaunch_test.xml`: This launch file tests all the launch files.

Visualizing the ABB robot model in RViz

After creating the robot model, we can test it using the `test_irb2400.launch` file. The following command will launch the test interface of the ABB IRB 2400 robot:

```
$ rosrun abb_irb2400_support test_irb2400.launch
```

It will show the robot model in RViz with a joint state publisher node as shown in the following screenshot:

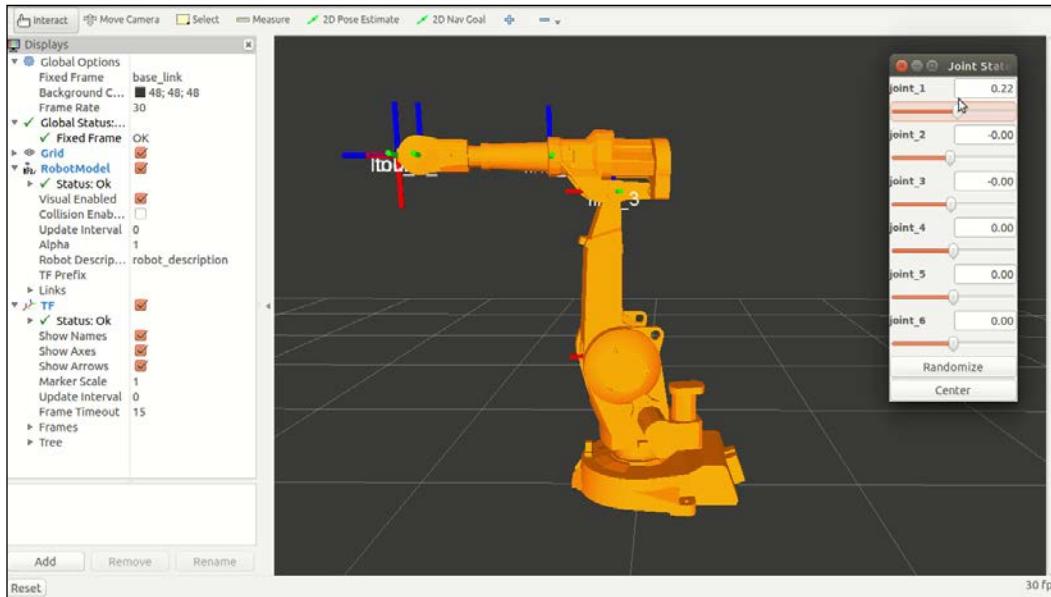


Figure 14: ABB IRB 2400 with joint state publisher on RViz

We can adjust the robot joints by adjusting the joint state publisher sliders' values. Using this testing interface, we can confirm whether the URDF design is correct or not.

ROS-Industrial robot client package

The industrial robot client nodes are responsible for sending robot position/trajectory data from ROS MoveIt! to the industrial robot controller. The industrial robot client converts the trajectory data to `simple_message` and communicates to the robot controller using the `simple_message` protocol. The industrial robot controller running a server and industrial robot client nodes are connecting to this server and start communicating with this server.

Designing industrial robot client nodes

The `industrial_robot_client` package contains various classes to implement industrial robot client nodes. The main functionalities that a client should have is, it can update the robot current state from the robot controller, and also it can send joint trajectories/joint position message to the controller.

There are two main nodes that are responsible for getting robot state and sending joint trajectory/position values.

- The `robot_state` node: This node is responsible for publishing the robot's current position, status, and so on
- The `joint_trajectory` node: This node subscribes the robot's command topic and sends the joint position commands to the robot controller via the simple message protocol

The following screenshot gives the list of APIs provided by the industrial robot client:

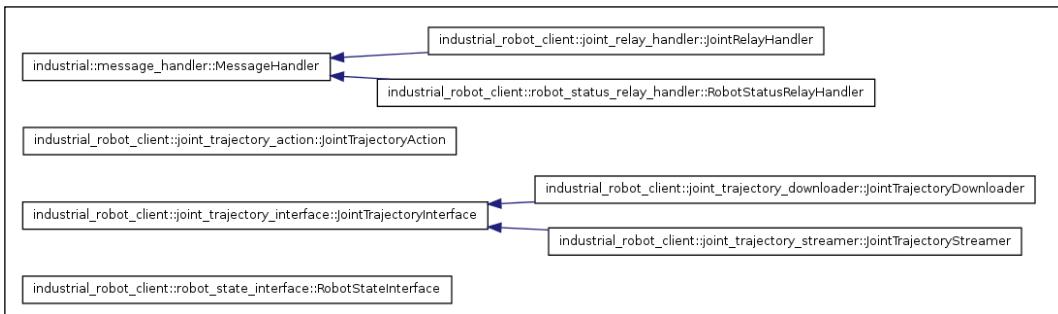


Figure 15: A list of the industrial robot client APIs

We can briefly go through these APIs and their functionalities as follows:

- `RobotStateInterface`: This class contains methods to publish the current robot position and status at regular intervals after receiving the position data from the robot controller.
- `JointRelayHandler`: The `RobotStateInterface` class is a wrapper around a class called `MessageManager`. What it does is, it listens to the `simple_message` robot connection and processes each message handling using `Messagehandlers`. The `JointRelayHandler` functionality is a `MessageHandler` and its function is to publish the joint position in the `joint_states` topic.
- `RobotStatusRelayHandler`: This is another `MessageHandler`, which can publish the current robot status info in the `robot_status` topic.

- `JointTrajectoryInterface`: This class contains methods to send the robot's joint position to the controller when it receives a ROS trajectory command.
- `JointTrajectoryDownloader`: This class is derived from the `JointTrajectoryInterface` class, and it implements a method called `send_to_robot()`. This method sends an entire trajectory as a sequence of messages to the robot controller. The robot controller will execute the trajectory in the robot only after getting all sequences sent from the client.
- `JointTrajectoryStreamer`: This class is the same as the preceding class except in the implementation of the `send_to_robot()` method. This method sends independent joint values to the controller in separate threads. Each position command is sent only after the execution of the existing command. In the robot side, there will be a small buffer for receiving the position to make the motion smoother.

The list of nodes inside the industrial robot client are as follows:

- `robot_state`: This node is running based on `RobotStateInterface`, which can publish the current robot states
- `motion_download_interface`: This node runs `JointTrajectoryDownloader`, which will download trajectory in sequence to the controller
- `motion_streaming_interface`: This node runs `JointTrajectoryStreamer`, which will send the joint position in parallel using threading
- `joint_trajectory_action`: This node provides a basic `actionlib` interface

ROS-Industrial robot driver package

In this section, we can discuss the industrial robot driver package. If we take the ABB robot as an example, it has a package called `abb_driver`. This package is responsible for communicating with the industrial robot controller. The package contains industrial robot clients and launches the file to start communicating with the controller.

We can check what's inside the `abb_driver/launch` folder. The following is a definition of a launch file called `robot_interface.launch`:

```
<launch>

<!-- robot_ip: IP-address of the robot's socket-messaging server -->
<arg name="robot_ip" />

<!-- J23_coupled: set TRUE to apply correction for J2/J3 parallel
linkage -->
```

```
<arg name="J23_coupled" default="false" />

<!-- copy the specified arguments to the Parameter Server, for use
by nodes below -->
<param name="robot_ip_address" type="str" value="$(arg robot_ip)"/>
<param name="J23_coupled" type="bool" value="$(arg J23_coupled)"/>

<!-- robot_state: publishes joint positions and robot-state data
      (from socket connection to robot) -->
<node pkg="abb_driver" type="robot_state" name="robot_state"/>

<!-- motion_download_interface: sends robot motion commands by
      DOWNLOADING path to robot
          (using socket connection to robot)
-->

<node pkg="abb_driver" type="motion_download_interface" name="motion_
download_interface"/>

<!-- joint_trajectory_action: provides actionlib interface for high-
level robot control -->
<node pkg="industrial_robot_client" type="joint_trajectory_action"
name="joint_trajectory_action"/>

</launch>
```

This launch file provides a socket-based connection to ABB robots using the standard ROS-Industrial `simple_message` protocol.

Several nodes are started to supply both low-level robot communication and high-level `actionlib` support:

- `robot_state`: This publishes the current joint positions and robot state data
- `motion_download_interface`: This commands the robot motion by sending motion points to the robot
- `joint_trajectory_action`: This is the `actionlib` interface to control the robot motion

Their usage is as follows:

```
robot_interface.launch robot_ip:=<value> [J23_coupled:=false]
```

We can see the `abb_irb6600_support/launch/ robot_interface_download_irb6640.launch` file and this is the driver for the ABB IRB 6640 model. This definition of launch is given in the following code. The preceding driver launch file is included in this launch file. In other support packages of other ABB models, use the same driver with different joint configuration parameter files:

```
<launch>
  <arg name="robot_ip" />
  <arg name="J23_coupled" default="true" />

  <rosparam command="load" file="$(find abb_irb2400_support)/config/
joint_names_irb2400.yaml" />

  <include file="$(find abb_driver)/launch/robot_interface.launch">
    <arg name="robot_ip" value="$(arg robot_ip)" />
    <arg name="J23_coupled" value="$(arg J23_coupled)" />
  </include>
</launch>
```

The preceding file is the manipulator-specific version of '`robot_interface.launch`' (of `abb_driver`).

- Defaults provided for IRB 2400: - `J23_coupled = true`
- Usage: `robot_interface_download_irb2400.launch robot_ip:=<value>`

We should run the driver launch file to start communicating with the real robot controller. For ABB robot IRB 2,400, we can use the following command to start bi-directional communication with the robot controller and the ROS client:

```
$ roslaunch abb_irb2400_support robot_interface_download_irb2400.launch
robot_ip:=<value>
```

After launching the driver, we can start planning using the MoveIt! interface. It should also be noted that the ABB robot should be configured and the IP of the robot controller should be found before starting the robot driver.

Understanding MoveIt! IKFast plugin

One of the default numerical IK solvers in ROS is KDL. KDL is mainly using $\text{DOF} > 6$. In robots $\text{DOF} \leq 6$, we can use analytic solvers, which is much faster than numerical solvers such as KDL. Most of the industrial arms are having $\text{DOF} \leq 6$, so it will be good if we make an analytical solver plugin for each arm.

The robot will work on the KDL solver too, but if we want fast IK solution, we can choose something such as the IKFast module to generate analytical solver-based plugins for MoveIt!. We can check which all are the IKFast plugin packages present in the robot, for example, universal robots and ABB.

- `ur_kinematics`: This package contains IKFast solver plugins of UR-5 and UR-10 robots from universal robotics
- `abb_irb2400_moveit_plugins/irb2400_kinematics`: This package contains IKFast solver plugins for the ABB robot model IRB 2400

We can go through the procedures to build an IKFast plugin for MoveIt!. It will be useful when we create an IK solver plugin for a custom industrial robotics arm. Let's see how to create a MoveIt! IKFast plugin for the industrial robot ABB-IRB6640.

Creating the MoveIt! IKFast plugin for the ABB-IRB6640 robot

We have seen the MoveIt! package for the ABB robot IRB 6640 model. But the robot is working using the KDL plugin, which is a default numerical solver. For generating IK solver plugin using IKFast, we can follow the procedure mentioned in this section. At the end of this section, we can run the MoveIt! demo of this robot using our custom `moveit-ikfast` plugin.

In short, we will build an IKFast MoveIt! plugin for robot ABB -IRB 66400. This plugin can be selected during the MoveIt! setup wizard or we can mention it in the `config/kinematics.yaml` file of the `moveit-config` package

Prerequisites for developing the MoveIt! IKFast plugin

The following is the configuration we have used for developing the MoveIt! IKFast plugin:

- Ubuntu 14.04.3 LTS x86_64 bit

- ROS-Indigo desktop-full, Version 1.11.13
- Open-Rave 0.9

OpenRave and IK Fast Module

OpenRave is a set of command line and GUI tools for developing, testing, and deploying motion planning algorithms in real-world applications. One of the OpenRave modules is **IKFast**, which is a robot kinematics compiler. OpenRave was created by a Robotic researcher called Rosen Diankov.

The IKFast compiler analytically solves the inverse kinematics of a robot and generates optimized and independent C++ files, which can be deployed in our code for solving IK. The IKFast compiler generates analytic solutions of IK, which is much faster than numerical solutions provided by KDL. The IK Fast compiler can handle any number of DOF, but practically it is well suited for `DOF <= 6`.

The IKFast is a Python script that takes arguments such as IK types, robot model, joint position of base link, and end effector.

The following are the main IK types supported by IKFast:

- `Transform6D`: This end effector should reach the commanded 6D transformation
- `Rotation 3D`: This end effector should reach the commanded 3D rotation
- `Translation 3D`: This end effector origin should reach the desired 3D translation

MoveIt! IK Fast

The `moveit-ikfast` ROS package contains tools to generate a kinematic solver plugin for MoveIt! using the OpenRave generated CPP file. We will use this tool to generate a IK Fast plugin for MoveIt!.

Installing MoveIt! IKFast package

The following command will install the `moveit-ikfast` package in ROS Indigo:

```
$ sudo apt-get install ros-indigo-moveit-ikfast
```

Installing OpenRave on Ubuntu 14.04.3

Installing OpenRave on the latest Ubuntu is a tedious task. We can install OpenRave from its repository or from the source itself. The repository installation has some issues so we have installed this application on Ubuntu 14.04.3 from the source code using the following procedure:

1. Clone the source code in the home folder. The file size is in the range of 300-400 MB.

```
$ git clone --branch latest_stable https://github.com/rdiankov/openrave.git
```

2. For compiling the source code, we need to install some packages:

- Installing boost, Python development packages and NumPy:

```
$ sudo apt-get install libboost-python-dev python python-dev  
python-numpy ipython
```

- Installing scientific Python and its package to handle symbolic mathematics:

```
$ sudo apt-get install python-scipy python-sympy
```

- Installing open asset import library to handle 3D file formats:

```
$ sudo apt-get install libassimp-dev assimp-utils python-pyassimp
```

3. Add the following lines on /etc/apt/source.list:

```
deb http://ppa.launchpad.net/openrave/testing/ubuntu trusty main  
deb-src http://ppa.launchpad.net/openrave/testing/ubuntu trusty main
```

4. Then, update the package list using the following command:

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

5. Install the following packages from the preceding repository using the following commands. It will install the collada file handling package and Qt4 GUI toolkit for the inventor app.

```
$ sudo apt-get install collada-dom2.4-dp*
```

```
$ sudo apt-get install libsoqt4-dev
```

Now, we'll see how to install cmake-gui for configuring and generating Makefiles from CMakeLists.txt.

The OpenRave project is based on CMake, so we need this tool for generating Makefiles.

```
$ sudo apt-get install cmake-qt-gui
```

Then, we'll perform the following steps:

1. The first procedure of installing OpenRave is to generate the UNIX Makefiles from `CMakeLists.txt` file.
2. Create a build folder inside the OpenRave cloned folder and open `cmake-gui` for configuring and building Makefiles.
3. Browse the source code and the build folder, as shown in the following screenshot, and after configuring uncheck support for Matlab and Octave interfaces:

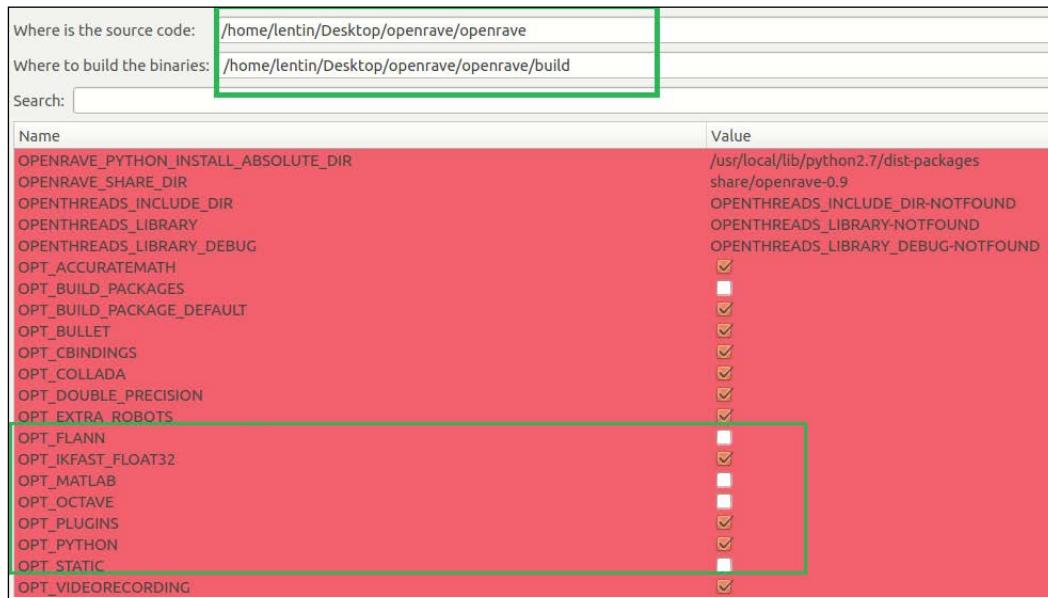


Figure 15: Configuring OpenRave with `cmake-gui`

4. Click on the **Generate** button to generate the Makefiles in the build folder.
5. Switch to the build folder and build the code and install using the following command:

```
$ make
$ sudo make install
```

6. After installing OpenRave, execute the following command to check OpenRave is working:

```
$ openrave
```

If everything works fine, it will open a 3D view port.

Creating the COLLADA file of a robot to work with OpenRave

In this section, we can discuss how to convert the robot URDF model to the collada file (.dae) format to work with OpenRave. There is a ROS package called `collada_urdf`, which contains nodes to convert URDF into collada files. The URDF file of ABB-IRB 6640 model is on `abb_irb6600_support/urdf` folder named `irb6640.urdf`. Copy this file into your working folder and run the following command for the conversion:

```
Start roscore  
$ roscore
```

Run the conversion command. We need to mention the URDF file and the output DAE file:

```
$ rosrun collada_urdf urdf_to_collada irb6640.urdf irb6640.dae
```

In most of the cases, this command fails because most of the URDF file contains STL meshes and it may not convert into DAE as we expected. If the robot meshes in, in DAE format, it will work fine. If it happens, follow this procedure:



- Install Meshlab tool for viewing and editing meshes using the following command:
`$ sudo apt-get install meshlab`
- Open meshes present at `abb_irb6600_support/meshes/irb6640/visual` in Meshlab and export the file into DAE with the same name.
- Edit the `irb6640.urdf` file and change the visual meshes in STL extension to DAE. This tool only process meshes for visual purpose only, so we will get a final DAE model.

We can open the `irb6640.dae` file using OpenRave with the following command:

```
$ openrave irb6640.dae
```

We will get the model in OpenRave as shown in the following screenshot:

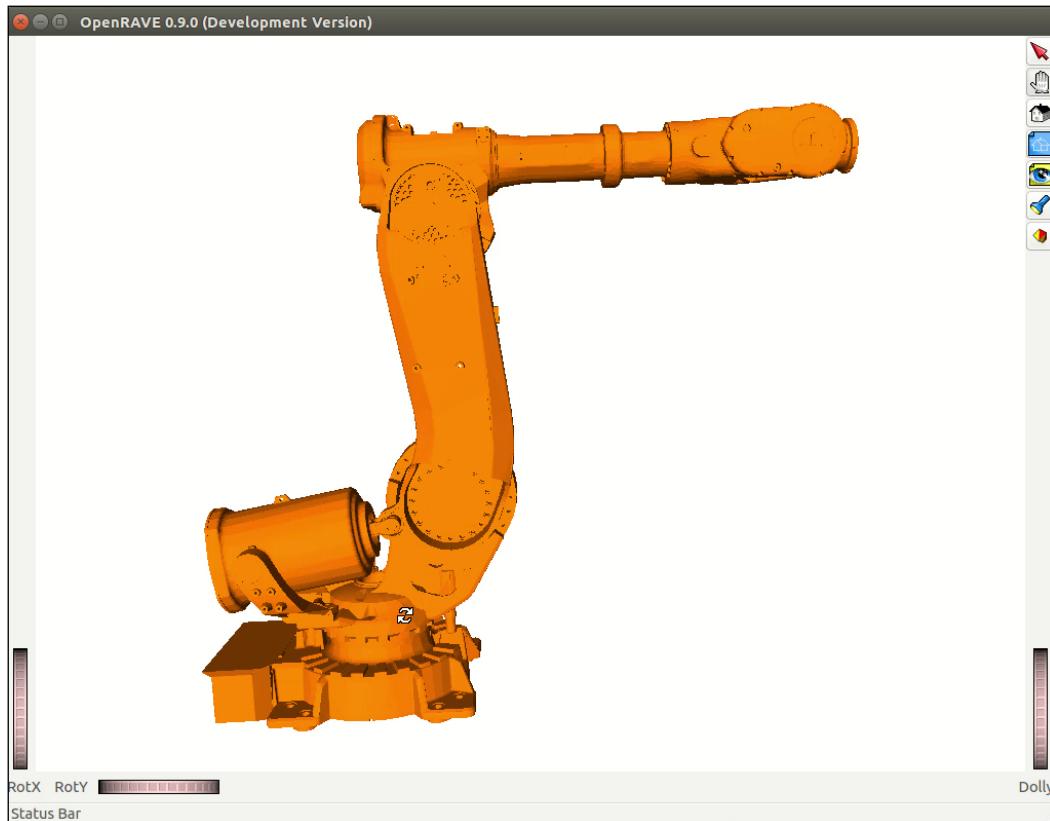


Figure 16: Viewing the ABB 6640 model on OpenRave

We can check the link information of the robot using the following command:

```
$ /usr/bin/openrave-robot.py irb6640.dae --info links
```

We can get link info about the robot in the following format:

```
name          index parents
-----
base_link      0
base           1      base_link
```

```
link_1      2      base_link
link_2      3      link_1
link_4      5      link_3
link_5      6      link_4
link_6      7      link_5
tool0       8      link_6
link_cylinder 9      link_1
link_piston 10     link_cylinder
-----
name          index parents
```

Generating the IKFast CPP file for the IRB 6640 robot

After getting the link info, we can start generating the IK solver CPP file for handling the IK of this robot.

Use the following command to generate the IK solver for the IRB 6640 robot:

```
$ python `openrave-config` --python-dir`/openravepy/_openravepy_ikfast.py --robot=irb6640.dae --iktype=transform6d --baselink=1 --ealink=8 --savefile=output_ikfast61.cpp
```

The preceding command generates a CPP file called `output_ikfast61.cpp` in which the IK type is `transform6d`, the position of the baselink is `1`, and the end effector link is `8`. We need to mention the robot DAE file as the robot argument.

We can test this file using the following procedure:

1. Download the IKFast demo code file from http://kaist-ros-pkg.googlecode.com/svn/trunk/arm_kinematics_tools/src/ikfastdemo/ikfastdemo.cpp.
2. Also, copy `IKFast.h` to the current folder. This file is present in the cloned file of OpenRave. We will get this header from `openrave/python`.
3. After getting `output_ikfast61.cpp`, `ikfastdemo.cpp`, and `ikfast.h` on the same folder, we need to edit `ikfastdemo.cpp` and change the following portion. Here, we are commenting a header, and instead of that, we add the CPP file that we have generated, that is `output_ikfast61.cpp`.

```
#define IK_VERSION 61
#include "output_ikfast61.cpp"
//#include "ikfast61.Transform6D.0_1_2_3_4_5.cpp"
```

4. Compile the edited file and check whether you are getting any errors. Here is the command to compile and execute this code:

```
$ g++ ikfastdemo.cpp -lstdc++ -llapack -o compute -lrt  
$ ./compute
```

If the demo is working, we can go to the next step. Now, we have successfully created the IK solver CPP file; the next step is to create a MoveIt! IK Fast plugin using this source code.

Creating the MoveIt! IKFast plugin

Creating a MoveIt! IKFast plugin is easy. There is no need to write code; everything can be generated using some tools. The only thing we need to do is to create an empty ROS package. The following are the procedures to create a plugin:

1. Switch to the `ros_industrial` workspace in the `src` folder:

```
$ cd ~/ros_industrial_ws/src
```
2. Create an empty package in which the name should contain the robot name and model number. This package is going to convert into the final plugin package using the plugin generation tool:

```
$ catkin_create_pkg abb_irb6640_moveit_plugins
```
3. Build the workspace using the `catkin_make` command.
4. After building the workspace, copy `ikfast.h` to `abb_irb6640_moveit_plugins/include`
5. Switch to the folder where we created the `output_ikfast61.cpp` file and the robot DAE file. Rename the `output_ikfast61.cpp` file to `abb_irb6640_manipulator_ikfast_solver.cpp`. This filename consists of robot name, model number, type of robot, and so on. This kind of naming is necessary for the generating tool.

After performing these steps, open two terminals in the current path where the IK solver CPP file exists. In one terminal, start the `roscore` command.

In the next terminal, we can enter the plugin creation command as follows:

```
$ rosrun moveit_ikfast create_ikfast_moveit_plugin.py abb_irb6640  
manipulator abb_irb6640_moveit_plugins  
abb_irb6640_manipulator_ikfast_solver.cpp
```

The `moveit_ikfast` ROS package consists of the `create_ikfast_moveit_plugin.py` script for the plugin generation. The first parameter is the robot name with the model number, the second argument is the type of robot, the third argument is the package name we have created earlier, and the fourth argument is the name of the IK solver CPP file. This tool needs the `abb_irb6640_moveit_config` package for its working. It will search this package using the given name of the robot. So, if the name of the robot is wrong, the tool for raising an error will say that it couldn't find the robot `moveit` package.

If the creation is successful, the messages in the following screenshot will be displayed:

```
lentin@lentin-Aspire-4755:~/ros_industrial_ws/src/abb_irb6640_moveit_plugins/src$ rosrun moveit_ikfast create_ikfast_moveit_plugin.py abb_irb6640 manipulator abb_irb6640 moveit_plugins abb_irb6640 manipulator_ikfast_solver.cpp
Warning: The default search has changed from OPTIMIZE_FREE_JOINT to now OPTIMIZE_MAX_JOINT!

IKFast Plugin Generator
Loading robot from 'abb_irb6640 moveit config' package ...
Creating plugin in 'abb_irb6640_moveit_plugins' package ...
  found 1 planning groups: manipulator
    found group 'manipulator'
    found source code generated by IKFast version 268435528

Created plugin file at '/home/lentin/ros_industrial_ws/src/abb_irb6640_moveit_plugins/src/abb_irb6640_manipulator_ikfast_moveit_plugin.cpp'

Created plugin definition at: '/home/lentin/ros_industrial_ws/src/abb_irb6640_moveit_plugins/abb_irb6640_manipulator_ikfast_plugin_description.xml'

Overwrote CMakeLists file at '/home/lentin/ros_industrial_ws/src/abb_irb6640_moveit_plugins/CMakeLists.txt'

Modified kinematics.yaml at /home/lentin/ros_industrial_ws/src/abb_irb6640_moveit_config/config/kinematics.yaml

Created update plugin script at /home/lentin/ros_industrial_ws/src/abb_irb6640_moveit_plugins/update_ikfast_plugin.sh
lentin@lentin-Aspire-4755:~/ros_industrial_ws/src/abb_irb6640_moveit_plugins/src$
```

Figure 17: Terminal messages of successful creation of IKFast plugin for MoveIt!

[ Note the possible errors that are discussed at https://github.com/ros-planning/moveit_ikfast/pull/48.]

Build `ros_industrial_ws` again, and we can see that a new plugin is building properly. If it is built, we can replace the default KDL IK solver in the `abb_irb6640_moveit_config/config/kinematics.yaml` file to the new solver as follows:

```
manipulator:
  kinematics_solver:
    abb_irb6640_manipulator_kinematics/IKFastKinematicsPlugin
  kinematics_solver_search_resolution: 0.005
  kinematics_solver_timeout: 0.005
  kinematics_solver_attempts: 3

#manipulator:
#  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
#  kinematics_solver_search_resolution: 0.005
```

```
# kinematics_solver_timeout: 0.005
# kinematics_solver_attempts: 3
```

After changing this kinematics solver, we can start working on the robot using the following command:

```
$ rosrun abb_irb6640_moveit_config demo.launch
```

We will get the planning window with a new IK solver as follows:

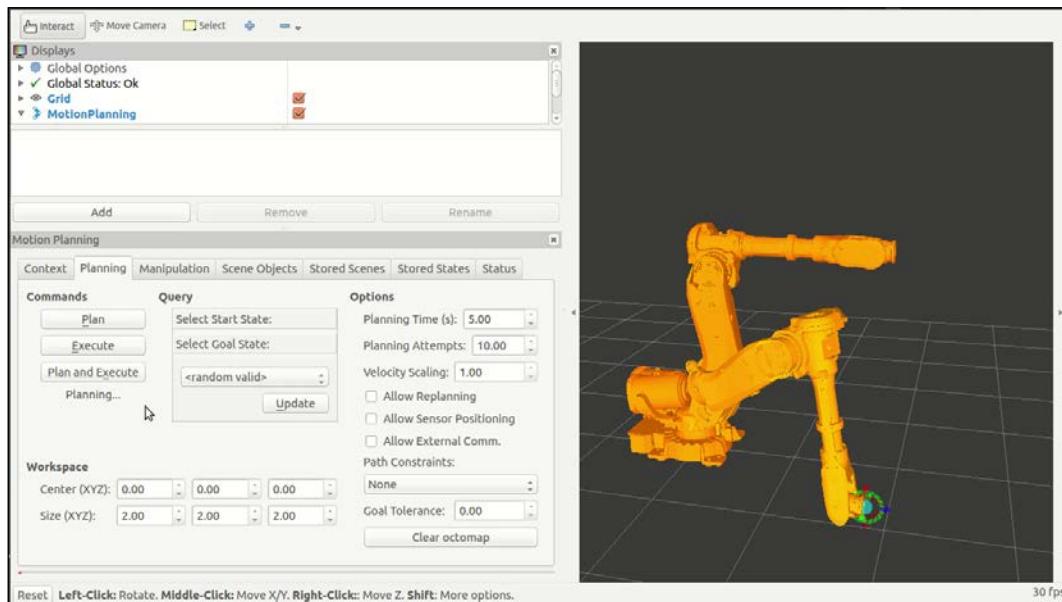


Figure 18: Motion planning of ABB 6640 using custom IKFast Plugin

Questions

Here are some common questions that will help you better learn and understand this chapter:

- What are the main benefits in using ROS-Industrial packages?
- What are the conventions followed by ROS-I in designing URDF for industrial robots?
- What is the purpose of ROS's support packages?
- What is the purpose of ROS's driver packages?
- Why we need an IKFast plugin for our industrial robot rather than the default KDL plugin?

Summary

In this chapter, we have been discussing a new interface of ROS for industrial robots called ROS-Industrial. We have seen the basic concepts in developing the industrial packages and installed it on Ubuntu. After installation, we have seen the block diagram of this stack and started discussing about developing the URDF model for industrial robots and also about creating the MoveIt! interface for an industrial robot. After discussing a lot on these topics, we have installed some industrial robot packages of universal robots and ABB. We have learned the structure of the MoveIt! package and then shifted to the ROS-Industrial support packages. We have discussed in detail and switched onto concepts such as the industrial robot client and about how to create MoveIt! IKFast plugin. In the end, we have used the developed plugin in the ABB robot.

In the next chapter, we look at the troubleshooting and best practices in ROS software development.

12

Troubleshooting and Best Practices in ROS

In the previous chapter, we discussed about ROS-Industrial and worked with motion planning of some industrial robots. In this chapter, we will discuss setting the ROS development environment in Eclipse IDE, best practices in ROS, and troubleshooting tips in ROS. This is the last chapter of this book, so before we start development in ROS, it will be good if we know the standard methods for writing code using ROS. Following are the topics that we are going to discuss in this chapter:

- Setting the ROS development environment in Eclipse IDE
- Best practices in ROS
- Best coding practices in ROS using C++
- Important troubleshooting tips in ROS

Before start coding in ROS, it will be good if we set ROS development environment in an IDE. Setting an IDE for ROS is not mandatory but it can save developer time. IDEs can provide auto completion features that can make programming easy. We can use any editors such as Sublime and VIM or simply gedit for coding in ROS. It will be good if you choose IDEs when you are planning a big project in ROS.

In this chapter, we will demonstrate how to set up the ROS development environment in Eclipse IDE. Let's see how to download, install, and the setting of ROS on the latest Eclipse IDE on Ubuntu 14.04.3.

Setting up Eclipse IDE on Ubuntu 14.04.3

Eclipse needs **Java Runtime Environment (JRE)** in order to work. The following command can install JRE in Ubuntu:

```
$ sudo apt-get install default-jre
```

The first step is to download the latest eclipse IDE. We can get the latest version of Eclipse at <https://www.eclipse.org/downloads/?osType=linux>.

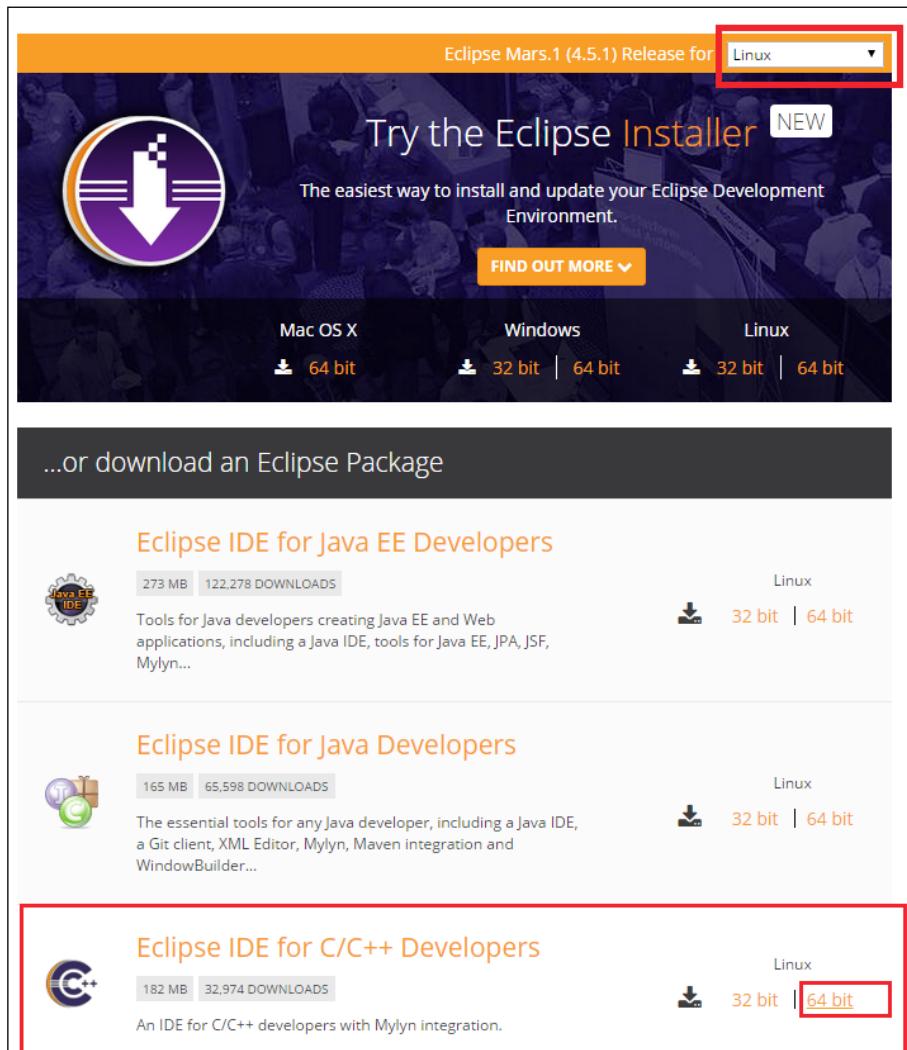


Figure 1: Eclipse IDE download page

Download and extract the **Eclipse IDE for C/C++ Developers** that is marked on the preceding image. Extract the Eclipse archive file using the following command. Here we are using Eclipse mars for Linux 64 bit:

```
$ tar -xvzf eclipse-cpp-<name_version>-linux-gtk-x86_64.tar.gz
```

We will get a folder called `eclipse` after extraction. Copy the `eclipse` folder to the `/opt` folder using the following command:

```
$ sudo cp -r eclipse /opt/
```

Create a desktop file for the `eclipse` for accessing from the Ubuntu search bar:

```
$ sudo nano /usr/share/applications/eclipse.desktop
```

Copy and paste the following content to this file. This file consists of the location of the `eclipse` executable and its icon:

```
[Desktop Entry]
Version=4.4.1
Name=Eclipse Mars Java EE
GenericName=IDE
Comment=Eclipse IDE for Java C++ Developers
Exec=/opt/eclipse/eclipse
Terminal=false
Icon=/opt/eclipse/icon.xpm
Type=Application
Categories=Utility;Application;
```

After saving this file, you can access Eclipse from the search bar itself.

Setting ROS development environment in Eclipse IDE

In this section, we can see the necessary settings that we need to do for compiling ROS C++ nodes in Eclipse. There are several methods available to configure ROS development environment in Eclipse. We are going to see one of the tested methods that is used to set the ROS environment.

Global settings in Eclipse IDE

Following are the global settings that we have to do in Eclipse IDE. We don't need to do these settings for each project. These are only one-time settings.

- Launch Eclipse IDE from the Ubuntu search bar.
- Go to **Window | Preferences**. from the **Preferences** Window, choose **C/C++ | Build | Settings** and then choose the **Discovery** tab. Select **CDT GCC Build Output Parser [Shared]**. Select the **Compiler command pattern** to `(.*gcc)|(.*[gc]\+\+)|(.*clang)`. Also check the **Project** option that is a part of **Container to keep discovered entries**. Click on the **Apply** button and then on the **OK** button to confirm the settings. These settings enable eclipse to find C++ 11 traits inside the package. The settings are shown in the following screenshot:

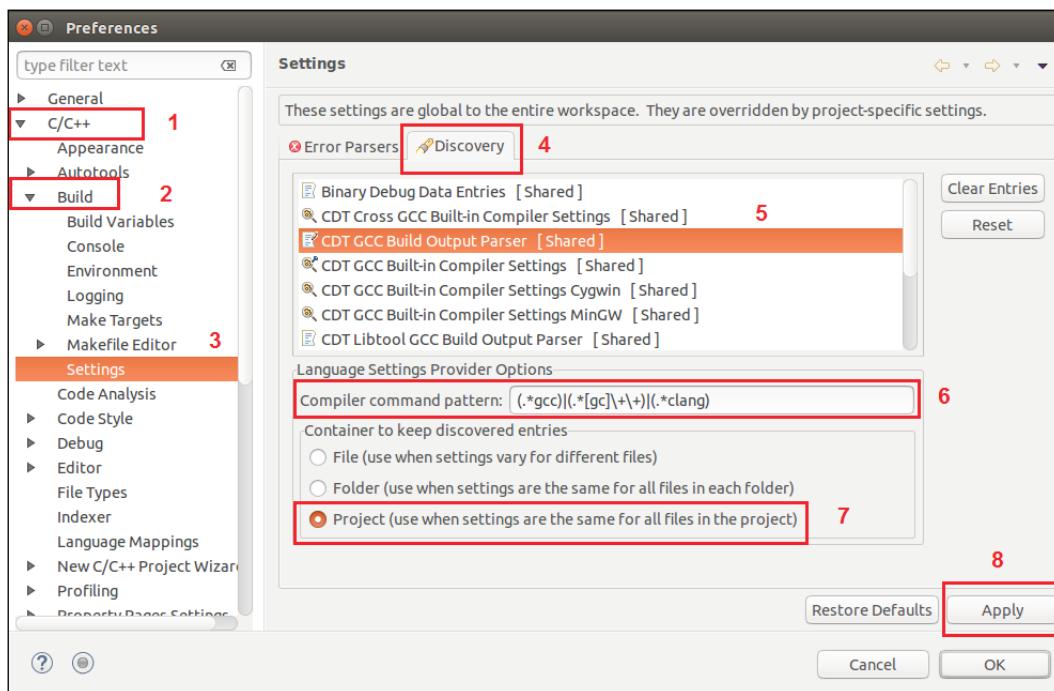


Figure 2: Settings inside Eclipse Preferences

- In the next step, click on the **CDT GCC Built-in Compiler Settings [Shared]** option from the **Discovery** tab and change the entry under **Command to get compiler specs** to `$(COMMAND) -E -P -v -dD -std=c++11 "$(INPUTS)"`.

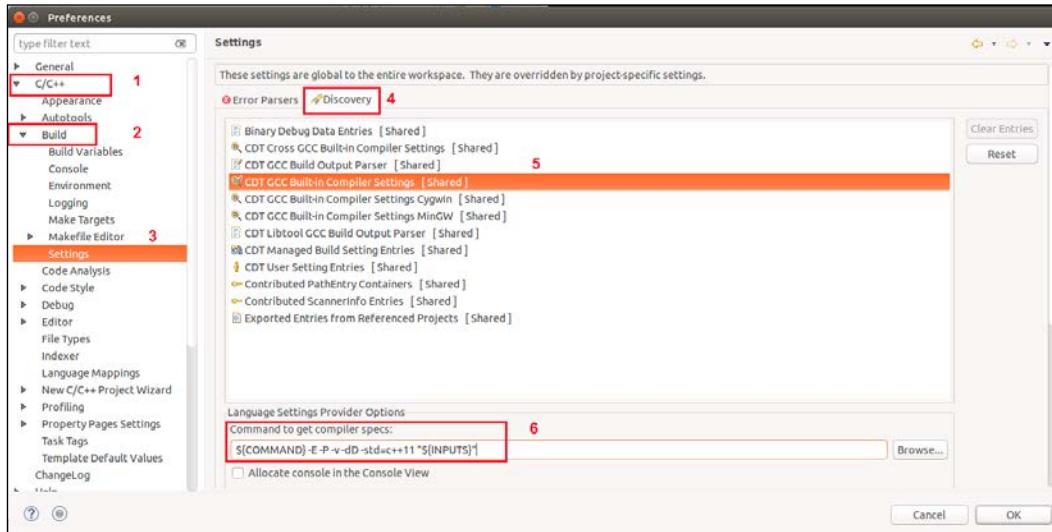


Figure 3: Eclipse Compiler settings

ROS compile script for Eclipse IDE

Compiling ROS nodes needs the ROS environment. We have to source `/opt/ros/indigo/setup.bash` to access the ROS environment in the current terminal. We can work from the system terminal because we already added this line to the `.bashrc` file, but when we work using Eclipse, we have to make a script to do this.

Create a file called `eclipsemake` in `/usr/local/bin` using the following command:

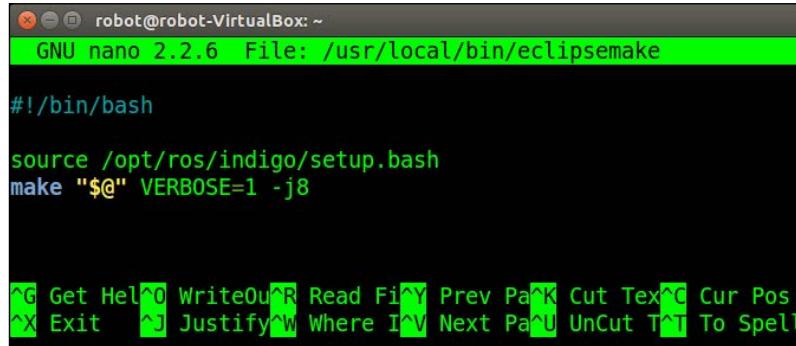
```
$ sudo nano /usr/local/bin/eclipsemake
```

Enter the following commands in this file:

```
#!/bin/bash

source /opt/ros/indigo/setup.bash

make "$@" VERBOSE=1 -j8
```



```
robot@robot-VirtualBox: ~
GNU nano 2.2.6  File: /usr/local/bin/eclipsemake

#!/bin/bash

source /opt/ros/indigo/setup.bash
make "$@" VERBOSE=1 -j8

^G Get Help^O WriteOut^R Read File^Y Prev Pag^K Cut Tex^C Cur Pos
^X Exit    ^J Justify^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

Figure 4: Script to source the ROS environment

Create another file called `eclipsemake-tests` for testing the purpose on the same path. Create a file using the following command:

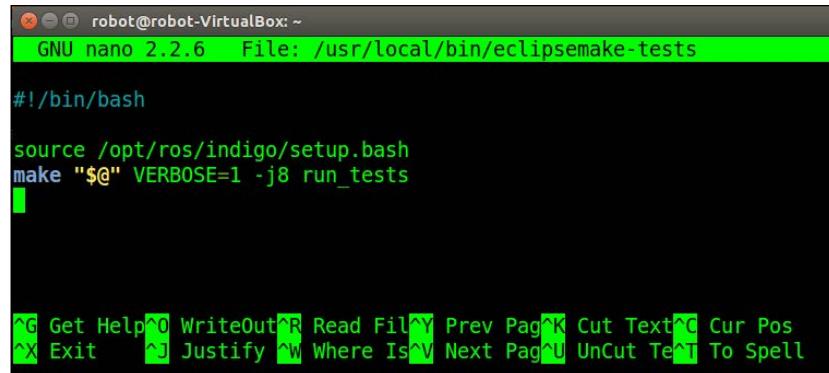
```
$ sudo nano /usr/local/bin/eclipsemake-tests
```

Enter the following content into the file:

```
#!/bin/bash

source /opt/ros/indigo/setup.bash

make "$@" VERBOSE=1 -j8 run_tests
```



```
robot@robot-VirtualBox: ~
GNU nano 2.2.6  File: /usr/local/bin/eclipsemake-tests

#!/bin/bash

source /opt/ros/indigo/setup.bash
make "$@" VERBOSE=1 -j8 run_tests

^G Get Help^O WriteOut^R Read File^Y Prev Pag^K Cut Tex^C Cur Pos
^X Exit    ^J Justify^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

Figure 5: Script to source the ROS environment and running test

If you are using Eclipse in Virtual Box, use `-j1` instead of `-j8`.

After creating these two files, change the permission of these two using the following command:

```
$ sudo chmod +x /usr/local/bin/eclipsemake*
```

Adding ROS Catkin package to Eclipse

After doing the preceding configuration, we can start adding ROS packages to Eclipse IDE. Click on **File Menu | Project...** from the **New Project** wizard, select **C/C++ | Makefile Project with Existing Code**:

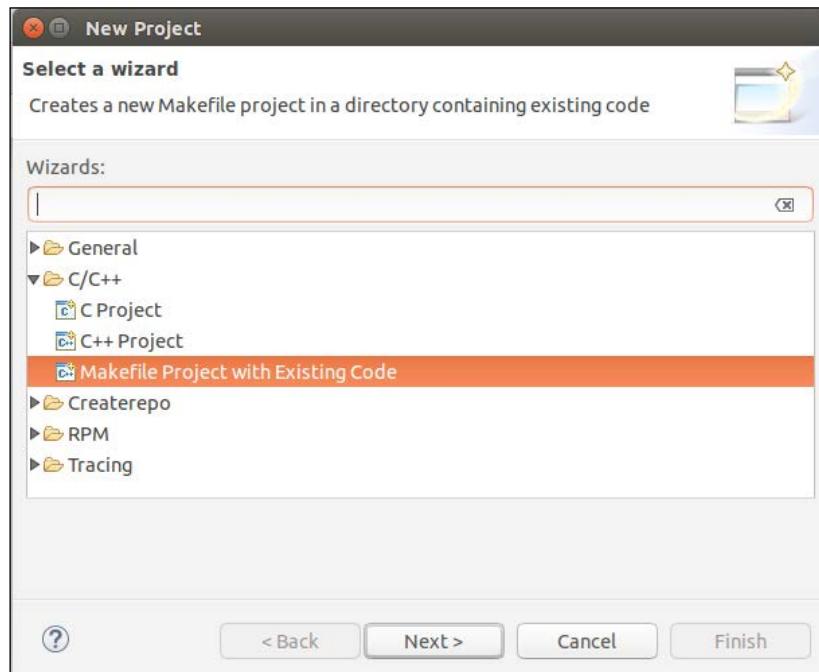


Figure 6: Open the ROS package from the catkin workspace

There was a `hello_world` ROS package in `ros_catkin_ws`; we are opening this project. This package consists of two ROS nodes, `talker.cpp` and `listener.cpp`. You can open any packages on your workspace.

Give a name for this project as `hello_world` and browse the ROS package from the catkin workspace as shown in the following screenshot:

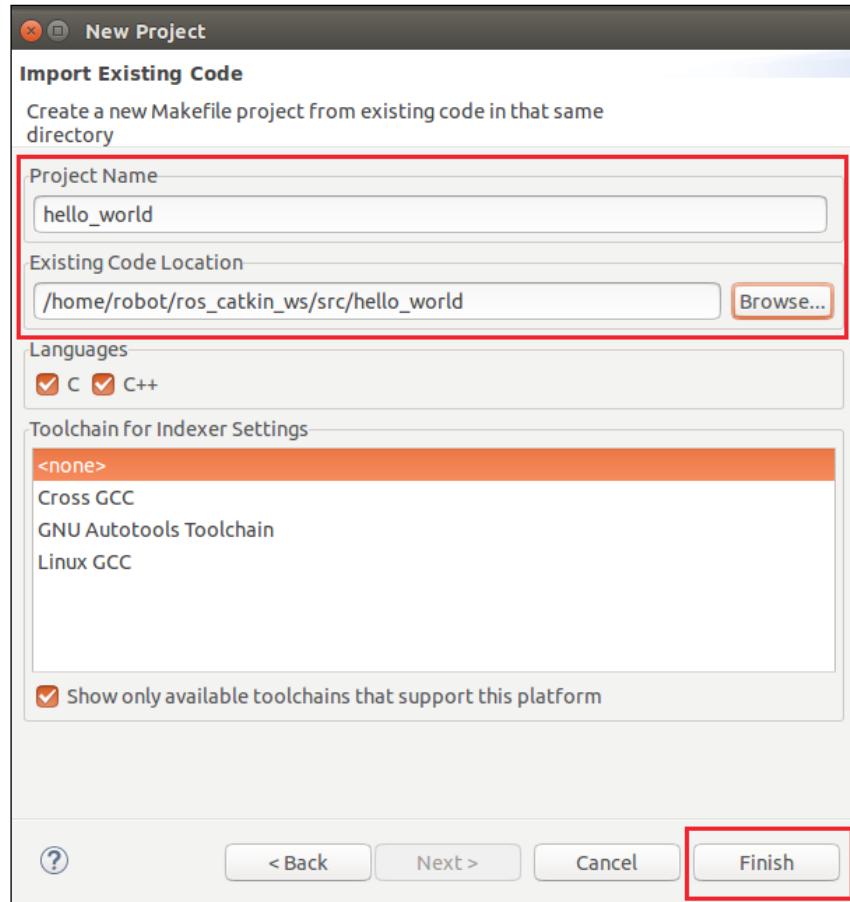


Figure 7: Giving project and location

After opening the project, right-click on the project and go to **Properties** of the project. From the **Properties**, click on the **C/C++ Build** option, and from the **Builder Settings** tab, change the **Build command** to the custom command called `eclipsemake`. Browse **Build location** of the ROS package by clicking on the **File System** button, as shown in the following screenshot:

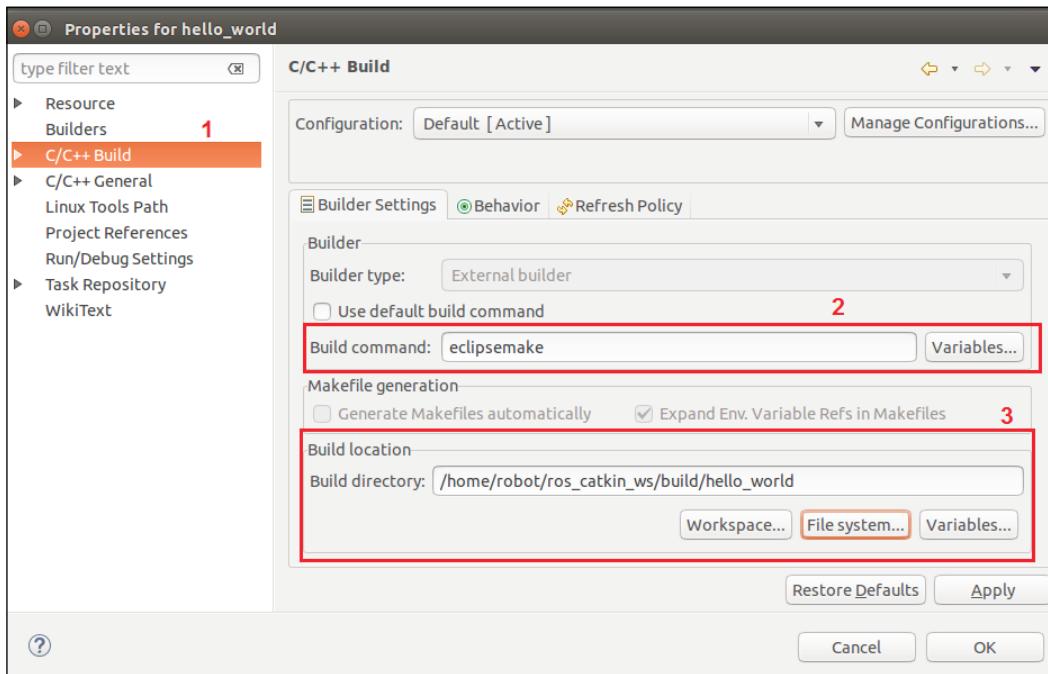


Figure 8: Eclipse build settings for ROS

In **C/C++ Build | Environment**, add a new variable called `VERBOSE` and set the value as `1`, as shown in the following screenshot:

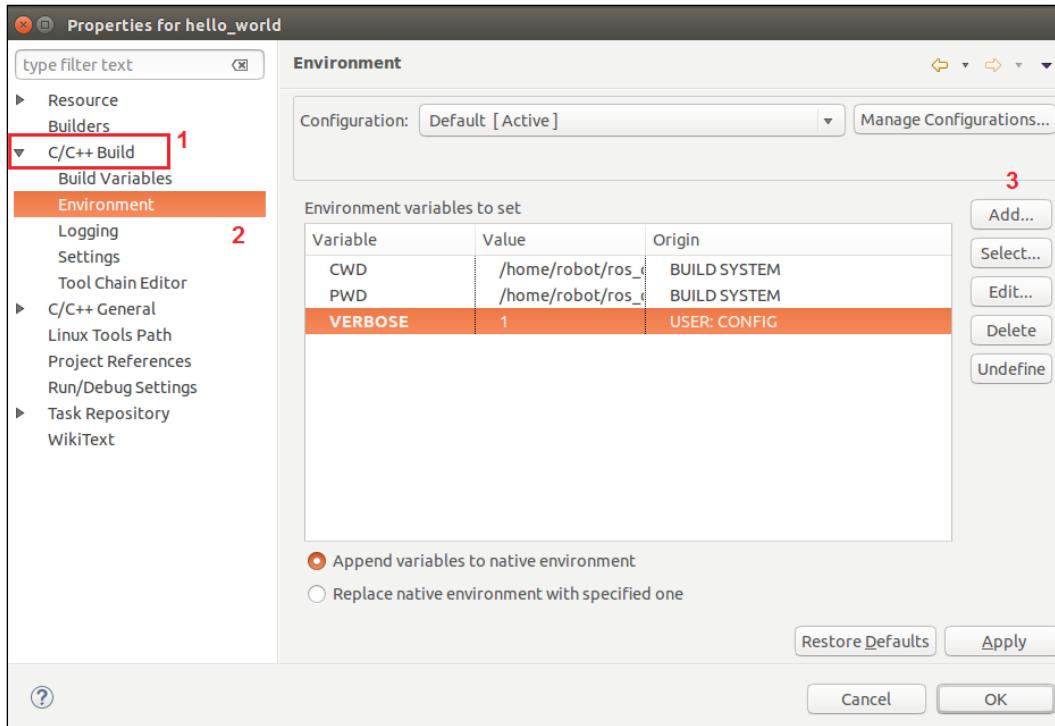


Figure 9: Setting `VERBOSE` in ROS project properties

In **C/C++ General**, select **Path and Symbols**, choose the **Symbols** tab, and add a symbol called `_GXX_EXPERIMENTAL_CXX0X_` in **GNU C++** with no values. Click on **Apply | OK** to confirm the settings, as shown in the following screenshot:

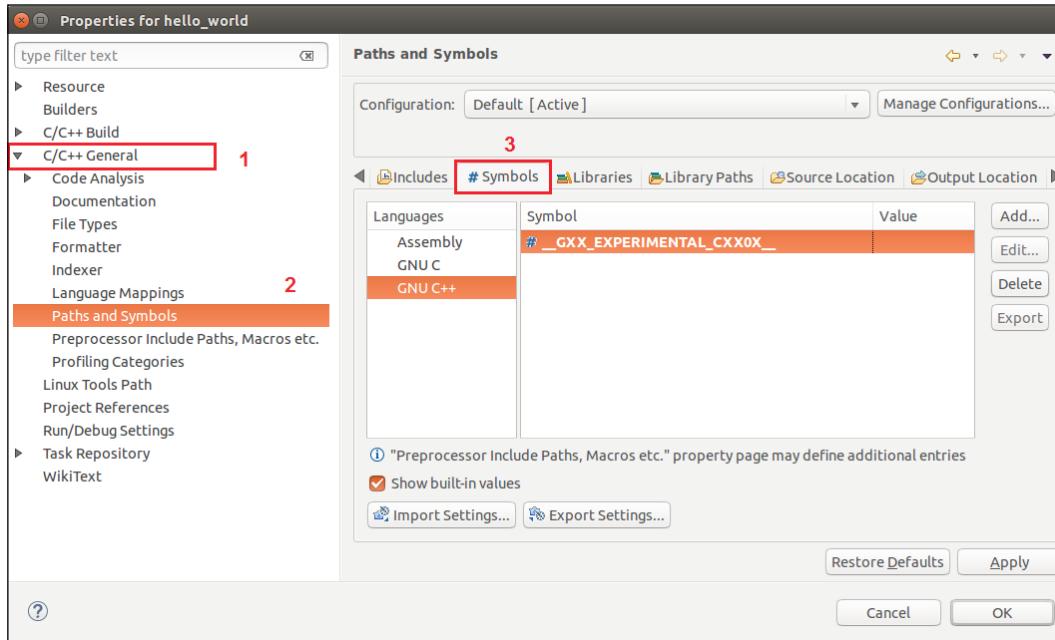


Figure 10: Setting path and symbols for the ROS project

In **C/C++ General**, choose **Preprocessor Includes Paths, Macros etc.** from the **Providers** tab, check the options **CDT GCC Build Output Parser [Shared]** and **GCC Built-in Compiler Settings [Shared]**. We should also verify the **Use global provider shared between projects** option in both. Click on **Apply** and then click on **OK**, as shown in the following screenshot:

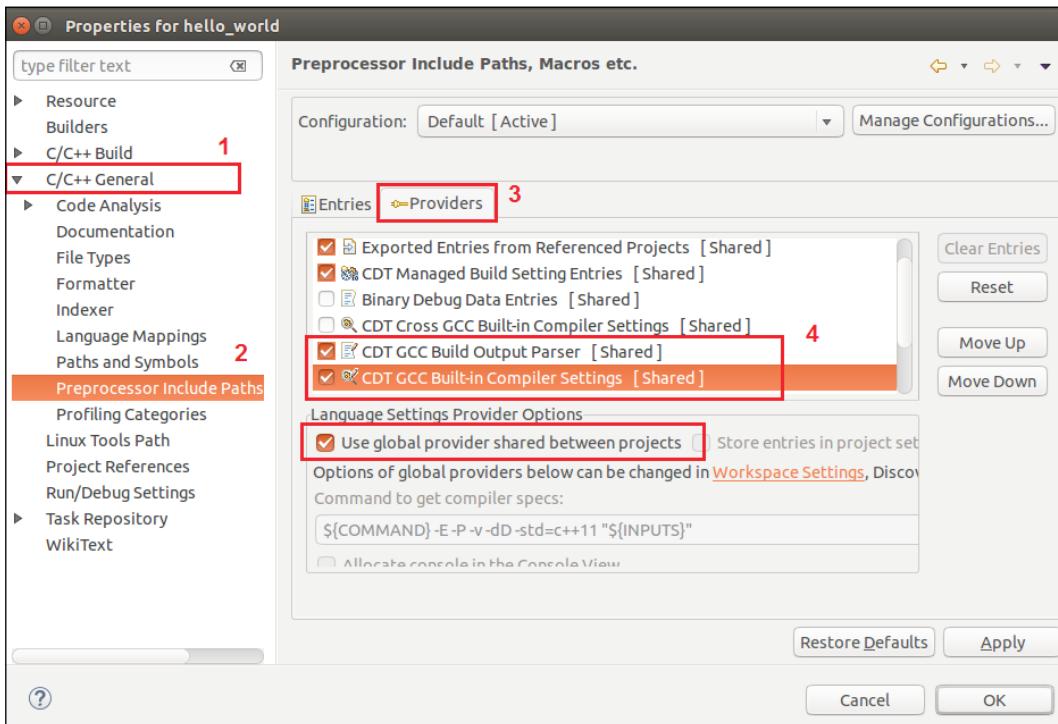


Figure 11: Setting Pre-processor in the project properties

After doing these all settings, we should clean the project by right-clicking on the **Project | Clean Project**. After cleaning, build the project (**Ctrl+B**).

Adding run configurations to run ROS nodes in Eclipse

After building the project, we may can run the node from Eclipse or from a terminal. For running the node inside Eclipse, right-click on the project and go to **Run as | Run Configurations**.

Create a **New Launch Configuration** under **C/C++ Application**. In the **Main** tab, browse the executable path in **C/C++ Application**. While we build the nodes in Eclipse, we can see the executable generating path. Browse the path of the executables here.

Here we are creating a launcher for the `talker` node, as shown in the following screenshot:

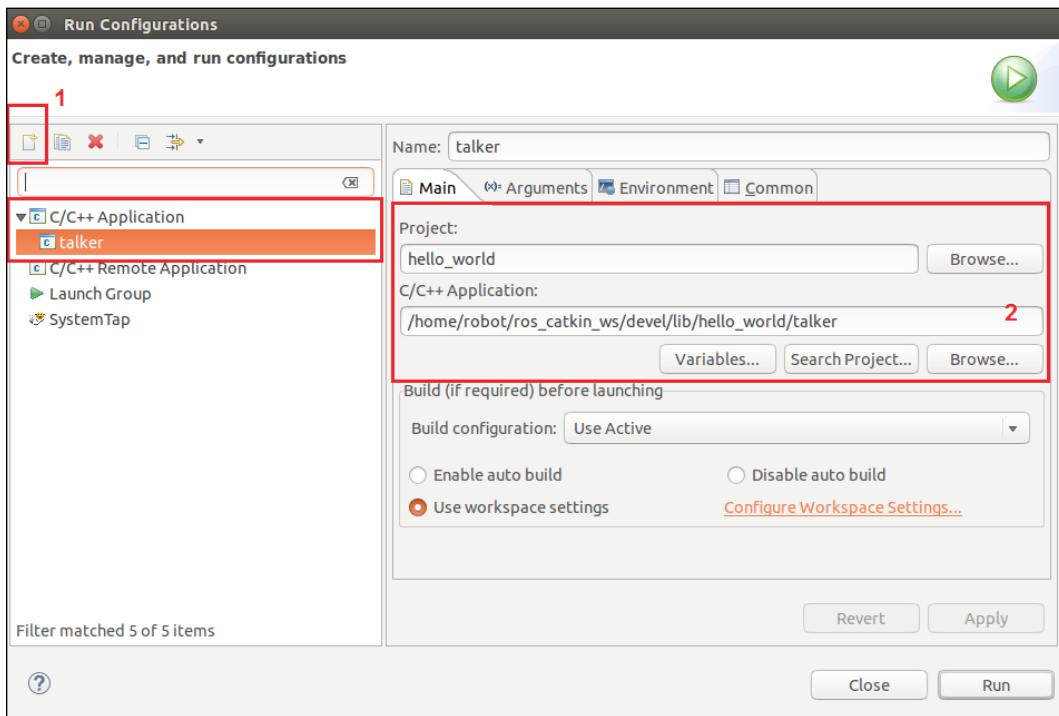


Figure 12: Creating the launcher for the talker node

After the preceding settings, click on the **Environment** tab and insert two variables:

```
ROS_MASTER_URI : http://localhost:11311  
ROS_ROOT : /opt/ros/indigo/share/ros
```

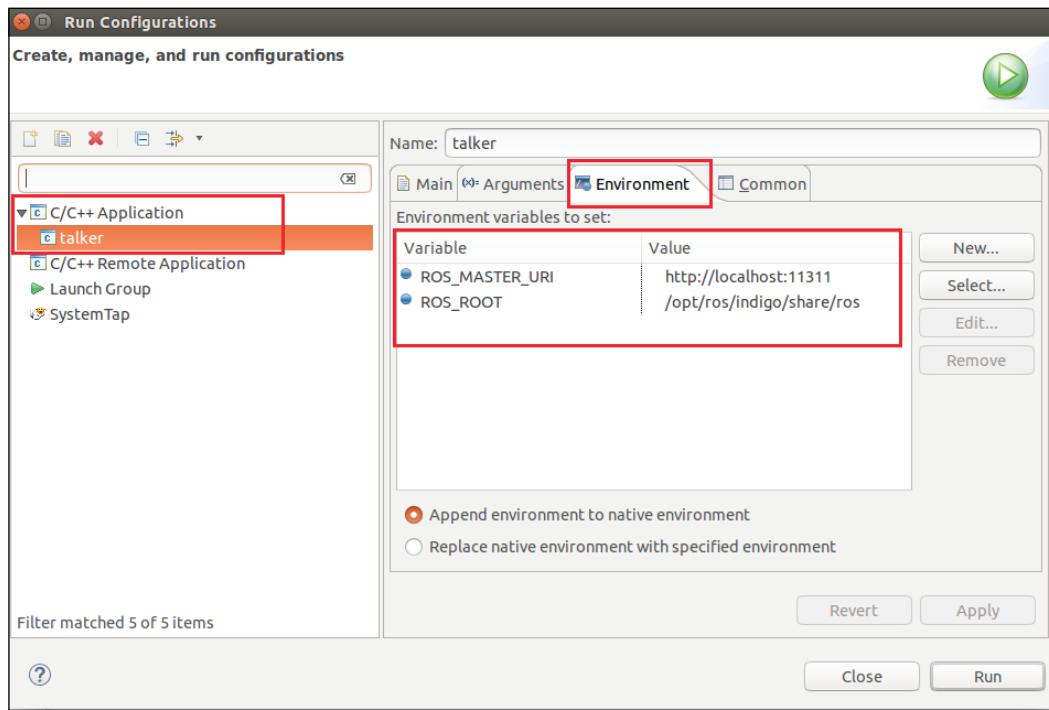


Figure 13: Setting the ROS environment variable inside the launcher configuration

After doing these setting, we can run the `talker` node by performing the following steps:

- Start `roscore` in one terminal.
- Start the `talker` node by pressing the **Run** key on the Eclipse, as shown in the following screenshot:

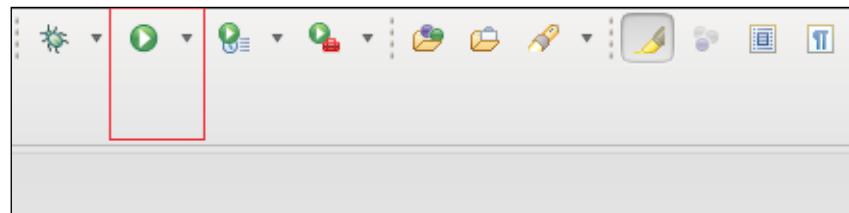
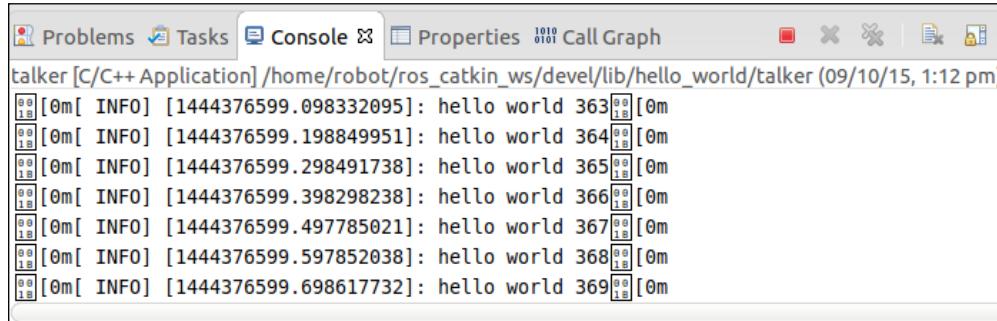


Figure 14: Launching the talker node

We can see the output in the Eclipse console, as shown in the following screenshot:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The title bar indicates the project is 'talker [C/C++ Application]' located at '/home/robot/ros_catkin_ws/devel/lib/hello_world/talker'. The timestamp is '09/10/15, 1:12 pm'. The console window displays multiple INFO messages from the 'hello_world' node, each containing the text 'hello world' followed by a timestamp and a sequence number (e.g., 363, 364, 365, 366, 367, 368, 369). The timestamps range from 1444376599.098332095 to 1444376599.698617732.

```
[0m[ INFO] [1444376599.098332095]: hello world 363[0m
[0m[ INFO] [1444376599.198849951]: hello world 364[0m
[0m[ INFO] [1444376599.298491738]: hello world 365[0m
[0m[ INFO] [1444376599.398298238]: hello world 366[0m
[0m[ INFO] [1444376599.497785021]: hello world 367[0m
[0m[ INFO] [1444376599.597852038]: hello world 368[0m
[0m[ INFO] [1444376599.698617732]: hello world 369[0m
```

Figure 15: Talker node running on the Eclipse terminal

In the next section, we will look at some of the best practices that should be followed when working with ROS.

Best practices in ROS

This section gives you a brief idea of the best practices that can be followed when we develop something with ROS. ROS provides detailed tutorials about its QA (Quality Assurance) process. QA process provides a detailed developers guide which mentions C++ and Python code style guides, naming conventions, and so on.

First we can discuss the ROS C++ coding styles.

ROS C++ coding style guide

ROS C++ nodes are following a coding style to make the code more readable, debuggable, and maintainable. If the code is properly styled, it will be very easy to re-use and contribute to the current code. In this section, we can quickly go through some commonly used coding styles.

Standard naming conventions used in ROS

Here we are using the text `Helloworld` to demonstrate the naming patterns we are using in ROS:

- **HelloWorld**: This name starts with an uppercase letter, and each new word starts with an uppercase letter with no space or underscores.
- **helloWorld**: In this naming method, the first letter will be lowercase but new words will be in uppercase letters without spaces.

- **hello_world:** This only contains lowercase letters. Words are separated with underscores.
- **HELLO_WORLD:** All are uppercase letters. Words are separated by an underscore.

The following are the naming conventions followed by each component in ROS:

- **Packages, Topics/Services, Files, Libraries:** These ROS components are following the `hello_world` pattern.
- **Classes/Types:** These classes are following the `HelloWorld` kind of naming conventions, for example, class `ExampleClass`.
- **Functions/Methods:** Functions follow `helloWorld` naming conventions and function arguments are following the `hello_world` pattern, for example, `void exampleMethod(int sample_arg);`.
- **Variables:** Generally, variables follow the `hello_world` pattern.
- **Constants:** Constants follow the `HELLO_WORLD` pattern.
- **Member variables:** The member variable inside a class follows the `hello_world` pattern, with a trailing underscore added, for example, `int sample_int_`.
- **Global variables:** Global variables follow `hello_world`, with a leading `g_`, for example, `int g_samplevar;`.
- **Namespace:** This follows the `hello_world` naming pattern.

Code license agreement

We should add a license statement on the top of code. ROS is an open source software framework and it's in the BSD license. The following is a code snippet of `LICENSE`, which has to be inserted on the top of the code. You will get the license agreement from any of the ROS nodes from the main repository. You can check the source code from the following ROS tutorial at https://github.com/ros/ros_tutorials.

```
*****
 * Software License Agreement (BSD License)
 *
 * Copyright (c) 2012, Willow Garage, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
```

```
* are met:  
*****
```

```
/* Author: Lentin Joseph */
```

For more information about various licensing schemes in ROS, refer to
<http://wiki.ros.org/DevelopersGuide#Licensing>.

ROS code formatting

One thing that needs to be taken care of while developing code is its formatting. One of the basic things in formatting is that each code blocks in ROS separated by two spaces. Given in the following is a code snippet showing the formatting:

```
if(a < b)  
{  
    // do stuff  
}  
else  
{  
    // do other stuff  
}
```

Given in the following is an example code snippet in the ROS standard formatting style:

```
#include <boost/tokenizer.hpp>  
#include <moveit/macros/console_colors.h>  
#include <moveit/move_group/node_name.h>  
  
static const std::string ROBOT_DESCRIPTION = "robot_description";  
// name of the robot description (a param name, so it can be changed  
externally)  
  
namespace move_group  
{  
  
class MoveGroupExe  
{  
public:  
  
    MoveGroupExe(const planning_scene_monitor::PlanningSceneMonitorPtr&  
    psm, bool debug) :
```

```
node_handle_("~")
{
    // if the user wants to be able to disable execution of paths,
    // they can just set this ROS param to false
    bool allow_trajectory_execution;
    node_handle_.param("allow_trajectory_execution", allow_trajectory_
execution, true);

    context_.reset(new MoveGroupContext(psm, allow_trajectory_
execution, debug));

    // start the capabilities
    configureCapabilities();
}

~MoveGroupExe()
{
```

ROS code documentation

The developer should be documented inside the code and should provide API documentation using tools such as Doxygen (www.doxygen.org/). The following is the method to generate documentation using Doxygen for a ROS package:

<http://wiki.ros.org/PackageDocumentation>

Console output

Avoid `printf` / `cout` statements for printing debug messages inside ROS nodes. We can use `rosconsole` (<http://wiki.ros.org/rosconsole>) for debugging, which provides five verbosity levels.

For detailed coding styles, refer to <http://wiki.ros.org/CppStyleGuide>.

Best practices in the ROS package

Following are the key points while creating and maintaining a package:

- **Version Control:** ROS supports version control using Git, Mercurial, and Subversion. We can host our code in GitHub and Bit bucket. Most of the ROS packages are in GitHub.

- **Packaging:** Inside a ROS catkin package, there will be a `package.xml`, and this file should contain the author name, description, and license.

The following is an example of a `package.xml`:

```
<?xml version="1.0"?>
<package>
    <name>roscpp_tutorials</name>

    <version>0.6.1</version>

    <description>
        This package attempts to show the features of ROS step-by-
        step,
        including using messages, servers, parameters, etc.
    </description>

    <maintainer email="dthomas@osrfoundation.org">Dirk Thomas</
    maintainer>

    <license>BSD</license>

    <url type="website">http://www.ros.org/wiki/roscpp\_tutorials</
    url>
        <url type="bugtracker">https://github.com/ros/ros\_tutorials/
        issues</url>
        <url type="repository">https://github.com/ros/ros\_tutorials</
        url>
    <author>Morgan Quigley</author>
```

Important troubleshooting tips in ROS

We will look at some of the common issues when working with ROS as well as tips to solve them.

One of the ROS inbuilt tools to find issues in a ROS system is `rosout`. `rosout`, which checks issues in following areas of ROS:

- Environment variables and configuration issues
- It can scan a package or meta-package to report potential issues
- It can check a launch file for its potential issues
- It can check system issues and online graph issues
- It can report warnings and errors – warnings can be avoided but are not necessary, errors should be addressed

Usage of rosdep

We can check the issues inside a ROS package by simply entering the package and entering rosdep. We can also check issues in the launch file using the following command:

```
$ rosdep <file_name>.launch
```

We may get a report if there are issues associated with the package.

```
Static checks summary:  
Found 1 error(s).  
  
ERROR Not all paths in PYTHONPATH [/home/robot/ros_industrial_ws/devel/lib/python2.7/dist-packages:/home/robot/ros_catkin_ws/devel/lib/python2.7/dist-packages:/home/robot/cool_arm_ws/devel/lib/python2.7/dist-packages:/home/robot/catkin_ws/devel/lib/python2.7/dist-packages:/opt/ros/indigo/lib/python2.7/dist-packages] point to a directory:  
* /home/robot/ros_catkin_ws/devel/lib/python2.7/dist-packages
```

Figure 16: rosdep command output for a ROS package

The wiki page of rosdep is available at <http://wiki.ros.org/rosdep>.

The following are some of the common issues faced when working with ROS:

- **Issue 1:**

Error message: Failed to contact master at [localhost:11311]. Retrying...

```
robot@robot-VirtualBox:~$ roslaunch ros_tutorials talker  
[ERROR] [1444418189.264516006]: [registerPublisher] Failed to contact master at [localhost:11311]. Retrying...
```

Figure 17: Failed to contact master error message

Solution: This message comes when the ROS node executes without running the roscore command.

- **Issue 2:**

Error message: **Could not process inbound connection: topic types do not match**

```
robot@robot-VirtualBox:~$ rostopic pub /chatter std_msgs/Int32 1
publishing and latching message. Press ctrl-C to terminate
[WARN] [WallTime: 1444419579.855744] Could not process inbound connection: topic types do not match: [std_msgs/String] vs. [std_msgs/Int32]{'topic': '/chatter', 'tcp_nodelay': '0', 'md5sum': '992ce8a1687cec8c8b
d883ec73ca41d1', 'type': 'std_msgs/String', 'callerid': '/listener'}
```

Figure 18: Inbound connection warning messages

Solution: This happens when there is a topic message mismatch, when we publish and subscribe a topic with different ROS message type.

- **Issue 3:**

Error message: **Couldn't find executables**

```
lentin@lentin-Aspire-4755:~/ros_catkin_ws$ rosrun hello_world talker
[rosrun] Couldn't find executable named talker below /home/lentin/ros_catkin_ws/src/hello_world
lentin@lentin-Aspire-4755:~/ros_catkin_ws$
```

Figure 19: Couldn't find executables

Solution: One of the reasons for this error is if we are not including `catkin_package()` inside `CMakeLists.txt`. In this situation, the executable will not build on the expected location, so `rosrun` will not find the executable. We can generate this error by commenting this line in `CMakeLists.txt`, as shown in the following:

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_world)

find_package(catkin REQUIRED COMPONENTS
    roscpp
    rospy
    std_msgs
)
#catkin_package()

include_directories(
    ${catkin_INCLUDE_DIRS}
)
```

Figure 20: `CMakeLists.txt` without `catkin_package()`

- **Issue 4:**

Error message: roscore command is not working

```
lentin@lentin-Aspire-4755:~$ roscore
^C... logging to /home/lentin/.ros/log/6d2860a2-6f48-11e5-b76e-9439e54d7dda/roslaunch-lentin-Aspire-4755-5045.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.
```

Figure 21: roscore command is not running properly

Solution: One of the reasons that can hang the roscore command is the definition of `ROS_IP` and `ROS_MASTER_URI`. When we run ROS in multiple computers, each computer has to assign its own IP as `ROS_IP`, and `ROS_MASTER_URI` as the IP of the computer, which is running `roscore`. If this IP is incorrect, `roscore` will not run. This error can be generated by assigning an incorrect IP on these variables.

```
export MY_IP=10.42.0.11
export ROS_IP=$MY_IP
export ROS_MASTER_URI="http://10.42.0.11:11311"
```

Figure 22: Incorrect `ROS_MASTER_URI`

- **Issue 5:**

Error message: Compiling and Linking Errors

```
Linking CXX executable /home/lentin/ros_catkin_ws/devel/lib/hello_world/talker
Linking CXX executable /home/lentin/ros_catkin_ws/devel/lib/hello_world/listener
CMakeFiles/talker.dir/src/talker.cpp.o: In function `main':
talker.cpp:(.text+0x61): undefined reference to `ros::init(int&, char**, std::string const&, unsigned int)'
talker.cpp:(.text+0xbd): undefined reference to `ros::NodeHandle::NodeHandle(std::string const&, std::map<std::string, std::string, std::less<std::string>, std::allocator<std::pair<std::string const, std::string> > > const&)'
talker.cpp:(.text+0x223): undefined reference to `ros::console::g_initialized'
talker.cpp:(.text+0x233): undefined reference to `ros::console::initialize()'
talker.cpp:(.text+0x288): undefined reference to `ros::console::initializeLogLocation(ros::console::LogLocation*, std::string const&, ros::console::levels::Level)'
talker.cpp:(.text+0x2c9): undefined reference to `ros::console::setLogLocationLevel(ros::console::LogLocation*, ros::console::levels::Level)'
talker.cpp:(.text+0x2d3): undefined reference to `ros::console::checkLogLocationEnabled'
```

Figure 23: Compiling and linking errors

Solution: If the `CMakeLists.txt` has no dependencies, which are required to compile the ROS nodes, it can show this error. We have to check the package dependencies in `package.xml` and `CMakeLists.txt`. Here we are generating this error by commenting `roscpp` dependencies.

```
find_package(catkin REQUIRED COMPONENTS
# roscpp
  rospy
  std_msgs
)
catkin_package()

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

Figure 24: `CMakeLists.txt` without package dependency

Some of the troubleshooting tips from ROS wiki are given at <http://wiki.ros.org/ROS/Troubleshooting>.

Questions

1. Why do we need an IDE to work with ROS?
2. What are the common naming conventions used in ROS?
3. Why is documentation important when we create a package?
4. What is the use of the `roswtf` command?

Summary

In this chapter, we discussed working with an Eclipse IDE and setting the ROS development environment inside IDE. After setting ROS in Eclipse, we discussed some of the best practices in ROS that consist of naming conventions, coding styles, best practices while creating a ROS package, and so on. After discussing best practices, we switched to ROS troubleshooting. In the troubleshooting section, we discussed various troubleshooting tips which can occur when we work with ROS.

Index

Symbol

3D sensor data
visualizing 97

A

ABB-IRB6640 robot
MoveIt! IK-Fast plugin, creating for 404
ABB robot model
visualizing, in RViz 398, 399
Accelerometer ADXL 335
interfacing 221-223
Adaptive Monte Carlo Localization (AMCL)
about 3, 116, 306
defining 317-319
used, for implementing autonomous navigation 151
Allowed Collision Matrix (ACM) 122, 342
AMCL launch file
creating 152-154
AMCL ROS package
URL 317
Apache 2.0 376
Arduino 204, 205
Arduino Publisher
example 213-217
Arduino Subscriber
example 213-217
arm
joint position controllers,
interfacing to 102, 103
joint state controllers,
interfacing to 102, 103

ASIMOV Robotics

URL 366

Asus Xtion Pro

interfacing, in ROS 265-268

autonomous navigation

implementing, AMCL used 151
implementing, static map used 151

B

basic real-time Joint controller, writing in ROS

about 184

CMakeLists.txt, updating 188

controller, building 188

controller configuration file, writing 189

controller header file, creating 184, 185

controller package, creating 184

controller, running with PR2

simulation 190

controller source file 187

controller source file, creating 185

launch file, writing for controller 189

package.xml, updating 188

plugin description file, creating 188

basic world plugin

creating 173-176

best practices, in ROS package

about 432

packaging 433

version control 432

blink LED

interfacing 218-220

bloom

reference link 51

Boost library 46
BSD (legacy) 376
button handling
 running, in Odroid-C1 246

C

calculator application
 plugins, creating for 158, 159
camera_calibration package 252
camera_calibration_parses package 252
camera_info_manager package 252
Chefbot
 defining 290-292
 Navigation stack, configuring for 306
Chefbot firmware
 flashing, Energia IDE used 293
Chefbot interface packages
 defining, on ROS 296-302
 files and folders 296
Chefbot simulation
 model files, adding in Gazebo model folder 331, 332
 room, building in Gazebo 329, 330
 working with 329
CMake (Cross Platform Make) 26
C++ nodes
 used, for running robot stand alone
 launch file 306
COLLADA file of robot
 creating, for OpenRave 408, 409
collision checking, MoveIt!
 about 122
 collision object, adding 342-346
 collision object, removing from planning scene 346
 performing 342
 self collision, checking 347, 348
collision properties
 adding, to URDF model 70
commands, ROS packages
 catkin_create_pkg 8
 catkin_make 8
 rosdep 8
 rospack 8

COOL arm robot
 about 367
 controller package, creating 368-372
 MoveIt! configuration 372, 373
custom messages
 creating, in ROS package 34-37
cv_bridge
 used, for converting between ROS and OpenCV 259
 used, for converting OpenCV-ROS images 262, 263

D

dependencies, ROS package
 actionlib 28
 actionlib_msgs 28
 roscpp 27
 std_msgs 27
Depth Image Occupancy Map Updater plugin 351
depth_image_proc package 253
depthimage_to_laserscan package
 about 270
 reference link 270
differential drive mobile robot
 robot model, creating for 82-86
differential drive robot
 SLAM, running on 148-151
differential wheeled robot
 simulating, in Gazebo 106, 107
Digital Motion Processor (DMP) 292
disk dump (dd) 234
Displays panel, RViz tool
 about 193
 reference link 193
DIY mobile robot
 defining 290-292
 hardware configuration 290-292
Dockable panels, RViz tool 193
Do It Yourself (DIY) 290
dynamic_reconfigure package
 reference link 21
dynamic link libraries (.DLL) 158
Dynamixel
 about 246
 URL 247

Dynamixel actuators
interfacing, to ROS 246

Dynamixel-ROS interface
about 366
dynamixel_controllers 366
dynamixel_driver 366
dynamixel_msgs 366

Dynamixel Servos
about 364, 365
ROS controllers 364
URL 366

E

Eclipse
ROS Catkin package, adding to 421-426
run configurations, adding to 427-429

Eclipse IDE
global settings 418, 419
ROS development environment,
setting in 417
setting, on Ubuntu 14.04.3 416, 417
URL, for downloading 416

edge detected image
visualizing 264

edges
finding, on image 263

encoder ticks
odometry, computing from 302-304

Energia IDE
URL 293
used, for flashing Chefbot firmware 293

Ethernet hotspot
configuring, for Odroid-C1 236, 237
configuring, for Raspberry Pi 2 236, 237

F

Fast SLAM 144

FCL (Flexible Collision Library)
reference 342

filtered_cloud_topic parameter 352

first URDF model
creating 64

folders, ROS packages
action 8
CMakeLists.txt 8
config 8

include/package_name 8
launch 8
msg 8
package.xml 8
scripts 8
src 8
srv 8

G

Gazebo
about 89
differential wheeled robot,
simulating in 106, 107
laser scanner, adding to 107-109
mobile robot, adding in 109, 110
robot joints, moving with ROS
controllers 99
ROS controller, interacting with 100-102
ROS controllers, launching with 103, 104
URL 89
used, for simulating robotic arm 90

gazebo_models
reference link 89

gazebo-msgs package 90

Gazebo plugins
about 172
model plugin 172
sensor plugin 172
system plugin 172
visual plugin 172
world plugin 172

gazebo-plugins package 90

gazebo-ros-control package 90

gazebo_ros_pkgs package 90

global settings, Eclipse IDE 418, 419

gmapping
Launch file, creating for 146
reference link 146

gmapping node
configuring 307, 308
URL 307

grasping
with MoveIt! 355-357

grasp object
creating, in MoveIt! 360-363

grasp table
creating, in MoveIt! 360-363

H

Hard kernel 231
hardware interfaces, ROS
Joint Command Interfaces 100
Joint State Interfaces 100
Hokuyo Laser
interfacing, in ROS 273-275

I

IK-Fast CPP file
generating, for IRB 6640 robot 410
IK Fast Module 405
image-common meta package
about 252
camera_calibration_parses 252
camera_info_manager 252
image_transport 252
polled_camera 252
image_geometry package 250
image-pipeline meta package
about 252
camera_calibration 252
depth_image_proc 253
image_proc 253
image_rotate 253
image_view 253
stereo_image_proc 253
image processing, with OpenCV 260
image processing, with ROS 260
image_proc package 253
image_rotate package 253
images
edges, finding on 263
publishing, image_transport used 261
subscribing, image_transport used 261
image_transport package
about 252
reference link 252
image-transport-plugins 253
image_view package 253
industrial-core stack
industrial-core 378

industrial_DEPRECATED 378
industrial_msgs 378
industrial_robot_client 378
industrial_robot_simulator 378
industrial_trajectory_filters 378
simple_message 378

industrial robot
MoveIt! configuration, creating for 382-385
URDF, creating for 380

industrial robot client nodes

designing 400, 401

installation instruction, ROS distribution
reference link 22

installation

MoveIt! 116
ROS-Industrial packages 377, 378
ROS Navigation stack 145

installation, rosserial packages
on Ubuntu 14.04/15.04 208-211

installation, Wiring Pi
on Odroid-C1 238
on Raspberry Pi 2 239

Intel Real Sense camera

interfacing, on ROS 268-270

interfacing packages, OpenCV
cv_bridge 250
image_geometry 250

interfacing packages, PCL
about 251
pcl_conversions 251
pcl_msgs 251
pcl_ros 251
pointcloud_to_laserscan 251

IRB 6640 robot
IK-Fast CPP file, generating for 410

J

Java Runtime Environment (JRE) 416

joint position controllers

interfacing, to arm 102, 103

joint state controllers

interfacing, to arm 102, 103

joint state publisher

about 80

adding, in launch file 110

reference link 81

K

Kinect

about 141
interfacing, in ROS 265-268

Kinematic and Dynamics Library (KDL) 61

kinematics handling, MoveIt! 121

L

laser-pipeline package

about 253
laser_assembler 253
laser_filter 253
laser_filters 253
laser_geometry 253

laser scanner

adding, to Gazebo 107-109

laser scans

point cloud, converting to 270-273

launch file

creating 48, 50
creating, for gmapping 146
creating, for nodelets 170-172
joint state publishers, adding in 110

LED blink

running, in Odroid-C1 246
running, in Raspberry Pi 2 246

Light Detection and Ranging (LIDAR) 275

Linux

OS Image, installing in 234

M

map

building, SLAM used 146

MeshLab

URL 381

methodology, in building RViz plugin

about 194
CMakeLists.txt, editing 198, 199
export tags, adding in package.xml 198
plugin, building 199, 200
plugin description file, creating 198
plugin, loading 199, 200
RViz plugin definition, creating 196, 197
RViz plugin header file, creating 195, 196
RViz plugin package, creating 195

mobile robot

moving, in Gazebo 109, 110

model plugin 172

motion planning, MoveIt!

about 118, 119
custom path, MoveIt! C++ APIs
used 340, 341
random path, MoveIt! C++ APIs
used 338, 339
with move_group C++ interface 338

motion planning request adapters, MoveIt! 120

motor velocities

computing, from ROS twist
message 305, 306

move_base node

about 143
packages 143, 144

move_group node 117, 118

MoveIt!

about 115
installing 116
URL 115

MoveIt! architecture

about 116
collision checking 121, 122
kinematics handling 121
motion planning 118, 119
motion planning request adapters 120
move_group node 117, 118
planning scene 120
reference link 116

MoveIt! configuration

creating, for Industrial robot 382-385
configuration files, updating 385, 386
testing 387

MoveIt! configuration, of ABB robots

working with 394-396

Moveit! configuration, of universal

robotic arm 390-393

MoveIt! configuration package

used, for motion planning of robot 130

MoveIt! configuration package, generating

with Setup Assistant tool

about 122

configuration files, generating 129

passive joints, adding 128

- planning groups, adding 126
robot end effector, setting up 128
robot poses, adding 127
Self-Collision matrix, generating 124, 125
Setup Assistant tool, launching 123
virtual joints, adding 125
- MoveIt! configuration package, interfacing to Gazebo**
about 134
controller configuration file, creating for Gazebo 136, 137
controller configuration file, writing for MoveIt! 134, 135
controller launch files, creating 135, 136
Gazebo-MoveIt! interface, debugging 139, 140
launch file, creating for Gazebo trajectory controllers 137, 138
- MoveIt! IK-Fast plugin**
about 404
creating 411, 412
creating, for ABB-IRB6640 robot 404
developing, prerequisites 404
installing 405
- MoveIt! Layer** 379
- MoveIt! package** 342
- MPU 6050 IMU** 292
- ## N
- Navigation packages**
Collision Recovery behaviour 145
Command Velocity, sending 145
goal, sending 145
localizing, on map 144
path planning, sending 145
working with 142
- Navigation stack**
about 115, 140
base_controller 142
configuring, for Chefbot 306
hardware requisites 141, 142
goal, sending to 333-336
odometry data 142
sensor source 142
sensor transforms/tf 142
used, for avoiding obstacle 328
- working with 144
- Navigation stack packages**
base local planner parameter, configuring 312
common configuration 310, 311
configuring 308, 309
DWA local planner parameters, configuring 313
global costmap parameters, configuring 311
local costmap parameters, configuring 312
move_base node parameters, configuring 314-316
parameters 316
- nodelet manager** 168
- nodelets**
about 165
building 168-170
CMakeLists.txt, editing 168
creating 165
explanation, of hello_world.cpp 166, 167
export, adding in package.xml 168
hello_world.cpp nodelet, creating 166
launch file, creating for 170-172
package, creating for 166
plugin description file, creating 167
running 168-170
- Non-Arduino boards**
interfacing, to ROS 230
- ## O
- OctoMap**
about 342
URL 342
- odometry**
computing, from encoder ticks 302-304
- Odometry Publisher**
example 228, 229
- Odroid**
webcam, streaming from 285-287
- Odroid-C1**
Button handling, running in 246
Ethernet hotspot, configuring for 236
LED Blink, running in 246
ROS, setting on 230-233
Wiring Pi, installing on 238

Odroid-C1 connection, from PC 235
official OS images, Raspberry Pi 2
 reference link 233

OMPL
 URL 119

OpenCV
 about 249, 250
 interfacing packages 250
 URL 250
 used, for image processing 260

OpenCV-ROS images
 converting, cv_bridge used 262, 263

OpenRave
 about 405
 COLLADA file of robot, creating
 for 408, 409
 installing, on Ubuntu 14.04.3 406, 407

OpenSLAM
 URL 146

Open Source Computer Vision. *See*
 OpenCV

operating system, supported on Odroid-C1
 reference link 233

OS Image
 installing, in Linux 234
 installing, in Windows 233, 234

P

package
 creating, for nodelets 166

packages, move_base node
 amcl 144
 clear-costmap-recovery 143
 costmap-2D 144
 global-planner 143
 gmapping 144
 local-planner 143
 map-server 144
 rotate-recovery 143

packet representation, rosserial protocol
 Checksum of Topic ID and data 206
 Checksum over message length 206
 Message Length 206
 Serialized Message data 206
 Sync Flag 206

Sync Flag/Protocol version 206
Topic ID 206

PCD file
 point cloud data, writing to 282
 point cloud, reading from 282-285

PCL
 about 249
 interfacing packages 251

pcl_conversions package 251
pcl_msgs package 251
pcl_ros package 251

perception
 working with, MoveIt! and Gazebo
 used 349-354

perception-pcl stack 253

physical properties
 adding, to URDF model 70

pick and place
 performing, in Gazebo 363, 364

planning request adapters, MoveIt!
 AddTimeParameterization 120
 FixStartStateBounds 120
 FixStartStateCollision 120
 FixStartStatePathConstraints 120
 FixWorkspaceBounds 120

planning scene, MoveIt!
 about 120
 reference link 120

pluginlib
 about 158
 used, for creating plugins for calculator
 application 158, 159

pluginlib_calculator package
 calculator_base header file, creating 160
 calculator_plugins header file,
 creating 160, 161
 CMakeLists.txt file, editing 164
 list of plugins, querying in package 164
 plugin description file, creating 162, 163
 plugin loader, implementing with
 calculator_loader.cpp 162
 plugin loader, running 164
 plugin, registering with ROS package
 system 163
 plugins, exporting with calculator_plugins.
 cpp 161, 162
 working with 159

plugins
about 158
calculator application, creating for 158, 159

point cloud
converting, to laser scans 270-273
processing 280, 281
publishing 278-285
reading, from PCD file 282-285
subscribing 280, 281

point cloud data
working with 278
writing, to PCD file 282

PointCloud Occupancy Map Updater
plugin 351

pointcloud_to_laserscan package
about 270
reference link 270

polled_camera package 252

Pololu DC Gear motor with Quadrature encoder
URL 291

Pololu motor drivers
URL 291

pr2_controller_interface package
about 181
Controller, stopping 183
initialization of Controller 182
ROS Controller, starting 182
ROS Controller, updating 183

pr2_controller_manager package 183

pr2_mechanism packages 181

pr2_mechanism stacks
pr2_controller_interface 180
pr2_controller_manager 180
pr2_hardware_interface 180
pr2_mechanism 180
pr2_mechanism_model 180
pr2_mechanism_msgs 180

prerequisites, ROS
about 22
ROS Jade/Indigo desktop full installation 22
Ubuntu 14.04.2 LTS / Ubuntu 15.04 22

probabilistic robotics
URL 317

Push Button
interfacing 218-220

Q

Qt 194

R

Raspberry Pi
reference link 231

Raspberry Pi 2
Ethernet hotspot, configuring for 236, 237
LED Blink, running in 246
ROS, setting on 230-233
Wiring Pi, installing on 239

Raspberry Pi 2 connection, from PC 235

raw image
visualizing 264

release
ROS package, preparing for 52

robot
components 291

robot 3D model
pan 69
tilt joints 69
visualizing, in RViz 68

robot arm specification, of seven DOF arm
about 75
joints 75

robot description
creating, for seven DOF robot manipulator 74
ROS package, creating for 64

robotic arm, simulating
Gazebo used 90
ROS used 90
Xtion Pro used 96

robotic arm simulation model, for Gazebo
3D vision, adding to Gazebo 94, 95
about 91, 92
colors, adding to Gazebo robot model 93
gazebo_ros_plugin, adding 94
textures, adding to Gazebo robot model 93
transmission tags, adding 93

robot joints
moving 105
moving, ROS controllers used 99

robot model
creating, for differential drive mobile robot 82-85

robot modeling, with URDF
about 61
gazebo tag 64
joint tag 62
link tag 61
robot tag 63

robot modeling, with xacro
about 71
macros, using 73
math expression, using 73
programmability 72
properties, using 72
simplify URDF 71

Robot Operating System. *See* **ROS**

robot pick and place task
working with, MoveIt! used 358, 359

robot stand alone launch file
running, C++ nodes used 306

robot state publisher 81

robot, with ROS navigation package
requirements 290-292

ROS
about 2
Asus Xtion Pro, interfacing in 265-268
benefits 2-4
best practices 429
Dynamixel Actuators, interfacing to 246
Hokuyo Laser, interfacing in 273-275
Intel Real Sense camera, interfacing
on 268-270
Kinect, interfacing in 265-268
limitations 4, 5
Non-Arduino boards, interfacing to 230
prerequisites 22
setting, on Odroid-C1 230-233
setting, on Raspberry Pi2 230-233
troubleshooting tips 433
USB webcams, interfacing in 254-256
used, for image processing 260
used, for simulating robotic arm 90
Velodyne LIDAR, interfacing in 275-277
webcam, starting from Odroid 285-287

ROS action client
building 46-48
creating 46

ROS actionlib
applications 50, 51

feedback 42
goal 42
result 42
working with 42, 43

ROS action server
building 46-48
creating 43-45

rosbag command
commands 18
reference links 19

ROS bags 12, 18

rosbash commands
reference link 8
rorun 8
roscd 8
roscl 8
rosed 8

ROS Camera Calibration
working with 256-258

ROS Catkin package
adding, to Eclipse 421-426

ROS C++ coding style guide
about 429
code documentation 432
code formatting 431
code license agreement 430
console output 432
standard naming conventions 429, 430

ros_comm
reference link 12

ROS Community Level
about 22
blog 22
bug ticket system 22
distributions 22
mailing lists 22
repositories 22
ROS Answers 22
ROS Wiki 22

ROS compile script, for Eclipse IDE 419-421

ROS Computation Graph Level
about 12
bags 14
master 13
messages 14
nodes 13
parameter server 13

services 14
topics 14

ROS controller
interacting, with Gazebo 100, 102
joint_effort_controller 100
joint_position_controller 100
joint_state_controller 100
launching, with Gazebo 103, 104
used, for moving robot joints 99

ROS controller, for PR2 robot
reference link 179

ros_control packages
about 99, 191
controller_interface 99
controller_manager 99
controller_manager_msgs 99
control_toolbox 99
hardware_interface 100
reference link 191
transmission_interface 100

roscore command output
checking 25, 26

ROS development environment
setting, in Eclipse IDE 417

rosdistro
reference link 53

ROS file system level
about 5
messages 6
meta packages 6
meta packages manifest 6
package manifest 6
packages 6
repositories 6
services 6

ROS Graph layer 13

ROS GUI layer 379

ROS-I Application Layer 379

ROS-I Controller Layer 380

ROS-I GUI 379

ROS-I Interface Layer 380

ROS-Industrial
benefits 377
goals 376
history 377

ROS-Industrial packages
about 376
block diagram 378
installing 377, 378
installing, of universal robotic arm 387, 388

ROS-Industrial robot driver
package 401-403

ROS-Industrial robot support
packages 396-398

ROS interface
installing, of universal robots 388-390

ROS-I Simple message Layer 380

ROS-I wiki page
URL 378

roslaunch 48

ROS Layer 379

ROS Master
about 12, 19, 20
running 23-25

ROS messages 10-16

ROS meta packages 9

rosmsg
parameters 16

ROS Navigation stack
installing 145

rosnode
usage 15

ROS Node APIs, in Arduino 211-213

ROS nodes
about 12-16
building 32-34
creating 28-32

ROS package
about 7
best practices 432
commands 8
creating 26, 27, 260
creating, for robot description 64
custom messages, creating in 34-37
dependencies 27, 28
folders 8
maintaining 51
preparing, for release 52
releasing 51-55
service definitions, creating in 34-37
URL, for source code 51

Wiki page, creating for 55, 57

ROS packages, for robot modeling

- about 60
- joint_state_publisher 60
- kdl_parser 61
- robot_model 60
- robot_state_publisher 61
- urdf 60
- xacro 61

ROS Parameter

- using 20, 21

ROS Parameter Server

- running 23, 24

rosparam tool

- commands 21

ROS perception

- installing 252

ROS perception stack

- image-common 252
- image-pipeline 252
- image-transport-plugins 253
- laser-pipeline 253
- perception-pcl 253
- vision-opencv 253

ROS project

- references 2

rosserial 206

rosserial_client libraries

- rosserial_arduino 207
- rosserial_embeddedlinux 207
- rosserial_java 207
- rosserial_python 207
- rosserial_server 207
- rosserial_windows 207

rosserial packages

- installing, on Ubuntu 14.04/15.04 208-211

rosserial_python package

- reference link 215

ROS services

- about 12-18
- applications 50, 51
- working with 37-42

rosservice tool

- usage 18

ROS Teleop node

- adding 111, 112

ROS topics

- about 12, 16
- applications 50, 51
- syntax 17
- working with 28

ROS Visualization Tool. *See RViz tool*

roswtf

- reference link 434
- using 434-436

rqt_bag

- reference link 19

rqt_graph tool

- reference link 14

run configurations

- adding, to Eclipse 427-429

RViz

- ABB robot model, visualizing in 398, 399
- robot 3D model, visualizing in 68

RViz Motion Planning plugin 345

RViz plugin

- writing, for teleoperation 194

RViz tool

- about 192
- Displays panels 193
- Dockable panels 193
- Time panel 193
- Views panel 193

RViz toolbar

- about 193
- reference link 193

RViz, with Navigation stack

- 2D Nav Goal button 322
- 2D Pose Estimate button 320
- defining 320
- global and local cost map, displaying 325, 326
- global plan, displaying 326
- goal 327
- local plan, displaying 326
- particle cloud, visualizing 321
- planner plan, displaying 326
- robot footprint, displaying 324
- static map, displaying 323

S

Semantic Description Format (SDF) 329
sensor plugin 172
serial communication (UART)
 used, for sending data from LaunchPad to PC 294
 used, for sending data from PC to Launchpad 295
services definitions
 creating, in ROS package 34-37
seven DOF Dynamixel based robotic arm
 interfacing, to ROS MoveIt! 366, 367
seven DOF arm, viewing in RViz
 about 79
 join state publisher 80
 robot state publisher 81
seven DOF robot manipulator
 robot description, creating for 74
shared objects (.so) 158
Simultaneous Localization And Mapping (SLAM)
 about 3, 116
 running, on differential drive robot 148-151
 used, for building map 146
Southwest Research Institute (SwRI)
 URL 377
static map
 used, for implementing autonomous navigation 151
stereo_image_proc package 253
support queries, ROS
 reference link 4
system plugins 172

T

TCPROS 17
teleoperation
 RViz plugin, writing for 194
time panel, RViz tool 193
Tiva C Launchpad Controller
 URL 291
troubleshooting tips, ROS
 about 433
 reference link 437

TTL (Transistor-Transistor Logic) 364

U

Ubuntu 14.04.3
 Eclipse IDE, setting on 416, 417
 OpenRave, installing on 406, 407
Ubuntu 14.04/15.04
 rosserial packages, installing on 208-211
UDPROS 17
ultrasonic distance sensor 224
ultrasonic range sensor
 using 225-227
Unified Robot Description Format (URDF)
 about 60
 creating, for Industrial robot 380
 Xacro, converting to 73
universal robotic arm
 ROS-Industrial packages, installing of 387, 388
universal robots
 ROS interface, installing of 388-390
universal robot stack, packages
 ur10_moveit_config/ur5_moveit_config
 389
 ur_bringup 388
 ur_description 388
 ur_driver 388
 ur_gazebo 389
 ur_kinematics 389
 ur_msgs 389
URDF design, followed by ROS-I
 Collision-Aware 381
 Standards Frames 381
 URDF Joint conventions 381
 Xacro Macros 381
URDF file 66-68
URDF model
 collision properties, adding to 70
 physical properties, adding to 70
URDF tags
 reference link 64
USB webcams
 interfacing, in ROS 254-256

V

Velodyne LIDAR

interfacing, in ROS 275, 276

Version Control System (VCS) 6

view panel, RViz tool 193

vision-opencv stack 253

visual plugin 172

W

webcam

streaming, from Odroid 285-287

Wiki page

creating, for ROS package 55-57

Win32diskimage tool

about 233

reference link 233

Windows

OS Image, installing in 233, 234

Wiring Pi, installing

on Odroid-C1 238

on Raspberry Pi 2 239

world plugin 172

X

Xacro

converting, to URDF 73

xacro model, of seven-DOF arm

about 75

constants, using 76

macros, using 76

meshes, using in links 77

other xacro files, using 77

robot gripper 78

XMLRPC (XML Remote Procedure Call) 19

Xtion Pro

robotic arm, simulating with 96

Y

Yaskawa Motoman Robotics

URL 377

YUV

URL 255



Thank you for buying **Mastering ROS for Robotics Programming**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

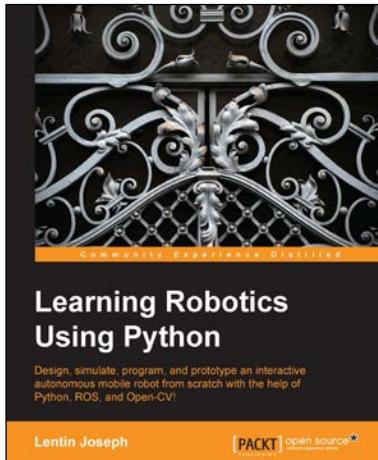
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



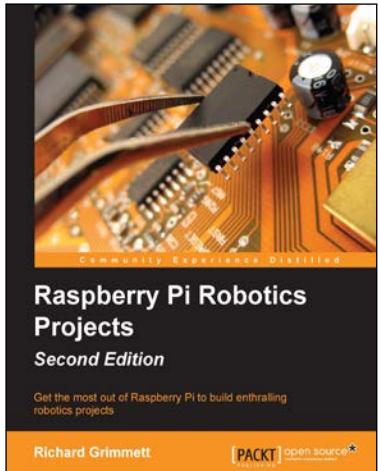
Learning Robotics Using Python

ISBN: 978-1-78328-753-6

Paperback: 330 pages

Design, simulate, program, and prototype an interactive autonomous mobile robot from scratch with the help of Python, ROS, and Open-CV!

1. Design, simulate, build and program an interactive autonomous mobile robot.
2. Program Robot Operating System using Python.
3. Get a grip on the hands-on guide to robotics for learning various robotics concepts and build an advanced robot from scratch.



Raspberry Pi Robotics Projects Second Edition

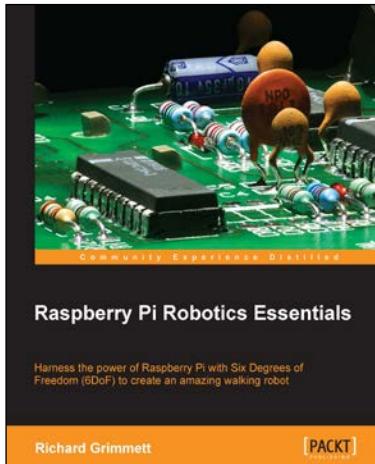
ISBN: 978-1-78528-014-6

Paperback: 300 pages

Get the most out of Raspberry Pi to build enthralling robotics projects

1. Make your projects talk and understand speech with the Raspberry Pi.
2. Use a standard webcam to make your projects see and enhance vision capabilities.
3. Full of simple, easy-to-understand instructions to bring your Raspberry Pi online to develop robotics projects.

Please check www.PacktPub.com for information on our titles



Raspberry Pi Robotics Essentials

ISBN: 978-1-78528-484-7 Paperback: 158 pages

Harness the power of Raspberry Pi with Six Degrees of Freedom (6DoF) to create an amazing walking robot

1. Construct a two-legged robot that can walk, turn, and dance.
2. Add vision and sensors to your robot so that it can "see" the environment and avoid barriers.
3. A fast-paced, practical guide with plenty of screenshots to develop a fully functional robot.



Mastering BeagleBone Robotics

ISBN: 978-1-78398-890-7 Paperback: 234 pages

Master the power of the BeagleBone Black to maximize your robot-building skills and create awesome projects

1. Create complex robots to explore land, sea, and the skies.
2. Control your robots through a wireless interface, or make them autonomous and self-directed.
3. This is a step-by-step guide to advancing your robotics skills through the power of the BeagleBone.

Please check www.PacktPub.com for information on our titles

